Lernzettel

Komponentenbasierte Entwicklung und Modularisierung

Universität: Technische Universität Berlin
Kurs/Modul: Softwaretechnik und Programmierparadigmen
Erstellungsdatum: September 19, 2025

Lernzettel: Komponentenbasierte Entwicklung und Modularisierung

(1) Definition.
Die komponentenbasierte Entwicklung (CBSE) beschreibt das Erstellen von Software durch Zusammensetzen bereits entwickelter, gut definierter Bausteine (Komponenten) mit klaren Schnittstellen. Ziel ist eine hohe Wiederverwendbarkeit, einfache Wartbarkeit und flexible, skalierbare Architekturen durch lose Kopplung und hohe Kohäsion der Komponenten.

(2) Grundbegriffe.

- *Komponente*: eigenständige, wiederverwendbare Softwareeinheit mit definierten Schnittstellen.
- *Schnittstelle* (Interface/Vertrag): Spezifikation des Verhaltens, der API und der Versionierung.
- *Schnittstellendesign*: Verträge statt Implementierungsdetails, erlaubt Austauschbarkeit.
- *Abstraktion*: Verbergen von Details hinter einer stabilen API.
- *Kopplung* und *Kohäsion*: lose Kopplung, hohe Kohäsion verbessern Wartbarkeit und Austauschbarkeit.
- *Modul* vs. *Komponente*: Module liefern Struktur; Komponenten liefern Verhalten über Schnittstellen.
- *Plug-in-Architektur*: Laufzeit-erweiterbare Systeme durch Austauschen/Ergänzen von Komponenten.

(3) Architekturprinzipien.

- *Lose Kopplung, hohe Kohäsion*: Komponenten sollten wenig voneinander abhängen, intern möglichst stark zusammenarbeiten.
- *Schnittstellenorientierung*: Klar definierte Verträge ermöglichen Austauschbarkeit ohne Re-Integration von Abhängigkeiten.
- *Information Hiding*: Implementierungsdetails bleiben verborgen; nur die Schnittstelle ist sichtbar.
- *Abstraktion und Verträge*: Verhalten wird über formale Verträge festgelegt (z.B. API, Spezifikationen).
- *Versionierung und Kompatibilität*: Verlässliche Abwärtskompatibilität durch SemVer o.ä.
- *Plug-in- und Extensibility-Patterns*: Systeme unterstützen Erweiterungen ohne Kernänderungen.

(4) Lebenszyklus und Vorgehensweise.

- *Anforderungsanalyse* und Domänenmodell für Component-Schnittstellen.

- *Interface-first Design*: Schnittstellen definieren, bevor Implementierung erfolgt.
- *Spezifikation der Verträge*: Formale oder halbstatische Spezifikationen der API und Laufzeitverhalten.
- *Auswahl oder Entwicklung von Komponenten*: Checks zu Funktionalität, Kompatibilität, Wartbarkeit.
- *Integration und Orchestrierung*: Zusammenspiel der Komponenten zur Gesamtanwendung.
- *Tests*: Komponententests, Integrations- und Systemtests; Vertragsprüfung der Schnittstellen.
- *Wartung und Evolution*: API-Kompatibilität sicherstellen, Migrationen planen.

(5) Techniken und Muster.

- *Dependency Injection (DI) / Inversion of Control (IoC)* zur Loslösung von Implementierungen von Clients.
- *Contract-first Entwicklung*: Verträge der Schnittstellen zuerst festlegen.
- *Service-Registry / Discovery*: Laufzeit-Erkennung von verfügbaren Komponenten.
- *Versionierung und Kompatibilität*: SemVer oder ähnliche Strategien.
- *Paket- und Modulmanagement*: z. B. Maven, Gradle, npm, NuGet, etc. zur Verteilung von Komponenten.
- *Plug-in-Architekturen*: Erweiterbarkeit durch unabhängige Plugins.
- *Verifikation der Verträge*: Tests, Formale Methoden oder Hoare-ähnliche Logik auf Schnittstellenebene.

(6) Vorteile und Herausforderungen.
Vorteile:

- Wiederverwendbarkeit von Komponenten über Projekte hinweg.
- Verbesserte Wartbarkeit durch klare Schnittstellen.
- Erleichterte Anpassung und Erweiterung durch Plug-ins.
- Skalierbare Entwicklung durch Verteilung der Arbeitslast.

Herausforderungen:

- Overhead durch Schnittstellendefinition und Vertragspflege.
- Komplexität der Build- und Laufzeit-Orchestrierung.
- Aufwand für Versionierung, Abwärtskompatibilität und Migration.

(7) Beispiele und Anwendungsfälle.

- Web- und Desktop-Plug-ins (z. B. Editoren, IDEs, CMS-Plugins).
- Plug-in-fähige Anwendungsplattformen mit Extensions (z. B. Browser-Add-ons, ERP-Systeme).
- Modulbasierte Serveranwendungen und Microservice-Architekturen mit klaren Schnittstellen.
- Bibliotheken und Frameworks, die als Komponenten wiederverwendet werden.

(8) Qualitätssicherung im CBSE-Kontext.

- Vertragsbasierte Tests der Schnittstellen (Schnittstellentests).
- Komponententests isoliert gegen definierte Interfaces.
- Integrations- und Systemtests, die das Zusammenwirken der Komponenten prüfen.
- Metriken zu Kopplung, Kohäsion, Wiederverwendung und Laufzeitverhalten.
- Regressionstests bei Komponenten-Austauschen oder API-Änderungen.

(9) Zusammenhang mit anderen Paradigmen.
Komponentenbasierte Entwicklung ergänzt objektorientierte, modellbasierte und funktionale Ansätze:

- OOP ergänzt CBSE durch Klassen und Objekte als Bausteine mit Schnittstellen.
- Modellbasierte Ansätze nutzen Abstraktionen, um Komponentenverträge zu modellieren.
- Funktionale Programmierung kann als Komponenten mit reinen Funktionen und unveränderlichem Zustand organisiert werden.
- Logische Programmierung kann über komponentenbasierte Orchestrierung in Systeme integriert werden.