Software Engineering II

7 Wiederverwendung

SS 2020

Prof. Dr. Dirk Müller

Übersicht

- Einführung
  - Einheiten der Wiederverwendung
  - Vor- und Nachteile
  - Ansätze
  - Schlüsselfaktoren
- Programmbibliotheken
  - Module und Klassen
  - *Callback*-Mechanismus
- Anwendungssysteme
  - Konfiguration vs. Integration
  - ERP-Systeme
- Zusammenfassung

Vier Grundprinzipien der Softwaretechnik

- Anwendung eines gemanagten und gut verstandenen SW-Entwicklungsprozesses (z. B. *V-Modell*, *XP*)
  - klare Ideen, was produziert wird und wann es fertiggestellt wird
  - geeigneter Prozess vom Typ der Software abhängig
- Funktionalität, Verlässlichkeit + Leistung
  - funktionale Anforderungen, keine Fehler, hohe Verfügbarkeit
  - Sicherheit im Sinne von *Safety* und *Security*
  - Effizienz, keine Verschwendung von Ressourcen
- Verständnis der Spezifikation + der Anforderungen
  - Erwartungen von Kunden- und Nutzergruppen managen
  - Lieferung innerhalb des Budgets zum Termin
- Nutzung vorhandener Ressourcen (wenn angebracht)
  - vorhandene Software(-teile) nutzen statt alles neu zu schreiben

nur so „Softwaretechnik“ berechtigt, kein *Code & Fix*

Qualität

agile Methoden für Zeit- und Kosten-management

Wiederverwendung

Quelle: [1], S. 26

Softwaretechnik für *Web*-Anwendungen

- Geschäftsanwendungen bis ca. 2000
  - monolithische Einzelplatzanwendungen
  - lokale oder auf ein LAN beschränkte Kommunikation
- Geschäftsanwendungen seit ca. 2000
  - hochgradig verteilte Software, manchmal über die ganze Welt
  - nicht mehr von Grund auf programmiert, sondern extensive Wiederverwendung von Komponenten und Programmen
- Zusammensetzung aus bestehenden Komponenten und Systemen, die oft in *Frameworks* gebündelt sind
- Anforderungen sind nicht alle am Anfang bekannt, sondern werden ständig aktualisiert: iterativ-inkrementell Entwicklg.
- dienstorientierte SW-Technik mit autonomen *Web*-Diensten
- *Rich-Client*-Schnittstellen direkt im Web-Browser nutzbar: *AJAX* und *HTML5*

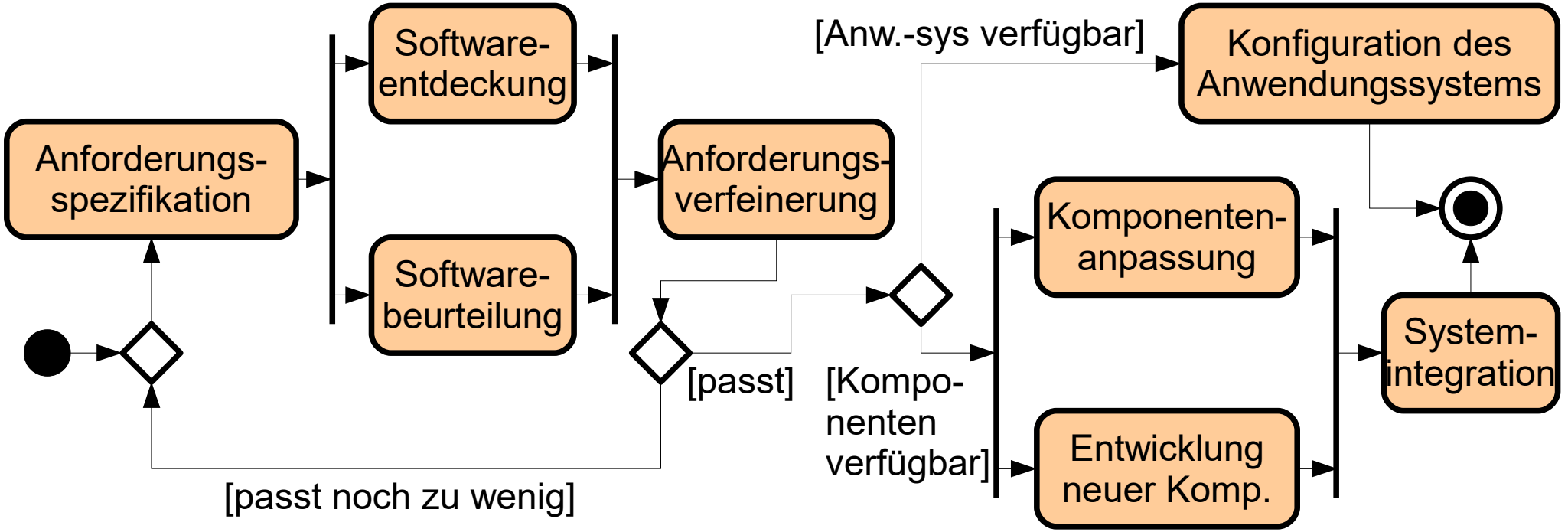
Quelle: [1], S. 27 f.

Reuse-oriented Software Engineering

- auf Wiederverwendung orientierte Softwaretechnik
  - Maximierung des Grades der Wiederverwendung als Ziel
  - setzt auf Integration und Konfiguration
- informelle Wiederverwendung: quelloffene Codestücke werden angeschaut, geändert und integriert
  - findet bei allen SW-Entwicklungsprozessen statt
- seit 2000 Prozesse, die Wiederverwendung explizieren
- Vorteile
  - Kosten- und Risikosenkung
  - schnellere Auslieferung
- Nachteile
  - Kompromisse bei den Anforderungen
  - Kontrollverlust über Evolution der wiederverwendeten Komponenten

Quelle: [1], S. 52 ff.

Workflow der auf Wiederverwendung orientierten Softwaretechnik



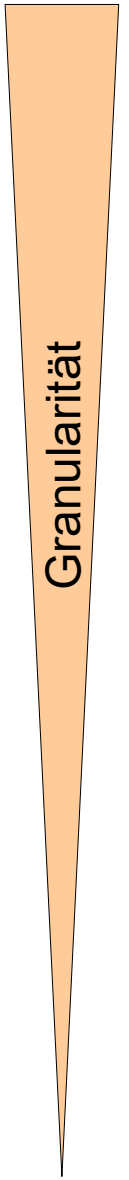
Meta-Ansätze für SW-Entwicklungsmodelle

- Wasserfallmodell — Basis für monumentale (plangetriebene) Modelle
- inkrementelle Entwicklung — Basis für agile Modelle
- Integration und Konfiguration

Quelle: [1], S. 52 f.

Einheiten der Wiederverwendung

- Systeme
  - Systeme, die aus verschiedenen Anwendungen bestehen
  - *System of systems*: min. 2 individuell gemanagte Systeme
- Anwendungen
  - müssen ggf. noch konfiguriert werden müssen (*Customizing*)
  - auch innerhalb von Produktlinien
- Komponenten
  - in der *Cloud* oder auf privaten Servern
  - Zugriff über API (*Application Programming Interface*) und damit Nutzung als Dienst
- Objekte und Funktionen
  - Ablage in Standardbibliotheken, seit ca. 1975 üblich
  - mathematische und Grafikbibliotheken sinnvoll, da komplizierte Optimierungen (Zeitverhalten, Interaktivität) dann einfach verfügbar sind



Quelle: [1], S. 437 ff.

Vorteile der Wiederverwendung

- schneller: *Time to Market* wird reduziert
  - Entwicklungs- und Validierungszeit verringert
- bessere Nutzung von Spezialisten
  - Fachwissen kann in wiederverwendbare SW gegossen werden
- Verlässlichkeit
  - wiederverwendete Teile schon länger da, somit sind viele Entwurfs- und Implementierungsfehler schon gefunden + korrigiert
- geringere Kosten
  - weniger Codezeilen (LOC) geschrieben, Einkauf meist günstiger
- geringeres Prozessrisiko
  - Streubereich der Kostenschätzung reduziert, Kosten eingekaufter Software sind bekannt
- Einhaltung von Standards
  - bes. Nutzerschnittstelle, weniger Fehlbedienung durch Nutzer

Quelle: [1], S. 439

Nachteile der Wiederverwendung

- Erstellung, Wartung + Nutzung einer Komponentenbibl.
  - Anpassung des Entwicklungsprozesses nötig
- Finden, Verstehen und Anpassen wiederverwendbarer Komponenten
  - neue Umgebung, Grad der nötigen Anpassung bzw. der Abstriche

Präzise Spezifikation nötig!

- erhöhte Wartungskosten
  - nicht quelloffene Bibliothek kann durch Änderungen am eigenen System inkompatibel werden
- keine Werkzeugunterstützung
  - Wiederverwendung nicht vorgesehen, z. B. bei der Entwicklung eingebetteter Systeme
- *Not-invented-here*-Syndrom (NIH)
  - lieber alles selbst schreiben, da mangelndes Vertrauen und Wille, sich zu beweisen und zu trainieren

Quelle: [1], S. 440

Ansätze zur Wiederverwendung

- Anwendungs-*Frameworks*
- Anwendungssystemintegration
- Architekturmuster
- Aspekt-orientierte Softwareentwicklung (AOSD)
- Komponentenbasierte SW-Entwicklung (CBSD)
- konfigurierbare Anwendungssysteme
- Entwurfsmuster
- ERP-Systeme (*Enterprise Resource Planning*)
- Kapselung von Altlast-Systemen (*Legacy Systems*)
- Modellgetriebene SW-Entwicklung (MDSD)
- Programmgeneratoren
- Programmbibliotheken
- dienstorientierte Systeme: SOA(P)
- Software-Produktlinien
- *Systems of systems*

hier behandelt

in „*Software Factories*“ behandelt

Quelle: [1], S. 442

Schlüsselfaktoren zur Planung

- je mehr Termindruck, desto mehr Wiederverwendung
  - Anforderungen werden nur ca. eingehalten, dafür der Termin genau
- erwartete Lebensdauer der Software
  - Änderbarkeit zentral bei langer Lebensdauer, würde gegen Wiederverwendung sprechen, Support kann einmal enden
  - quelloffene Komponenten anpassbar, andere nicht
- Hintergrund, Fähigkeiten + Erfahrung des Entwicklerteams
  - komplexe Ansätze, während Einarbeitungszeit kein Nettonutzen
- kritische Systeme, nicht-funktionale Anforderungen
  - Zertifizierung ohne Quellcode kaum möglich
  - manuelle Optimierung ist MDSD bzgl. Zeitverhalten noch überlegen
- Anwendungsdomäne
  - konfigurierbare Anwendungssysteme im Maschinenbau oder für medizinische Informationssysteme als Standard
- Plattform
  - nicht frei kombinierbar, *Lock-in*-Effekt, z. B. .NET bei *Microsoft*

Quelle: [1], S. 441 f.

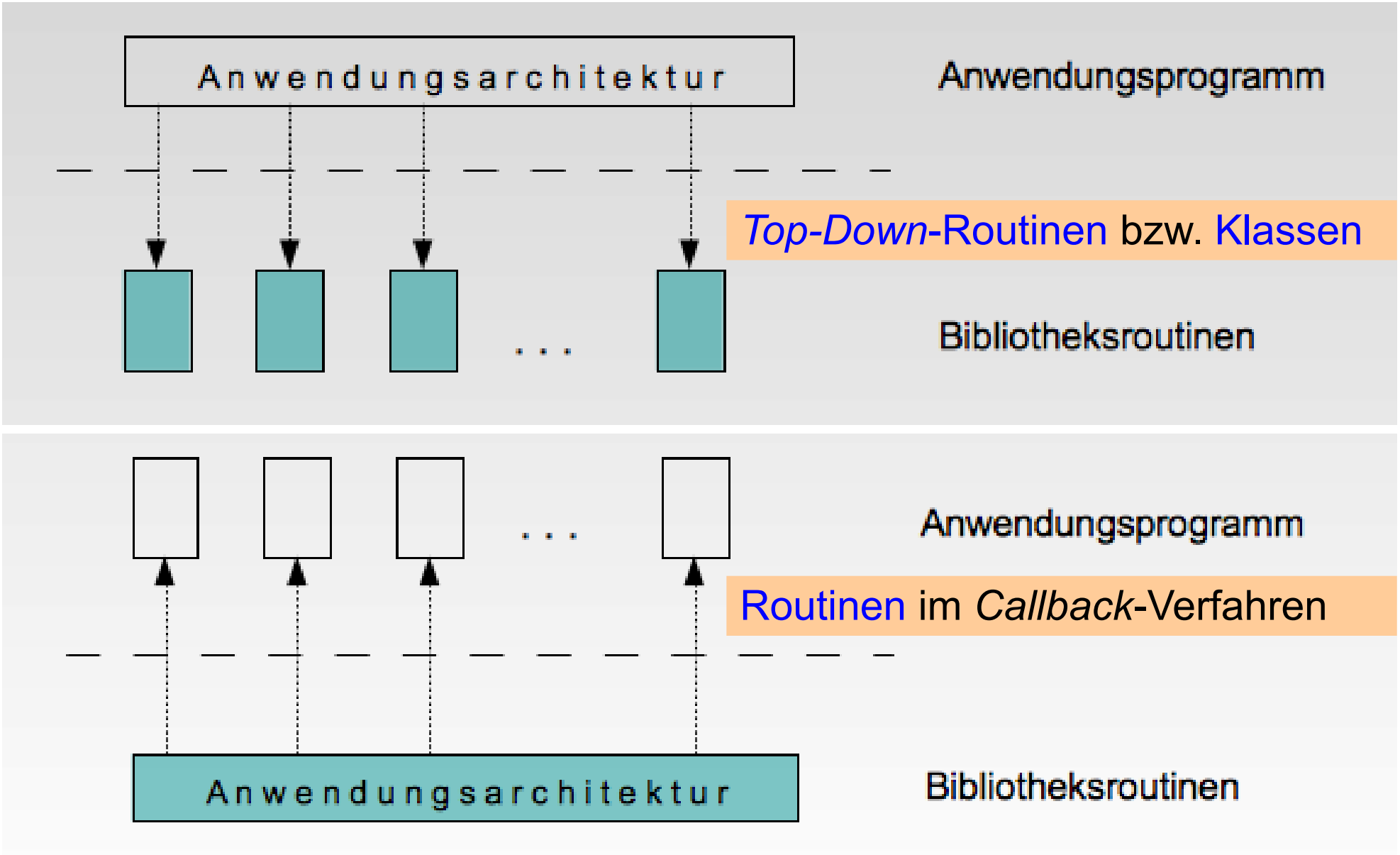
Programmbibliotheken

- Modul als abstrakte Datenstruktur (ADS), z. B. *Modula 2*
  - werden leicht modifiziert bei Wiederverwendung
- Klasse als abstrakter Datentyp (ADT), z. B. *Java*
  - Objekterzeugung in Objektfabriken
  - können abgeleitet werden (Vererbung, keine Änderung des bestehenden Quelltextes/Modells, nur der Modellinstanzen; auch Mehrfachvererbung, in *Java* über *Interfaces*)
  - Polymorphismus bei Methoden und Operatoren (Überladen) liefert einheitlichen Zugriff und damit intuitive Anwendung, z. B. liefert 3 + 2 = 5 und „AL“ + „PHA“ = „ALPHA“ => Wiederverwendung der Schnittstelle

aber:Verkettung mit Pluszeichen beschrieben, nicht kommutativ

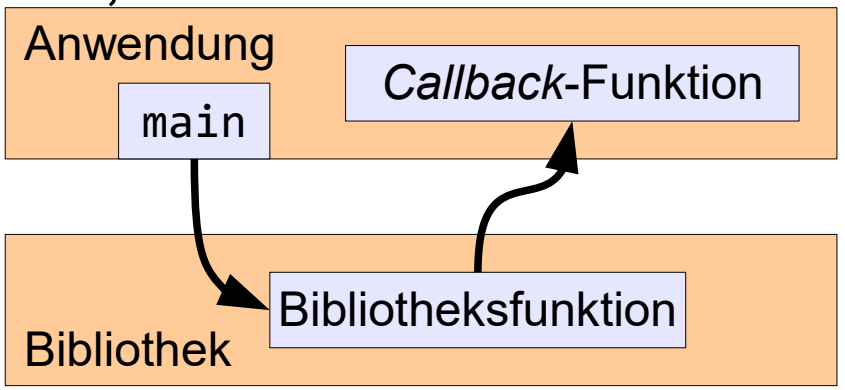
- objektorientierte Sprachen wie *C++* oder *Java* sind „Werkzeuge“, die nur bei korrekter Anwendung Wiederverwendung ermöglichen

Architektur von Programmbibliotheken



Callback-Mechanismus

- *Callback*: ein Stück ausführbarer Code, das als Argument an anderen Code übergeben wird, um ihn dort zu geeigneter Zeit auszuführen
- Funktion soll als Variable abstrahiert werden
  - z. B. als Funktionspointer in *C*



- *Hollywood*-Prinzip „*Don't call us, we will call you!*“
  - Umkehrung der Kontrolle
- typische Anwendung: grafische Benutzeroberfläche
  - Ereignisschleife in Bibliothek fängt Maus- und Tastatureingaben ab
  - anwendungsspezifische Routinen reagieren dann entsprechend nach einem Aufruf (sog. Einschubmethode, engl. *hook method*)
  - einfacher, aber weniger flexibel: Funktionalität nur an vorgesehenen Stellen anpassbar

Grenzen der Wiederverwendung von Klassen

- Enthusiasten des OO-Ansatzes sagten in der 1990ern, dass Klassen in verschiedenen Systemen wiederverwendet werden können.
- aber: häufig zu fein-granular, zu speziell
- typische Situation: mehr Zeit für Verständnis und Anpassung der vorhandenen Klasse als für Neuimplementierung, somit nicht wirtschaftlich
- Ausweg: Wiederverwendung auf höherer Granularitätsstufe, z. B. in Form von *Frameworks*, Produktlinien, Diensten, konfigurierbaren Anwendungen oder sogar ganzen Systemen

Quelle: [1], S. 443

Wiederverwendung von Anwendungssystemen

- Anwendungssystem
  - Serienfertigung, Verkauf in großer Stückzahl
  - Standardsoftware, keine Individualsoftware
  - auch: *Commercial Off-the-Shelf System*, COTS
  - auch: O*ut-of-the-box-Software*
  - für Wiederverwendung: *Modifiable Off-the-Shelf System*, MOTS
- Anpassung an Bedürfnisse beim Kunden möglich
  - ohne Änderung/Ergänzung von Quellcode: Einstellungen, Konfigurationsdateien
  - mit Ergänzung von Quellcode: Plug-ins
- schnellere und kostengünstigere Entwicklung

Quelle: [1], S. 453

Konfiguration vs. Integration

konfigurierbares Anwendungssystem

- 1 Produkt für den Kunden
- basiert auf generischer Lösung + standardisierten Prozessen
- Konfiguration
- Systemanbieter zuständig für Wartung
- Systemanbieter liefert Plattform für das System

Anwendungssystem-integration

- mehrere Anwendungs-systeme für Kunden-bedürfnisse integriert
- flexible Lösungen für Kunden
- Integration
- Systemeigner zuständig für Wartung
- Systemeigner liefert Platt-form für das System

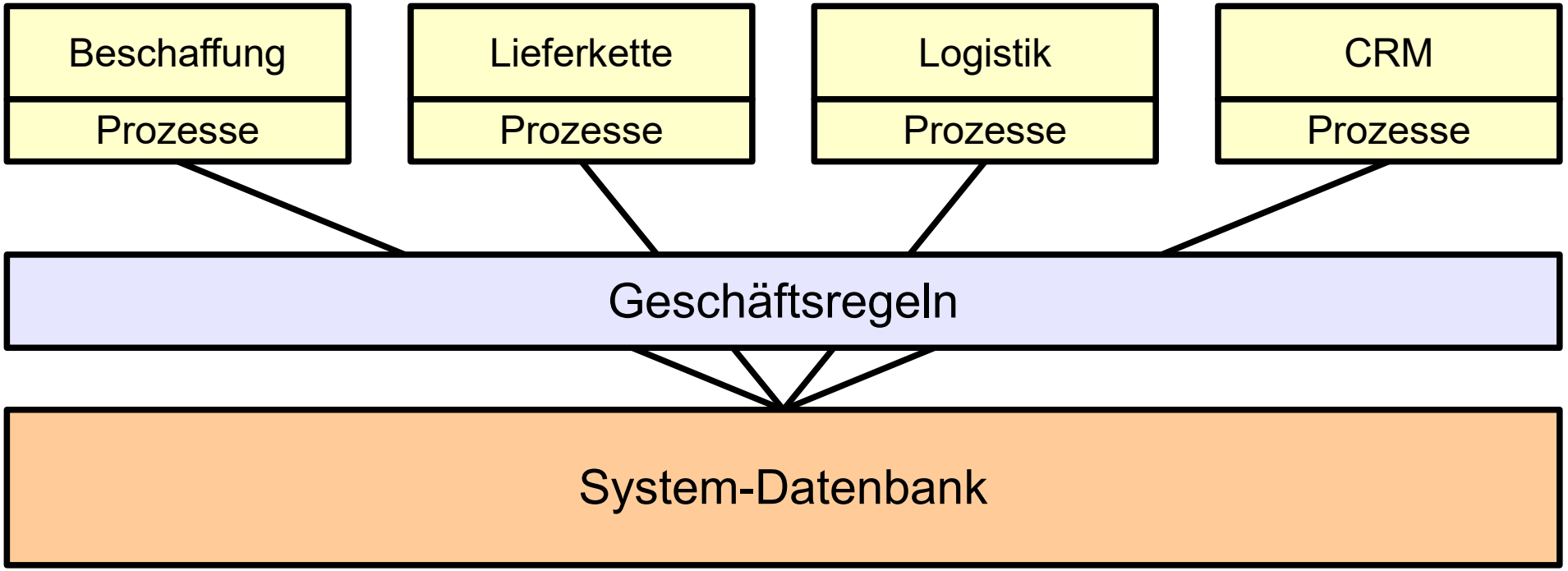
Quelle: [1], S. 454

Konfigurierbare Anwendungssysteme

- Beispiel Zahnarztsoftware
  - Termine, Erinnerungen, Patientendaten, Rechnungen
- *Enterprise Resource Planning* (ERP)
  - Produktion, Bestellung, Kapital, Personalwesen, IT-Systeme, *Customer Relationship Management* (CRM), etc.
  - Marktführer: *SAP* und *Oracle*
  - Module werden ausgewählt, konfiguriert und zusammengesetzt
  - Geschäftsprozesse und Geschäftsregeln definieren
- generische Funktionalität deckt nicht alle Fälle ab
  - z. B. Studentenmanagement bei Doppelabschlüssen
  - Studenten sind keine richtigen Kunden
- Testen ist sehr schwierig
  - keine API, Einsatz von *xUnit* nicht möglich
  - Missverständnisse zwischen Konfigurationsteam u. Entscheidern

Quelle: [1], S. 454 ff.

Architektur eines ERP-Systems

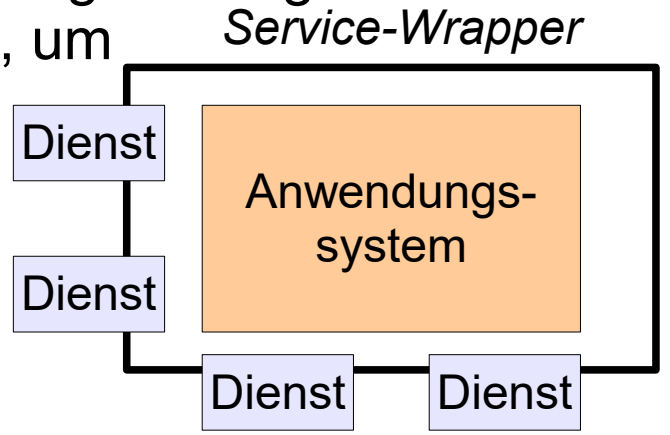


- Trend zum *Cloud*-Computing
- Datenschutz ist ein Problem, schon jetzt Industriespionage
- schnelle Anpassung an geänderte Geschäftsprozesse als Schlüssel zum Erfolg: Daten in Echtzeit

Quelle: [1], S. 455 f.

Integrierte Anwendungssysteme

- Motivation
  - gefundene Anwendungssysteme zu weit weg von Anforderungen
  - Integration neuer Systeme und bereits genutzter Systeme
- Entwurfsentscheidungen
  - Auswahl der zu integrierenden Anwendungssysteme (möglichst nah an den Anforderungen)
  - Schnittstellen für Datenaustausch, ggf. Adapter entwerfen
  - Entscheidung, welche Funktionalität durch welches Anw.-sys abgedeckt werden soll, dann Deaktivierung der ungenutzten Funktionalität in den anderen Anw.-sys., um Interferenzen/Störungen zu vermeiden
- dienstorientierter Ansatz kann Architektur verbessern
  - besonders gut für Altlast-Systeme



Quelle: [1], S. 459

Herausforderungen bei der Systemintegration

- Kontrollverlust über Funktionalität und Leistung
  - Hochglanzprospekte enthalten oft Halbwahrheiten
  - Wechselwirkung zwischen verschiedenen Systemen führen zu Leistungseinbrüchen unter bestimmten Betriebsbedingungen
- Systeme mit eigenen Annahmen über Anwendungsfälle und Schnittstellen
  - Interoperabilität nur mit Standards vernünftig möglich
  - z. B. ereignisgesteuerte Kommunikation bei 3 Systemen, aber
    - 3 verschiedene Modelle eines Ereignisses
    - jedes mit der impliziten Annahme, dass es exklusiven Zugriff auf die Ereignis-Warteschlange hat
    - Projekt benötigte 2 Jahre statt 6 Monate
- Kontrollverlust über System-Evolution
  - neue Versionen inkompatibel zu älteren oder mit unnötigen Fkt.
- *Support* unsicher: Pleite, Übernahme, Produkteinstellung

Quelle: [1], S. 459 f.

Zusammenfassung

- Wiederverwendung als eines von 3 Prozess-Paradigmen
- Granularität von Methoden und Klassen bis zu Systemen
- Vorteile
  - geringere Kosten, schnellere Entwicklung, geringeres Risiko
  - System-Verlässlichkeit steigt
  - effizienterer Einsatz von Spezialisten zur Erstellung wiederverwendbarer Komponenten (Fachwissen konserviert)
- Nachteile
  - Finden, Verstehen und Anpassen wiederverwendbarer Komponenten anspruchsvoll
  - erhöhte Wartungskosten, mangelhafte Werkzeugunterstützung
- Kontrollverlust durch Einkauf von Produkten und Leistungen (*Outsourcing*)

Literatur

[1] Ian Sommerville: „Software Engineering 10“, Addison-Wesley, 2016