Integrationsbegriff

Entwicklung eines Integrationsbegriffs

- Integration impliziert Kommunikationsfähigkeit
- Kommunikationsfähigkeit impliziert gemeinsame Sprache bzw. gemeinsame Begriffe/Konzepte
- Konsequenz: die Integration von Teilsystemen impliziert ein System gemeinsamer Konzepte → semantisches Referenzsystem (z.B. elementare Datentypen, Klassen, spezielle Datenstrukturen)
- Komplexität empfiehlt Differenzierung in Integrationsdimensionen

Niveau der Integration: Semantik

- Die dimensionsspezifische, konzeptionelle Integration zweier Komponenten nimmt tendenziell mit der Semantik der Konzepte des jeweiligen semantischen Referenzsystems zu.
- verwendeter Semantikbegriff: Informationsgehalt → umso größer, je mehr Interpretationen ausgeschlossen sind

Statische Integration

- zu integrierende Komponenten, die Daten/Objekte aufnehmen und ausgeben können
  - Funktionen/Prozeduren/Objekte/Programme/Anwendungen/Komponenten
- Integration durch einheitlich festgelegte Semantik gemeinsam verwendeter Daten

Interoperabilität

- interoperable Komponenten können Nachrichten austauschen
- Voraussetzungen
  - geeignete Kommunikationsinfrastruktur (z.B. Sockets, Pipes, RPC …)
  - Identifikation der beteiligten Komponenten durch gemeinsame Namensräume
  - einheitliche Bezeichnung der Konzepte im gemeinsamen semantischen Referenzsystem

Funktionale Integration

- zwei Komponenten sind funktional integriert, wenn sie gemeinsame Funktionen sinnvoll nutzen können
- Voraussetzungen
  - statische Integration und Interoperabilität
  - Erweiterung des semantischen Referenzsystems um funktionale Aspekte (Funktionen, …)
- semantisches Referenzsystem für Schnittstellen oder für Implementierungen möglich

Dynamische Integration oder zielgerichtete Kooperation

- bezeichnet die Einbettung von Komponenten in Prozesse
- die zu integrierenden Komponenten sollen innerhalb eines Prozesses bestimmte Aufgaben bzw. Funktionen wahrnehmen
- Voraussetzungen
  - funktionale Integration
  - Erweiterung des semantischen Referenzsystems um dynamische Aspekte (Ereignisse, Ausführungsregeln, evtl. zeitliche Beziehungen)
- Ausführungsregeln entsprechen Regeln zur Koordination von Handlungssystemen
  - Option direkte Kopplung: jede Anwendung kennt relevante Teile des Kooperationsschemas und sorgt für die Fortsetzung
  - Option Koordination über Steuerungseinheit/Controller: jede Anwendung erzeugt Ereignisse nach Maßgabe eines Ereignisschemas, Controller setzt Prozess fort

Integrationsdimensionen

	Semant. Referenzsystem	Beispielkonzepte	Integrationstechnologien
statische Integration	Datentypen Datenmodell → Datenbankschema	Datentyp „Integer“ Entitätstyp „Kunde“	DBMS Data Warehouse
funktionale Integration	Funktionenmodell → Funktionsbibliothek	Sortierfunktion Funktion zur Berechnung der MwSt.	Funktionsbibliotheken Web Services
OO-Integration	Objektmodell → Klassenbibliothek, Schema von OODBMS	Klasse „Window“ Klasse „Rechnung“ Klasse „Kunde“	CORBA J2EE
dynamische Integration	Ereignismodell → Verzeichnis von Ereignis-klassen und ggfs. Aktionen	„Auftrag eingegangen“ „Prüfung abgeschlossen“	WFMS

Integration auf Instanzenebene

- setzt konzeptionelle Integration voraus
- ergänzend zu gemeinsamen Konzepten: gemeinsame Instanzen
  - gemeinsamer Namensraum für Instanzen
  - gemeinsame Instanzen für alle integrierten Komponenten zugreifbar
- Beispiel: statische Integration von Daten mittels gemeinsamer Datenbank

Gründe für Einschränkungen der Integration auf Instanzenebene

- unterschiedliche interne Repräsentation (anwendungsspezifische Schemata): unzureichende konzeptionelle Integration
- Bandbreitenrestriktion (z.B. bei Mobilanwendungen Kopien statt immer Verbindung zur DB)
- eingeschränkte Verfügbarkeit von Kommunikationskanälen
- Datenschutz
- Konsequenz: man kann nicht von der Verteilung abstrahieren, sondern muss bei der Implementierung darauf achten, wo welche Daten abgelegt werden

Integration der Phasen des Software-Lebenszyklus

- Ziel: Überwindung von Friktionen zwischen den Phasen
- gemeinsames semantisches Referenzsystem: Konzepte, die in allen Phasen verwendet bzw. referenziert werden
- dabei u.U. allerdings variierender Detaillierungs- und Formalisierungsgrad

Organisatorische Integration

- Integration des Handlungssystems einer Organisation mit dem Informationssystem
- auch „IT-Business-Alignment“
- erfolgt ebenfalls sprachlich: je mehr Begriffe, auf denen ein Informationssystem beruht, mit denen korrespondieren, die die Diskurswelt des Handlungssystems bestimmen, desto höher ist das Integrationsniveau

Bewertung von Integration

- grundsätzlich positiv belegt: integrierte IS im Zweifel besser bewertet als nicht integrierte
- aber: in der Praxis sind warnende Stimmen zu verzeichnen
- Integration bedeutet Aufwand und Risiko, auch Wartung kann erschwert werden

Etablierung semantischer Referenzsysteme

- bilaterale/multilaterale Konventionen, Industriestandards, Standards/Normen
- jeweils verschiedene Abstraktionsgrade und Anpassungsaufwände

Wiederverwendbarkeit

- von Komponenten/Objekten, Datenstrukturen, Funktionen, Modellen, Architekturen
- aber auch: von Entwurfs- und Implementierungswissen, Domänenwissen
- impliziert i.d.R. Integrierbarkeit
- Wiederverwendungskomfort
  - nimmt tendenziell mit der Semantik der angebotenen Artefakte zu
  - Voraussetzung: Artefakte entsprechen den Anforderungen des Verwenders
- Wiederverwendungsreichweite
  - nimmt tendenziell mit der Semantik der angebotenen Artefakte ab

Architekturen betrieblicher Informationssysteme

Begriff des Informationssystems

- System, das der Erfassung, Aufbewahrung, Verarbeitung und Bereitstellung von Informationen und Wissen dienst, die für zweckgerichtetes Handeln in einem Unternehmen von Bedeutung sind
- Fokus auf rechnergestützte Informationssysteme

Architektur eines betrieblichen Informationssystems

- eine Architektur ist eine Abstraktion eines Informationssystems, also ein Modell, das den Aufbau des Systems aus Funktionseinheiten sowie deren Zusammenwirken darstellt
- im Unterschied zu konzeptuellen Modellen wird dabei von fachlichen Besonderheiten der Anwendung abstrahiert!
- ein System kann durch verschiedene Architekturen dargestellt werden

Informationssysteme: Allgemeine Gestaltungsziele

- Effektivität/Integrität/Integration/Ergonomische Eignung/Transparenz/Wartbarkeit
- Offenheit/Robustheit/Verlässlichkeit/Sicherheit/Wirtschaftlichkeit

IS-Architekturen: Allgemeine Entwurfsprinzipien

- Abstraktion
  - von spezifischen Eigenschaften solcher Systemteile, die Veränderungen unterworfen sind
  - von spezifischen Eigenschaften solcher Systemteile, die in der Zuständigkeit anderer liegen
- Modularisierung
  - Unterteilung des Gesamtsystems in unabhängig voneinander zu pflegenden Einheiten → „separation of concerns“
  - auf verschiedenen Granularitätsstufen durchzuführen

IS-Architekturen: Angemessenheit und Anschaulichkeit

- Gestaltung der Architektur eines Informationssystems erfordert Ziele
- Architektur sollte auf die Erfüllung dieser Ziele gerichtet sein (Angemessenheit)
- Anschaulichkeit
  - die Visualisierung einer Architektur sollte besonders auf solche Merkmale abstrahieren, die in einem engen Zusammenhang mit jeweils betrachteten Zielen stehen

- der Grad der Detaillierung bzw. Vereinfachung hängt von Fähigkeiten und Interessen der Betrachter ab → möglicherweise verschiedene Sichten notwendig

IS-Architekturen: Bewertung

- Angemessenheit, Anschaulichkeit
- Authentizität
  - Beschreibt die Architektur tatsächlich wesentliche Systemmerkmale?
  - Welche Intention haben die Autoren?
  - Ist die Abstraktion (Vereinfachung) für das Verständnis förderlich oder hinderlich?
- Aufgabe ist i.d.R. die Kommunikation: Klärung von Mehrdeutigkeiten und von kritischen Fragen

Blickwinkel des Managements

- Verdeutlichung genereller Systemmerkmale
- organisatorische Integration
  - Unterstützung betrieblicher Funktionen, von GPs und von Entscheidungsszenarien
- Gestaltung unternehmensübergreifender Prozesse
  - Anbahnung von Transaktionen, Durchführung (Logistik), Abwicklung von Zahlungsströmen

Unternehmensarchitektur

- seit einiger Zeit in der (Beratungs-)Praxis verwendeter Begriff
- betont eine integrative Sicht auf Unternehmensorganisation und IS
- Architektur im Großen, keine Detaillierung
- soll dazu beitragen, die Funktion des IS besser verstehen und bewerten zu können
- allerdings: keineswegs einheitlich verwendet, Vielfalt ganz unterschiedlicher Abstraktionen

Unternehmensarchitektur vs. Unternehmensmodell

- ähnliches Anliegen: Abstraktion über Handlungssystem und IS
- allerdings unterschiedliche Foki
  - Unternehmensarchitektur betont hohes Aggregationsniveau, auf Management gerichtet; korrespondierende Modellierungssprachen zumeist nicht sehr differenziert
  - Unternehmensmodellierung bietet i.d.R. verschiedene Detaillierungsgrade und wendet sich an verschiedene Zielgruppen; beinhaltet (mitunter) elaborierte Modellierungssprachen

Zusammenhang: konzeptuelle Modelle und Architekturen

- grundsätzlich unterschiedliche Abstraktionen
- Fachkonzepte der konzeptuellen Modelle („Anwendungslogik“) mitunter auf verschiedene Anwendungen verteilt (Abstraktion auf Fachkonzepte)
- Beschreibung von Anwendungen in einer Architektur abstrahiert von Fachkonzepten
- Zusammenhang dann gut darstellbar, wenn konzeptuelle Modelle auf (zentrale) Schemata abgebildet werden
  - Datenmodell auf DBMS-Schema, GP-Modell auf Workflow-Schema

Evolution von Architekturen

- erste Rekonstruktion: Fokus auf Integration und Wiederverwendung; keine dedizierte Unterstützung verteilter Systeme
  - Software-Entwickler können nicht von Verteilung abstrahieren
- zweite Rekonstruktion: Fokus auf Verteilung
  - Ziel: weitgehende Abstraktion von Verteilung ermöglichen

Prototypische Evolutionsstufen: Fokus auf Integration und Wiederverwendung

- Stufe 1
  - Isolierte Anwendungen bilden *einzelne Aufgabenbereiche* des Unternehmens ab. Sie nutzen jeweils die Dienste eines Betriebssystems. → keine/kaum Integrationsansätze
- Stufe 2
  - Anwendungen werden vor dem Hintergrund von *partiellen Datenmodellen* entworfen. Anwendungen nutzen ein DBMS.
- Stufe 3
  - Anwendungen basieren auf einem *unternehmensweiten Datenmodell* und nutzen ein DBMS für die Datenverwaltung. → statische Integration durch einheitliches DB-Schema, höhere Wiederverwendung der Datenstrukturen und entsprechenden Funktionen, Wartbarkeit hängt von der Qualität der Abstraktionen ab
- Stufe 4
  - *unternehmensweites Datenmodell* sowie *unternehmensweites Funktionsmodell*
  - Anwendungen unterteilen sich ebenfalls in Datenmodell und Funktionsmodell, die jeweils *Projektionen auf die unternehmensweiten Modelle* darstellen
  - Integration gefördert durch DBMS mit einheitlichem Datenmodell (typischerweise: relationales Modell), zugehörigem CASE-Werkzeugen sowie generelle Benutzerinteraktionsmodelle, die in *GUI-Bibliotheken* verfügbar sind
  - SAP geht in diese Richtung, Vorteil: kommt aus einem Haus
  - Herausforderung: unterschiedliche Anbieter, unterschiedliche Entwicklerteams
- Stufe 5: objektorientierte Informationssysteme
  - in einem zentralen *Objekt-Managementsystem* (OMS) werden ein *unternehmensweites Objektmodell und Vorgangsmodelle* sowie die zugehörigen Instanzen verwaltet
  - großes Netz interagierender Objekte, es gibt keine Anwendungen mehr, nur noch Nutzungsszenarien innerhalb von Interaktionskontexten
- Stufe 6: Self-Referential Enterprise System (SRES)
  - in die Unternehmenssoftware sind nicht nur konzeptuelle Systemmodelle, sondern auch Modelle des Handlungssystems integriert
  - das *Informationssystem* enthält also ein *Unternehmensmodell*, dass mit den Instanzdaten integriert ist
  - zwischen Modell- und Instanzenebene kann hin- und hernavigiert werden
  - konzeptuelle Modelle nicht nur als Dokumentation oder Vorstufe der Entwicklung, sondern als wichtigen Teil der Implementierung bzw. der Laufzeit

Zusammenfassende Betrachtung

- Welche Auswirkungen haben die Architekturen jeweils auf das Integrationsniveau des IS? je mehr eine Architektur Gemeinsamkeiten zwischen den einzelnen Teilen eines IS betont, desto größer ist ihr Beitrag für die Förderung des Integrationsniveaus
- Wie ist der Umfang der Wiederverwendung jeweils einzuschätzen? je größer die Gemeinsamkeiten, desto größer die Chance Wiederverwendung zu realisieren
- Wie ist der Aufwand für Erstellung und Pflege jeweils zu beurteilen? bei Stufe 1 geringer Aufwand zur Erstellung, aber Pflege aufwändig, bei Stufe 6 andersrum (bei langfristig stabilen Gemeinsamkeiten)

Fokus auf Verteilung

- Stufe 1: Mainframe
  - Anwendungen laufen zentral, verteilte Nutzung über Terminals
  - Problem: Mainframe-OS ursprünglich für Stapelbetrieb entwickelt, deshalb keine Transaktionssicherung bei großer Zahl von Dialogen
- Stufe 2: Mainframe + Transaktionsmonitor
  - Transaktionsmonitor als Aufsatz des OS sorgt für die Verwaltung parallel genutzter Ressourcen (Anwendungen, Daten) und stellt die Durchführung von Transaktionen sicher
- Stufe 3: Microcomputer, verteilte Rechnersysteme
  - Zwei-Schichten-Architektur („2-Tier“): Server für die Datenverwaltung, „Fat Clients“ für Anwendungen und Präsentation
- Stufe 4: Trennung von Präsentation, Anwendungslogik und Datenverwaltung
  - Drei-Schichten-Architektur („3-Tier“): Server für die Datenverwaltung, Anwendungen als Server ausgelegt, Präsentation getrennt von Anwendungen, ergänzt um Transaktionsmonitor

Bewertung der verschiedenen Architekturen

Mainframe-Architektur	Zwei-Schichten-Architektur	Drei-Schichten-Architektur
▪ zentrale Verwaltung von Anwendungen und sonstigen Ressourcen – günstige Voraussetzung für Integration, Integrität, Wartung und Skalierbarkeit ▪ u.U. eingeschränkte Performanz auf Client-Seite ▪ hohe Kosten ▪ keine Trennung von Anwendungen und Präsentation	▪ Kostensenkung durch Skaleneffekte in der Produktion von Microcomputern ▪ attraktivere GUIs ▪ hoher Wartungsaufwand: Anwendungen verteilt installiert, Trennung von Anwendung und Präsentation nicht notwendig ▪ eingeschränkte Skalierbarkeit: begrenzte Ressourcen auf Client-Systemen, keine Unterstützung verteilter Transkationen	▪ Separierung von Anwendungslogik unterstützt Integrität, Integration, Wartung, Skalierbarkeit ▪ Herausforderung/Risiken durch: Komplexität (u.U. Ressourcen auf einer Vielzahl vernetzter Rechner) und Heterogenität (Plattformen, Sprachen)

Middleware: Wesentliche Funktionen

Fokus auf Verteilung: Kategorien von Informationssystemen

- Verteilung: Daten, Programmausführung auf verteilten Rechnern
- Heterogenität: unterschiedliche Plattformen und Implementierungssprachen
- Mehrbenutzerbetrieb: exklusive Nutzung ausgewählter Ressourcen, Integritätsgefährdung durch mehrfachen Zugriff

Ziele: Abstraktion von Verteilung und Heterogenität

- Benutzersicht
  - Ressourcen des verteilten Systems sind lokal verfügbar
  - tatsächlicher Speicher- und Ausführungsort ist transparent
  - gleichzeitige Benutzung gemeinsamer Ressourcen mit anderen Benutzern erfolgt transparent, ggfs. erfolgt ein Schutz von integritätsverletzenden Zugriffen
- Entwicklersicht
  - die Entwicklung von Systemen, die in verteilten Umgebungen eingesetzt werden, erfolgt weitgehend so wie die Entwicklung lokal eingesetzter Systeme

- die Entwicklung von Systemen, die in heterogener Umgebung eingesetzt werden, erfolgt weitgehend so wie in homogener Umgebung

Middleware

- Software-System, das Funktionen für verteilte Systeme bereitstellt, die i.d.R. OS für einzelne Rechner anbietet
- Abstraktion von Verteilung und Heterogenität
- transparenter Zugriff auf verteilte Ressourcen durch gemeinsamen Namensraum
- vereinfacht: Zwischenschicht zwischen Anwendungen und darunter liegenden Schichten (OS, DB)
- Trennung vom OS nicht immer eindeutig
- Vorsicht: Begriff durch Anbieter geprägt, erhebliche Funktionsunterschiede
- Abstraktion der Middleware muss stabiler sein als die darunterliegenden Schnittstellen (z.B. SQL)

Middleware: wirtschaftliche Motivation

- Befreiung der Anwendungsentwickler von Aufwand und Risiko, die mit der Bewältigung von Verteilung und Heterogenität verbunden sind
- Kostenvorteile durch Wiederverwendung
- durch weite Verbreitung und Standardisierung Beitrag zur überbetrieblichen Integration von IS
- ggfs. Beitrag zum Schutz von Investitionen (abhängig von der Pflege der Middleware)

Middleware: idealtypische Funktionen

- Überwindung von Heterogenität, Verteilungstransparenz, Vermittlung von Dienstgebern, Dienstaktivierung und -terminierung, Lastverteilung, Sicherheit, Persistenz, Transaktionsschutz

Kommunikation durch Middleware: Perspektiven auf das Thema

- Anforderungen
- Architektur: Typsystem/Objekt(meta)modell, Funktionseinheiten/Dienste, Datenstrukturen
- Funktionsweise
- Entwicklung verteilter Anwendungen: Vorgehensmodell, Werkzeuge
- Beispielsysteme

Ermittlung entfernter Objekte

- Verwaltung von einzelnen Objekten erfolgt mithilfe entsprechender Verwaltungsobjekte
- Adresse des Verwaltungsobjekts erhält man durch Namensdienst (Name Service), welcher Zugriff auf die White Pages (Implementation Repository: Zuordnung von logischen Servern zu konkreten Netzwerkadressen) hat
- Vermittlungsdienst (Trading Services) zur automatisierten Suche von geeigneten Komponenten, hat Zugriff auf Yellow Pages (Interface Repository: strukturierte Beschreibung von Diensten)
- Namens- und Vermittlungsdienst können auf einem Server zusammengefasst sein

Semantisches Referenzsystem

- gemeinsames Objektmodell (Klassenschema) zur Überwindung unterschiedlicher Typsysteme, in dem gemeinsame Interfaces definiert werden
- die Middleware liefert jedoch nicht spezielle Klassen wie Kunde oder Rechnung, da diese schon zu speziell wären um generisch eingesetzt zu werden
- sondern Verwendung einer Interface Definition Language (IDL): erlaubt die sprachunabhängige Spezifikation von Klassen (und Methoden, Attribute normalerweise nicht vorgesehen)

transparenter Zugriff: Abstraktion von Verteilung

- Falls das referenzierte Objekt außerhalb des eigenen Namensraums liegt, was wird dann zurückgegeben? Wie wird darauf zugegriffen?
- der Client kommuniziert lokal mit einem für diesen Zweck instanziierten, zustandslosen Stellvertreterobjekt (Stub) im lokalen Namensraum, das das entfernte Objekt repräsentiert
- der Stub kommuniziert mit einem Proxy auf Serverseite (Skeleton), welcher die Anfragen an das eigentliche Zielobjekt weiterleitet
- die Proxys sind in IDL spezifiziert

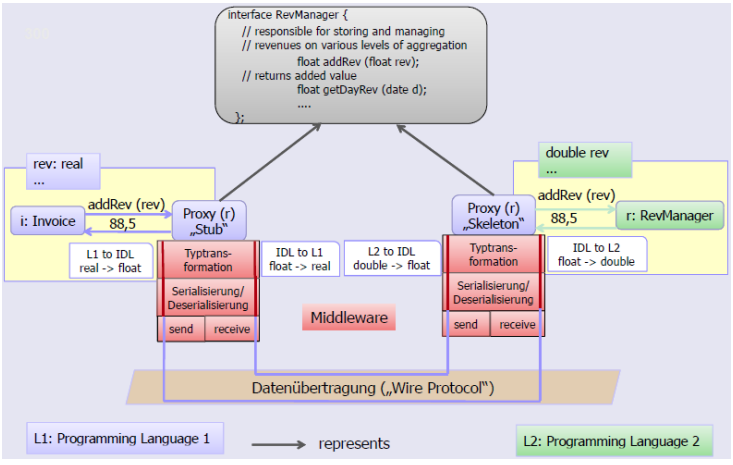
Aktivierung/Deaktivierung entfernter Objekte

- in verteiltem System häufig große Zahl möglicher Dienstgeber
- nicht sinnvoll, alle Dienstgeber ständig aktiv zu halten
- nicht aktive Dienstgeber können allerdings keine Aufrufe entgegennehmen
- deshalb: Aktivierungsdienst als Bestandteil des Objektlebenszyklus-Dienstes, der eine Menge von Dienstgebern nach außen repräsentiert und einen Dienstgeber bei Bedarf aktiviert
  - im Implementation Repository wird der Status der Objekte verwaltet (active, inactive)
- Terminierung erfordert geeignete Regelungen, z.B. nach einer definierten Zeit ohne Aufrufe
- Wahrung referentieller Integrität: wird das entfernte Objekt nicht mehr benötigt, wird das lokale Stub-Objekt freigegeben, daraufhin erhält der Objektlebenszyklus-Dienst die entsprechende Nachricht und schickt eine Freigabe an das Skeleton-Objekt, welches dann die Referenz auf das entfernte Objekt löscht, erst dann kann das entfernte Objekt ebenfalls freigegeben werden

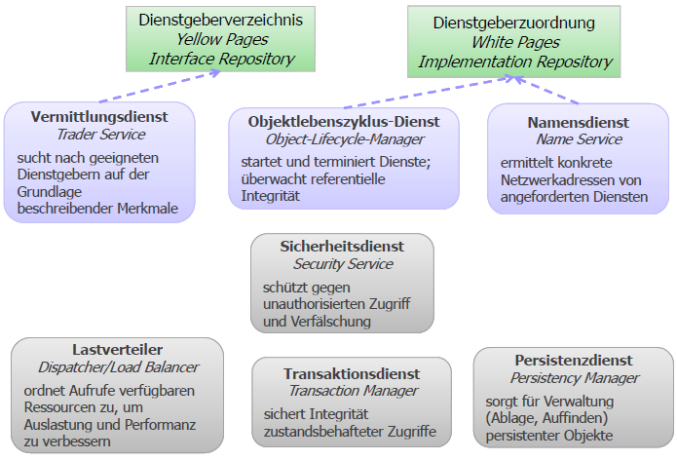
Funktionsweise

- Vorbereitung
  - Erstellung öffentlicher Objekt-Schnittstellen mit IDL
  - Eintrag in Implementation Repository und Interface Repository
- Sprachanbindung
  - Auswahl/Erstellung der benötigten öffentlichen Schnittstellen (Klassen)
  - Erstellung/Erzeugung korrespondierender Klassen in der lokalen Programmiersprache und Anbindung an Schnittstelle
  - Typkonversion (IDL → lokale Sprache et vice versa)
- Kommunikation
  - Serialisierung/Deserialisierung
  - Versenden über Kommunikationsinfrastruktur („wire protocol“)

Funktionsweise: Beispiel



Referenzarchitektur: zentrale Komponenten



Lastverteilung

- für den Gesamtdurchsatz eines verteilten Systems u.U. von großer Bedeutung
- erfordert die Berücksichtigung der aktuellen Ressourcenbeanspruchung auf den involvierten Systemen sowie Annahmen über die Entwicklung der Ressourcenbeanspruchung
- unterschiedliche Strategien implementiert

Sicherheit

- Voraussetzung: sichere Kommunikation (Schutz von Mitlesen und Verfälschen)
  - digitale Unterschriften, Verschlüsselungsverfahren
- Fokus auf Identifikation und Authentifizierung
  - zentrale Verwaltung von Benutzernamen und Zugriffsrechten

Persistenz

- direkter Zugriff auf DBMS
  - erlaubt Einsatz von SQL oder anderen verbreiteten Zugriffssprachen
  - erfordert Abbildung des Typsystems der Programmiersprache auf Datenbankzugriffssprache
- Zugriff über Middleware
  - erfordert Abbildung des Typsystems auf Datenmodell (z.B. relationales Modell)
  - erlaubt Abstraktion von DBMS
- differenzierte Bewertung erforderlich
  - Investitionsschutz: Stabilität von Schnittstellen
  - Produktivität (z.B. durch Abstraktion von Typdifferenzen)
  - Performanz

Transaktionsschutz

- einzelne Dienstaufrufe können Bestandteil einer Transaktion sein
- Dienstnehmer zeigt dem Transaktionsverwalter Beginn und Ende der Transaktion an
- Transaktionsverwalter überwacht Durchführung nach Maßgabe eines geeigneten Protokolls – garantiert ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability)
- falls innerhalb einer Transaktion nur Datenbankdienste liegen, kann alternativ die Transaktions-verwaltung des DBMS genutzt werden

erforderliche Funktionen

- Message Queuing: Zwischenspeicherung von Nachrichten bis Ressourcen verfügbar sind
- Lock Verwaltung: sperrt Teile einer Datenbasis (Objekte oder Objekteigenschaften)

- Log Verwaltung: protokolliert alle Änderungen persistenter Daten (Voraussetzung für Rollback)
- Transaktionsprotokoll: Sicherstellung der Integrität bei Zugriff auf verteilte Datenbanken (z.B. Two Phase Commit: erst Vorbereitung, dann nach Rückmeldung Durchführung)
- Rollback: bei nicht erfolgreicher Abschluss einer Transaktion wird alter Zustand wiederhergestellt

Entwicklung verteilter Anwendungen mit Middleware

- Middleware unterstützt weitgehende Abstraktion von Verteilung und Heterogenität
- Entwicklung verteilter Systeme erfordert dennoch spezielle Vorkehrungen
  - Kenntnisse der Funktionsweise der Middleware
  - Nutzung der Werkzeuge der Middleware
- in jedem Fall angeraten: Nutzung einer geeigneten Methode (Sprache + Vorgehensmodell)

Entwicklung verteilter Anwendungen mit Middleware (1)

1) Anforderungsanalyse
2) Entwurf eines systemweiten Objektmodells
   - Hinzufügen von Management-Klassen
   - Spezifikation von Navigationspfaden
   - Erzeugung/Generierung der Signaturen von Zugriffsoperationen (set-/get-Methoden)
   - Erstellung weiterer Methoden
3) Erzeugen von IDL-Schnittstellen und Code
   - das Objektmodell dient als Grundlage für die Implementierung, dabei ist es i.d.R. sinnvoll, Werkzeuge zur Generierung von Code (Klassenspezifikation, Zugriffsoperationen) zu nutzen
   - das Objektmodell erlaubt die Erzeugung von Code in der jeweiligen Implementierungssprache (z.B. Java) und die Erzeugung korrespondierender Schnittstellenbeschreibungen in IDL
   - ggfs. kann der Umfang der öffentlichen (also verteilt nutzbaren) Klassen und Schnittstellen reduziert werden (durch Markierung im Objektmodell)
4) Implementierung
5) Test

Bewertung

- Objekte jeder Klasse können verteilt genutzt werden
- grundsätzlich positiv: hohe Abstraktion und Flexibilität
- beliebige Verteilung erzeugt allerdings erheblichen Verwaltungsaufwand und damit: Komplexität, Risiko
- Auswirkungen auf Performanz i.d.R. nicht akzeptabel
- Wartung und Test mit besonderen Herausforderungen verbunden
  - Test von Software aufwändig, Kompilierung hat systemweite Auswirkungen
- Alternative: Nutzung von Komponenten

Werkzeuge: IDL Compiler

- die Abbildung der Klassen einer Programmiersprache auf eine IDL wie auch umgekehrt ist mühsam und gleichzeitig gut zu automatisieren
- Werkzeuge zur Erzeugung und Einbindung von Programmcode für verschiedene Zielsprachen aus IDL-Spezifikationen gängiger Bestandteil von Middleware-Systemen
- vor allem sinnvoll für die Entwicklung von Anwendungen, die auf existierende Ressourcen (Klassen, Objekte) im verteilten System zugreifen
- besser wäre jedoch ein Generator, der aus einem Objektmodell Code und IDL erzeugt (Attribute)

Grundformen des Nachrichtenaustausches

- synchron, asynchron (ressourcenschonender, aber aufwändiger), publish & subscribe
- Zustellen/Empfangen von Nachrichten bei asynchroner Kommunikation: polling, ereignisbasiert
- Middleware-Systeme unterstützen nicht immer alle Formen

vergleichende Bewertung

- synchrone Kommunikation Spezialfall der asynchronen
  - asynchrone Nachricht wird abgeschickt
  - Dienstnehmer wartet bis Ereignis Antwort anzeigt
- ergo: mächtigeres, flexibleres Kommunikationsmodell
- publish & subscribe erhöht Flexibilität
- mitunter allerdings synchrone Kommunikation erforderlich
- Performanz hängt von spezifischem Kontext ab

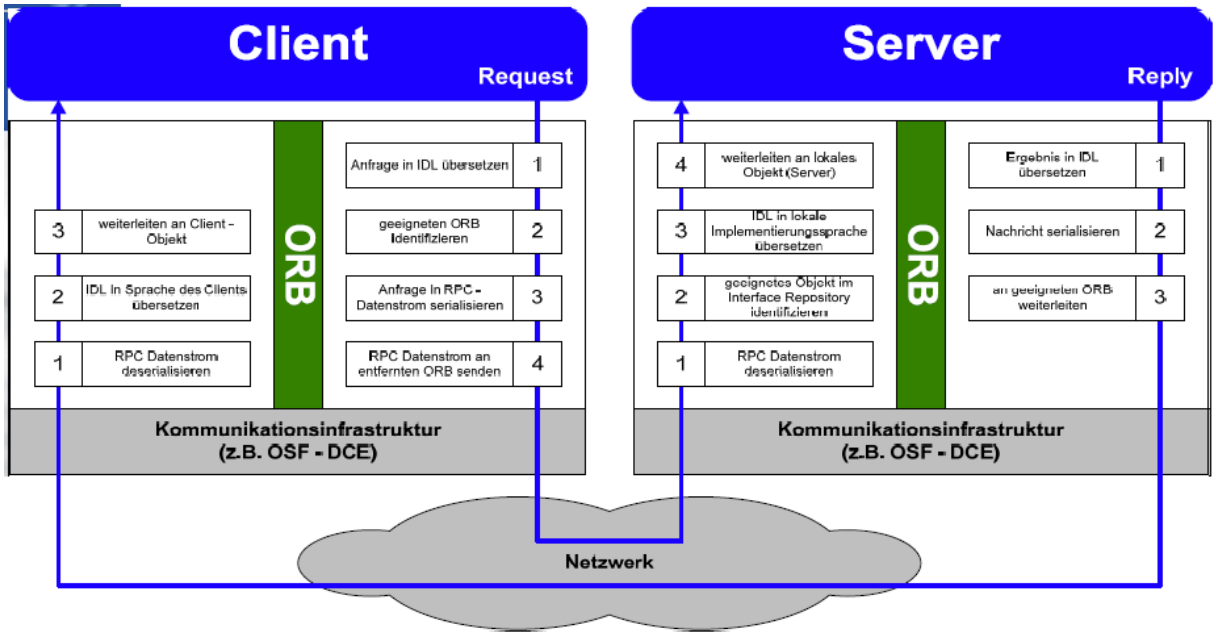
Kommunikationsplattformen

- OMG CORBA, Web Services, Java Enterprise Edition
- Persistenzframeworks (z.B. OO → RDBMS)

CORBA: Bestandteile

- Semantisches Referenzsystem
  - Objektmodell („Core Objekt Model“) → Metamodell zur Beschreibung von Klassen
  - Basistypen („Non-Objekt Types“)
  - repräsentiert durch IDL (dient der sprachunabhängigen Beschreibung von Schnittstellen)
- Object Management Architecture (OMA)
  - Referenzarchitektur für verteilte Systeme
  - Common Object Request Broker
  - Object Services (Object Life Cycle, Event Handling …)
  - Common Facilities (Basisdienste: Drucken, Persistenz …) → fördert Plattformunabhängigkeit

Basismodell der Kommunikation innerhalb der OMA



Middleware: Zwischenstand

- Middleware verkapselt die Heterogenität der im System enthaltenen Plattformen (OS, HW) – setzt Implementierungen der Middleware auf den jeweiligen Plattformen heraus
- idealtypisch: Anwendungen beziehen keine Dienste direkt vom jeweiligen OS, sondern indirekt über die Middleware
- ergo: Beitrag zur Integration (Kommunikationsfähigkeit) und Portabilität von Anwendungen
- Integrationsförderung hängt wesentlich vom Typsystem bzw. Objektmodell ab
  - sollte so reichhaltig sein wie Typsysteme der abzudeckenden Programmiersprachen
  - Problem: Paradigmenbruch zwischen prozeduralen und objektorientierten Sprachen
- unterschiedliche Konkretisierungen von Middleware
  - Kommunikationsplattformen (z.B. CORBA)
  - Application Server („Anwendungsdienstgeber“)
  - Application Server + Webserver
  - …
- je mehr Dienste von einer Middleware bereitgestellt werden, desto größer der Beitrag zur Wiederverwendung
- Standardisierung wünschenswert
  - verspricht Investitionsschutz
  - fördert Offenheit gegenüber externen Partnern
  - aber: Unterschied zwischen Norm und sog. Industriestandard
- Nutzung erfordert Anpassung der Anwendungsentwicklung
  - Architektur, Code, ggfs. konzeptuelle Modelle
- komplexe Systeme: Bewertung und Auswahl als Herausforderung
- Bewertung von Entwurfs-/Nutzungsalternativen ohne aufwändigen Tests kaum möglich
  - Art der Lastverteilung, Art des Zugriff auf DBMS
- ex-ante Bewertung der Skalierbarkeit für Anwender kaum möglich

Bezugsrahmen zur Bewertung

- generelle Kriterien
  - Korrektheit, Zuverlässigkeit, Robustheit, Performanz …
- Interoperabilität
  - existierende Sprachanbindungen, Verwendung von (Kommunikations-)Standards, Mechanismen für asynchrone Kommunikation, Mächtigkeit des Typsystems/Objektmodells
- Aufwand
  - (sichtbare) Komplexität (Rüstzeiten), Umfang angebotener Funktionen, Verfügbarkeit qualifizierter Entwickler
- Investitionsschutz
  - Anzahl der abgedeckten Plattformen, Verwendung von (Kommunikations-)Standards, Umfang der Unterstützung durch Software-Hoster

Komponententechnologien

Komponenten: Objekte, Klassen oder …

- erhebliche Aufmerksamkeit für Komponenten – in Forschung und Praxis
- vor allem für weit verbreitete Artefakte verwendet – z.B. ActiveX, JavaBeans etc.
- inspiriert durch Analogien zur HW
- mitunter als eine Evolutionsstufe über der Objektorientierung betrachtet

Komponenten – Definition

- „A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject of composition by third parties."
- jedoch keine einheitliche Begrifflichkeit

Komponenten vs. Objekte

	Komponenten	Klassen
Wiederverwendung	▪ Bündelung spezieller Ressourcen (Klassen) in einer Komponente erhöht Wiederverwendungschancen in anderen Kontexten ▪ Wiederverwendungsreichweite und -komfort hängen von der Semantik der Schnittstellen ab ▪ i.d.R. keine Spezialisierung/Generalisierung zwischen Komponenten – mit entsprechenden Nachteilen für die Wartung ▪ durch (i.d.R.) geringe Abhängigkeiten zwischen Komponenten allerdings Wartungsvorteile	▪ Je spezieller die Klasse (je höher ihr semantisches Niveau) desto geringer die Wiederverwendungschance in anderen Kontexten - desto höher der Wiederverwendungsnutzen in ähnlichen Kontexten. ▪ Spezialisierung/Generalisierung zwischen Klassen erleichtern die Wartung von Systemen – sofern die Spezialisierungshierarchie von hoher Stabilität ist.
Verteilung	▪ Durch Bündelung von Klassen, zwischen deren Objekte ein intensiver Nachrichtenaustausch besteht – und gleichzeitige Trennung von anderen Klassen, mit deren Objekte nur ein geringer Nachrichtenaustausch besteht, wird Performanz in verteilten Systemen weniger belastet. ▪ Die Komplexität, der sich die Wartung und das Systemmgmt gegenübersehen, wird reduziert.	▪ Jedes Objekt kann durch geeignete Middleware in einem verteilten System genutzt werden. ▪ Bei häufigen Aufrufen über Maschinengrenzen hinweg drohen damit allerdings erhebliche Performanzeinbußen.

Entwicklung verteilter Anwendungen mit Middleware (2)

1) Entwurf eines systemweiten Objektmodells
2) Abgrenzung von Komponenten
   - Entwurf erfolgt i.d.R. objektorientiert, also durch Klassen (gleicht der Realwelt)
   - Wiederverwendung und Verteilung empfehlen allerdings mitunter eine weitere Abstraktion
     - Klassen, die andere Klassen verwenden, können nicht isoliert wiederverwendet werden
     - Objekte einer Klasse, die intensiv mit Objekten anderer Klassen kommunizieren, sollten aus Performanzgründen eher auf der gleichen Plattform verwaltet werden
3) Identifikation öffentlicher Klassen und Methoden
4) Erzeugen korrespondierender IDL-Schnittstellen
   - bei Schnittstellen mit höherem semantischen Niveau mehr Wiederverwendungskomfort und höhere Integrität, jedoch mir geringerer Wiederverwendungsreichweite
5) Implementierung
6) Zuordnung von Komponenten zu Plattformen

Entwurf von Komponenten

- objektorientierte Analyse & Entwurf basieren auf der Annahme einer „natürlichen" Korrespondenz realweltlicher Objekte und SW-Objekten
- Komponenten werden vorzugsweise objektorientiert implementiert
- damit stellt sich die Frage: Wie werden komponentenorientierte System entworfen?
  - zentrale Abstraktionen bei Analyse und Entwurf
  - Zusammenfassung von Klassen zu Komponenten

mögliches Vorgehen

1) Entwurf eines Objektmodells
2) Ermittlung von Clustern durch (semi-)automatisches Verfahren
   - Berechnung von Objekt- bzw. Klassen-Clustern
   - Cluster: fasst Klassen mit enger Kopplung zusammen
3) brauchbare Cluster-Kandidaten für die Bildung von Komponenten

semi-automatisches Verfahren

- Grundannahme
  - Komponenten lassen sich durch formale, softwaretechnische Merkmale in einem Objektmodell identifizieren
- SW-Maße zur Analyse von Systemen
  - Autonomie von Klassen: z.B. coupling between objects (CBO)
  - Kopplung zwischen Klassen: Distanz zwischen Klassen (Kohäsionsmaß für Komponenten)
- Klassen-Cluster als Indiz für Komponenten
- aber: sagt nichts über tatsächliche Aufrufhäufigkeiten aus, auch fachliche Nähe wird nicht berücksichtigt → kann nur als Hinweis verwendet werden

Alternative: Ergänzung von Top Down durch Bottom Up Elemente

- Top down Entwurf empfiehlt objektorientierte Abstraktionen
- häufig erfolgt der Entwurf aber nicht ausschließlich top down, da existierende Artefakte wiederverwendet oder eingebunden werden sollen
- wann immer es sinnvoll erscheint, existierende Artefakte in eine Entwurfsmodell zu integrieren, kann die Verwendung von Komponenten als eine Abstraktion dieser Artefakte sinnvoll sein

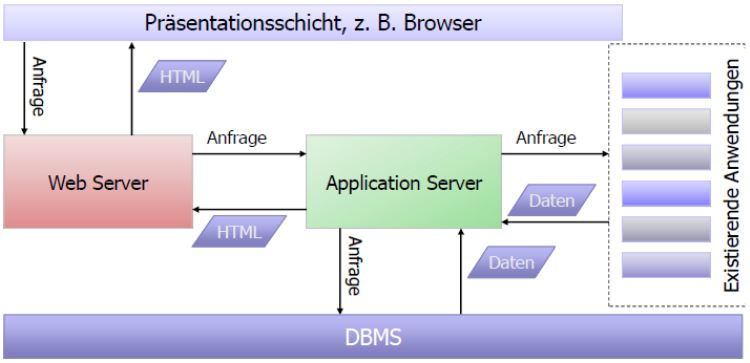
komponentenorientierte Architektur

- Grundidee: Aufbau eines IS aus Komponenten
- Komponenten laufen i.d.R. in einem speziellen Laufzeitsystem, das häufig verteilt eingesetzt kann („Application Server“)
- das Laufzeitsystem ist verbunden mit einer korrespondierenden Entwicklungsumgebung, die u.a. Navigation und Inspektion (Methoden, Code) von Komponenten unterstützt
- zusätzlicher Reiz: Komponenten können verwendet werden, um existierende Anwendungen zu kapseln
  - Anwendungen können in einem verteilten System genutzt werden
  - einheitlicher Zugriff auf alle Anwendungen – setzt allerdings geeignete Schnittstellen der Anwendungen voraus

Application Server

- Bestandteil von Middleware-Systemen
  - beinhaltet Namensdienste, Lastverteilung, Objektlebenszyklusdienst, Transaktionsschutz etc.
  - im Rahmen mehrschichtiger Architekturen eingesetzt
  - häufig mit Web-Server integriert
- Laufzeitumgebung für verteilte, komponentenorientierte Systeme
  - Verteilungstransparenz
  - Skalierbarkeit: weitere Instanzen eines AS können auf zusätzlicher HW eingesetzt werden
- ein sog. Komponentenmodell definiert eine Basisschnittstelle, die alle Komponenten nach außen anbieten

mehrschichtige Architektur: generisches Modell



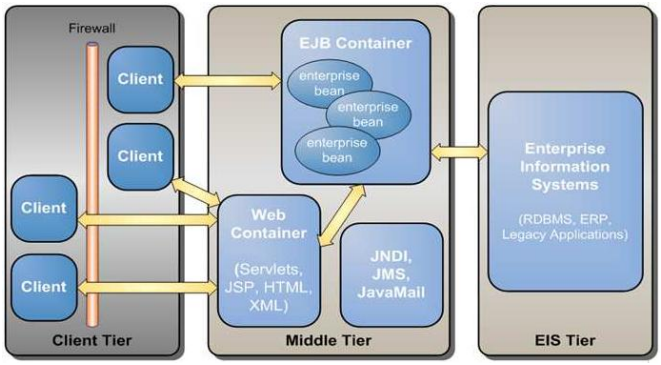
- erreichbares Integrationsniveau hängt von den Schnittstellen bzw. von den Implementierungen der Altanwendungen ab
- Ablösung von Altsystemen nicht immer eine Option, wenn sie funktionieren und keiner mehr Einsicht in den Quellcode hat
- Problem wenn zwei Anwendungen unterschiedliche Konzepte von Objekten haben (z.B. Kunden) und nicht gemeinsam die DB nutzen

Komponenten: Fazit

- aus konzeptueller Sicht (top down) gibt es keinen Bedarf für eine Abstraktion „Komponente“
- Verfügbarkeit von verteilten Laufzeitsystemen und korrespondierenden Entwicklungsumgebungen fördert die Produktivität
- Komponenten als (weitgehend unabhängige) SW-Module unterstützen Einsatz in anderen Kontexten (erforderliche Ressourcen weitgehen von Application Server bereitgestellt)
- Komponenten als Abstraktion über existierende Anwendungssysteme
  - Beitrag zum Investitionsschutz, Förderung verteilter Systeme
- Komponenten tragen tendenziell zu einer geringen Redundanz bei, wenn sich die wiederverwendbaren Komponenten gut in unterschiedlichen Kontexten einsetzen lassen
- Komponenten tragen tendenziell zu einer größeren Redundanz bei, wenn Klassen von vielen Komponenten eingesetzt werden (z.B. für Persistenz, Sicherheit)
- Komponenten fördern die Integrität eines Systems, wenn sie sorgfältig entworfen und implementiert sowie gut getestet sind
- Komponenten gefährden die Integrität eines Systems, wenn eine Klasse immer wieder in den Komponenten vorkommt → siehe Redundanz

→ die Qualität komponentenorientierter Architekturen hängt wesentlich davon ab, wie die jeweilige Konfiguration von Komponenten diesen Konflikten begegnet (Redundanz tut nicht weh, wenn diese Klasse langfristig stabil ist)

JEE-Architektur



Service-Orientierte Architekturen

SOA: Motivation

- motiviert durch Unzulänglichkeiten monolithischer SW-Systeme
  - häufig eher an betrieblichen Funktionen orientiert → unzureichende Unterstützung von GPs
  - Änderung von Anforderungen erfordert aufwändige, mitunter nicht finanzierbare Wartungsprojekte
- vordergründig attraktive Vision
  - Gestaltung von IS auf der Basis optimierter GPs
  - Realisierung der benötigten Funktionalität durch Nutzung von Services

Verheißungen

- Potentiale durch standardisierte Infrastruktur
  - Kommunikationsinfrastruktur, Schnittstellen, ggfs. Beschreibung von Diensteigenschaften
- Wiederverwendung
  - SW kann durch Rückgriff auf existierende Dienste realisiert werden
  - ggfs. weltweit wachsende Bibliothek von Diensten
- Integration
  - existierender Anwendungen
  - vor allem: unternehmensübergreifende Integration zur Unterstützung von Kooperationsprozessen

SOA: Merkmale

- Begriff vielfach vom Marketing instrumentalisiert
- keine eindeutige Begriffsabgrenzung
- Bereitstellung von Anwendungsfunktionalität durch Dienste
  - einfacher Fall: Clients greifen auf einzelne Dienste (z.B. von gekapselten Anwendungen) zu
  - anspruchsvoller Fall: Anwendungsmetapher wird weitgehend aufgegeben. Anwendungsszenarien werden durch die Koordination von Diensten (aus mehreren Quellen) unterstützt
- Betonung von Standards als wesentliches Merkmal
  - Architektur offen für Erweiterungen über Unternehmensgrenze hinweg
- Herausforderungen: Systeme müssen geeignete Schnittstellen bereitstellen, außerdem ist die Integration (auf konzeptueller und der Instanzen-Ebene) verschiedene Services nicht trivial → semantisches Referenzsystem und gemeinsamer Namensraum benötigt, aber die Services haben i.d.R. keine gemeinsame Datenbasis

Koordinationsprotokolle

- beschreiben Kontrollfluss für den Aufruf von Diensten zur Unterstützung eines bestimmten Nutzungsszenarios
  - mitunter „Orchestrierung“ genannt
  - „Choreographie“: Abstimmung zwischen verschiedenen Prozessen
  - können durch Modelle (z.B. Aktivitätsdiagramme) veranschaulicht werden
  - sollten durch standardisierte Sprachen spezifiziert werden
- Alternative zur Spezifikation des Kontrollflusses mittels der Programmiersprachen der aufrufenden Anwendungen (Nachteil: feste Verdrahtung, schlechtere Wartbarkeit)

Web Services – Merkmale

- Identifizierung der Dienste durch URI (u.a. URL, Port, Service Name)

- Art der Interaktion ist durch Beschreibung vorgegeben
- Nachrichtenaustausch mittels WSDL
- Gebrauch von vorhandenen Transportprotokollen (z.B. HTTP)
- besonders gut geeignet, um SOA zu unterstützen
  - große Offenheit durch Nutzung von Internet-Standards, große Verbreitung
- i.d.R. zustandslos, keine Speicherung des Ergebnisses
  - dienstausführende SW abstrahiert vom Kontext (Zustand) des Dienstaufrufers
- quasi-zustandsbehaftete Services allerdings möglich
  - Verwendung von Ressourcen auf Server-Seite, deren Identifikation durch eine zustandslose Nachricht erfolgt (Client muss sich Zustände merken)
- Kommunikation i.d.R. synchron
- asynchrone Kommunikation erfordert Identifikation des Dienstaufrufers und Polling bzw. dedizierten Web Service

Web Services – Vorteile

- Standardisierung und große Verbreitung korrespondierender Technologien
  - zahlreiche (kostengünstige) Werkzeuge verfügbar, die Produktivität fördern
  - relativ große Zahl einschlägig qualifizierter Entwickler
- Chance auf hohen Wiederverwendungsnutzen
  - wiederverwendbare Dienste relativ leicht zu nutzen (standardisierte Schnittstelle)
  - potentiell große, weiter wachsende Bibliothek von Diensten
- durch offene Standards wird die Realisierung unternehmensübergreifender IS gefördert

Web Services – Nachteile

- relativ geringes semantisches Niveau der Kommunikation
  - i.d.R. Semantikverlust bei Transformation in WSDL
  - impliziert Integritätsprobleme
- Standardisierung zunächst auf WSDL beschränkt
- Parsen von WSDL belastet Performanz der Kommunikation
- Integration gemeinsamer Datenbestände wird nicht unterstützt!
- ergo: ggfs. Redundanz und daraus resultierende Integritätsprobleme

Web Services – Einsatzvoraussetzungen

- Performanzverluste durch eingeschränkte Bandbreiten sind hinnehmbar
- geringes Integrationsniveau ist hinnehmbar
- Implementierung von Diensten im Zeitverlauf variant, Schnittstelle bleibt stabil
- Zuständigkeit (Verantwortung, Kompetenz) für die Implementierung und Wartung von Diensten lässt sich klar abgrenzen
- Verlässlichkeit: die mit einem Dienst verbundenen Zusicherungen sind präzise spezifiziert; ihre Einhaltung wird (vertraglich) garantiert

Web Services – Einsatzgebiete

- allgemeine Dienste, die Systemzustände wiedergeben, die sich im Zeitverlauf ändern (und deren Erfassung aufwändig ist)
  - Wetterauskunft, Adress-/Telefonverwaltung, Uhrzeit
- komplexere, in sich geschlossene Dienste
  - Erstellung einer Steuererklärung oder Bilanz, Optimierung in der Produktionsplanung

populäre Vision: Web Services in GPs

- Unternehmen erwerben keine Anwendungssysteme mehr
- Fokus liegt vielmehr auf der Gestaltung/Optimierung von GPs sowie deren Unterstützung durch geeignete Web Services („best of breed“)
- wird auch als Architekturrahmen für die Integration existierender, heterogener Anwendungslandschaften diskutiert (EAI)
  - Anwendungssysteme werden unter Rückgriff auf Web Services verkapselt
  - auf diese Weise können Anwendungssysteme bzw. Teile der selben im Zeitverlauf ohne Seiteneffekte ausgewechselt werden
- Verheißung der freien Wahl von Services wird dadurch beeinträchtigt, dass man bei gewünschter Datenintegration nur die Services wählen kann, die eine gemeinsame Datenbasis und gemeinsame Ereignistypen besitzen

Business Process Execution Language for Web Services (BPEL4WS)

- Definition von Prozessen mittels Komposition von Web Services
- zwei verschiedene Aspekte
  - Bereitstellung von Funktionen durch Web Services (WSDL, UDDI)
  - Definition des Kontrollflusses zwischen Web Services mittels einer dedizierten Sprache (BPEL4WS: Kontrollfluss)
- ergo: Kontrollflussspezifikation unter Berücksichtigung synchroner Funktionsaufrufe
- Standardisierung der Sprachelemente (nicht grafische Notation) durch OASIS

BPEL: wesentliche Aspekte

- Schnittstellen zwischen Partnern
  - Port Types (Bestandteil der Web Services): definieren Schnittstelle eines WS-Anbieters
  - Partnerlinks (BPEL): legen Beziehungen zwischen konkreten Partnern fest
- lokaler Kontrollfluss
  - lokale Prozesse werden meist angestoßen durch den Eingang einer Nachricht
  - Spezifikation des lokalen Kontrollflusses durch Graph-basierten Ansatz
    - Link: definiert eine Beziehung
    - Activity: kann Quelle oder Senke bzgl. eines Links sein

BPEL: kritische Würdigung – Vorteile

- Definition einer Integrationsbasis für etablierte Technologien (Web Service) durchaus positiv
  - weite Verbreitung von Web Services, Standardisierung
- grundlegende Aspekte des Kontrollflusses können durch den Knoten-Link-basierten Ansatz dargestellt werden
- ergänzend stehen typische Kontrollstrukturen (Iteration, Verzweigung, …) bereit
- durch Rückgriff auf Web Services prinzipiell gut für die Unterstützung unternehmensübergreifender GPs geeignet

BPEL: kritische Würdigung – Einschränkungen

- keine graphische Notation definiert
  - aber: XML-basiertes Format vorhanden
- Einschränkungen des Anwendungsbereiches auf Web Services und somit keine dedizierte Berücksichtigung von sonstigen Ressourcen und Informationsobjekten außerhalb der WS-Schnittstellen

- Bezug zu klassischen WfMS nicht eindeutig
  - ergänzende Abstraktion (Integration interorganisationaler Workflows) oder
  - orthogonales Konzept (Integration von WS zusätzlich zu existierenden WfMS)

SOA - Beurteilung

- Zugriff auf verteilt implementierter Dienste alter Hut (z.B. RPCs)
- Web Services zwar nicht notwendig für SOA, aber für deren Nutzen von erheblicher Bedeutung
  - Offenheit über Unternehmensgrenzen hinaus
  - durch BPEL Unterstützung (unternehmensübergreifender) GPs
  - Aussicht auf weltweite Bibliothek von Diensten
- keine Abstraktionskonzepte für Services (z.B. Generalisierung)
  - eingeschränkte Wartbarkeit und Wiederverwendbarkeit
- von Persistenz wird abstrahiert, ist für IS aber von zentraler Bedeutung
- objektorientierte oder komponentenorientierte Kommunikationsplattform softwaretechnisch überlegen
  - Abstraktionsmechanismus
  - Nachrichtenaustausch performanter
- deshalb zur Konstruktion eines Systems im Detail wenig geeignet
- eher als ergänzende Schnittstelle von Systemen, um große Offenheit zu erreichen

Open Application Group (OAG) – die Vision

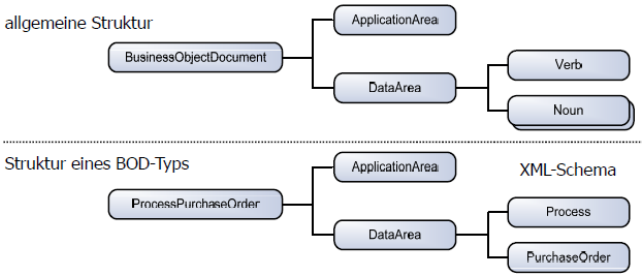
- Komposition eines integrierten IS aus verschiedenen Anwendungen
- „Best of Breed“: für jede Teilfunktion wird die am besten geeignete Anwendung am Markt erworben
- ursprüngliche Vision: einfache Erweiterung des IS durch „Anstecken“ weitere Anwendungen an einen zentralen Kommunikationsbus
- Integration bestehender „Insellösungen“ in ein unternehmensweites IS

OAG Integration Specification (OAGIS)

- OAGIS definiert ein dokumentenbasiertes Protokoll zur (losen) Kopplung betrieblicher Anwendungen
- Kern des Standards sind die sog. Business Object Documents (BOD)
  - allgemeine Struktur, Schemata für ca. 400 BOD-Typen (in XML Schema)
- BOD dienen als Vehikel zur Sendungen von Botschaften zwischen (verteilten) Anwendungen
- von technischer Implementierung wird weitgehend abstrahiert, Verweis u.a. auf Web Services

OAGIS BODs: Semantisches Referenzsystem für die Kommunikation

- OAGIS BODs nutzen XML um eine einfache, generische betriebswirtschaftliche Sprache zu definieren
- BOD Architektur
  - definiert die allgemeine XML-Struktur und das Verhalten aller OAGIS-Nachrichten



→ definiert Verhalten (Get, Cancel …)
→ entsprechen Objekten (Dokumente, Kontrolldaten, Inhalte)

- BOD Definition (z.B. ProcessPurchaseOrder.XSD)
  - definiert Layout bzw. Struktur einer bestimmten Nachricht
  - häufig durch Erweiterung (Spezialisierung) spezifiziert
- BOD Instanz (z.B. ProcessPurchaseOrder.XML)
  - ist eine tatsächliche Nachricht, die reale Daten enthält

Szenarios und Nachrichten

- Szenarios
  - sind relativ abstrakte Prozessbeschreibungen
  - liefern eine Kontext für Nachrichten
  - dienen als Bibliothek für wieder verwendbare Prozesse
- BODs sind die Nachrichten der Szenarios

Integrationsniveau

- statische Integration: Struktur der auszutauschenden Dokumente (BOD-Typen)
- lose Kopplung
  - interne Datenstruktur losgelöst von der Nachrichtendatenstruktur
  - Anwendungen sind isoliert für Datenverwaltung zuständig
  - Kommunikation erfolgt asynchron
  - BODs werden ohne Angabe eines Empfängers auf den Integrationsbus geschickt
- Vorteil der losen Kopplung durch asynchrone Kommunikation: höhere Performanz möglich
- Nachteil: kein direktes Feedback, Kommunikation wird aufwändiger, höheres Risiko

Vergleich mit objektorientierter Middleware, z.B. CORBA

- ähnliche Ziele: Integration von Anwendungen
- wichtiger Unterschied: OAGIS umfasst die Spezifikation anwendungsnaher Objekte (BODs)
- aber: unterschiedliche Einsatzszenarien
  - CORBA: Entwicklungen der zu integrierenden Anwendungen sollte gemeinsame Klassen berücksichtigen (relativ enge ex ante Integration)
  - OAGIS: Anwendungen werden eher ex post integriert – in loser Kopplung
- lose Kopplung vereinfacht die Integration ist aber i.d.R. mit einem geringen Integrationsniveau verbunden
- Datenintegration meist nicht möglich, da Altanwendungen eigene Datenspeicherung haben → redundante Datenhaltung

kritische Würdigung und Ausblick

- weltweites, herstellerunabhängiges Protokoll
- Integration betrieblicher (Standard-)Anwendungen über Betriebs- und OS-Grenzen hinweg
- der Botschaftsmechanismus BOD legt eine „Semantische Kapsel“ um die einzelnen Anwendungen und erleichtert so deren Wartung (wenn die Schnittstellen stabil sind)
- Angemessenheit und Vollständigkeit der BODs bisher nicht umfassend überprüft
- das Fehlen einer gemeinsamen Datenhaltung fördert Redundanz und Dateninkonsistenz
- die OAG-Mitglieder kommen überwiegend aus dem angelsächsischen Raum
- neuere Marketing-Verlautbarungen von SAP betonen die „Best of Breed“-Vision – Konsequenzen allerdings schwer absehbar
- „Plug and Play“-Zusammenspiel OAGIS-kompatibler Anwendungen wird nicht erreicht, aber auch nicht angestrebt (dafür fehlt Definition der Systemdynamik mit Ereignissen)

Persistenztechnologien

Objekte in DBMS

- Ziel: komfortable und sichere Verwaltung persistenter Objekte
- dabei: Erfüllung wesentlicher Anforderungen an DBMS
  - Persistenz, Nebenläufigkeit von Transaktionen, Recovery Mechanismen, Anfragesprachen, Verteilung
- Problem: noch keine einheitliche und leistungsfähige Technologie am Markt verfügbar
- Herausforderung: bei Speicherung von Objekten muss die transitive Hülle mit gespeichert werden, dies soll aber transparent für den Entwickler/Nutzer erfolgen
- Herausforderung: auch das Schema muss gespeichert werden → semantische Abbildung (Klassen auf Schema, Objekte auf DB-Objekte), idealerweise mit der gleichen Sprache

Arten von DBMS zur Speicherung von Objekten

- Relationale DBMS (RDBMS)
- Objektrelationale DBMS (ORDBMS)
- Objektorientierte DBMS (ODBMS)

RDBMS für persistente Objekte

- relationale DBMS heute weit verbreitet
  - große Zahl ausgereifter Produkte, Vielzahl kompetenter Fachleute, mathematische fundiert
- generell: Objekte (besser: Objektzustände) können in RDBMS gespeichert werden
- aber: Bruch zwischen Paradigmen
  - Objekt als Instanz einer Klasse kapselt Zustand und kann auf Nachrichten reagieren (Klasse definiert Struktur und Verhalten der Klasse)
  - Entitäten in RDBMS sind Tupel einer Relation – keine Kapselung der enthaltenen Attribute
- Probleme bei Abbildung von Klassen auf das Schema
  - Speicherbegrenzung bei Strings in der DB: VARCHAR(255)
  - keine Abbildung der Zugriffsmethoden in der DB
  - Mengenlehre erlauben keine identischen Tupel, künstliche Schlüssel nötig

Abbildung von Vererbung

- Vererbung i.d.R. nicht durch RDBMS unterstützt
- Strategie 1: Abbildung auf eine universelle Tabelle
  - eine Tabelle für alle Typen der Hierarchie
  - enthält alle Attribute, nicht benötigte werden frei gelassen
  - bei disziplinierter Anwendung können Anomalien von Vererbungsbeziehungen in OO-Programmiersprachen vermieden werden
    - Person wird Mitarbeiter: Hinzufügen entsprechender Attributwerte
    - Person hört auf, Student zu sein: Löschen der entsprechenden Attributwerte
  - aber: erheblicher Verlust von Semantik
    - OO-Klassen gehen als Abstraktion verloren
    - impliziert erheblichen Aufwand und damit verbundene Risiken beim Zugriff aus Programmiersprache
- Strategie 2: Abbildung auf mehrere Tabellen (1)
  - Objekte einer Klasse auf jeweils eine Tabelle abgebildet
  - enthält alle Attribute der jeweiligen Klasse (auch die geerbten)

  - entspricht hinsichtlich der Attribute der verwendeten Entitätstypen der (unbefriedigenden) Semantik objektorientierter Programmiersprachen
  - vordergründig positiv: kein Bruch zur OO-Programmiersprache
  - aber: erheblicher Verlust an Semantik
    - Spezialisierungsbeziehungen werden nicht abgebildet
    - Änderungen in Oberklassen führen nicht zwingende zu Änderungen in Tabellen, die Unterklassen repräsentieren
    - ergo: erhöhter Abbildungsaufwand, erhöhtes Risiko
- Strategie 3: Abbildung auf mehrere Tabellen (2)
  - Objekte einer Klasse auf jeweils eine Tabelle abgebildet
  - enthält nur die spezifischen Attribute der jeweiligen Klasse (nicht die geerbten)
  - Vererbungsrelationen werden durch Primärschlüssel dargestellt
  - bildet im Hinblick auf die Objektzustände das Rollenkonzept (Delegation ab)
  - beugt damit den Anomalien der Spezialisierungsbeziehungen in OO-Programmiersprachen vor
  - aber: semantischer Bruch zur OO-Programmiersprache – „Unterrelation“ unterscheidet sich in ihrer Struktur von korrespondierender Klasse
  - Abbildungsaufwand und resultierende Risiken erheblich

ORDBMS

- Erweiterung relationaler Systeme um objektorientierte Konzepte
- Motivation: wachsende Popularität der OO-Programmiersprachen in den 1990er Jahren
- erste Systeme mit proprietären Erweiterungen bereits früh verfügbar
- erste Standardisierung objektrelationaler Konzepte durch SQL:1999
  - benutzerdefinierte Prozeduren (stored procedures)
  - (strukturierte) Datentypen: Large Objects, Subtypen, Multimengen als Typen von Attributen
  - Anfragearten: rekursive Anfragen, Anfragen mit Methodenaufrufen
- aber: Relation bleibt weiterhin das dominierende Konzept

Vererbung

- basiert auf extensionalem Klassenbegriff
  - eine Entität (Objekt) kann gleichzeitig Element mehrerer Entitätstypen (Relationstypen) sein
  - entspricht dem alltagsweltlichen Verständnis von Spezialisierung
  - unterscheidet sich jedoch subtil, aber erheblich von der Spezialisierungssemantik der OO-Programmiersprache
- Konsequenz: bei unbedachter Abbildung von Vererbungshierarchie drohen Verzerrungen

ORDBMS: Bewertung

- Erweiterung relationaler Systeme bietet mögliche ökonomische Vorteile
  - relationale Systeme bewährt und ausgereift
  - Produkte können weiter gepflegt und angeboten werden
  - Kunden müssen nicht migrieren → Investitionsschutz
- Paradigmenbruch existiert immer noch
  - keine Verkapselung
  - abweichende Semantik von Spezialisierungsbeziehungen
  - Abfragen in anderer Sprache (SQL)
- Integration des persistenten Speichers nicht transparent für den Entwickler

- Verkapselung aufgebrochen → Abstraktionsnachteile
- es müssen Konzepte in zwei Namensräumen gewartet werden
  - DB: Typen und Routinen
  - Programm: Klassen
  - → Änderungen müssen in beiden Namensräumen vorgenommen werden
- deshalb ökonomisch u.U. riskant
  - fördert Verharren in alter Technologie
  - behindert frühzeitige Auseinandersetzung mit fortschrittlichen Ansätzen

ODBMS

- grundlegende Konzepte: Klassen/Objekte und Beziehungen zwischen Klassen/Objekten
- Ziel: Elimination des „Impedance mismatch“,
  - d.h. dem Bruch zwischen Paradigmen
  - Reduktion von Friktionen aufgrund der Abbildung zwischen Paradigmen
- Modell für lesenden Zugriff
  - Navigation über Referenzen zwischen Objekten
  - nicht: mengenorientierte Anfragen in SQL

Ideal: transparente Persistenz

- Principal of Data Type Orthogonality
  - keine Unterscheidung zwischen persistenten und transienten Typen/Klassen
  - Persistenz also unabhängig vom Typ eines Objektes
- Principle of Persistence Identification
  - keine expliziten Lade- und Schreiboperationen erforderlich
  - persistente Objekte sollen automatisch ermittelt werden
- Principle of Persistence Independance
  - persistente Objekte sollen genauso behandelt werden wie transiente
  - im Programmcode wird nicht zwischen persistenten und transienten Objekten unterschieden

Persistenz in OO-Anwendungen

- klassenabhängige Persistenz
  - Klassen werden im Code explizit als persistent gekennzeichnet („persistente Klassen“)
  - alle Instanzen solcher Klassen sind automatisch persistent
  - analog: Instanzen nicht persistenter Klassen können nicht persistent sein
- objektabhängige Persistenz
  - Definition persistenzfähiger Klassen
  - Instanzen solcher Klassen können persistent sein, sind es aber nicht per se

Deklaration persistenzfähiger Klassen

- Persistenzfähigkeit durch Vererbung
  - es existiert eine abstrakte Klasse als Oberklasse für alle persistenzfähigen Klassen, diese definiert Schnittstellen und Verhalten für persistente Objekte
  - Einschränkungen bei Sprachen ohne Mehrfachvererbung
- Persistenzfähigkeit durch explizite Auszeichnung
  - persistenzfähige Klassen werden explizit als solche gekennzeichnet
  - Veränderungen bei Compiler/Interpreter oder Einsatz eines Präprozessors nötig
- typunabhängige Persistenzfähigkeit

  - Instanzen aller Klassen können persistent gemacht werden
  - massive Änderungen der Laufzeitumgebung notwendig
- persistente Programmiersprachen
  - alle Objekte werden als persistente Objekte erstellt
  - ergo: kein Synchronisationsbedarf zwischen transienten und korrespondierenden persistenten Objekten
  - ODBMS stellt Laufzeitumgebung für die Ausführung von Programmen bzw. Laufzeitumgebung erlaubt Umgang mit persistenten Objekten → aber schlechte Performanz

objektabhängige Persistenz: Arten der Erzeugung persistenter Objekte

- Instanziierung als persistentes Objekt
  - bereits bei der Instanziierung wird festgelegt, dass ein Objekt persistent sein muss
  - spätere Änderung nur bedingt möglich
- Persistentmachung bei Methodenaufruf
  - Senden einer festgelegten Nachricht, um ein Objekt persistent zu machen
  - Zeitpunkt der Entscheidung bzgl. Persistenz flexibel
  - häufig in Persistenzframeworks zu finden
- Zuweisung eines Bezeichners in der DB
  - ein Objekt kann explizit persistent gemacht werden, indem es dem Namensraum der DB übergeben wird
  - bspw. kann ein DB-Bezeichner (Variable) definiert werden, welchem ein konkretes Objekt zugewiesen wird
  - ODBMS verfügt über eine Laufzeitumgebung, verwaltet die Namensräume von Objekte ähnlich wie das Laufzeitsystem einer Programmiersprache den Arbeitsspeicher

Persistenz durch Erreichbarkeit

- neben expliziter Persistenz erfordert Integrität auch implizite Persistenz
- ein Objekt ist genau dann persistent, wenn es von einem persistenten Objekt referenziert wird
- Beispiel
  - ein Objekt explizit als persistent (persistente Wurzel) definiert
  - alle von diesem Objekt direkt oder indirekt referenzierten Objekte sind wiederum persistent

ODBMS – kritische Würdigung

- kein Bruch der Paradigmen: erleichterter SW-Entwicklung und Wartung
- transparente Integration möglich zwischen Programmiersprache und DBMS
- nur wenige Systeme verfügbar
  - Konzentration auf Java, i.d.R. keine aktiven Objekte unterstützt
- Standard zwar definiert (ODMG), aber nicht auf breiter Basis umgesetzt
  - für jede Programmiersprache würde eine eigene ODBMS benötigt → Middleware wäre ein Ausweg, aber Performanzeinbußen

vergleichende Bewertung

	Verfügbarkeit	Zuverlässigkeit	Integration	Performanz
RDBMS	++	++	-	o
RDBMS + Framework	+	+	o	o
ORDBMS	++	++	o	o
ODBMS	-	o	+	o
persistente Progr.Sprache	--	o	++	--

Methodische Unterstützung von Integrationsvorhaben

Enterprise Application Integration (EAI)

- zentrale Botschaft von großer Attraktivität: existierende Systeme können weiter verwendet werden, werden aber integriert
- großer Bedarf bei vielen Unternehmen
  - über lange Zeit gewachsene Anwendungslandschaften, häufig unzureichend integriert
  - gleichzeitig mitunter hohe Abhängigkeit von Altsystemen
- deshalb attraktiver Markt für Anbieter einschlägiger Werkzeuge und Dienstleistungen
- Begriff nicht einheitlich verwendet
  - einerseits Fokus auf (Middleware-)Systemen, die die Integration heterogener Anwendungslandschaften unterstützen
  - andererseits auf Projekten zur Integration heterogener Anwendungslandschaften
  - bzw. eine Kombination aus beidem
- Formen von EAI: Application-to-Application, Business-to-Business, Business-to-Consumer

Prozessorientierung als wesentliches Merkmal

- nicht nur eine statische datenorientierte Integration
- sondern dynamische Integration von Anwendungen entlang von GPs
- betont prozessorientierter Sicht
  - verspricht eine Ausrichtung an Unternehmenszielen bzw. Kundenwünschen
  - empfiehlt u.U. Reorganisation von GPs
  - Integration nicht nur, aber auch, als technische Aufgabe

EAI aus Sicht der Unternehmensleitung

- häufig große betriebswirtschaftliche Bedeutung
  - für die Wettbewerbsfähigkeit
  - für den Schutz von Investitionen in existierende Systeme
- sehr komplexer Gegenstand
  - vielfältige Abhängigkeiten zwischen IT-Ressourcen
  - Integrationsziele häufig unklar und schwer zu spezifizieren
  - spezifischer Nutzen für das Unternehmen schwer fassbar
  - verwirrende Vielfalt von Integrationstechnologien; Erfassung und Bewertung mit großem Aufwand verbunden
  - ergo: hohes Risikopotenzial

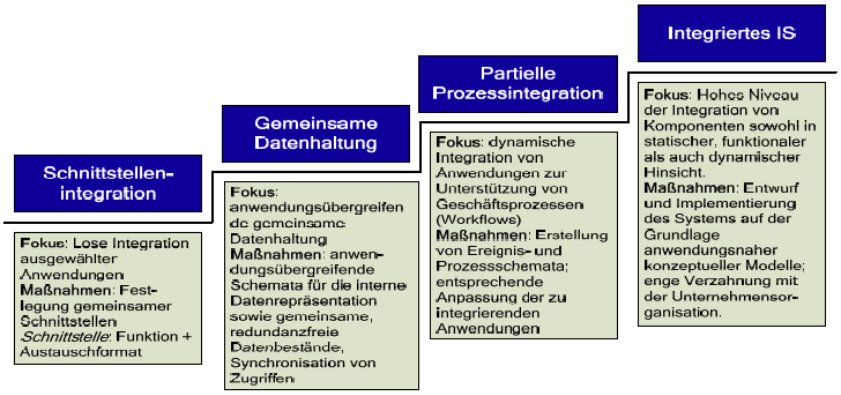
Anforderungen an einen Lösungsansatz

- wirksame Reduktion von Komplexität
  - Fokussierung auf wesentliche Integrationsaspekte
  - weitgehende Vernachlässigung spezieller, zeitlich variierender Eigenschaften (Technologie …)
  - übersichtliche Darstellung auch komplexer Zusammenhänge
- Unterstützung einschlägiger Analysen
  - Bereitstellung geeigneter Konzepte und damit korrespondierender Vorgehensweise

beispielhaftes Vorgehensmodell

1) Klärung der IT-Strategie
   - zielt auf
     - Entwurf einer sinnstiftenden langfristigen Orientierung

- dargestellt z.B. in einem Evolutionsstufenmodell, für verschiedene Teile des IS können unterschiedliche Evolutionspfade vorgesehen werden



- Schutz der Investitionen in frühe Integrationsprojekte
- beinhaltet Analyse
  - der Unternehmensstrategie
  - insbesondere: unternehmensübergreifende GPs
  - der zukünftigen Wettbewerbsintensität von Kernprozessen

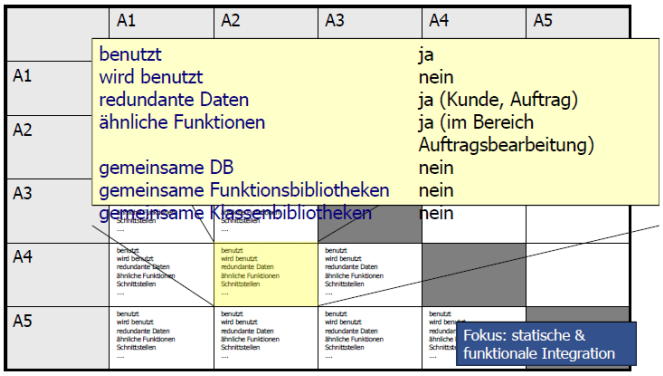
2) Modellierung der relevanten Prozesse
3) Beschreibung der beteiligten Systeme
4) Analyse des aktuellen Integrationsniveaus
5) Analyse des Integrationsbedarfs
6) Erstellung eines Integrationsplans (mit Rücksprung zur dritten Phase)

Skizze einer dedizierten Modellierungsmethode (Auswahl oder selbst erstellt)

- Modellierungssprachen
  - Sprache zur statischen Modellierung von IT-Landschaften
  - korrespondierende Sprache zur Modellierung von GPs
- Vorgehensmodell für unterschiedliche Arten von Integrationsprojekten, z.B.
  - kontextunabhängige Analyse der IT-Landschaft
  - prozessbasierte Analyse der IT-Landschaft
- Unterstützung bei der Auswahl geeigneter Technologien

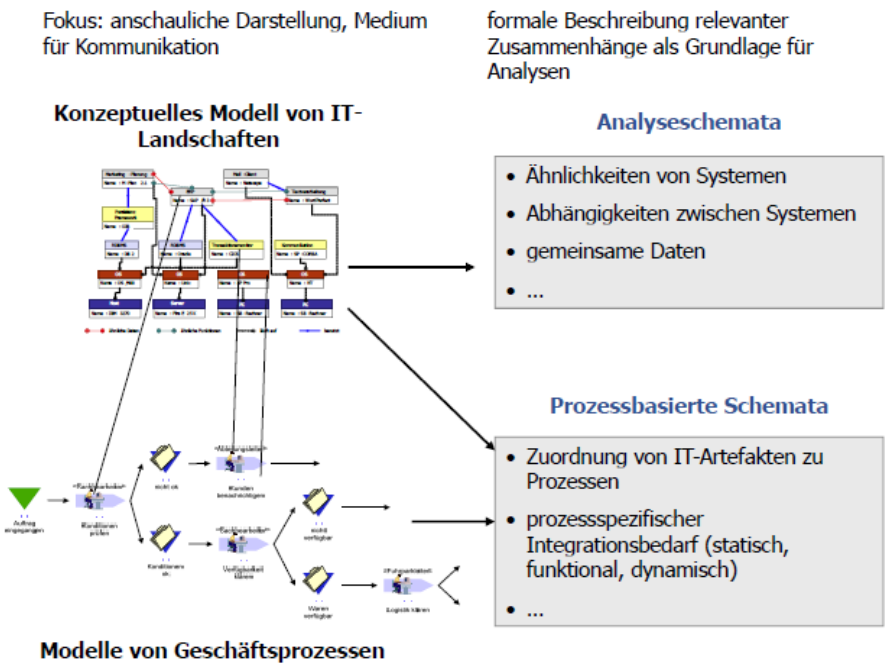
Skizze einer Methode zur Integration heterogener Anwendungslandschaften

- Analyse des Integrationsstatus
  - Modellierung der Systemlandschaft auf geeignetem Abstraktionsniveau (grafische Modellierung als Option zur Verbesserung der Anschaulichkeit)
  - Erstellung/Generierung eines entsprechenden Schemas (z.B. mit Excel oder DBMS)



  - Instanziierung und Durchführung geeigneter Auswertungen

- zwei Ausprägungsformen
  - kontextunabhängig
  - im Kontext der Nutzung durch GPs, Fokus erweitert auf dynamische Integration und vor allem betriebswirtschaftlichen Nutzen der Integration → hier Durchführung einer betriebswirtschaftlichen Bewertung eher möglich



Durchführung von Integrationsprojekten: Analyseschritte

Analysephase	Dokumente	Angaben
Beschreibung der Anwendungssysteme	Modelle der Anwendungen	Datenhaltung (File/ RDB/…), Datenmodell (offen/verborgen), Source Code (ja/nein), Ereignisschema (ja/nein)
Erfassung des Ist-Zustands der statischen Integration von Anwendungssystemen	Modelle der Anwendungen	Systeme, ähnliche Daten, ähnliche Funktionen
Erfassung des Ist-Zustands der funktionalen Integration von Anwendungssystemen	Modelle der Anwendungslandschaft	
Erfassung des Ist-Zustands der dynamischen Integration von Anwendungssystemen	Modelle der Anwendungen und Prozessmodelle	ausgelöst durch, terminiert durch, Ausnahmebehandlung (ja/nein)
Erfassung des Integrations-bedarfs (Kernprozesse) und des Integrationsaufwands	Modelle der Anwendungen und Prozessmodelle, Dokumente der strateg. Integrationsplanung	

Entwurf von Integrationslösungen

- Basis: analysierter Integrationsbedarf
- weitere Entscheidungsaspekte
  - wirtschaftliche Dringlichkeit der Integration
  - strategische Bedeutung der Integration
  - Integrationsaufwand/-risiko

- dazu: Integrieren alter Systeme oder Neuentwicklung
- Make or Buy?

- Erfassung und Bewertung unterstützender Integrationstechnologien (Werkzeuge)
  - zunächst Fokus auf Technologiekategorien
  - anschließend Auswahl konkreter Systeme

Integrationstechnologien

		Integrationspotenzial	geeignet für ex post Integration	Anmerkungen
statische Integration	RDBMS	mittel – hoch, begrenzt durch semantisches Potenzial relationaler Schemata	kaum, da interne Datenstrukturen geändert werden müssen	weit verbreitet, allerdings mit konzeptbedingten Schwächen; Potential nur bei Neuentwicklung auszuschöpfen
	DWH	hängt auch von der Semantik der zu integrierenden Daten ab	ja, allerdings erheblicher Aufwand	„aufgesetzte“ Integration, keine gemeinsame Datenhaltung
	Kommunikations-Middleware	hoch: gemeinsame Klassenbibliotheken	eingeschränkt, ggfs. mit Wrappern	Potential nur bei Neuentwicklung auszuschöpfen
funktionale bzw. OO-Integration	gemeinsame Schnittstellen	mittel, hängt im Einzelfall von der Semantik der Schnittstelle ab	ja, setzt allerdings Eingriff in Anwendungen voraus	bei Berücksichtigung geeigneter Standards gute Potentiale für unternehmensübergreifende Integration
	Funktionsbibliothek	hoch, hängt allerdings von der Semantik der Schnittstelle ab	kaum	auf anwendungsnaher Ebene kaum verfügbar
	Klassenbibliothek	hoch	kaum	auf anwendungsnaher Ebene kaum verfügbar
	Kommunikationsmiddleware	hoch: Unterstützung des verteilten Zugriffs, Transaktionsschutz	eingeschränkt, ggfs. mit Wrappern	Potential nur bei Neuentwicklung auszuschöpfen
dynamische Integration	Kommunikationsmiddleware	mittel: u.U. Unterstützung anwendungsübergr. Ereignisverwaltung; zentraler Controller zur Verwaltung von Ereignissen	eingeschränkt, ggfs. mit Wrappern	Potential nur bei Neuentwicklung auszuschöpfen
	WfMS	hoch: setzt allerdings elaboriertes Ereignis und Prozessschema voraus	eingeschränkt, nur auf geringem semantischem Niveau	z.Z. noch kein dominanter Standard zur Spezifikation von Workflows

Integrationstechnologien: Werkzeuge

	Unterstützungspotential	Erfolgsfaktoren	Anmerkungen
Modellierungswerkzeuge	hoch: hängt vor allem von den in der Modellierungssprache verwendeten Konzepte ab	leistungsfähige Analysen, Integration mit Integrationstechnologien (z.B. WfMS)	Fokus liegt vor allem auf der Modellierung von Prozessen; Entwicklung noch in vollem Gang; Standardisierung nicht in Sicht
Werkzeuge zur Erstellung/Pflege von Schnittstellenrepositories	hoch: allerdings nur bei Schnittstellenintegration in großen Systemen	Wahrung referentieller Integrität, Integration mit Integrationstechnologien (z.B. WfMS)	Nutzen hängt wesentlich von der Qualität der Datenpflege ab