Karlsruher Institut für Technologie

Kapitel 3.4 – Architekturstile

SWT I – Sommersemester 2021
Walter F. Tichy, Christopher Gerking, Tobias Hey

Abstrakte/virtuelle Maschine (engl. Abstract/Virtual Machine)

Def.: Eine **abstrakte Maschine** oder **virtuelle Maschine** ist eine Menge von Softwarebefehlen und -objekten, die auf einer darunterliegenden (abstrakten oder realen) Maschine aufbauen und diese ganz oder teilweise verdecken können.

- Beispiele für abstrakte Maschinen:
  - Programmiersprache, Betriebssystem, Anwendungskern, GUI-Bibliothek, Java VM.
- Die Benutztrelation zwischen mehreren abstrakten Maschinen ist hierarchisch, d. h. zyklenfrei.

Abstrakte Maschine

- Eine abstrakte Maschine wird in der Regel von einem oder mehreren Modulen oder Paketen implementiert. Dabei werden die Softwarebefehle und -objekte von den Schnittstellen dieser Module bereitgestellt.
- Die darunter liegende Maschine muss ganz oder teilweise verdeckt werden, um Inkonsistenzen der beiden Maschinen zu vermeiden.
- Die Befehle der abstrakten Maschine sollen so gewählt werden, dass sie in einer Vielzahl von Programmen verwendet werden können.
- Eine abstrakte Maschine ist oft als eine Programmbibliothek oder Application Programming Interface (API) verwirklicht.

Beispiele für abstrakte Maschinen

- **Ein Betriebssystem** stellt eine mächtige abstrakte Maschine zur Verfügung mit
  - Prozessverwaltung,
  - virtuellem Speicher,
  - Datenhaltung auf Hintergrundspeicher,
  - Kommunikation
  - Kommandosprachen, grafische Benutzeroberfläche
- Das Betriebssystem verdeckt gewisse privilegierte Befehle, die zur Implementierung der Betriebssystemdienste benötigt werden.
- Die Nicht-Nutzung der privilegierten Befehle wird zur Laufzeit überprüft und führen zu einer Unterbrechung oder werden ignoriert.

Beispiele für abstrakte Maschinen

- Java-Virtuelle-Maschine: interpretiert Bytecode
- Java-Übersetzer
  - Bietet abstrakte Maschine, die in Java programmiert wird
  - Setzt selbst auf einer weiteren abstrakten Maschine, der Java VM, auf
  - Dabei sind Maschinenbefehle, Bytecode verdeckt
  - Die Verdeckung wird vom Übersetzer überprüft
- (Analog: Interpretierer, Dokumentformatierer)

Weitere Beispiele für abstrakte Maschinen

- System zur Bearbeitung elektronischer Post
  - Bietet Befehle zum Empfangen und Senden von Nachrichten, verdeckt die darunterliegenden Protokolle (Pufferung, Stückelung und Zusammensetzung von langen Nachrichten, Quittungen, Fehlerbehandlung), die für zuverlässigen Nachrichtenaustausch über Kommunikationskanäle erforderlich sind.
- Makros
  - Makros bieten parametrisierbare Textersetzung an. Der „Aufruf“ eines Makros bewirkt eine Ersetzung des Aufrufs mit dem Ersatztext des Makros
  - Beispiel in C: *#define Swap(X,Y) temp=X; X=Y; Y=temp;*
  - *Swap(a,b)* wird ersetzt durch den Text *temp=a; a=b; b=temp*;
  - Makros Sind i. d. R. ungeeignet, die unterliegende Maschine zu verdecken. Z. B. können Call-Return-Assembler-Makros nicht verhindern, dass andere Instruktionen die Invarianten bez. des Aufrufkellers stören.

Weitere Beispiele für abstrakte Maschinen

- Virtualisierung ist bei Dienstgebern im Internet („Cloud Computing“) von großer Bedeutung.
  - Ein Kunde kann z.B. einen IBM-Rechner mieten. Dieser kann auf einem Rechner anderen Typs „virtualisiert“ sein, d.h. der Befehlssatz des IBM-Rechners wird komplett als virtuelle Maschine auf einem anderen Rechner simuliert. Der Kunde kann auf dieser virtuellen Maschine sein gewünschtes Betriebssystem installieren.
  - Pro echtem Rechner können auch mehrere (sogar unterschiedliche) virtuelle Maschinen virtualisiert werden.
  - Die Virtualisierung wird entweder durch Software geleistet, aber auch durch dynamische Übersetzung von Befehlsfolgen der virtuellen Maschine in Befehlsfolgen der simulierenden Maschine (vergl. Just-in-time compiler für Java VM)
  - Mit Virtualisierung wird eine gute Auslastung der Rechnerresourcen gewährleistet. Auch das „Umziehen“ von Anwendungen von einem Rechner auf einen anderen bei Lastschwankungen ist möglich.

Programmfamilie / Software-Produktlinie (engl. Program Family / Software Product Line)

Def.: Eine **Programmfamilie** oder **Software-Produktlinie** ist eine Menge von Programmen, die erhebliche Anteile von Anforderungen, Entwurfsbestandteilen oder Softwarekomponenten gemeinsam haben.

- Neue Mitglieder der Produktlinie können rasch entwickelt werden, in dem man Anteile von Anforderungen, Entwurf, Bibliotheken oder Komponenten wieder verwendet (evtl. leicht abgewandelt).
- Es ist also wesentlich kostengünstiger, ein neues Mitglied der Produktlinie zu erzeugen, als das Programm völlig von vorne zu entwickeln.

Programmfamilie / Software-Produktlinie

- Ziel:
  - Ausnutzung der Gemeinsamkeiten,
  - Wiederverwendung von Entwürfen, Spezifikationen und Anforderungen, Softwarekomponenten, Bibliotheken
  - zur Reduktion der Entwicklungs- und Wartungskosten.
- Variationspunkte:
  - Stellen in der Architektur, an denen Alternativen in Funktionalität, benutzte Plattform, etc. eingeführt werden können.

Wie unterscheiden sich Mitglieder einer Programmfamilie?

- Extern:
  - Sie laufen auf verschiedenen Hardware- und Betriebssystem-Konfigurationen.
  - Sie unterscheiden sich im Format der Ein-/Ausgabe.
  - Sie unterscheiden sich im Funktionsumfang, da manche Benutzer nur eine Teilmenge der Funktionen benötigen (und nicht gezwungen sein sollten, für nicht benötigte Funktionalität zu bezahlen, Betriebsmittel bereitzustellen oder Effizienzeinbußen hinzunehmen).

Wie unterscheiden sich Mitglieder einer Programmfamilie?

- Intern:
  - Sie unterscheiden sich in Datenstrukturen und Algorithmen,
    - wegen Unterschieden in den zur Verfügung stehenden Betriebsmitteln oder Unterschieden in den Leistungsanforderungen
    - wegen Unterschieden im Umfang der Eingabedaten oder der relativen Häufigkeit gewisser Ereignisse (z.B. Anfragen an das System)

Programmfamilie / SW-Produktlinie

- Wichtiger Unterschied zwischen Allgemeinheit und Flexibilität:
  - Ein Programm ist allgemein, falls es in vielen Situationen ohne Änderung benutzt werden kann. (Allgemeinheit bedeutet meist hohe Laufzeit- und Speicherplatzkosten.)
  - Ein Programm ist flexibel, falls es leicht für viele Situationen abgeändert werden kann. (Flexibilität erfordert höhere Entwurfskosten.)
- Alle Überlegungen bis jetzt zielen darauf ab, ein System flexibel zu machen und dadurch Wartung zu vereinfachen.
- Ein flexibler Entwurf ist änderungsfreundlich.

Architekturstile

- Architekturstile legen den Grobaufbau eines Softwaresystems fest. Wir behandeln folgende Stile:
  - Schichtenarchitektur
  - Klient/Dienstgeber (engl. client/server)
  - Partnernetze (engl. peer-to-peer)
  - Datenablage (engl. repository)
  - Modell-Präsentation-Steuerung (engl. Model-View-Controller)
  - Fließband (engl. pipeline)
  - Rahmenarchitektur (engl. framework)
  - Dienstorientierte Architektur (engl. service oriented architecture)

Der Klassiker: Die Schichtenarchitektur (engl. *Layered Architecture*)

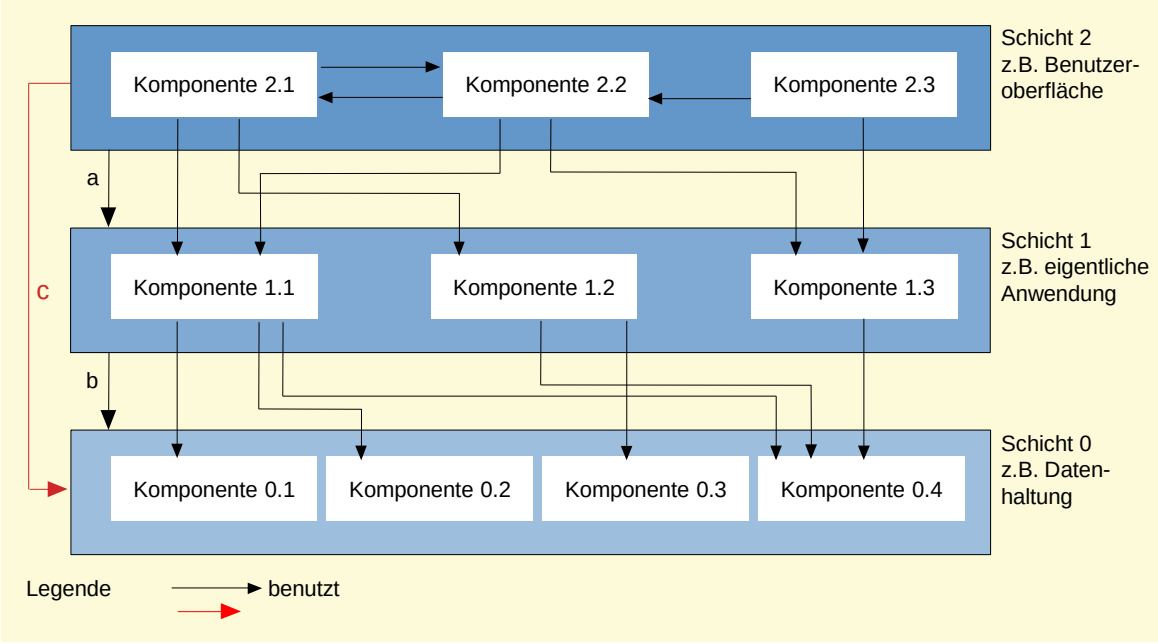
Def.: Eine **Schichtenarchitektur** ist die Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten.

Eine Schicht besteht aus einer Menge von Software-Komponenten (Module, Klassen, Objekte, Pakete) mit einer wohldefinierten Schnittstelle, nutzt die darunter liegenden Schichten und stellt seine Dienste darüber liegenden Schichten zur Verfügung.

Zwischen den einzelnen Schichten ist die Benutztrelation linear, baumartig, oder ein azyklischer Graph. Innerhalb einer Schicht ist die Benutztrelation beliebig.

Schichtenarchitektur

- Beispiel für eine Drei-Schichten-Architektur

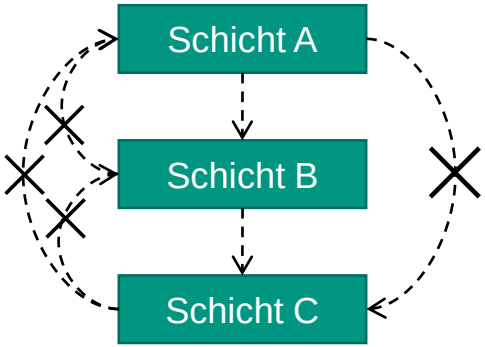


Schichten

- Eine Schicht (engl. *layer*, *tier*) ist ein Subsystem, welches Dienste für andere Schichten zur Verfügung stellt, mit folgenden Einschränkungen:
  - Eine Schicht nutzt nur Dienste von niedrigeren Schichten
  - Eine Schicht nutzt keine höheren Schichten
- Eine Schicht kann horizontal in mehrere, unabhängige Subsysteme, auch Partitionen genannt, aufgeteilt werden
  - Partitionen bieten Dienste für andere Partitionen der gleichen Schicht an

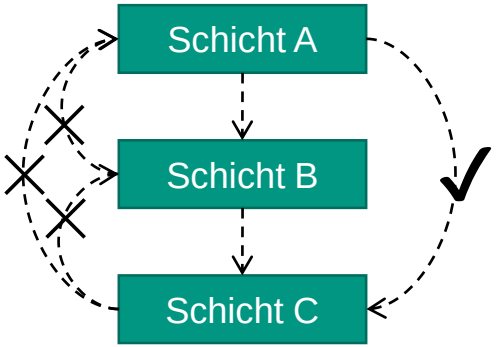
Intransparente Schichtenarchitektur

- Bei einer intransparenten Schichtenarchitektur werden undurchlässige Schichten (engl. *opaque layers*) verwendet
- Eine Schicht in einer solchen Architektur kann nur auf Dienste der Schicht direkt unter ihr zugreifen



Transparente Schichtenarchitektur

- Bei einer transparenten Schichtenarchitektur werden durchlässige Schichten (engl. *transparent layers*) verwendet
- Eine Schicht in einer solchen Architektur kann auf Dienste jeder Schicht unter ihr zugreifen.



Schichtenarchitektur

- Schichtenarchitektur, Vorteile:
  - Übersichtliche Strukturierung in Abstraktionsebenen oder virtuelle Maschinen (die Schichten stellen abstrakte Maschinen dar)
  - Keine zu starke Einschränkung des Entwerfenden, da er neben einer strengen Hierarchie noch eine liberale Strukturierungsmöglichkeit innerhalb der Schichten besitzt
  - Es werden die Wiederverwendbarkeit, die Änderbarkeit, die Wartbarkeit, die Portabilität und die Testbarkeit unterstützt (Schichten können ausgetauscht, hinzugefügt, portiert, verbessert und wieder verwendet werden; Testen von unten oder von oben her).

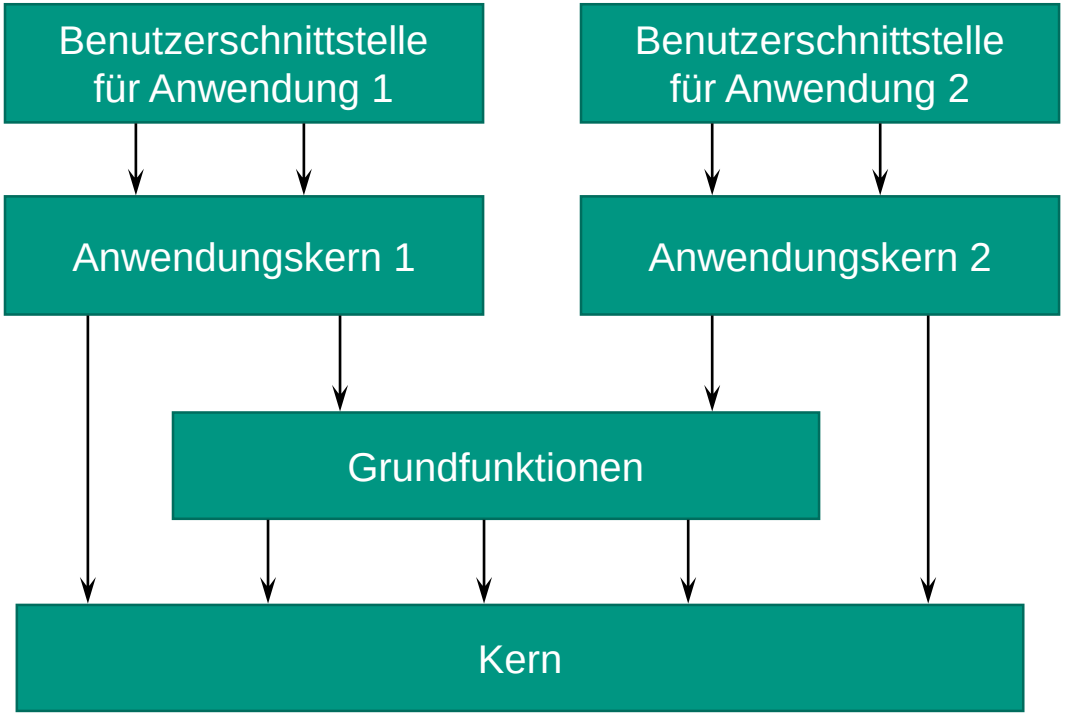
Schichtenarchitektur

- Schichtenarchitektur, Nachteile/Probleme:
  - Bei intransparenter Schichtung kann es Effizienzverluste geben, da Aufrufe, die mehrere Schichten überschreiten, durch mehrere Schichten weitergereicht werden müssen, und mehrfache Parameterübergabe und Ergebnisrückgabe erfordern.
    - Dies gilt auch für Fehlermeldungen (mit catch/throw aber kein Problem mehr)
  - Eindeutig voneinander abgrenzbare Abstraktionsschichten lassen sich nicht immer definieren.
  - Innerhalb einer Schicht darf kein Chaos herrschen!

3-Schichten-Architektur (1)

- **Def.:** 3-Schichten-Architektur
  - Ein Architekturstil, bei welchem die Anwendung aus 3 hierarchisch geordneten Subsystemen besteht
    - Eine Benutzerschnittstelle, Anwendungskern und einem Datenbanksystem.
- **Def.:** 3-stufige Architektur (engl. *3-tier architecture*)
  - Eine 3-Schichten-Architektur, bei welcher die Schichten auf unterschiedlichen Rechnern laufen.
    - Beispiel: Benutzerschicht läuft auf einem Klienten, Anwendungskern und Datenbank auf einem Dienstgeber

4-stufige Schichtenarchitektur



Noch eine 4-Schichten-Architektur

- 4-Schichten-Architekturen werden oft bei Webdiensten für elektronischen Handel verwendet:
  - Der Webbrowser stellt die Benutzerschnittstelle dar.
  - Der Webserver liefert statische HTML-Seiten.
  - Ein Anwendungs-Dienstgeber verwaltet Sitzungen (engl. Sessions) und erzeugt dynamische HTML-Seiten.
  - Eine Datenbank verwaltet die Daten.

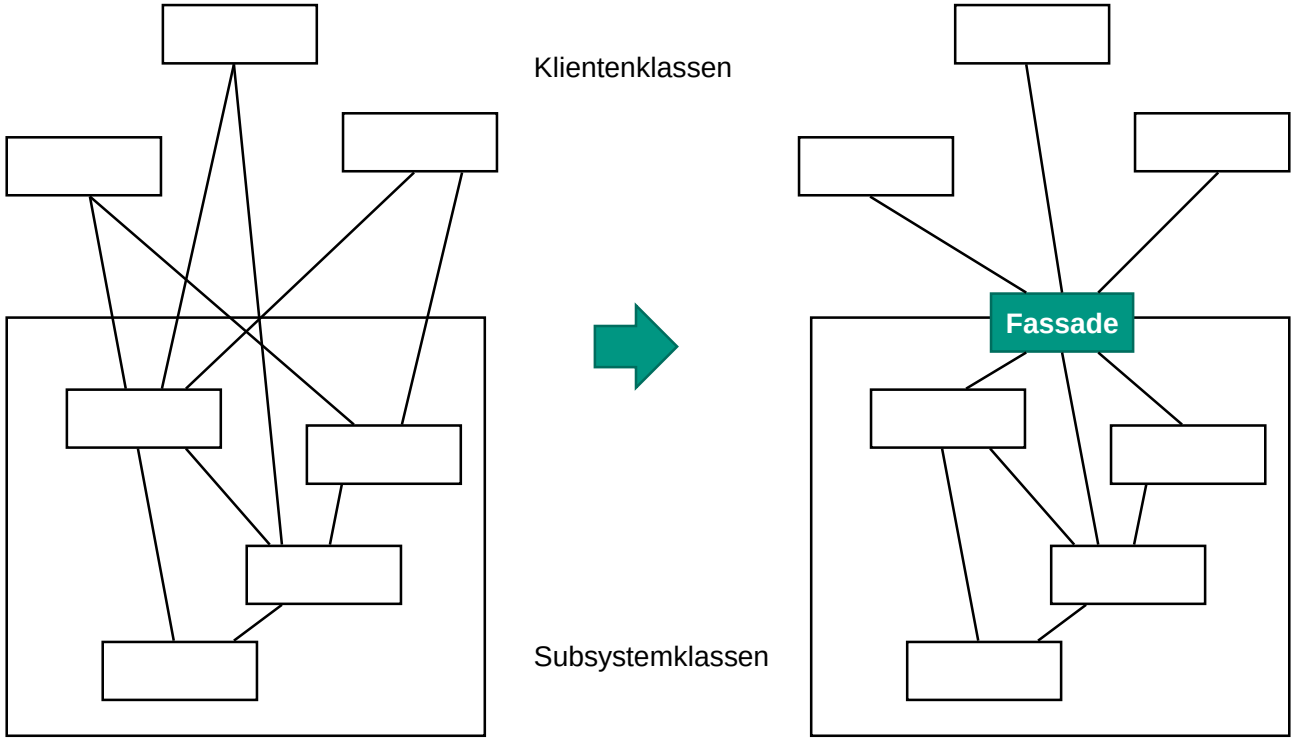
Weitere Beispiele

- Betriebssysteme: die wichtigsten Schichten sind:
  - Prozessverwaltung,
  - Speicherverwaltung,
  - Dateiverwaltung,
  - Kommunikation (Netzschnittstellen),
  - Kommandoschnittstelle,
  - graphische Benutzeroberfläche,
  - Anwendungen
- Protokolltürme bei der Datenfernübertragung
- Informationssysteme (auf Datenbanken aufbauend)

Schichtenarchitektur und das Entwurfsmuster Fassade (engl. *Facade*)

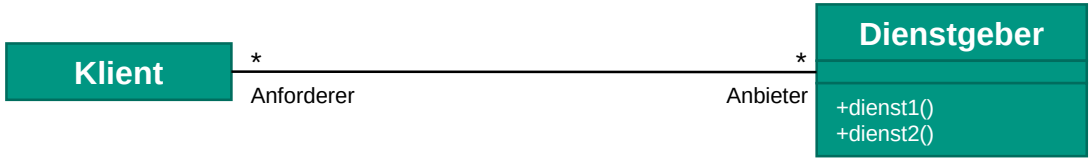
- Eine Schicht besteht oft aus einer Vielzahl von Elementen (Pakete, Module, Klassen, Objekte und Funktionen), die nicht alle an die darüberliegenden Schichten zur Verfügung gestellt werden sollen (weil zu komplex in der Bedienung oder nicht nötig)
- Um eine bereinigte und vereinfachte Schnittstelle zur Verfügung zu stellen, setzt man eine Fassade ein.
- Die Fassade ist eine oder mehrere Klassen, die nur die zur Verfügung stehenden Elemente enthält und an die eigentlichen Elemente in der Schicht delegiert.
- Die Fassade kann auch bestimmte Befehlsfolgen, die gerne zusammen benutzt werden, in neuen Methoden zusammenfassen (Bequemlichkeitsmethoden)

Schichtenarchitektur und das Entwurfsmuster Fassade



Klient/Dienstgeber (engl. *Client/Server*)

- Ein oder mehrere Dienstgeber bieten Dienste für andere Subsysteme, Klienten genannt, an.
- Jeder Klient ruft eine Funktion des Dienstgebers auf, welcher den gewünschten Dienst ausführt und das Ergebnis zurückliefert.
  - Dazu muss der Klient die Schnittstelle des Dienstgebers kennen.
  - Umgekehrt muss der Dienstgeber die Schnittstelle des Klienten nicht kennen.
- Ein Beispiel einer 2-stufigen, verteilten Architektur



Klient/Dienstgeber

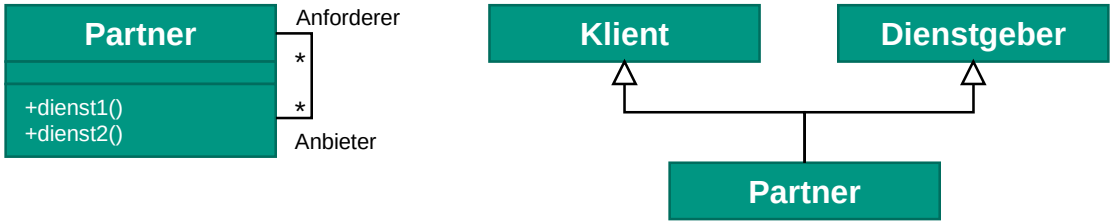
- Wird oft beim Entwurf von Datenbank-Systemen verwendet:
  - Front-End: Benutzeroberfläche für den Benutzer (Klient)
  - Back-End: Datenbankzugriff und Manipulation (Dienstgeber)
- Funktionen, die der Klient ausführt:
  - Eingaben des Benutzers entgegennehmen
  - Vorverarbeitung der Eingaben
- Funktionen, die der Dienstgeber ausführt:
  - Datenverwaltung
  - Datenintegrität und -Konsistenz
  - Sicherheit
- Klient und Dienstgeber laufen i.d.R. auf unterschiedlichen Rechnern, aber nicht unbedingt.

Klient/Dienstgeber: Beispiel

- Dateiübertragung auf einen FTP-Server:
  - Der Klient (bspw. ein FTP-Programm wie Filezilla) initiiert das Übertragen einer Datei.
  - Der Dienstgeber reagiert auf die Anfrage des Klienten und empfängt bzw. sendet die Datei.

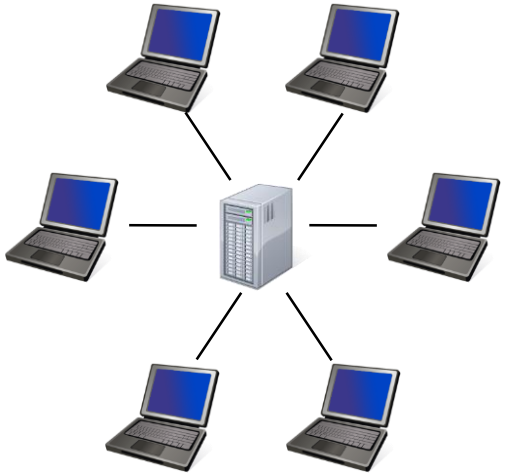
Partnernetze (engl. *Peer-to-Peer Networks*)

- Verallgemeinerung des Klient/Dienstgeber-Architekturstils.
- Bei einem Partnernetz sind alle Subsysteme gleichberechtigt („Peer“ bedeutet „Gleichgestellter“ , „Gleichaltriger“, s.a. „peer pressure“). Partner laufen auf unterschiedlichen Rechnern.
- Vereinfacht dargestellt:

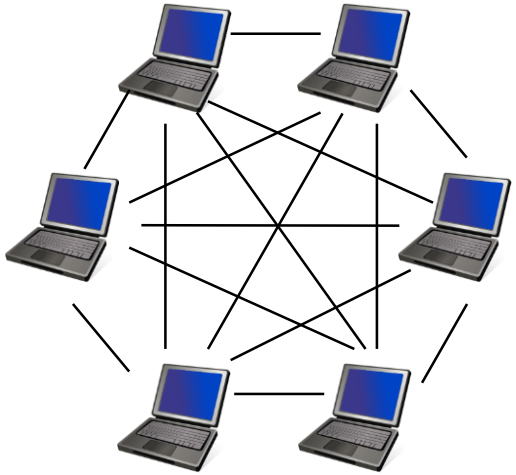


Partnernetz vs. Klient/Dienstgeber

Klient/Dienstgeber



Partnernetz



Partnernetz – Eigenschaften (1)

- Rollensymmetrie:
  - Jeder Partner ist sowohl Klient als auch Dienstgeber
- Dezentralisierung:
  - Es gibt keine zentrale Koordination und keine zentrale Datenbasis. Jeder Partner kennt i.d.R. nur eine Untermenge der übrigen Partner (seine „Nachbarschaft“)
- Selbstorganisation:
  - Das Gesamtverhalten des Systems setzt sich aus der Interaktion zwischen dein einzelnen Partnern zusammen

Partnernetz – Eigenschaften (2)

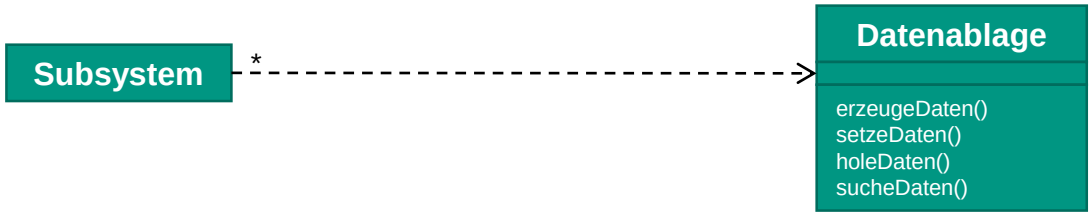
- Autonomie:
  - Partner treffen Entscheidungen autonom und verhalten sich autonom
- Zuverlässigkeit:
  - Partner sind unzuverlässig (z.B. nicht immer angeschaltet). D.h. es müssen Mechanismen gefunden werden, welche diese Unzuverlässigkeit kompensieren
- Verfügbarkeit:
  - Alle im System gespeicherten Daten müssen redundant verfügbar sein, wegen Unzuverlässigkeit (mancher) Partner

Partnernetz – Beispiel

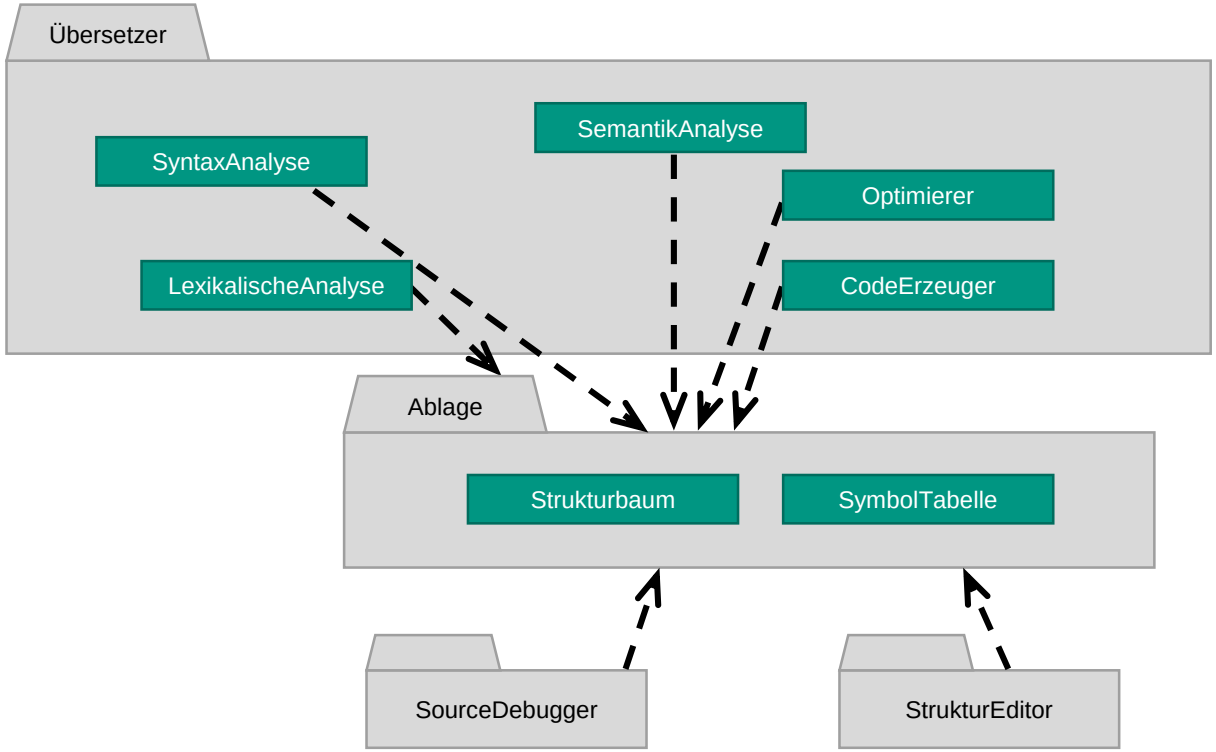
- Auf Dienstebene: Austausch von Dateien in einem Partnernetz (bspw. Bittorrent):
  - Ein Partner ist sowohl Klient als auch Dienstgeber:
    - Er kann Dateien von anderen Partnern anfordern (→ Klient)
    - Er kann anderen Partnern Dateien (und weitere Dienste) anbieten (→ Dienstgeber)
    - Wenn eine Anfrage nicht befriedigt werden kann, wird sie an einen anderen Partner weitergereicht.
- Auf Netzwerkebene: TCP/IP, DNS:
  - Anfragen werden anwendungsunabhängig über das physikalische Netzwerk verteilt.

Datenablage, Depot (engl. *Repository*)

- Subsysteme modifizieren Daten von einer zentralen Datenstruktur, der Datenablage.
- Subsysteme sind lose gekoppelt und interagieren nur über die Datenablage.
- Subsysteme können parallel oder hintereinander auf die Datenablage zugreifen. In parallelem Fall ist Synchronisation nötig (siehe Datenbanken).
- Kann mit lokalem oder Fernzugriff verwirklicht werden.



Datenablage: Beispiel (1)



Datenablage: Beispiel (2)

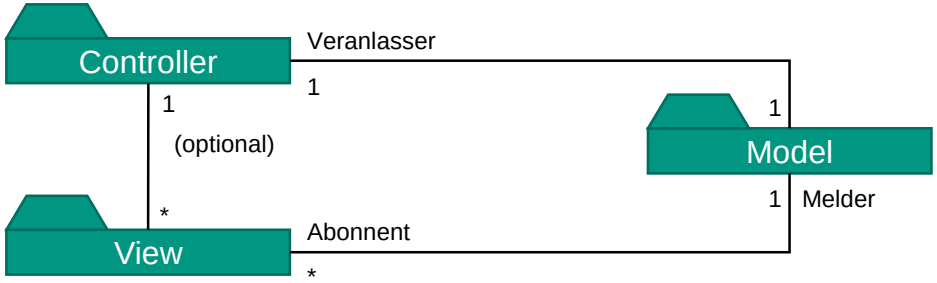
- Erklärung:
  - Die drei Werkzeuge werden vom Benutzer in beliebiger Reihenfolge, auch parallel ausgeführt.
  - Die Datenablage stellt sicher, dass gleichzeitige Zugriffe auf gemeinsame Daten geordnet werden.
    Transaktionskonzept: bei gleichzeitigen Zugriffen sieht es immer so aus, als wären sie in einer sequenziellen Reihenfolge jeweils vollständig abgewickelt worden.
  - Der Zustand der Datenablage kann Einfluss auf die Ausführung der verschiedenen Werkzeuge haben.
    - Z.B. kann das Bearbeiten von Dateien mit dem Editor während eines Übersetzvorgangs verboten werden.
- Weiteres Beispiel: Subversion, Git

Model/View/Controller (MVC)

- **Problem:** Angenommen, ein geg. System hat eine hohe Kopplung, so dass Änderungen an der Benutzerschnittstelle zu Änderungen an den Daten-Objekten führen.
  - Die Benutzerschnittstelle kann nicht neu implementiert werden, ohne die Darstellung der Daten-Objekte zu ändern.
  - Die Daten-Objekte können nicht neu organisiert werden, ohne die Benutzerschnittstelle zu ändern.
- **Lösung:** Der Architekturstil „Model-View-Controller“, welcher Daten von deren Darstellung trennt.
  - Das Subsystem zur Datenspeicherung heißt (Daten-)Modell. Es enthält neben den Daten oftmals auch die Anwendungslogik.
  - Das Subsystem zur Darstellung der Daten wird View (Präsentation, Sicht) genannt.
  - Das Subsystem für die Entgegennahme der Benutzereingaben und Steuerung der Interaktion zwischen Modell und Präsentation wird Controller (Steuerung) genannt. Die Steuerung kann zusätzlich einen Teil der Anwendungslogik enthalten.

Modell-Präsentation-Steuerung (engl. Model-View-Controller, MVC)

- **Modell (Model)**: Verantwortlich für anwendungsspezifische Daten.
- **Präsentation (View)**: Verantwortlich für die Darstellung der Objekte der Anwendung.
- **Steuerung (Controller)**: Verantwortlich für Benutzerinteraktion; aktualisiert das Modell; stößt ferner das Melden von Änderungen der Modelldaten an die Präsentation(en) an.



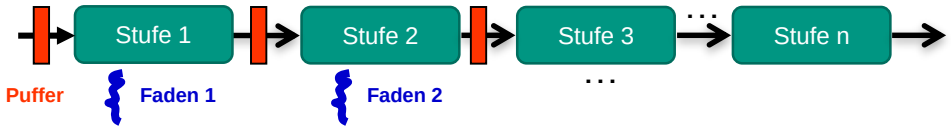
Mehr zu MVC

- Die Bekanntgabe von Änderungen im Datenmodell geschieht über das Entwurfsmuster „Beobachter“. Das Modell informiert dabei die Präsentation, dass sich relevante Daten geändert haben, die sich die Präsentation holt und anzeigt (mehr dazu siehe Kap. Entwurfsmuster). Dieser Aktualisierungsvorgang wird von der Steuerung veranlasst.
- In der Regel wird die Präsentation mit den Schaltflächen der Steuerung in einer einzigen graphischen Anzeige zusammengefasst. Es kann auch ein zusätzliche Interaktion zwischen Präsentation und Steuerung geben (muss es aber nicht).
- Mit MVC kann das Modell wiederverwendet und mit einer anderen Präsentationsimplementierung, z.B. für Mac oder Linux, versehen werden. Die Präsentation kann ebenfalls wiederverwendet werden, wenn eine andere Anwendung ähnlich Daten anzeigen möchte.

Fließband (engl. *Pipeline*, *Pipe and Filter*)

- Jede Stufe oder Filter ist ein eigenständig ablaufender Prozess oder Faden, mit eigenem Befehlszähler
- Daten fließen durch das Fließband, wobei jede Stufe von der vorigen Daten empfängt, sie verarbeitet, und an die nächste weiterreicht.
- Aufeinanderfolgende Stufen sind mit einem größenbeschränkten Puffer verbunden, um Geschwindigkeitsschwankungen auszugleichen.

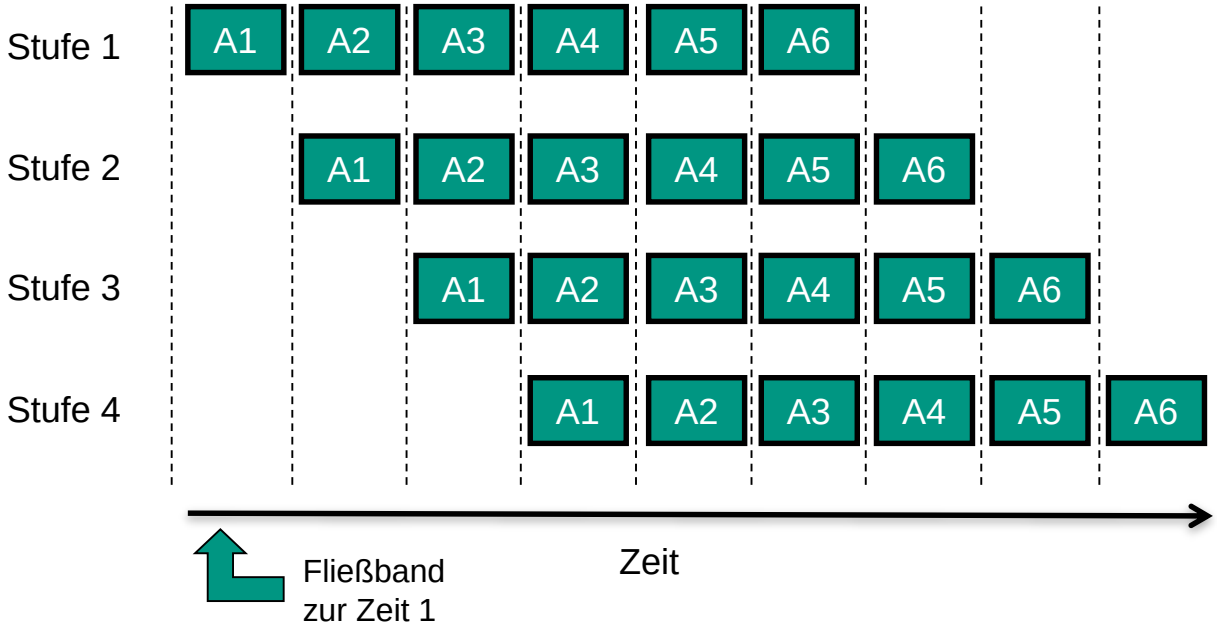
Fließband



- Bei Parallelrechnern können die einzelnen Stufen echt parallel ablaufen und damit zu einer Beschleunigung führen.
- Bei sequentiellen Rechnern werden die einzelnen Prozesse abwechselnd ausgeführt.

Fließband

- Verarbeitungsprinzip



Beispiele für Fließband

- Ein Beispiel für diesen Architekturstil ist die Unix-Shell.

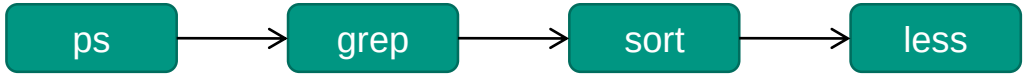
```
ps auww | grep hans | sort | less
```

Zeigt alle laufenden Prozesse an

Filtert Zeilen mit „hans“ heraus

Sortiert Eingabe

Erlaubt Blättern in der Ausgabe



Fließband: Anwendbarkeit

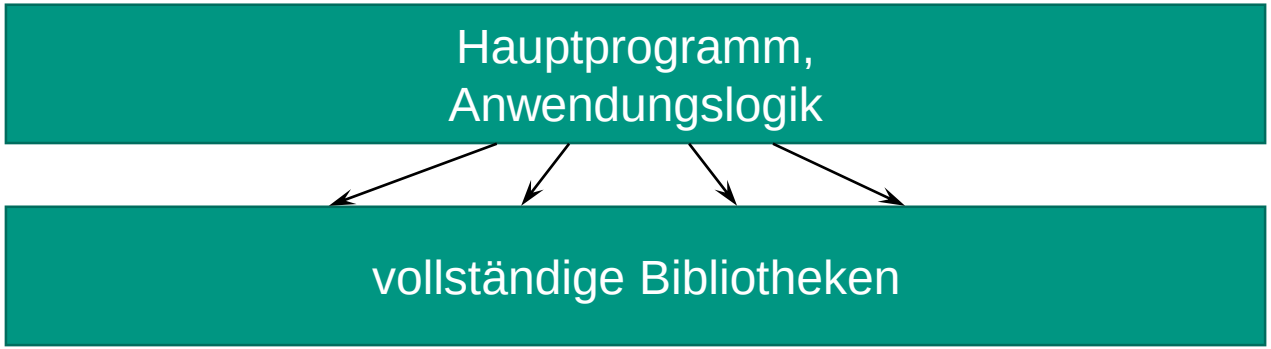
- Gut geeignet für Verarbeiten von Datenströmen, z.B. für Videocodierung und -decodierung, Videobearbeitung, Übersetzer, Stapelverarbeitung.
- Für gute Leistung auf Parallelrechnern sollten die einzelnen Stufen etwa gleich schnell laufen (unerheblich für Einzelprozessoren)

Rahmenarchitektur (engl. *Framework*)

- Bietet ein (nahezu) vollständiges Programm, das durch Einfüllen geplanter „Lücken” oder Erweiterungspunkten erweitert werden kann
- Es enthält die vollständige Anwendungslogik, meistens sogar ein komplettes Hauptprogramm
- Von einigen der Klassen in dem Programm können Benutzer Unterklassen bilden und dabei Methoden überschreiben oder vordefinierte abstrakte Methoden implementieren
  - Das Rahmenprogramm sieht vor, dass die vom Benutzer gelieferten Erweiterungen richtig aufgerufen werden.
  - Die Erweiterungen werden auch Einschübe (engl. Plug-ins) bezeichnet.

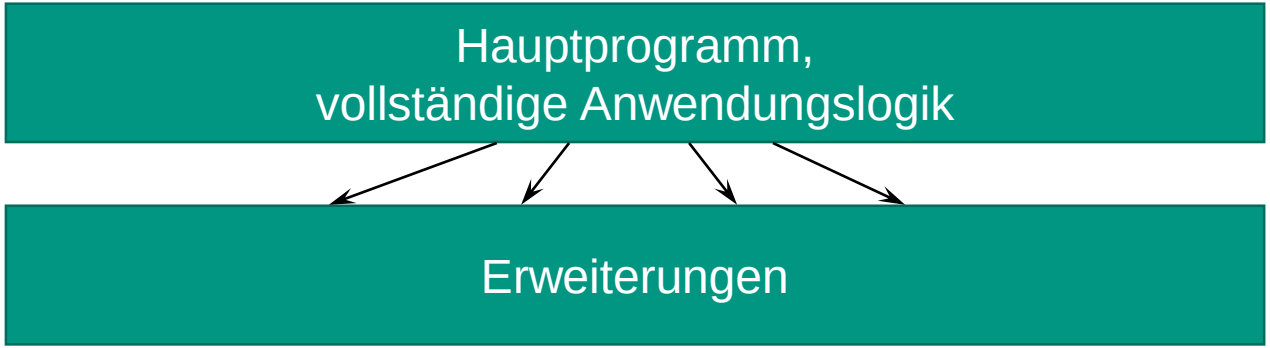
Rahmenarchitektur: Struktur (1)

- Herkömmliche Systemstruktur:
  - Hersteller liefert Bibliotheken
  - Benutzer schreibt Hauptprogramm und Anwendungslogik



Rahmenarchitektur: Struktur (2)

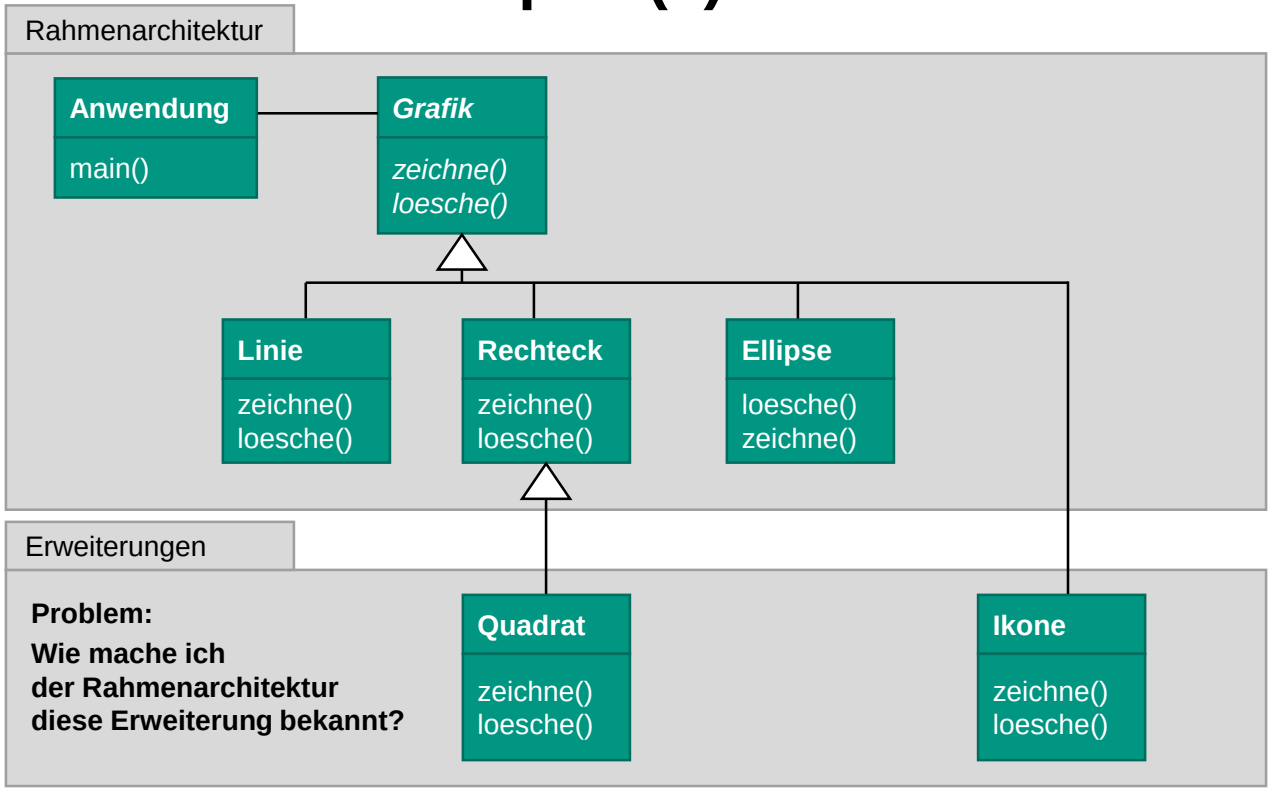
- Rahmenarchitektur:
  - Ein Rahmenprogramm befolgt das „Hollywood-Prinzip“: „Don’t call us – we call you“.
  - Das Hauptprogramm besteht bereits und ruft die Erweiterungen der Benutzer auf.



Rahmenarchitektur: Beispiel (1)

- Ein Zeichen-Rahmenprogramm sieht eine Klasse Grafik mit einigen Unterklassen (z.B. Linie, Rechteck, Ellipse, usw.) vor.
- Der Benutzer darf neue Unterklassen von Grafik und seinen Unterklassen bilden, z.B. Quadrat oder Ikone, muss dazu aber eine Methode `zeichne()` bereitstellen.
- Das Rahmenprogramm sorgt dann dafür, dass Objekte der neuen Klassen richtig erzeugt, auf dem Zeichenbrett positioniert, verschoben, gesichert, usw. werden können.

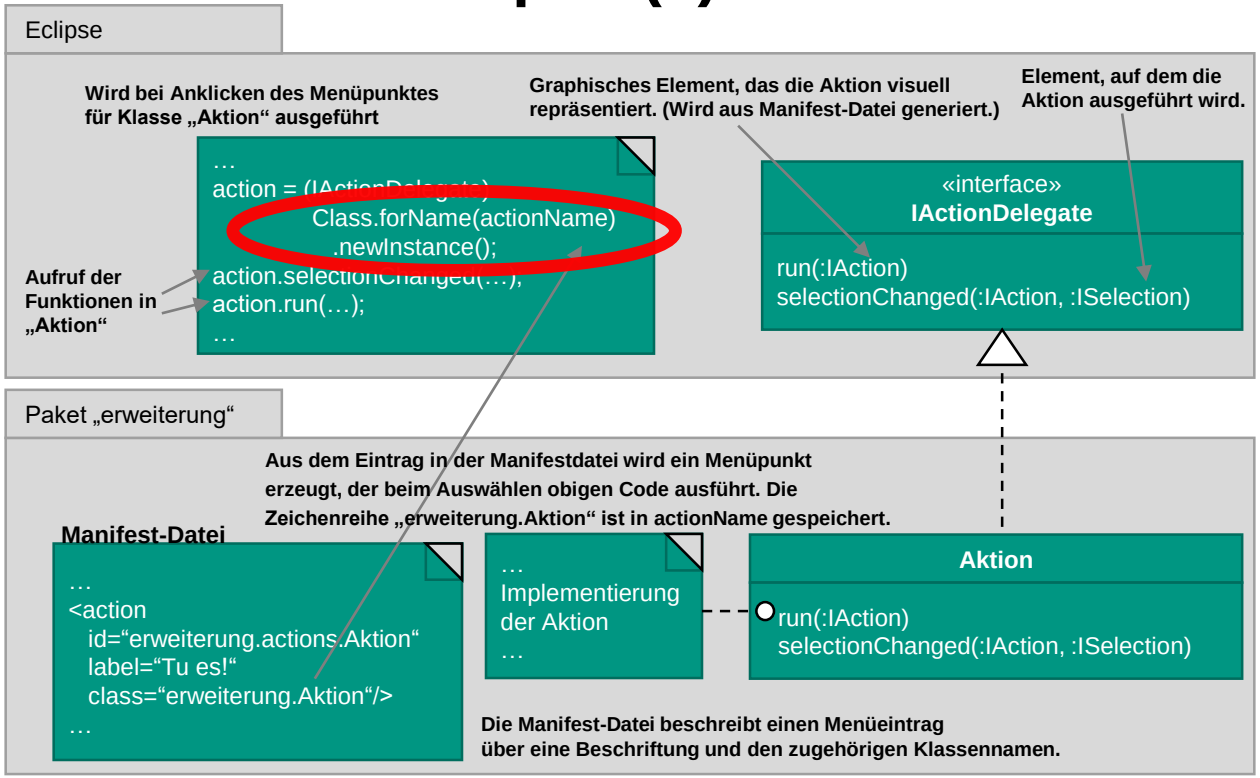
Rahmenarchitektur: Beispiel (2)



Rahmenarchitektur: Beispiel (3)

- **Eclipse:** Quelloffene Entwicklungsplattform
- Besteht aus einem relativ kleinen Kern mit einer großen Zahl an Erweiterungspunkten für Einschübe (Plug-Ins), die wiederum weitere Erweiterungspunkte bieten können.
- Große Teile der Standard-Distribution von Eclipse bestehen aus Einschüben.
- Einschübe werden als Jar-Dateien gespeichert, deren Manifest-Datei vom Eclipse-Kern gelesen wird, um die Bezeichner der Erweiterungsklassen zu erhalten.

Rahmenarchitektur: Beispiel (4)



Rahmenarchitektur: Beispiel (5)

- Die Manifest-Datei enthält die Namen der Erweiterungsklassen, Bezeichnung der gewünschten Menüeinträge, und weitere Einzelheiten.
- Elemente wie Menüeinträge und Knöpfe in der Entwicklungsumgebung werden daraus generiert.
  - Daher ist die Darstellung dieser Elemente möglich ohne die Klassen sofort laden zu müssen. (→ Vermeidet lange Wartezeiten beim Start)
- Wird eine neue Methode über einen Menüeintrag aufgerufen, dann lädt Eclipse die Klasse über `Class.forName(…)`, initialisiert sie und ruft die in der Schnittstelle definierte Methode auf.
  - Bei der Initialisierung (`selectionChanged`) wird der Kontext des Methodenaufrufs übergeben. (Editor, Selektion, Projekt, …)

Rahmenarchitektur: Anwendbarkeit

- Wenn eine Grundversion der Anwendung schon funktionsfähig sein soll
- Wenn Erweiterungen möglich sein sollen, die sich konsistent verhalten
  - Anwendungslogik im Rahmenprogramm
- Wenn komplexe Anwendungslogik nicht neu programmiert werden soll
- Die Entwurfsmuster Strategie, Fabrikmethode, abstrakte Fabrik und Schablonenmethode werden häufig in Rahmenarchitekturen benötigt
  - siehe nächster Abschnitt

Dienstorientierte Architekturen (engl. Service Oriented Architecture, SOA)

- SOA ist ein Architekturstil, bei dem Anwendungen aus (unabhängigen) Diensten (engl. services) zusammengestellt werden
- SOA ist ein abstraktes Konzept einer Softwarearchitektur

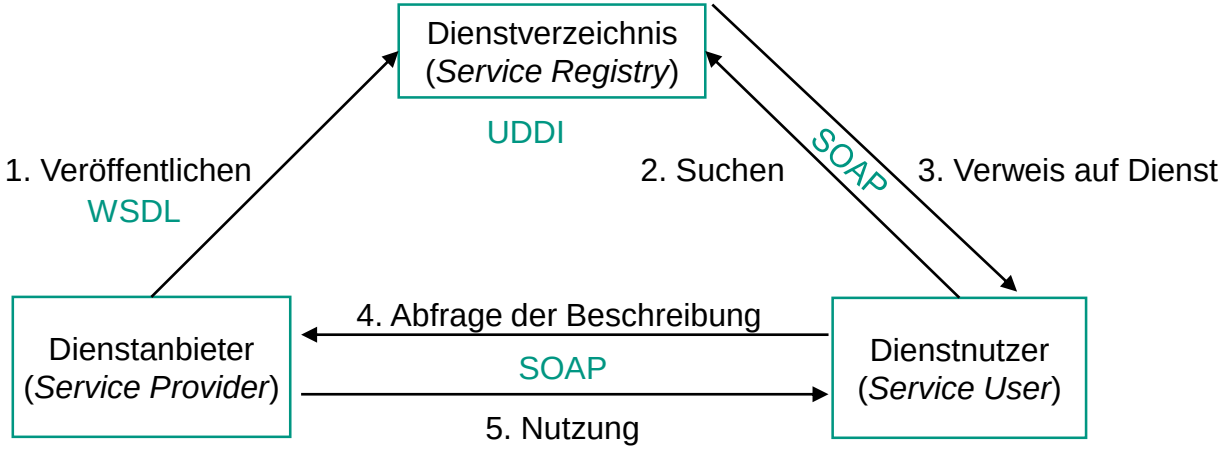
- Dienste werden als zentrale Elemente eines Unternehmens betrachtet (Stichwort: Dienstleistungen)
  - Bereitstellen gekapselter Funktionalität an andere Dienste und Anwendungen
  - Erlaubt Kapseln von Alt-Systemen und (späteres) Austauschen
  - Dienstbeschreibung (Schnittstelle) liegt in standardisierter Form vor
  - Sprach- und technologieunabhängig

Dienste als Ergebnis einer evolutionären Entwicklung der Informationstechnologie

- Seit Mitte des 20. Jahrhunderts ist der Trend hin zur Zerlegung und Verteilung in der Informationstechnologie klar erkennbar
- Programme entwickelten sich von wenigen großen, inhärent komplexen Software-Bestandteilen zu zahlreichen kleinen, überschaubaren Codestücken
- Die Akzeptanz der serviceorientierten Architekturen kann als ein mit dem Internet vergleichbarer Entwicklungsschritt auf der Ebene der Software angesehen werden
- Der SOA liegt die Idee eines Marktplatzes zugrunde
- In einer SOA sind die Dienste (d.h. Software-Komponenten) lose gekoppelt

Dienstmodell als der Kern einer SOA

- Benötigte Funktionalität kann in Form eines Dienstes zur Laufzeit eingebunden werden
- Die Basis für die dynamische Bindung ist das Servicemodell



Merkmale und Ziele der Dienstorientierung

- Lose Kopplung
  - Ziel ist das einfache Herauslösen und Ersetzen eines Dienstes zur Laufzeit
  - Dynamisches Binden wird durch das Dienstverzeichnis als zentralem Bestandteil des Dienstmodells ermöglicht
- Unterstützung von Geschäftsprozessen
  - Dienste kapseln geschäftsrelevante Funktionalität
  - (Komplexe) Anwendung = Komposition von Diensten
- Verwendung von (offenen) Standards
  - Programmiersprachen- und Plattform-unabhängige Bereitstellung der Dienste und Dienstkompositionen

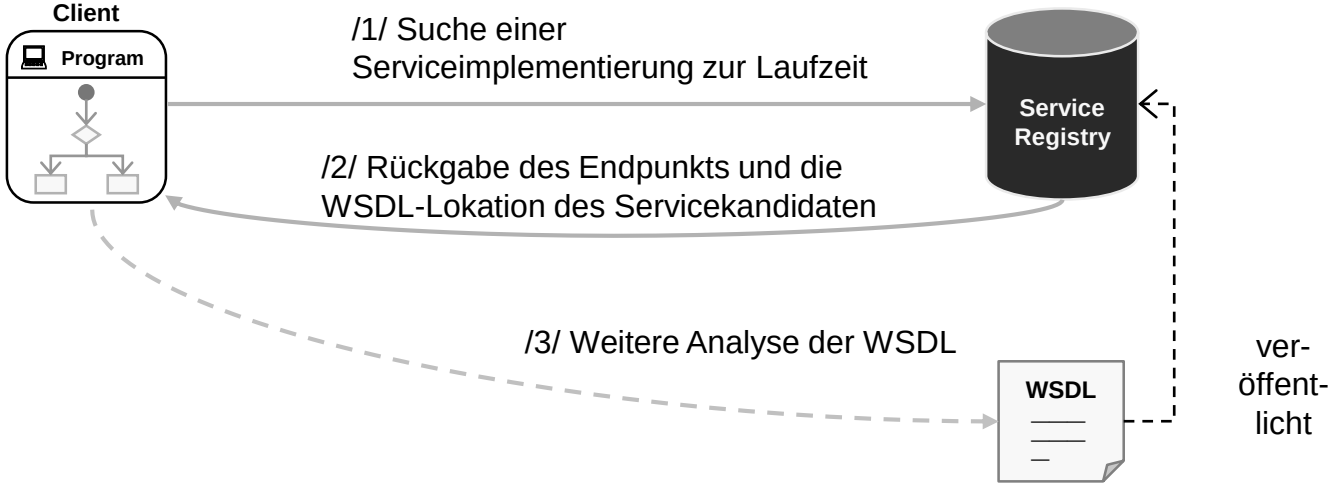
Universal Description, Discovery and Integration (UDDI)

- Durch UDDI wird die Registrierung und das Auffinden von Services unterstützt
  - Industrieübergreifende Initiative zur Erzeugung eines Verzeichnis-Standards für Web-Services
- Anforderungen an ein fortschrittliches Serviceverzeichnis
  - Maximierung der Web-Service-Wiederverwendung
  - Erzeugung einer Management- und Governance-Struktur
  - Speicherung aller Metadaten zu einem Web-Service und den assoziierten Dokumenten
    - Dokument-basiert oder Metadaten-basiert
  - Bereitstellung von flexiblen Verwaltungs- und Zugriffsschnittstellen
  - Anpassung an sich ändernde Geschäftsanforderungen und eine wachsende Anzahl an Services und Nutzern

Von UDDI bereitgestellte Funktionen und Eigenschaften

- Funktionen
  - Servicebereitsteller können ihre Services beschreiben (Publishing)
  - Servicenutzer können Informationen über Unternehmen, den von diesen angebotenen Web-Services und technische Informationen entdecken (Discovery)
- Eigenschaften
  - Bereitstellung von Netzadressen zu Ressourcen
  - XML-Dokument zur Beschreibung einer Geschäftsentität und deren Web-Services
    - Weiße Seiten: WER
    - Gelbe Seiten: WAS
    - Grüne Seiten: WO und WIE
  - Technologieunabhängig

Dynamischer Serviceaufruf



- Dynamisches Auffinden von Services aufgrund einer benötigten Funktionalität ist momentan in der Praxis nicht üblich
- Gemäß dem Stand der Technik lassen sich dynamisch das Zugriffsprotokoll oder der Endpunkt festlegen

Dienstorientierte Architekturen
Umgesetzt mit Web-Services