Verteilte Systeme und Komponenten

Zusammenfassung Frühlingssemester 2018

Patrick Bucher

21.06.2018

Inhaltsverzeichnis

1 Komponenten

Herkunft: *componere* (lat.) = zusammensetzen

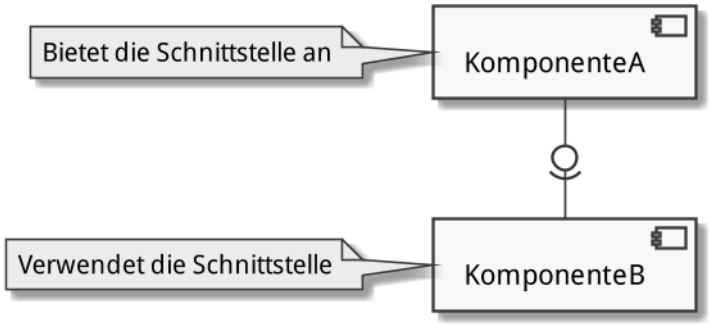

Abbildung 1: Komponentendiagramm (UML2)

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 Versionskontrolle

SCM: Source Code Management

- hält zeitliche Entwicklung von Artefakten fest
- erlaubt Rückgriff auf frühere Revisionen (Änderungsstände)
- ermöglicht Zusammenarbeit an gemeinsamen Quellen im Team
- automatisiertes Zusammenfügen von Änderungen (Merging) – soweit möglich (Konflikte)
- zentrale oder verteilte Datenhaltung – oder beides
- können fehlende Koordinaten *nicht* ersetzen
- machen Änderungen an Artefakten nachvollziehbar
  - als *Changesets* innerhalb einer Transaktion gespeichert
  - $1..n$ Dateiartefakte werden vom konsistentem Zustand z_1 in den konsistenten Zustand z_2 überführt
- sind *keine* Backupsysteme

2.1.1 Arbeiten mit SCM

- Grundlegende Operationen:
  - `checkout`: lokale Arbeitskopie von Repository erstellen
  - `update`: Änderungen von Repository in der lokalen Arbeitskopie aktualisieren
  - `log`: Revisionen von Artefakten einsehen
  - `diff`: Vergleich verschiedener Revisionen zweier Artefakte
  - `commit`: Änderungen in das Repository schreiben
- Operationen verteilter SCM:
  - `clone`: lokale Kopie eines entfernten Repositories erzeugen
  - `fetch`: Änderungen eines entfernten Repositories herunterladen
  - `pull`: Änderungen eines entfernten Repositories lokal nachtragen/einpflegen
  - `push`: Lokale Änderungen auf entferntes Repository schreiben
- Tagging: Markieren eines bestimmten Revisionsstands mit Namen oder Versionsnummer
  - CVS: Markierung auf Dateiebene
  - Subversion (SVN): Kopie in ein Verzeichnis
  - `git`, Mercurial (hg): Identifikation der Revision über das ganze Dateisystem des Repositories
- Branching: Voneinander unabhängig bearbeitbare Entwicklungszweige
  - für Bugfixing, Prototypen (Tests, Experimente), neue Features
  - Branch wird entweder verworfen (abgebrochene Experimente) …
  - … oder in den Hauptzweig eingepflegt (Merge)
- Inhalte: Was gehört ins Repository?
  - Ja: Quellcode (`*.java`), Konfigurationsdateien (`*.xml`), Dokumentation (`*.md`, `*.txt`)
  - Nein: Kompilate (`*.class`), generierte Dokumente (HTML-Reports)
  - Filterregeln: `.gitignore` (`git`) bestimmt, was ins Repository kommt

2.1.2 Verschiedene SCM

- Unterschiede zwischen verschiedenen SCM
  - zentral oder verteilt
  - optimistische oder pessimistische Lockverfahren
  - Versionierung anhand Datei, Verzeichnisstruktur oder Changeset
  - mit oder ohne Transaktionsunterstützung
  - weitere Features: Zugriffsprotokolle, Sicherheitsmassnahmen, Webserver-Integration
- Beispiele bekannter SCM:
  - CVS: zentral, robust, stabil, alt, Verbreitung rückläufig
  - Subversion: zentral, transaktionsorientiert, Versionierung auf Verzeichnisebene, flexibler und anspruchsvoller als CVS, als CVS-Nachfolger angetreten (bereits wieder rückläufig)
  - `git`: dezentral, weit verbreitet, «billiges» Branching (schnell und einfach), skalierbar (einfach und schnell bei kleinen Repositories, funktioniert auch für gigantische Repositories – Linux-Kernel), aufgrund dezentraler Struktur nicht ganz einfach zu verstehen

- Code-Hosting-Plattformen: GitLab, GitHub
  - Verwaltung von Teams (Berechtigungen), Projekten (Issue-Tracking)
  - Werkzeuge zur Bearbeitung von Merge-Requests (GitLab), Pull-Requests (GitHub)
  - Webseiten, Wikis, Durchsuchen des Source-Trees
  - Community-Gedanke (GitHub), Social Network für Nerds
- Praktische Empfehlungen für die Arbeit mit SCM
  - Vor Arbeitsbeginn das lokale Repository auf den neuesten Stand bringen (`pull` bzw. `checkout`)
  - Aussagekräftige `commit`-Messages verwenden, am besten mit Verweis auf Issue-/Tasknummer
  - Lieber öfters kleinere Änderungen als selten grössere Änderungen schreiben (weniger und kleinere Merge-Konflikte)

2.2 Buildautomatisierung

Buildprozess: aus Quellartefakten ein fertiges Produkt erstellen

- Generieren, Kompilieren, Testen, Packen, JavaDoc erzeugen etc.
- Per IDE möglich (manuell), mühsam und fehleranfällig bei mehreren Buildvorgängen pro Tag
- Per Skript
  - Vorteile: automatisierter Ablauf, reproduzierbare Ergebnisse, nächtlich getriggerte Ausführung möglich, Unabhängigkeit von der IDE
  - Nachteile: unflexibler Ablauf – oder aufwändige Skripte, Abhängigkeit von Shell und Plattform, aufwändige Wartung und Erweiterung
- Per Build-Werkzeug: spezialisiertes Werkzeug mit eigener Skript- oder Definitionssprache
  - für Build-Aufgaben (Generieren, Kompilieren etc.) optimiert
  - vereinfachte Handhabung von Ressourcen (Dateimengen)
  - automatische Prüfung von und Steuerung durch Abhängigkeiten
  - Abstraktion der plattformspezifischen Eigenheiten: plattformübergreifend funktionierend
- Unterschiedliche Ansätze: imperativ vs. deklarativ – Vorgeben der Schritte oder Beschreibung des Resultats?
- Vorteile von Build-Werkzeugen
  - einfache und einheitliche Definition des Builds
  - einfache Handhabung mit Build-Targets
  - optimierte Abläufe: nur bei Änderungen neu kompilieren/generieren etc. (Auflösung des Abhängigkeitenbaums)
  - Erweiterbarkeit für neue Aspekte und projektspezifische Bedürfnisse
  - geringer Ressourcenverbrauch (ohne GUI, auf Server ausführbar)
  - reproduzierbarer Ablauf mit reproduzierbaren Ergebnissen
- Beispiele für Build-Werkzeuge
  - `make`: Urvater der Build-Tools, v.a. für C/C++ verwendet, sehr flexibel einsetzbar
  - Ant: alt und bewährt, für Java mit XML

  - Maven: populär und etabliert, für Java mit XML
  - Gradle: populär und junges, mit Groovy-Script und DSL

2.2.1 Apache Maven

Apache Maven: deklaratives (XML), in Java entwickeltes und weit verbreitetes Build-Werkzeug

- schlank: Funktionalität aus dynamisch geladenen Plugins (Maven-Core-Plugins und grosse Auswahl von Drittanbietern)
- zentrales Binär-Repository search.maven.org zum Einbinden von Libraries
- Project Object Model: in `pom.xml` deklarierte Metainformationen (Targets, Plugins, Dependencies) für das Projekt
- Lifecycle-Phasen: generalisierter Ablauf mit typischen Build-Phasen
  - `validate`: Projektdefinition (`pom.xml`) überprüfen
  - `compile`: Quellen kompilieren
  - `test`: Ausführung der Unit-Tests
  - `package`: Packen der Distribution (`.jar`, `.ear`, `.war`)
  - `verify`: Ausführen der Integrationstests
  - `install`: Deployment (lokales Repository)
  - `deploy`: Deployment (zentrales Repository)
- Lokales Repository: `$HOME/.m2/repository` mit `$HOME/.m2/settings.xml` (Repository-Konfiguration)
- Module: Aufteilung des Projekts in Untermodule, Definition ihrer Abhängigkeiten, Vererbung von Modulkonfigurationen an Untermodule

2.3 Dependency Management

Dependency Management: Organisation und Techniken für den Umgang mit Abhängigkeiten zu anderen Modulen

- Abhängigkeiten: meist in Binärform (kompiliert), Zugriff über Binär-Repositories mit Paketmanagern
  - intern: Modul im selben Projekt
  - extern: Modul aus einem anderen Projekt, evtl. von anderer Organisation
- Merkmale von Dependency-Management-Software:
  - zentrale Ablage auf Server (oftmals mit verschiedenen Mirrors)
  - standardisiertes Paketformat
  - zusätzliche Metainformationen
  - definierte Abhängigkeiten
  - Konsistenzsicherung (automatisches überprüfen von Prüfsummen)
  - Suchfunktion und weitere Hilfswerkzeuge
- Beispiele für Dependency-Management-Software:
  - apt: Adcanced Packaging Tool (für Debian GNU/Linux und Derivate)

  - pip: Pip Installs Packages (Paketverwaltung für Python)
  - npm: Node Package Manager (Paketverwaltung für Node.js/JavaScript)
- Java Dependency Management
  - binäre Module: `.jar`, `.ear`, `.war`
  - kein Verfahren zur Definition von Abhängigkeiten, keine Modulverwaltung
  - Modularisierung ab Java 9, ohne Versionierung
  - Manuelles Verwalten von `CLASSPATH` und `.jar`-Dateien im `lib/`-Verzeichnis: *JAR Hell*
  - ab 2001: Maven als Buildsystem mit Dependency Management

2.3.1 Apache Maven

- Maven Repository
  - zahlreiche öffentliche Repositories mit Leserechten (z.B. Maven Central)
    - schreibrechte für ausgewählte Personen gemäss definierten Prozessen
  - interne Repositories bei professionellen Organisationen
    - diverse Produkte: Apache Archiva, Sonatype Nexus
    - HSLU: RepuHub Nexus
  - Zwischenspeicherung (Caching) in lokalem Repository `$HOME/.m2/repository`
- Weltweit eindeutige Identifikation: über *Maven Coordinates*, drei Attribute:
  1. GroupId: *Reverse Domain Name* der Organisation mit Zusatz: `ch.hslu.vsk.g05`
  2. ArtifactId: Projektname, Modulname: `logger-server`, `logger-viewer`
  3. Version: Empfohlen nach *Semantic Versioning*: `4.0.1`
  - Identifikation: `ch.hslu.vsk.g05:logger-server:1.1.3`
  - Deklaraton von Dependencies
- Dependency Scopes: Geltungsbereich (Scope) kann pro Abhängigkeit angegeben werden
  - `compile`: für die Kompilierung und Programmlaufzeit (Standard)
  - `test`: für die Kompilierung und Ausführung der Testfälle
  - `runtime`: für die Laufzeit, aber nicht für die Kompilierung
  - in IDEs mehr oder weniger gut umgesetzt (Referenzprüfung)
- Transitive Abhängigkeiten ($\rightarrow$: hängt ab von): $m_a \rightarrow m_b \rightarrow m_c$
  - m_a hängt direkt von m_b ab
  - m_b hängt direkt von m_c ab
  - m_a hängt *transitiv* von m_c ab
  - Maven erstellt einen Dependency-Graph, um die Abhängigkeiten aufzulösen; Konflikte und Zyklen zu erkennen und – soweit möglich – automatisch aufzulösen
- Versionierung und Snapshots: Dependencies sind grundsätzlich versioniert, dies ermöglicht:
  - Erkennung neuer Versionen
  - Automatische Verwendung von Aktualisierungen (Verbesserungen, Bugfixes)
  - Angabe kompatibler Versionsbereiche
  - Versionen sollen nicht überschrieben werden können (Nachvollziehbarkeit)
- Snapshots: erneuerbare, nicht stabile Version während Entwicklungsphase
  - mit dem Suffix `-SNAPSHOT` markierte Abhängigkeiten werden bei jedem Build aufgelöst und aktualisiert

  - Beispiel: `1.0.0-SNAPSHOT` ist die Weiterentwicklung von Version `1.0.0`, die noch nicht stabil ist und später als `2.0.0`, `1.1.0` oder `1.0.1` freigegeben wird.
- Multimodul-Projekte: Projekte, die aus mehreren Submodulen bestehen
  - Submodule haben gleiche Dependencies (`log4j`, `JUnit`) und sollten unbedingt die gleiche Version davon verwenden!
  - Problem: Konfiguration des Dependencies in jedem Submodul (Konflikte, mühsam nachzutragen)
  - Lösung: Übergeordnetes Master-POM mit `dependencyManagement` als Grundeinstellung für Version und Scope

Beispiel für `pom.xml`:

```
<project>
    <groupId>ch.hslu.vsk.g05</groupId>
    <artifactId>logger-server</artifactId>
    <version>1.1.3</version>
    <dependencyManagement>
        <!-- nur in Parent-POM -->
        <dependencies>
            <dependency>
                <groupId>org.apache.logging.log4j</groupId>
                <artifactId>log4j-api</artifactId>
                <version>2.10.0</version>
                <scope>compile</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>ch.hslu.vsk.g05</groupId>
            <artifactId>logger-common</artifactId>
            <version>1.1.0</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <!-- in Child-POM ohne Version und Scope -->
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
        </dependency>
    </dependencies>
</project>
```

2.4 Buildserver

Buildserver: Serversoftware, die Builds automatisch erstellt und das Resultat den Entwicklern zur Verfügung stellt

- Auslösung des Builds
  - automatisch aufgrund von Änderungen im SCM
  - automatisch zu bestimmten Zeitpunkten
  - manuell durch Anwender
- Vorteile von Buildservern
  - Entlastung der Entwickler und ihrer Rechner von repetitiven Aufgaben
  - regelmässige und häufige Verifikation (Build, Test, Deployment)
  - statistische Informationen über Entwicklungsprozess
  - automatische Informationen über den Projektzustand
- Produkte und Dienste
  - Jenkins/Hudson: sehr verbreitet, Open Source
  - Continuum: speziell für Maven-Projekte, Open Source
  - Bamboo: mit JIRA verknüpft, kommerziell
  - GitLab: Kombination von SCM und CI, Cloud
  - Travis: Mit GitHub und anderen Diensten kombinierbar, Cloud
- Konfiguration
  1. klassisch: Konfiguration von Projekt getrennt, vom Server vorgegeben
  2. DevOps-Ansatz: Konfiguration im Projekt (`.yml`-Datei) durch Entwickler
- Voraussetzungen
  - Automatisierung des Builds (Ant, Maven, Gradle)
  - Einsatz eines Versionskontrollsystems (Git, Subversion, CVS)
- Aufgabentrennung im CI-Prozess
  - *Wann* wird ein Build durchgeführt: Buildserver, Anwender
  - *Was* wird gebaut: Versionskontrollsystem
  - *Wie* wird gebaut: Buildautomatisierung (Ant, Maven, Gradle)
  - *Wohin* gelangen die Artefakte: Binary-Repo, Roll-Out auf Testsystem
  - Das *wie* sollte nicht mit dem Buildserver sondern mit der Buildautomatisierung umgesetzt werden, damit auch auf anderen Umgebungen (Entwicklerrechner) ein gleichwertiger Build durchgeführt werden kann.
- Verschiedene Buildszenarien
  - Continuous Build: automatisch bei Änderungen im SCM für schnelles Feedback
  - Nightly Build: zeizlich gesteuert (meistens nachts) für vollumfängliches Feedback
  - Release Build: manuell oder durch Versionstag ausgelöst für auslieferbares Produkt
- Integration und Verknüpfung von Buildservern
  - Buildtools
  - Versionskontrollsysteme
  - Kommunikationstechnologien für Notifikation
  - Auswertungen, Visualisierungen (auch für IDEs)
  - mit Issue-Tracking- und Code-Review-Systemen

2.5 Continuous Integration

Ziele von Continuous Integration:

- lauffähiges Buildresultat immer vorhanden (kontinuierliche Tests möglich)
- schnelles Feedback bei Fehlern (Unit- und Integrationstests, Kompilierung, Codeprüfung)
- parallele Entwicklung im Team (gemeinsamer, aktueller Stand und Überblick)
- agile Software-Entwicklung ermöglichen

2.5.1 10 Praktiken der Continuous Integration

1. Einsatz eines zentralen Versionskontrollsystems
    - alles, was für einen Build benötigt wird, aber nichts, was mit einem Build erstellt werden kann, liegt im SCM vor
    - sinnvolle Commit-Kommentare mit Referenz auf Issue
    - Tagging von Versionen zur Identifikation eines Release
    - temporäre Branches für parallele Entwicklung von Features und bei Bugs
2. Automatisierter Buildprozess
    - auf einer kontrollierten, stabilen, «sauberen» Umgebung
    - nur auf Basis der aktuellen Quellen aus dem SCM
    - inklusive Ausführung der automatischen Testfälle
3. Automatisierte Testfälle
    - möglichst hohe Abdeckung durch automatisierte Tests anstreben (möglichst mit Unit-Tests, sekundär mit Integrationstests)
    - schnelles Erkennen fehlerhafter und unvollständiger Implementierungen
    - ständige Performance-Tests zur Erkennung imperformanter Neuentwicklungen
    - Tests müssen immer laufen und sollen im Fehlerfall als erstes korrigiert werden
4. Änderungen des Quellcodes auf dem Hauptzweig
    - Feature-Branches oft in den Hauptzweig mergen
    - oft kleine statt selten grosse Changesets mergen
5. Automatischer Build bei Änderungen
    - Änderungen im SCM per push-Verfahren (hook) oder polling ermitteln und sofort ein Build aufgrund der neuesten Revision starten
    - Kommunikation der Buildergebnisse inkl. Metriken
    - Build sollte immer funktionieren; Buildfehler sollten als erstes korrigiert werden
    - Zielkonflikt: umfassender Build (inkl. Integrationstests) vs. schneller Build (kompilieren, Unit-Tests)
6. Schneller Build-Prozess
    - Entwickler bekommen sofort Feedback, gerade im Fehlerfall
    - nicht alle Tests werden lokal vor dem Commit ausgeführt (zeitintensive Integrationstests)
    - möglichst viele Tests sollen auf dem Buildserver ausgeführt werden
    - gestaffelte Builds: kleiner, sofortiger Build; grosser, nächtlicher Build
7. Tests auf Produktivumgebungen (oder mit Kopien davon)

    - Build- und Testumgebungen möglichst ähnlich zur Produktivumgebung (Hardware, Betriebssystem, Laufzeitumgebung, Netzwerkzugriff, Datenmenge, Datenqualität, Berechtigungen)
    - finanzieller Aspekt: Produktivsysteme oft besser ausgestattet
8. Einfacher Zugriff auf Buildartefakte
    - Bereitstellung der aktuellen Buildresultate für weitere (manuelle) Tests
    - Archivierung von Buildartefakten auf Buildserver (Tests mit *bestimmter* Version)
    - zusätzliches Deployment in binäres Repository (z.B. Maven-Repository)
9. Offensive Information über den aktuellen Zustand
    - Urheber und Zeitpunkt jeder Änderung sowie deren Auswirkung sind jederzeit für jeden einsehbar
    - nicht als Kontrollinstrument, sondern zur gegenseitigen Unterstützung
    - gemeinsames Ziel: funktionierender Build und fehlerfreie Software
10. Automatisches Deployment
    - erfolgreiche Buildergebnisse auf repräsentative Zielsysteme verteilen
    - abgestuftes Deployment: jeder Build auf Entwicklungssystem, nächtlicher Build auf Testsystem, versionierter Build auf Staging-System
    - Ziel: aktueller Build für manuelle Tests auf repräsentativer Umgebung bereitstellen

2.6 Integrations- und Systemtesting

- Testen und das V-Modell: Zu jeder Disziplin (absteigende Flanke im V) gibt es eine bestimmte Art von Tests (aufsteigende Flanke im V)
    1. Customer Requirements: Acceptance Tests
    2. Detailed Requirements: System Tests
    3. System Design: Integration Tests
    4. Detailed Design: Unit Tests
- Test-Design: Nur dokumentierte oder automatisierte Tests lassen sich wiederholen (Regressionstests)
    - Die Komplexität von Software erfordert einen Testentwurf nach Modellen.
    - *«Testing by poking around is a waste of time.»* (Robert Binder)
- Mit keiner Testart können alle Fehler gefunden werden:
    - Unit Test: Überprüfung konkreter Implementierungen.
    - Integrationstest: Spielen die Implementierungen richtig zusammen?
    - Systemtest: Arbeiten die einzelnen Systemkomponenten richtig zusammen?
    - Aktepzanztest: Wurde überhaupt das entwickelt, was der Kunde wollte?
    - Vollständiges Testen ist nicht machbar: Mit welcher Kombination von Tests wird das optimale Verhältnis von entdeckten Fehlern gegenüber Aufwand und Kosten erreicht?

2.6.1 Integrationstests

- Integrationstests prüfen die Schnittstellen und das Zusammenspiel von Systemkomponenten.

    - iterative Entwicklung = iterative Integration
    - Anstreben eines stabilen Systems und stabiler Teilsysteme
    - einzelne Komponenten sollten nach Möglichkeit schon getestet sein (Unit-Tests)
    - gewisse Fehler können nur statistisch geprüft werden (Race Conditions, Performance)
- Durch Integrationstests abzudeckende Aspekte:
    - Schnittstellen
        * Kompatibilität der Objekte: Typen, Wertbereiche
        * Aufruf-Sequenzen
        * Validierung von Inputs
    - Datenabhängigkeiten der Komponenten
    - Abdeckung (Call Graph, verschiedene Aufrufvarianten)
- Integrationsstrategien
    - Bottom-Up the Small: Integration kleinerer Teilsysteme
    - Top-Down the Controls: Integration aufwändiger Kontrollstrukturen (mithilfe von Stubs)
    - Bing-Bang the Backbone: Alles weitere in einem grossen Schritt
    - Continuous Integration: Bei iterativ-inkrementeller Entwicklung
- Platzhalter: Ersetzen (noch) fehlender Komponenten durch Test Doubles
    - Stubs: Ersatzbaustein mit identischem Interface und statischem Verhalten
    - Mocks: Intelligenter Stub mit zusätzlicher Logik

2.6.2 Systemtests

- Systemtests prüfen die gesamte Wirkungskette im Softwareprodukt.
    - Ziel: potenziell auslieferbares Softwareprodukt am Ende jedes Sprints
        * ausserhalb der Entwicklungsumgebung lauffähig
        * mit funktionierender Benutzerschnittstelle
        * Interaktion mit anderen Applikationen/Systemen möglich
    - benötigen Testfälle, die in einer möglichst realitätsnahen Testumgebung ausgeführt werden
    - Testen mit verschiedenen Konfigurationen nötig (Konfigurationsmanagement)
- Systemtestfälle: können abgeleitet werden aus
    - dem Backlog (formulierte Anforderungen und Abnahmekriterien)
    - den Use-Case-Beschreibungen
    - der Definition of Done
- Testen nicht funktionaler Anforderungen
    - Last-, Performance-, Stress-, Sicherheit-, Robustheitstests
    - Anforderungen werden oft gar nicht oder zu wenig spezifisch festgehalten
    - sollen auch mit Systemtests abgedeckt werden
- Test-First-Ansatz: Formulieren der Systemtests zu Beginn fördert das Verständnis der Anforderungen und deren Testbarkeit

2.6.3 Regressionstests

Regressionstest: Wiederholtes Ausführen bestehender (und erfolgreich durchlaufener) Testfälle zu einem späteren Zeitpunkt, um sicherzustellen, dass durch die Neuentwicklung keine bestehende Funktionalität beeinträchtigt wurde.

- Regressionstests beschreiben keine bestimmte Art von Tests, sondern das wiederholte Ausführen bestehender Tests.
- Wiederholbarkeit von Tests erfordert nachvollziehbare Dokumentation der Testfälle:
  - Vorbedingungen für die Testausführung (*given*)
  - Handlungen und Eingaben bei der Testdurchführung (*when*)
  - erwartete Ergebnisse und Nachbedingungen (*then*)
  - Die beste Dokumentation von Tests liegt bei automatischen Tests vor: Code lügt nicht – kann aber auch falsch sein (falsche Vorbedingungen, falsche Handlungen/Eingaben, falsche Erwartugen).

2.6.4 Agiles Testing

- Die vier Quadranten des agilen Testens:
  1. Unit- und Komponententests (automatisiert)
     - Ziel: Vertrauen in den Code
     - Verifikation: Code funktioniert, Komponenten spielen zusammen
     - Tests: automatische Testfälle, liegen im SCM vor
     - Werkzeuge: JUnit, Stubs, Mocks
     - Wer: Team
     - Wann: bei jeder Änderung, in jedem Sprint
  2. Funktionale Tests (automatisiert und manuell)
     - Ziel: Vertrauen in den Weg
     - Verifikation: Software macht, was Kunde erwartet
     - Tests: teilweise automatisiert, dokumentiert und protokolliert
     - Werkzeuge: Prototypen, Mockups
     - Wer: Team und Product Owner
     - Wann: in jedem Sprint
  3. Abnahmetests (manuell)
     - Ziel: Vertrauen in das Produkt
     - Verifikation: Software macht, was Kunde erwartet
     - Tests: manuell, dokumentiert (Testplan) und protokolliert (Test- und Abnahmeprotokoll)
     - Werkzeuge: Testdaten, Logs
     - Wer: Team und Product Owner, Kunde bei Abnahme
     - Wann: nach einem Sprint, vor einem Release
  4. Performance-, Last-, Sicherheitstests (mit speziellen Tools)
     - Ziel: Vertrauen in Performance, Sicherheit etc.

  - Verifikation: Code läuft stabil und ist leistungsfähig
  - Tests: Tool-gestützte, teilweise automatisierte Testfälle

2.6.5 Testing in SoDa

- Testplanung und -Organisation
  - Ziel: Abnahmetests am Ende des Sprints
  - Einplanung von Tests zu jeder User Story
  - Durchführung ständig zunehmender Regressionstests
- Testaufgaben im Scrum-Team
  - Planning-Meeting: Aufwandsschätzung für das Testing
  - Sprint: Tests möglichst bald durchführen (Anhäufung vermeiden)
  - Sprint-Abnahme: Demonstration getesteter Features (inkrementelle Validierung)
  - Retrospektive: Stolpersteine ermitteln, Verbesserungsvorschläge erarbeiten

2.7 Entwurfsmuster

- Aus der Architektur entlehnter Begriff: bewährte Entwürfe (Schablonen) für wiederkehrende Entwurfsprobleme.
- Von der *Gang of Four* (GoF) – Erich Gamme, Richard Helm, Ralph Johnson und John Vlissides im Buch *Design Patterns. Elements of Reusable Object-Oriented Software* (1995) gesammelt (23 Entwurfsmuster) und popularisiert.

2.7.1 Wiederverwendung

- Ziel: Wiederverwendung bewährter Lösungen
  - Wiederverwendung von Objekten zur Laufzeit für höhere Effizienz
    - Threads in einem Thread-Pool
    - Datenobjekte in einem Cache
  - Wiederverwendung von Klassen zur Entwicklungszeit für geringeren Entwicklungsaufwand und geringere Fehlerrate
    - Copy & Paste: schlecht
    - Vererbung: je nach Kontekt mehr oder weniger gut geeignet
    - Aggregation und Komposition: meist besser als Vererbung
  - Wiederverwendung von Komponenten für geringeren Entwicklungsaufwand und geringere Fehlerrate
    - Libraries und Frameworks
    - Entwicklung eigener Komponenten für verschiedene Projekte
- Problem: Wiederverwendung von Code ist schwierig
  - verschiedene Kontexte
  - verschiedene Technologien
  - verschiedene Designkonzepte

  - Weiterentwicklung und Wartung
  - aufwändige Verwaltung
  - Abhängigkeiten von Dritten
- Alternative: Wiederverwendung von Konzepten
  - Konzepte ändern sich selten
  - meist unabhängig von Sprache und Implementierung
  - gut erprobt, weit verbreitet und allgemein bekannt

2.7.2 Klassifikation von Entwurfsmustern

Entwurfsmuster werden nach ihrem Zweck klassifiziert.

1. Creational Patterns (Erzeugungsmuster): Erzeugung von Objekten abstrahieren (Typ, Zeitpunkt, Art); Objekterzeugung delegieren und Details auslagern
   - Abstrakte Fabrik (Abstract Factory)
   - Erbauer (Builder)
   - Fabrikmethode (Factory Method)
   - Prototyp (Prototype)
   - Einzelstück (Singleton)
2. Structural Patterns (Strukturmuster): Objekte/Klassen zu neuen Strukturen zusammenfassen; verschiedene Strukturen aneinander anpassen und miteinander verbinden
   - Adapter (Adapter)
   - Bridge (Brücke)
   - Composite (Kompositum)
   - Decorator (Dekorierer)
   - Facade (Fassade)
   - Flyweight Fliegengewicht
   - Proxy (Stellvertreter)
3. Behavioral Patterns (Verhaltensmuster): Interaktionen zwischen Objekten beschreiben; Kontrollflüsse zwischen Objekten festlegen; Zuständigkeiten und Kontrolle delegieren
   - Chain of Responsibility (Zuständigkeitskette)
   - Command (Befehl)
   - Interpreter (Interpreter)
   - Iterator (Iterator)
   - Mediator (Vermittler)
   - Memento (Memento)
   - Observer (Beobachter)
   - State (Zustand)
   - Strategy (Strategie)
   - Template Method (Schablonenmethode)
   - Visitor (Besucher)

- Sekundäre Unterteilung:
  - Klassenmuster: Beziehungen zur Kompilierzeit festgelegt

  - Objektmuster: Beziehung zur Laufzeit dynamisch veränderbar

2.7.3 Singleton (Einzelstück)

Zweck: Gewährleistet, dass es von einer Klasse nur eine Instanz geben kann und stellt einen globalen Zugriffspunkt auf diese bereit. Siehe UML-Diagramm Singleton.

- Klassifikation: Erzeugungsmuster, objektbasiert
- Merkmale
  - Speichert die Objektinstanz als privates, statisches Attribut ab.
  - Verfügt über einen privaten Konstruktor zur Verhinderung Neuinstanziierung von aussen.
  - Erlaubt den Zugriff auf die eine Instanz über eine öffentliche, statische Methode.
- Problematik: Singleton führt zu starker Kopplung und lässt sich schwer austauschen.

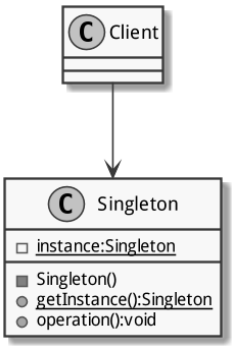

Abbildung 2: Singleton (Entwurfsmuster)

Implementierung:

```
public class Singleton {
    private static final Singleton instance = new Singleton();
    private Singleton() {
    }
    public static Singleton getInstance() {
        return instance;
    }
    public void operation() {
        // ...
    }
}
```

2.7.4 Facade (Fassade)

Zweck: Bietet eine einheitliche Schnittstelle für eine Reihe von Schnittstellen in einem Untersystem. Eine Fassade definiert eine Schnittstelle auf einer höheren Abstraktionsstufe, welche die Handhabung des Untersystem einfacher macht. Siehe UML-Diagramm Facade.

- Klassifikation: Strukturmuster, objektbasiert
- Merkmale
  - Vereinfacht die Anwendung mehrerer Subsysteme.
  - Entkoppelt die Subsysteme vom Client.
    - geringere Kopplung
    - einfachere Austauschbarkeit
- Problematik: Sollte weder zu reinem Durchlauferhitzer verkommen noch wesentliche Logik beinhalten.

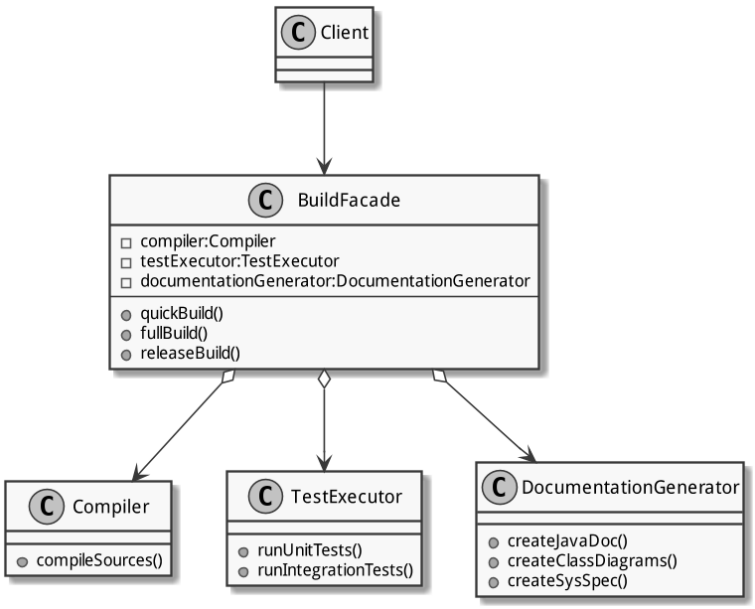

Abbildung 3: Facade (Entwurfsmuster)

Implementierung:

```
public class BuildFacade {
    private Compiler compiler = new Compiler();
    private TestExecutor testExecutor = new TestExecutor();
    private DocumentationGenerator documentationGenerator = new DocumentationGenerator();
    public void quickBuild() {
        compiler.compileSources();
        testExecutor.runUnitTests();
    }
```

```
    public void fullBuild() {
        quickBuild();
        testExecutor.runIntegrationTests();
        documentationGenerator.createJavaDoc();
    }
    public void releaseBuild() {
        fullBuild();
        documentationGenerator.createSysSpec();
        documentationGenerator.createClassDiagrams();
    }
}
public class Compiler {
    public void compileSources() { /* ... */ }
}
public class TestExecutor {
    public void runUnitTests() { /* ... */ }
    public void runIntegrationTests() { /* ... */ }
}
public class DocumentationGenerator {
    public void createJavaDoc() { /* ... */ }
    public void createSysSpec() { /* ... */ }
    public void createClassDiagrams() { /* ... */ }
}
```

2.7.5 Strategy (Strategie)

Zweck: Definiert eine Familie von Algorithmen, kapselt jeden davon und macht sie austauschbar. Mit einer Strategie kann der Algorithmus unabhängig vom Client, der ihn verwendet, austauschen. Siehe UML-Diagramm Strategy.

- Klassifikation: Verhaltensmuster, objektbasiert
- Merkmale
  - Bietet unterschiedliche Varianten/Implementierungen von Algorithmen an
  - Fasst Klassen mit gleichem Interface zusammen, die sich in ihrem Verhalten unterscheiden.
  - Erlaubt das Hinzufügen weiterer Implementierungen ohne Änderungen am bestehenden Code.
- Problematik: Der oftmals lohnende Einsatz bei kleinen Methoden wird gerne übersehen.

Implementierung:

```
public interface SortingStrategy {
    public <T> void sort(List<T> items);
}
```

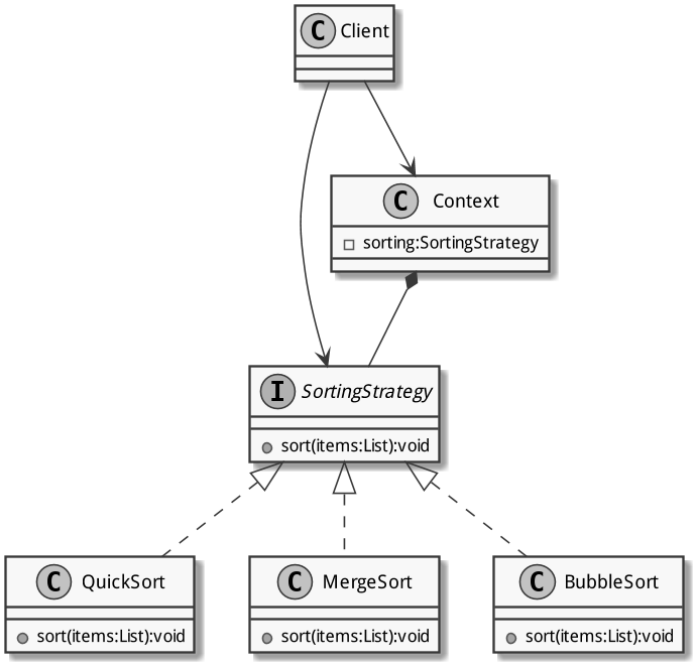

Abbildung 4: Strategy (Entwurfsmuster)

```
public class QuickSort implements SortingStrategy {
    public <T> void sort(List<T> items) { /* ... */  }
}
public class MergeSort implements SortingStrategy {
    public <T> void sort(List<T> items) { /* ... */  }
}
public class BubbleSort implements SortingStrategy {
    public <T> void sort(List<T> items) { /* ... */  }
}
```

2.7.6 Observer (Beobachter)

Zweck: Definiert eine eins-zu-viele-Abhängigkeit zwischen Objekten, damit bei der ÄNderung des Zustands des einen Objekts alle von ihm abhängigen Objekte automatisch benachrichtigt und aktualisiert werden. Siehe UML-Diagramm Observer.

- Klassifikation: Verhaltensmuster, objektbasiert
- Merkmale
  - Realisiert eine lose Kopplung zwischen einem Subjekt und einer theoretisch beliebigen Anzahl von Beobachtern.
  - Erlaubt Kommunikation entgegen der Abhängigkeitsrichtung.

  - Hilft bei der Auflösung von zyklischen Referenzen.
  - Bildet die Grundlage für das Model-View-Controller-Pattern (MVC) und das Event-Handling in Java-GUI-Frameworks.
- Problematik: Eine enge funktionale Kopplung von Subjekt und Beobachter führt zu grossen Schnittstellen. Das Observer-Pattern ist kein Ersatz für schlechte Aufgabenteilung zwischen Klassen!

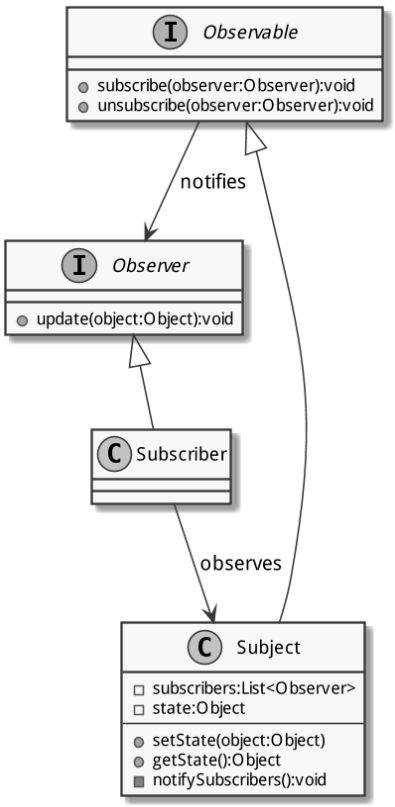

Abbildung 5: Observer (Entwurfsmuster)

Implementierung:

```
public interface Observable {
    public void subscribe(Observer observer);
    public void unsubscribe(Observer observer);
}
public interface Observer {
    public void update(Object state);
}
public class Subject implements Observable {
    private List<Observer> subscribers = new ArrayList<Observer>;
    private Object state = null;
```

```
    public void subscribe(Observer observer) {
        subscribers.add(observer);
    }
    public void unsubscribe(Observer observer) {
        subscribers.remove(observer);
    }
    public void setState(Object state) {
        this.state = state;
        notifySubscribers();
    }
    public Object getState() {
        return state;
    }
    private void notifySubscribers() {
        subscribers.stream().forEach(s -> s.update(state));
    }
}
public class Subscriber implements Observer {
    public void update(Object state) { /* ... */  }
}
```

2.7.7 Adapter (Adapter)

Zweck: Konvertiert die Schnittstelle einer Klasse in eine andere Schnittstelle, die von einem Client erwartet wird. Ein Adapter lässt Klassen zusammenarbeiten, die es ansonsten aufgrund von inkompatibler Schnittstellen nicht könnten. Siehe UML-Diagramm Adapter.

- Klassifikation: Strukturmuster, klassen- oder objektbasiert
- Merkmale
  - Erlaubt die Wiederverwendung von existierenden Klassen und Schnittstellen trotz inkompatibler/ungeeigneter Schnittstellen.
  - Erlaubt die Definition einer möglichst allgemeinen Schnittstelle und die spätere Anpassung für die Bedürfnisse unterschiedlicher Clients.
- Problematik: Ein Adapter ist kein Ersatz für die sinnvolle Ausgestaltung kompatibler Schnittstellen! Sein Einsatz ist oft sinnvoll im Umgang mit APIs von Dritten. Bei eigenen (internen) APIs ist ein Refactoring einem Adapter meist vorzuziehen.

Implementierung:

```
public interface MessageStore {
    public void store(Message message);
}
public interface KeyValueStore {
    public void put(String id, String message);
```

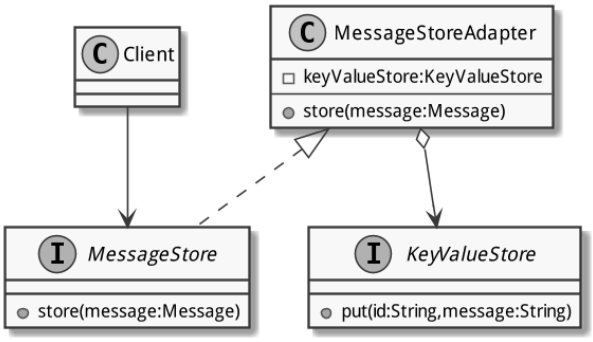

Abbildung 6: Adapter (Entwurfsmuster)

```
}
public class MessageStoreAdapter implements MessageStore {
    private KeyValueStore keyValueStore = KeyValueStore.getInstance("messages");
    public void store(Message message) {
        keyValueStore.put(message.getId(), message.getPayload());
    }
}
```

2.7.8 Factory Method (Fabrikmethode)

Zweck: Definiert eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts. Die Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Siehe UML-Diagramm Fabrikmethode.

- Klassifikation: Erzeugungsmuster, klassenbasiert
- Merkmale
  - Erlaubt das Erstellen von Instanzen der Klassen einer bestehenden Klassenhierarchie.
  - Stellt eine einheitliches Schnittstelle zur Erzeugung dieser Klassen zur Verfügung.
  - Delegiert die Details der Objekterzeugung an die Unterklassen/Implementierungen.
- Problematik: Bei der Erweiterung der Klassenhierarchie muss auch die Hierarchie der Factory-Klassen erweitert werden. Erweiterungen wirken sich so immer auf mehrere Stellen im Code aus.

Implementierung:

```
public interface Document {
    public String getMimeType();
    public String getExtension();
}
public class JSONDocument implements Document {
    public String getMimeType() {
```

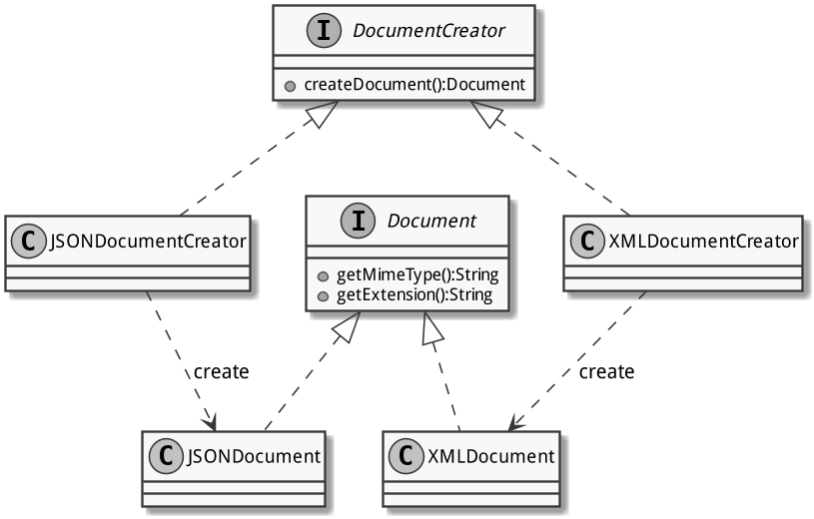

Abbildung 7: Fabrikmethode (Design Pattern)

```
        return "application/json";
    }
    public String getExtension() {
        return ".json";
    }
}
public class XMLDocument implements Document {
    public String getMimeType() {
        return "application/xml";
    }
    public String getExtension() {
        return ".xml";
    }
}
public interface DocumentCreator {
    public Document createDocument();
}
public class JSONDocumentCreator {
    public Document createDocument() {
        return new JSONDocument();
    }
}
public class XMLDocumentCreator {
    public Document createDocument() {
        return new XMLDocument();
```

```
    }
}
```

2.7.9 Prototype (Prototyp)

Zweck: Entkoppelt die Objekterzeugung vom eigentlichen System. Gibt die Möglichkeit beliebig komplexe Prototypen aus einzelnen, einfachen Prototypen zusammenzubauen. Siehe UML-Diagramm Prototyp.

- Klassifikation: Erzeugungsmuster, objektbasiert
- Merkmale
  - Erlaubt die Erzeugung von Objekten aufgrund bereits existierender Objekte.
  - Steigert die Effizient bei der Erstellung neuer Objekte, indem bestehende Objekte kopiert und nicht komplett neu aufgebaut werden.
  - Wird oftmals im Zusammenhang mit einem Prototyp-Cache verwendet, aus dem bestehende Objekte zum Klonen herausgelesen werden können.
- Problematik: Das Klonen von Objekten kann fehleranfällig sein, gerade wenn viele Referenzen auf andere Objekte bestehen. Man muss sich bewusst sein, ob und wo eine *deep copy* (Klonen sämtlicher referenzier Objekte) oder eine *shallow copy* (blosses Kopieren aller Referenzen) vorgenommen werden soll.

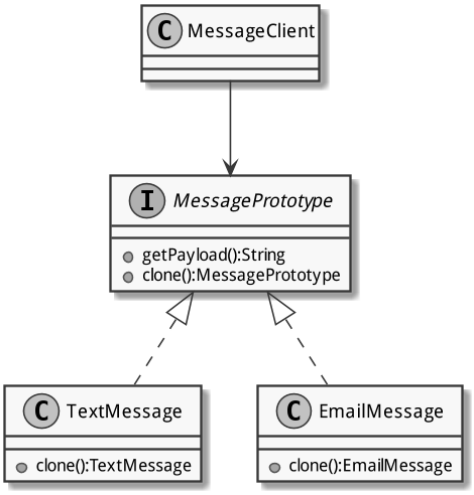

Abbildung 8: Prototyp (Design Pattern)

Implementierung:

```
public interface MessagePrototype {
    public String getPayload();
    public MessagePrototype clone();
}
```

```
public class TextMessage implements MessagePrototype {
    public String getPayload() {
        // ...
        return payload;
    }
    public MessagePrototype clone() {
        return super.clone();
    }
}
public class EmailMessage implements MessagePrototype {
    public String getPayload() {
        // ...
        return payload;
    }
    public MessagePrototype clone() {
        return super.clone();
    }
}
public MessageClient() {
    // provide new instances of MessagePrototype by cloning the existing ones
}
```

2.7.10 Einsatz von Entwurfsmustern

- Voraussetzungen: Man muss die Entwurfsmuster kennen und verstehen!
  - Quelle: Literatur (GoF-Buch), Internet
- Sinnvolle Auswahl: Kein Entwurfsmuster löst jedes Problem (*no golden hammer*).
  - Anhand Kategorie vorselektieren (Erzeugung, Struktur, Verhalten)
  - Vor- und Nachteile abwägen (siehe *Consequences*-Sektion im GoF-Buch)
  - Im Zweifelsfall das Muster auswählen, das eine grössere Flexibilität bietet.
- Überlegter Einsatz: Macht der Einsatz eines Entwurfsmuster den Code besser und einfacher verständlich – oder wird der Code dadurch nur aufgebläht und unflexibel?
  - Verifikation anhand fiktivem oder (besser) realem Beispiel durchführen!
- Im Umgang mit Entwurfsmustern braucht es viel Erfahrung. Im Zweifelsfall sollte man besser auf den Einsatz eines Entwurfsmusters verzichten.
  - Habe ich ein Problem, auf das ein Entwurfsmuster passt?
  - Oder habe ich ein Entwurfsmuster, wofür ich ein passendes Problem suche?
  - Entwurfsmuster sind Konzepte, die erweitert und kombiniert (MVC) werden können.
  - Ein stures Festhalten an originalen Konzepten im GoF-Buch ist nicht sinnvoll!

Der Vorteil von Entwurfsmustern liegt nicht nur im durch sie besser strukturierten Code, sondern darin, dass die Entwickler *eine gemeinsame Sprache* mit kurzen und prägnanten Begriffen für ansonsten schwer vermittelbare Konzepte haben.

2.8 Automatisiertes Testing

- Testen geniesst unter Entwicklern und Projektleitern einen schlechten Ruf:
  - Entwickler wollen lieber programmieren als testen und glauben, dass sie keine Fehler machen.
  - Projektleiter sehen das Testen als unnötigen Arbeitsschritt, der keinen Mehrwert schafft.
- Problem: Falscher Testing-Ansatz in der Entwicklung:
  - Getestet wird erst am Schluss: Suche nach Fehlern, die man lieber nicht finden würde.
  - Konsequenz: Es wird schlecht getestet, Fehler werden verschwiegen oder Entwickler verhalten sich defensiv.
  - Werden keine Fehler gefunden heisst dies nicht, dass es keine Fehler gibt!
- Lösung: Kontinuierlich testen; nicht um Fehler zu finden, sondern um Gewissheit zu haben, was bereits funktioniert.
- Test-First-Ansatz (Test Driven Development)
  - Idee: Mögliche Fehler finden, bevor sie gemacht werden, und korrigieren, bevor sie implementiert werden. (Fehler bereits im Ansatz erkennen.)
  - Ursprung: Extreme Programming (Kent Beck, Erich Gamma)
  - Ansatz: Zuerst Testfälle (Code) schreiben, dann den produktiven Code.
  - Vorteil: Der Test kann automatisch erfolgen, sobald der produktive Code geschrieben wurde.
  - Folge: Schnelles Feedback und erhöhte Motivation!

2.8.1 Unit- und Integrationstests

- Unit-Tests (oder Komponenten-, Modul-, Entwicklertests): funktionale Tests einzelner, in sich abgeschlossener Einheiten (Klasse, Komponente, Modul)
  - Ziel: Testen einzelner Einheiten ohne Abhängigkeiten zu anderen Einheiten
  - Vorteile: schnell, einfach ausführbar, selbstvalidierend (Assertions) und automatisiert
  - Ausführung in der Entwicklungsumgebung durch den Entwickler und in der CI-Umgebung während des Buildprozesses
  - Nutzen:
    * schnell ausführbare Tests neuer Komponenten oder veränderter Komponenten (Regressionstests)
    * Testen erfolgt bereits während der Implementierungsphase
    * Möglichkeit des Test-First-Ansatzes: gute Testbarkeit von Anfang an gegeben
    * automatisiertes, schnelles und übersichtliches Feedback (Reporting)
    * Messung von Codeabdeckung möglich
  - Probleme:
    * für GUI-Komponenten aufwändig
    * Qualität und Nachvollziehbarkeit der Tests wichtig (aber zeitaufwändig)
    * es lässt sich nicht «alles» mit Unit-Tests abdecken
  - JUnit: Am häufigsten eingesetztes Test-Framework in JAva
    * Version 4.12 am stärksten verbreitet und am besten integriert

        * Version 5.0 seit September 2017 mit Java 8 (für neue Projekte)
- Integrationstests: Testen das Zusammenspiel verschiedener Komponenten
  - Abgrenzug zu Unit-Tests oft kontrovers und nicht eindeutig
  - Verwendung von Umgebungsressourcen (Netzwerk, Dateisystem): Integrationstest!
  - Fehlschlagen aufgrund von Fremdeinflüssen möglich: Integrationstest!
  - Faustregel: Unit-Tests sind auf einem beliebigen System jederzeit lauffähig.
- JUnit-Namenskonvention: Suffix `Test` für Unit-Tests, Suffix `IT` für Integrationstests; erlaubt die Unterscheidung für verschiedene Verifikationsphasen und unterschiedlich umfassende Builds
  - Vor dem Commit/Push in der Entwicklungsumgebung: schneller Build mit Unit-Tests
  - Nightly Build auf der CI-Umgebung: umfassender Build mit Unit- und Integrationstests

2.8.2 Codeabdeckung

- Problem: Wie erreicht man mit möglichst wenig Aufwand möglichst umfassende Tests?
- Lösung: Durch Messung der Codeabdeckung durch Testfälle!
  - Dadurch gezielte Entwicklung von Tests möglich, die mit wenig Testcode viel produktiven Code testen.
  - Statistische Auswertung möglich, auch mit zeitlicher Entwicklung (Verbesserung oder Verschlechterung der Testabdeckung).
- Verfahren: Testumgebung merkt sich, welcher Code in einem Testdurchlauf ausgeführt wurde.
- Es gibt verschiedene Messtechniken:
  - Welche Zeilen oder Statements wurden ausgeführt? (Line Coverage)
  - Welche Verzweigungen wurden genommen? (Branch Coverage)
  - Welche Bedingungen wurden evaluiert? (Decision Coverage)
  - Welche Programmablaufpfade wurden durchlaufen? (Path Coverage)
    * Inpraktikabel, da die Anzahl Pfade exponentiell mit der Anzahl Entscheidungen wächst.
  - Welche Funktionen wurden ausgeführt? (Function Coverage)
  - Welche Codestellen laufen parallel ab? (Race Coverage)
  - Diese Metriken sind unterschiedlich aussagekräftig und sollten am besten kombiniert verwendet werden.
- Technische Umsetzung:
  1. Instrumentierung des Quellcodes: Compiler fügt Statements zur Coverage-Messung in den Code ein. Zu vermeiden, da der Quellcode zuvor manipuliert und dadurch das Debugging erschwert wird.
  2. Instrumentierung des Bytecodes: Der Bytecode wird nach der Kompilierung um Coverage-Messungen ergänzt. Die ergänzten `.class`-Dateien müssen von den «produktiven» Kompilaten gesondert werden.
  3. Just-in-time Instrumentierung zur Laufzeit: Die Laufzeitumgebung instrumentiert den Bytecode über den Classloader. Bester Ansatz.

2.8.3 Dependency Injections

- Schlechte Testbarkeit von Komponenten aufgrund hoher Kopplung.
  - Komponente A hängt von Komponente B ab, Komponente B hängt von Komponente C ab.
  - Ein Unit-Test der Komponente A erfordert und testet sogleich Komponente B und C – und ist also kein eigentlicher Unit-Test mehr.
- Lösung: Dependency Injection
  - Die Komponenten werden nicht länger fest verdrahtet (A instanziert B, B instanziert C).
  - Stattdessen können jeder Komponente ihre Abhängigkeiten über einen zusätzlichen Konstruktor mitgegeben werden.
  - Dadurch wird die Kopplung reduziert; idealerweise über ein Interface.
  - Die Testbarkeit wird erhöht, erfordert aber die Entwicklung von *Test Doubles*, die als abhängige Komponenten eingesetzt werden können.

2.8.4 Test Doubles

- Anforderungen und Vorbedingungen für den Einsatz von Test Doubles:
  - Einsatz von Interfaces: Test Double und Produktivkomponente müssen das gleiche Interface implementieren.
  - Ersetzung der Komponente zur Laufzeit: mittels Dependency Injection!
  - Sicherungsmassnahmen, um die Verwendung von Test Doubles in der Produktivumgebung zu verhindern. (Aufteilung in Test- und Produktivprojekt)
- Dummy: primitive, funktionslose Attrappe; häufig eine «leere» Implementierung eines Interfaces
  - Beispiel: Eine Komponente benötigt einen Logger, das Logging ist aber nicht Gegenstand des Tests
  - Lösung: Ein `LoggingDummy` implementiert die `Logging`-Schnittstelle, ohne tatsächlich etwas zu loggen, und wird der Komponente mitgegeben.
- Stub: einfachste Implementierung eines Interfaces, die mit konstante, vordefinierte Rückgaben liefert
  - Beispiel: Eine Komponente hängt von einem Loginmechanismus ab, doch es stehen keine Credentials für Tests zur Verfügung.
  - Lösung: Es wird ein `LoginStub` entwickelt, das die Authentifikatonsschnittstelle implementiert, und einfach alle Loginversuche akzeptiert. (Ein zweiter Stub könnte alle Loginversuche ablehnen, je nach Anforderungen.)
- Spy: alternative Implementierung, welche dynamische Werte zurückliefert und sich die Aufrufe der Methoden merkt (Anzahl, Häufigkeit, Parameter, Zeitpunkt, Exceptions)
  - Einsatz: Wenn nicht der Rückgabewert einer Aktion, sondern deren Verhalten getestet werden, wozu die Aufzeichnungen des Spies ausgewertet werden können.
- Mock: Spezialisierung von Spy, welche die Verifikation der gesammelten Daten selber vornehmen kann
  - Einsatz: Mithilfe von Mocking-Frameworks (z.B. Mockito) werden zur Laufzeit Mock-Objekte erstellt.

- Fake: vollständige und korrekte – aber einfachere! – Implementierung einer Komponente
  - Beispiel: Eine Komponente benötigt einen Webservice als Abhängigkeit, welcher in der Testumgebung nicht zur Verfügung steht.
  - Lösung: Es wird ein Fake-Webservice erstellt, der die Anfragen direkt und nicht über das Netzwerk zurückgibt. (Der Fake-Webservice könnte auf Aspekte wie Logging, Persistenz und Concurrency verzichten.)
- Einsatzzweck der verschiedenen Verfahren:
  - Dummy und Stub: Erreichen einer besseren Testisolation und höhreren Selektivität der Testfälle.
  - Spy und Mock: Universeller Ansatz zur Durchführung von Behaviour-Testing.
  - Fake: Aufwändige Implementierung zur vollständigen Entkopplung vom Original.

Die Test Doubles sind nicht der Gegenstand der Tests sondern nur Mittel zum Zweck! Es sollte nur so viel Aufwand für deren Umsetzung betrieben werden, bis dass die Funktion der eigentlich zu testenden Komponente möglichst vollständig getestet werden kann!

2.9 Review

2.10 Konfigurationsmanagement

2.11 Deployment

Deployment: Bereitstellung, Auslieferung von Software

- 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.)

2.11.1 Aspekte des Deployments

- 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

2.11.2 Deployment in 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

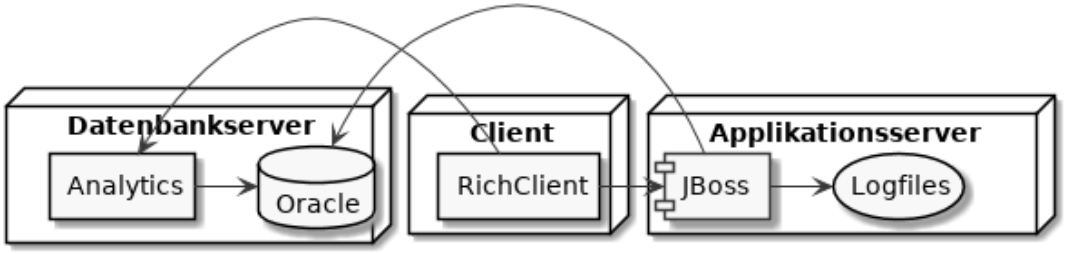

Abbildung 9: Deploymentdiagramm

2.12 Code-Qualität

2.12.1 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`

2.12.2 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 lOlOlOlOlOlOfAc70r; 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

2.12.3 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 entsprechendem 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.

2.12.4 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.

2.12.5 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

- Netzwerkschichten:
    - Applikationsschicht (application layer): HTTP, SMTP, FTP, DNS
    - Transportschicht (transport layer):
        * Transmission Control Protocol (TCP): verbindungsorientiert, zuverlässig
        * User Datagram Protocol (UDP): verbindungslos, unzuverlässig
    - Internetschicht (internet layer): Internet Protocol (IP)
    - Netzwerkschickt (network layout): Ethernet, WLAN, DSL, UMTS, LTE
- Begriffe:
    - Host: An ein Netzwerk angeschlossener Computer mit netzwerkweit eindeutiger IP-Adresse
    - Socket: Kommunikationsendpunkt, durch IP-Adresse und Port-Nummer definiert
    - Server (Diener): Dienstleister, der Daten/Ressourcen zur Verfügung stellt; umfasst einen oder mehrere Computer
        * Software: Server-Programm, das einen Dienst implementiert
        * Hardware: Server-Computer, auf dem ein oder mehrere Server-Programme laufen
    - Client (Kunde): Dienstnehmer, der Dienste von Servern verwendet

3.1.1 Java Sockets

Package `java.net.*` mit Klassen:

- `InetAddress`
    - `static InetAddress getByName(String host)`: IP-Adresse von Host anhand Namen finden
    - `String getHostName()`: liefern Hostnamen
    - `String getHostAddress()`: liefert IP-Adresse
    - `String getCanonicalHostName()`: liefert voll qualifizierten Hostnamen (mit Domäne)
    - `boolean isReachable(int msec)`: prüft, ob ein Host innerhalb eines Timeouts antwortet

    - `static InetAddress getLocalHost()`: IP-Adresse des lokalen Hosts finden
    - `boolean isSiteLocalAddress()`: Prüft, ob es sich um den lokalen Host handelt
- `Socket`
    - `Socket(String host, int port)`: Socket-Verbindung zu `host:port` erstellen
    - `OutputStream getOutputStream()`: zum schreibenden Zugriff
    - `InputStream getInputStream()`: zum lesenden Zugriff
- `ServerSocket`
    - `ServerSocket(int port)`: Socket, der auf `port` hört
    - `ServerSocket(int port, int backlog, InetAddress addr)`: mit Grösse der Warteschlange und spezifischer IP-Adresse
    - `Socket accept()`: Verbindung entgegennehmen (blockierend)
- `NetworkInterface`
    - `static Enumeration<NetworkInterface> getNetworkInterfaces()`
    - `String getDisplayName()`
    - `Enumeration<InetAddress> getInetAddresses()`

3.1.2 Socket-Lebenszyklus

1. Server: Socket erzeugen und an lokalen Port binden
2. Server: Mit `accept` auf eingehende Verbindung warten
3. Client: Verbindung mit Server herstellen (mit IP-Adresse und Port-Nummer)
4. Client/Server: Daten über Socket lesen/schreiben
5. Client: Verbindung schliessen
6. Server: Socket schliessen

3.2 Serialisierung

Serialisierung: Objekte in eine persistente Form bringen, damit der Objektzustand später wiederhergestellt werden kann

- Anforderungen an Objektpersistenz
    - Transparenz: gleiche Handhabung von persistenten und transienten Objekten durch den Entwickler
    - Interoperabilität: Verwendung persistenter Objekte unabhängig von Laufzeitumgebung
    - Wiederauffindbarkeit: Auffinden von persistenten Objekten ohne explizite Suche (transparent)
- Grundidee: Umwandlung eines Objekts in Bytestrom zur persistenten Abspeicherung

3.2.1 Serialisierung in Java

- Java-Serialisierungsverfahren: Ablauf und Umfang

    1. Metadaten (voll qualifizierter Klassenname, Signatur, Versionsnummer) in den Bytestrom schreiben
    2. Rekursive Serialisierung nicht-statischer, nicht-transienter Attribute (`private`, `protected`, `public`) und aus Oberklassen geerbte Attribute
    3. Zusammenfassen der entstandenen Byteströme zu einem bestimmten Format
    - alle verwendeten Klassen müssen vollständig im `CLASSPATH` vorliegen!
- Objektserialisierung in Java
    - Java Object Serialization (JOS): binäres Format
    - Java Bean Persistence (JBP): Abspeicherung von Java Beans als XML
    - Java Architecture for XML Binding (JAXB): Abbilden von Objektstrukturen auf XML
- Standard-Serialisierung: Java-Klassen, Interfaces und Methoden
    - Idee: Dekoration eines Input- oder Output-Streams (`FileInputStream`, `BufferedOutputStream`, etc.)
    - Interface `java.io.ObjectOutput`
        * `void ObjectOutputStream.writeObject(Object obj)`: rekursives Abarbeiten des Objekt-Parameters
    - Interface `java.io.ObjectInput`
        * `Object ObjectInputStream.readObject()`: Aufbauen der Objekthierarchie aus Bytestrom (Cast notwendig)
    - Marker-Interface `java.io.Serializable` (keine Methoden): muss implementiert werden, damit Objekte der jeweiligen Klasse serialisiert werden können
        * dadurch automatische Serialisierbarkeit von erbenden Klassen gegeben
    - spezielle Methoden Auf dem jeweiligen Objekt:
    - `Object readResolve()`: erlaubt Manipulation des deserialisierten Objekts
        * zur Wiederherstellung transienter Objekte
        * zum Garantieren der Eindeutigkeit bei Singleton-Objekten
    - `void writeObject(final ObjectOutputStream oos)`
        * eigens definierte Serialisierung anhand von Bytestrom
        * Aufruf von `oos.defaultWriteObject()` zum Erweitern der Serialisierung
    - `void readObject(final ObjectInputStream ois)`
        * eigens definierte Deserialisierung anhand von Bytestrom
        * Aufruf von `ois.defaultReadObject()` zum Erweitern der Deserialisierung
- Klonen von beliebigen Objekten per Serialisierung und anschliessender Deserialisierung
    - `ByteArrayOutputStream` $\rightarrow$ `ObjectOutputStream` $\rightarrow$ `ByteArrayInputStream` $\rightarrow$ `ObjectInputStream`
- Transiente Attribute: für «berechnete», zwischengespeicherte Felder
    - Schlüsselwort `transient`: `private transient String fullName;`
    - Deserialisierung: Implementierung der Methode `readResolve()`
- Serialisierung und Vererbung
    - private Felder nicht-serialisierbarer Oberklassen werden nicht serialisiert
    - Deserialisierung: Suche und Aufruf des parameterlosen Konstruktors der ersten nicht-serialisierbaren Oberklasse: es muss ein solcher existieren!
    - Implementierung von `readObject()` und `writeObject()` muss sich um Zustand der Oberklasse kümmern

  - Verhinderung von Serialisierung bei Unterklassen durch Überschreibung von `writeObject()` und `readObject()` möglich
- Versionierung: aufgrund langer Zeitspannen zwischen Serialisierung und Deserialisierung mit wechselnden Umgebungen (Laufzeitumgebung, Softwareversion) unbedingt nötig
  - Angabe über Attribut: `public static long serialVersionUID = ...;`
  - Andernfalls automatische Berechnung eines Hash-Werts zur Laufzeit anhand verschiedener Klassenparameter (nicht empfohlen).
  - Neuerstellung der `serialVersionUID` bei inkompatiblen Änderungen der Klasse.
  - Ein Objekt kann nur deserialisiert werden, wenn seine Klasse die gleiche Version hat, wie sie die zugrundeliegende Klasse des serialisierten Objekts bei der Serialisierung hatte.

3.3 Message Passing

Message Passing: Kommunikationsparadigma zum Versenden einer Nachricht von einem Sender zu einem oder zu mehreren Empfängern.

- Arten von Nachrichten: (Remote) Method Invocation, Signale, Datenpakete
- Designentscheide bei der Entwicklung eines Message-Passing-Systems:
  1. zuverlässige oder unzuverlässige Nachrichtenübertragung
  2. garantierte oder beliebige Übertragungsreihenfolge der Nachrichten
  3. Sender-Empfänger-Beziehungen:
     - Unicast: ein Sender, ein Empfänger
     - Multicast/Broadcast: ein Sender, mehrere Empfänger
     - Client-Server: mehrere Sender, ein Empfänger
     - All-to-all: mehrer Sender, mehrere Empfänger
  4. synchrone oder asynchrone Kommunikation
     - synchron: Sender blockiert, bis der Empfänger die Nachricht entgegengenommen hat (Telefonie, Instant Messaging)
     - asynchron: Sender führt seine Aufgabe nach Versenden einer Nachricht fort (E-Mail, Diskussionsforum)
  5. persistente oder transiente Kommunikation
     - persistent: Nachricht bis zur Bereitschaft des Empfängers zwischenspeichern (E-Mail)
     - transient: Nachricht während Ausführung von sendender und empfangender Applikation zwischengespeichert (Router, Socket)
- Kommunikationsformen: Nachricht $A \rightarrow B$
  - persistent und asynchron: HTTP/REST
    1. A sendet eine Nachricht und wird fortgesetzt
    2. B läuft nicht
    3. B wird gestartet und empfängt die Nachricht
    4. A ist fertig
  - persistent und synchron: E-Mail
    1. A sendet eine Nachricht und wartet, bis diese akzeptiert wurde

    2. B speichert die Nachricht für die spätere Auslieferung ab
    3. B meldet A, dass die Nachricht akzeptiert wurde
    4. B wird gestartet und empfängt die Nachricht
  - transient und asynchron: UDP
    1. A sendet eine Nachricht und wird fortgesetzt
    2. B muss laufen, damit Die Nachricht gesendet werden kann
    3. B empfängt die Nachricht
  - transient und synchron (empfangsbasiert): TCP
    1. A sendet eine Nachricht und wartet auf die Empfangsbestätigung
    2. B läuft, macht aber gerade etwas anderes
    3. B nimmt die Nachricht entgegen und bestätigt dies A
    4. B verarbeitet die Nachricht später
  - transient und synchron (auslieferungsbasiert): asynchroner RPC
    1. A sendet eine Nachricht und wartet, bis diese akzeptiert wurde
    2. B empfängt die Nachricht, macht aber gerade etwas anderes
    3. B bestätigt die Annahme der Nachricht
    4. B verarbeitet die Nachricht später
  - transient und synchron (antwortbasiert): RMI
    1. A sendet eine Nachricht und wartet auf eine Antwort
    2. B läuft, macht aber gerade etwas anderes
    3. B nimmt die Nachricht entgegen
    4. B verarbeitet die Nachricht und sendet eine Antwort
- Message Passing Interface (MPI): Standard für den Nachrichtenaustausch
  - definiert eine API, ist kein konkretes Protokoll und keine Implementierung

3.3.1 Nachrichtenverarbeitung

- Nachrichten
  - werden über einen Kommunikationskanal gesendet
  - enthalten eine Anzahl Elemente von bestimmten Datentypen
    * ID: Identifikation der Nachricht, nicht immer nötig
    * Argumente: einfache Datentypen (Integer, String) oder mit zusätzlichen Informationen/Instruktionen versehen
  - Solche Nachrichten sind Protokollen wie HTTP, RMI etc. vorzuziehen, wenn:
    * die Kommunikation und zu übermittelnden die Datenstrukturen simpel sind
    * der Transaktionsdurchsatz kritisch ist (Echtzeit-Anwendungen)
    * die Entwicklungsressourcen limitiert sind (schnelle Entwicklung wichtiger als Flexibilität)
    * spezielle Protokolle benötigt werden
    * andere Protokolle (HTTP, RMI etc.) nicht verfügbar sind
  - Können mittels Factory-Method-Design-Pattern flexibel und transparent erstellt werden.
- Prinzipien der Nachrichtenverarbeitung
  - Trennung zwischen Kommunikations- und Applikationsdetails

  - Kommunikation scheint auf Applikationsebene vonstatten zu gehen (untere Layer transparent)

3.3.2 Protokollarten

- Fixe Protokolle: Parameter zu Beginn der Sitzung bekannt, keine Änderungen während der Kommunikation
  - Menge möglicher Kennungen (IDs)
  - mögliche Argumente (Anzahl und Typ)
- Adaptive Protokolle: Parameter können während einer Sitzung ändern (Länge der Argumentliste, Argumenttypen, Nachrichttypen)
  - Änderungen per anpassbarem Message Handler zur Laufzeit bewältigt
  - Kann mit dem Prototyp-Design-Pattern umgesetzt werden: Neue Arten von Nachrichten müssen nicht als Klassen umgesetzt, sondern können über die Erweiterung von Prototyp-Objekten erzeugt werden.
  - Die Liste bekannter Nachrichtentypen wird zur Laufzeit erweitert.

3.4 Verteilung & Kommunikation: RMI

Verteiltes System: System, in dem sich Hardware- und Softwarekomponenten auf vernetzten Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren.

Verteilte Anwendung: Anwendung, die ein verteiltes System als Kommunikationsinfrastruktur für ihre verteilten Komponenten nutzt.

3.4.1 Middleware

Middleware: anwendungsneutrale Vermittlungssofware, die zwischen Anwendungen vermittelt, und dabei den Anwendungen ihre eigene Komplexität und diejenige der Infrastruktur verbirgt.

- Arten von Middleware:
  1. kommunikationsorientierte Middleware: abstrahiert Netzwerkprogrammierung (RPC, RMI, Web Service)
  2. nachrichtenorientierte Middleware: arbeitet über den Austausch von Nachrichten (messages) mithilfe von Warteschlangen (queues): JMS, SOAP
  3. anwendungsorientierte Middleware: unterstützt verteilte Anwendungen: JEE, .NET, CORBA
- Eine Middleware schafft Transparenz:
  - Ortstransparenz: Der Benutzer braucht nicht zu wissen, wo sich ein Dienst oder eine Ressource befindet.
  - Zugriffstransparenz: Der Zugriff erfolgt immer gleich, egal von welchem Netzwerk aus (lokal, remote).

  - Nebenläufigkeitstransparenz: Mehrere Benutzer können gleichzeitig auf die Dienste und Ressourcen zugreifen. Das System ermöglicht exklusive Zugriffe, Synchronisation und Replikation von Daten.
  - Fehler- und Ausfalltransparenz: Fehler, die durch die Verteilung auftreten können (Übertragungsfehler, Komponentenausfall) bleiben der Anwendung weitgehend verborgen.
  - Sprachtransparenz: Die Kommunikation zwischen den Komponenten ist nicht von deren Implementierungssprache abhängig.
  - Replikationstransparenz: Ressourcen werden bei Bedarf automatisch repliziert, um die benötigte Performance bieten zu können.
- Architekturmodelle:
  - Client-Server vs. Peer-to-Peer
    * Client-Server: langlebiger Serverprozess, kurzlebige Client-Prozesse
    * Peer-to-Peer: Austausch zwischen gleichberechtigten Prozessen
  - Fat- vs. Thin-Client
    * Fat-Client: enthält Verarbeitungslogik und Benutzeroberfläche
    * Thin-Client: bezieht Verarbeitungslogik und Benutzeroberfläche von einem Server
    * Kombination: Fat-Client & Thin-Server; Thin-Client & Fat-Server
  - 2-, 3-, n-Tier
    * 2-Tier: Datenhaltung von Präsentation/Anwendungslogik getrennt
    * 3-Tier: Je ein Tier für Datenhaltung, Anwendungslogik und Präsentation
    * n-Tier: Weitere Verteilung von Datenhaltung und Anwendungslogik

3.4.2 RMI: Remote Method Invocation

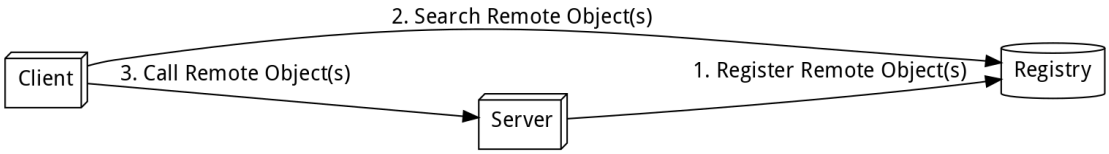

Abbildung 10: Arbeitsweise von RMI

- Prinzipielle Arbeitsweise (siehe Arbeitsweise von RMI)
  1. Remote-Interface: eine oder mehrere Methoden zur Bereitstellung als Dienst
  2. Server-Klasse: implementiert Remote-Interface, ≥ 1 Instanzen (Remote-Objekte)
  3. Registry: Registrierung der Remote-Objekte unter eindeutigem Namen
  4. Suche: Client findet Remote-Objekt über dessen Namen
  5. Aufruf: Verteilung für Client (Parameterübergabe) und Server (Rückgabe) transparent
- Technische Umsetzung von RMI
  - Client- und Server-Stub: Stellvertreterobjekte, die das Remote-Interface implementieren
    * Automatische Erzeugung seit JDK 1.5, zuvor mit speziellem RMI-Compiler (`rmic`)
  - Kommunikation über Remote Method Protocol auf Basis von TCP/IP-Verbindungen
  - RMI-Transportschicht erfolgt über Stubs (Server-Stub kennt Remote-Objekt)
  - Parameterübergabe: primitive Datentypen und serialisierbare Objekte (call by value)

  - Übergaben von Verweisen auf Remote-Objekte ohne Serialisierung (call by reference)
- Vorgehen bei der Entwicklung von RMI-Anwendungen
  1. Definition des Remote-Interfaces: grundsätzlich normales Java-Interface
     - muss von Interface `Remote` erben
     - jede Methode soll eine `RemoteException` werfen können
     - Parameter und Rückgabewert: primitive Datentypen oder serialisierbare Klassen
  2. Implementierung des Remote-Interfaces
     - Ableitung von `UnicastRemoteObject` oder Export per `UnicastRemoteObject.exportObject()`
  3. Aufstarten der RMI-Registry: `rmiregistry`, Standardport 1099
     - per automatisch generierter URL erreichbar
     - programmatischer Zugriff mittels `java.rmi.registry`-Paket
     - Server kann nur eine Registry manipulieren, die auf dem gleichen Rechner läuft (Sicherheit)
     - Registry unterstützt keine hierarchiche Namensräume oder dynamische Namen
  4. Erzeugung und Registrierung von Remote-Objekten: `Naming.bind(url, remoteObject)`
     - URL: `rmi://localhost:1099/calcSum`
     - Dynamisches Laden von Klassen erfordert einen `SecurityManager`
  5. Implementierung des Clients: Remote-Objekt finden und aufrufen
     - `Naming.lookup(url)` mit entsprechendem Cast liefert Referenz auf Remote-Objekt
- Codebase: Unter Einsatz eines Security Managers kann Code auf von einem entfernten Rechner geladen werden. Die Codebase wird über das JVM-Property `java.rmi.codebase` definiert, welches beim Aufstarten der Anwendung oder im Code derselben gesetzt werden kann:
  - `java -Djava.rmi.server.codebase=http://localhost:8080 -jar anwendung.jar`
  - `System.setProperty("java.rmi.server.codebase", "http://localhsot:8080");`
  - Der Code muss per HTTP zur Verfügung gestellt werden (`tool.jar` vom JDK).
    * `java -jar tool.jar -port 8080 -dir anwendung/`
- Security Manager: Einschränkungen von Code, der aus dem Netz geladen wird (Netzwerkverbindungen, Dateisystemzugriff, Lesen und Schreiben von Properties, Ausführen externer Programme, Nachladen externer Libraries etc.).
  - `-Djava.security.manager`: beim Aufstarten der Anwendung
  - `System.setSecurityManager(new SecurityManager());`: im Code (zuerst mit `getSecurityManager()` prüfen, ob dieser bereits aktiviert ist)
  - Definition der Sicherheitsrichtlinie in `policy`-Datei:
    * `-Djava.security.policy=my.policy` per Kommandozeile
    * `System.setProperty("java.security.policy", "my.policy");` im Code (`my.policy` muss im akzuellen Verzeichnis liegen)
    * Gewährung von Berechtigungen per `grant` (siehe Beispiele unten)
  - Zusätzliche Angaben (optional):
    * Codebase: Rechte von Klassen aus bestimmten Quellen
    * Signierung: Rechte werden nur für signierten Code gewährt

    * Principal: Sonderrechte für authentifizierte Benutzer

Security-Policy, die alle Berechtigungen gewährt (zu Testzwecken oder innerhalb eines Containers):

```
grant {
    permission java.security.AllPermission;
};
```

Security-Policy für Sockets (ausgehende Verbindungen zu `localhost:1099` und eingehende Verbindungen auf Port 1024 erlauben):

```
grant {
    permission java.net.SocketPermission "localhost:1099", "connect,resolve";
    permission java.net.SocketPermission "*:1024", "accept,resolve";
};
```

3.4.3 RMI: Codebeispiel

Definition des Remote-Interfaces:

```
public interface RemoteSum extends Remote {
    public int sum(int a, int b) throws RemoteException;
}
```

Implementierung des Remote-Interfaces:

```
public class RemoteSumImpl extends UnicastRemoteObject extends RemoteSum {
    @Override
    public int sum(int a, int b) throws RemoteException {
        return a + b;
    }
}
```

Erzeugung der Registry:

```
public class RegistrySetup {
    public static void main(String[] args)
            throws RemoteException, InterruptedException {
        Registry registry = LocateRegistry.createRegistry(
            Registry.REGISTRY_PORT); // 1099
        synchronized (reg) {
            reg.wait(); // inaktiver Wartezustand, nicht terminieren!
        }
    }
}
```

Registrierung des Remote-Objekts:

```
public class SumServer {
    public static void main(String[] args) throws RemoteException,
            AlreadyBoundException, MalformedURLException
        RemoteSumImpl sum = new RemoteSumImpl();
        String url = "rmi://localhost:1099/sum";
        Naming.bind(url, sum);
    }
}
```

Implementierung des Clients:

```
public class SumClient {
    public static void main(String[] args) {
        try {
            int a = 65;
            int b = 23;
            String url = "rmi://localhost:1099/sum";
            RemoteSum sum = (RemoteSum) Naming.lookup(url);
            int result = sum.sum(a, b);
            System.out.println(sum); // 88
        } catch (RemoteException | NotBoundException | MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

3.4.4 Push-Prinzip

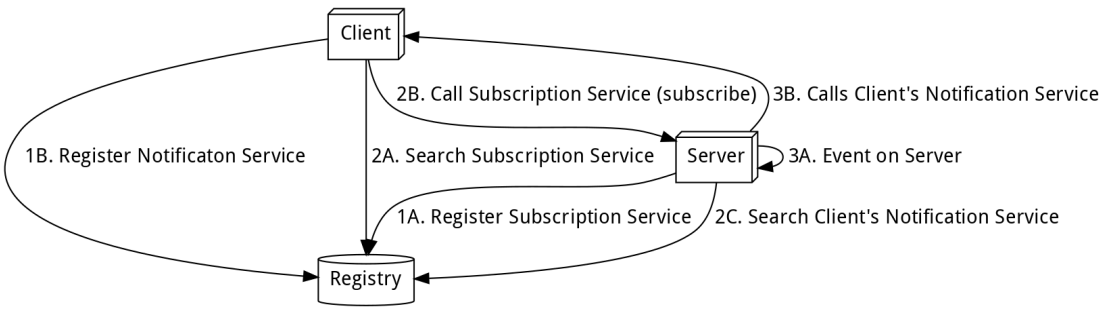

Abbildung 11: RMI: Push-Prinzip

- Umsetzung des Push-Prinzips mittels RMI (siehe RMI: Push-Prinzip):
  1. Registrierung der Services
     - A. Der Server bietet einen Dienst an, auf welchem sich Clients für Notifikationen registrieren können.

    - B. Der Client bietet einen Dienst an, welchen der Server für die Notifikation aufrufen kann. Jeder Client registriert sein Remote-Objekt unter einem eindeutigen Namen.
  2. Vorbereiten der Push-Kommunikation
    - A. Der Client findet den serverseitigen Service für die Registrierung.
    - B. Der Client registriert sich beim Server, um Notifikationen zu erhalten. Er übergibt dem Server den eindeutigen Namen, unter dem sein Remote-Objekt auf der Registry zu finden ist.
    - C. Der Server findet den clientseitigen Service für die Notifikation anhand des zuvor übergebenen eindeutigen Namens.
  3. Notifizierung des Clients
    - A. Auf dem Server tritt ein Ereignis ein, das für die Clients von Interesse ist.
    - B. Der Server notifiziert alle Clients über deren Notifikationsservice.
- Mit dieser Architektur lassen sich verteilte Observer und Callback-Aufrufe realisieren.

3.5 Uhrensynchronisation

3.6 Verteilung: Data Grid