SEBA

TUM WS19/20

Basierend auf IN2085

(Florian Matthes, Chair of Software Engineering for Business Information Systems)

Inhalt

Zusammenfassung

1: IT-Unterstützung betrieblicher Anwendungen

1.1 – Klassifikation betrieblicher Anwendungen

Was ist eine betriebliche Anwendung?
Eine betriebliche Anwendung ist im engeren Sinne die Gesamtheit aller Programme, d.h. die Anwendungssoftware, und die dazugehörigen Daten für ein konkretes betr. Anwendungsgebiet.

Betriebliche Anwendungen in Unternehmen
- Supporter: unterstützt bestehende betriebliche Prozesse
- Enabler: zur Umsetzung neuer Produkte & Ideen (Business Process Reengineering)

1.2 – Standard- und Individualsoftware

Individualsoftware
- Speziell für ein Unternehmen entwickelt & an Prozesse angepasst
- Individuell geplegt & an Veränderungen angepasst

Standardsoftware
- Für einen Markt entwickelt
- Standardgeschäftsprozesse, daher in vielen Unternehmen einsetzbar

Stamm- und Bewegungsdaten
- Stammdaten: über längere Zeiträume beständige und gespeicherte Daten
- Bewegungsdaten: Transaktionsdaten, Aufträge, etc. (sehr volatil)

Adaptionstechniken betrieblicher Standardsoftware (Customizing)
- Konfiguration: Einstellungen, die zur individuellen Variation der Software führen, obligatorisch
  > Herausforderung: Abbildung vielfältiger Unternehmenstrukturen & Prozesse
- Erweiterung: Software, die den Funktionsumfang erweitert (vom selben Hersteller)
- Kopplung: Anbindung von externen Systemen (in Form von Schnittstellen vordefiniert)
  > Middleware

1.3 – Charakteristika betrieblicher Anwendungen

Anforderungen und Stakeholder
- Anforderungen in Lasten- und Pflichtenheft definieren
- Modellierung zur Formalisierung und Dokumentation
- Aufwandsschätzung als Basis der Projektplanung (Preis des Codes?)
- Herausforderung: Change, Release, Version & Build Management

Daten Persistenz
- Daten machen großen Teil von Unternehmen(-swert) aus.
- Datenkonsitenz: parallele Zugriffe auf Daten, Daten dürfen nicht verloren gehen
- Herausforderungen: Netzwerk zur Bereitstellung, interne Datenverwaltung (Mapping)

Verteilung
- Zugriff auf Daten von verschiedenen Orten zu unterschiedlichen Zeiten
- Client-Server Architektur: Native Client, mittlerweile meist Web Clients
- Herausforderungen: Serialisierung & Bandbreite im Netzwerk, Authorisierte Zugriffe (Sicherheit), parallel Zugriffe

Integration
- Verschiedene Anwendungen auf gleicher Datenbank, Kommunikation über Middleware
- Herausforderung: Vermeidung von zu engler Kopplung (unflexibel), versch. Programmiersprachen

Skalierbarkeit
- Betr. Anwendungen können weltweit von Mitarbeitern genutzt werden (evtl. +Kunden)
- Unterschiedliche (saisonale) Auslastung
- Herausforderung: Load balancing, zeitversetzte Ausführung ressourcenintensiver Operationen

2: Requirements Engineering

2.1 – Grundlagen

Woran scheitern IT-Projekte?
Ein Großteil der Projekte wird nicht wie geplant abgeschlossen. Häufige Ursachen sind der fehlende Userinput & sich ändernde Anforderungen. (Software wird am Markt vorbei gebaut)

Requirementsengineering
- Teilprozesse: Anforderungsermittlung (Was möchte man?), Anforderungs- und Systemanalyse (Was ist möglich?)
- Requirement: Bedingung/Fähigkeit/Eigenschaft, die ein Stakeholder für ein Produkt fordert.
- Aufgabenbereiche des Requirementsengineering
  > Indentifizieren & erfassen (Elicitation)
  > Bewerten, priorisieren, konsolidieren
  > Dokumentieren, modellieren, strukturieren
  > Analysieren, validieren (bzgl. Inhalt und Darstellung des Artifakts)
- Aufgabenbereiche Requiremens Management
  > Änderungsmanagement
  > Änderungen verifizieren
- Ziele: Effiziente Erarbeitung einer qualitativ hochwertigen Anforderungs- und Systemspezifikation

Anforderungsklassifikation
- Funktionale Anforderungen: Interaktion zwischen System & Umgebung unabhängig von deren Realisierung
- Nicht-funktionale Anforderungen: Eigenschaften d. Systems, welche keine Interaktionen sind
- Beschränkungen (Pseudoanforderungen): Bestimmen Lösungsraum für Realisierung

Lastenheft
- Fachlisches Ergebnisdokument der Anforderungsermittlungsphase
- Entspricht dem „Fachkonzept"
- Anforderungsvalidierung
  > Gültigkeitsprüfung: Kann das System die Funktionen erbringen
  > Konsitenzprüfungen: Gibt es gegensätzliche Anforderungen
  > Vollständigkeitsprüfungen: Decken die Anforderungen alle gewünschten Funktionen ab
  > Realitätsprüfung: Können Anforderungen in Softwaresystem umgesetzt werden
  > Nachweisbarkeit: Sind Anforderungen konkret & nachweisbar dokumentiert

2.2 – Pflichtenheft

Pflichtenheft
Das Pflichtenheft beschreibt in konkreterer Form, wie der Auftragnehmer die Anforderungen aus dem Lastenheft umzusetzen gedenkt. Es dient als Basis für den Vertrag zwischen Auftragnehmer und Auftraggeber. Entspricht dem „IT-Konzept".

Aufbau & Inhalt eines Pflichtenheftes

1. Zielbestimmung
   > Muss-Kriterien, Wunschkriterien & Abgrenzungskriterien
2. Produkt-Einsatz
   > Anwendungsbereiche, Zielgruppen, Betriebsbedingungen
3. Produkt-Umgebung
   > Software, Hardware, Orgware, Produkt-Schnittstellen
4. Produkt-Funktionen
   > Definiert abstrakte Funktionalität aus Benutzersicht
   > Grafische/Schematische Darstellung: Anwendungsfalldiagramme, UI-Skizzen
5. Produkt-Daten
   > Beschreibung langfristig zu speichernder Daten aus Benutzersicht
6. Produkt-Leistungen
   > Zeit, Umfang, Benutzerzahl, Genauigkeit
7. Benutzeroberfläche (wenn gewünscht)
   > Mock-Ups, Screenshot aus ähnlichen Anwendungen
8. Qualitäts-Zielbestimmung
   > Messung d. Lösungsgüte (Ergebnosinterpretation)
9. Testszenarien
   > Abnahmetests, überprüfen fachlicher Korrektheit (gemäß Lastenheft)
10. Entwicklungsumgebung
    > Vorgeschriebene Entwicklungswerkzeuge, Code-Richtlinien
11. Ergänzungen
12. Glossar

3: Konzeptionelle Modellierung in UML

3.1 – Methodisches Vorgehen

Methodisches Vorgehen zum Erstellen von Klassendiagrammen

Modellierung: aus informeller Sprache funktionale & konzeptionelle Abstraktionen der Wirklichk.
Iterativer Prozess, bis Auftragenehmer & Auftraggeber zufrieden sind

- Finde neue Klasse, Attribute, Assoziationen
- Dokumentiere Klassen/Assoziationen
- Übersetze Ergebnis in Sprache des Auftraggebers
- Bespreche das Ergebnis mit dem Auftraggeber (intensiv) nach

Klassen finden

Abbot's Technik

1. Finde Substantive in informeller Beschreibung > pot. Klassen
2. Identifiziere Synonyme, Akronyme (Abkürzungen) und Homonyme (gleiche Wörter mit unterschiedlichen Bedeutungen)
3. Kategorisiere Substantive und streiche die, die keine eigenständigen konzeptuellen Klassen bezeichen > auf extra Liste sammeln (pot. Funktionen, Assoziationen)
4. Aussagefähige und verbindliche Klassennamen wählen Klassen dokumentieren (knapp)

Attribute finden

1. Identifiziere relevante Attribute für jede Klasse
2. Überprüfe Attributnamen (Substantiv, kein Homonym)
3. Definiere Typ der Attribute (komplexe Attribute durch eigenständige Klasse ersetzen)

Assoziationen finden

1. Suche nach Verben, die relevante Substantive verbinden, identifiziere beteiligte Klassen
2. Kategorisiere Assoziationen und streiche die irrelevanten & implementierungsbezogenen
3. Definiere Assoziations- und Rollennamen, Bestimme die Muliplizität
4. Klassifiziere Kompositionen
   > „besteht aus", Zugriff ausschließlich über Aggregat-Objekt, Lebensdauer durch Aggretat-Objekt beschränkt?
5. Dokumentiere Restriktionen z.B. *{ordered}, {derived as…}*

Konzeptionelle und Implentierungsnahe Klassendiagramme

	Konzeptionell	Implentierungsnah
Sichtbarkeit (public)	Nein	Ja
Datentypen	Ja	Ja
Methoden	Nein	Ja
Vererbung	Sparsam	Wenn sinnvoll
Abtrakte Klassen	Nein	Wenn sinnvoll
Assoziationsklassen	Ja	Nein (aufgelöst)

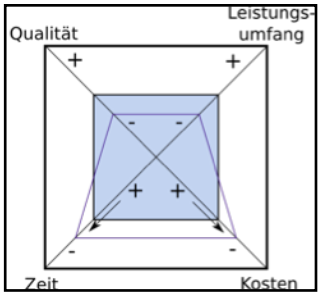
4: Aufwandsschätzung

4.1 – Übersicht

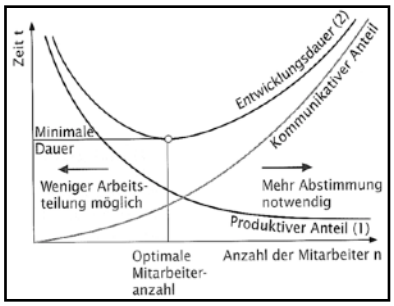
Aufwandsschätzung
Aufwände möglichst früh und möglichst genau vorhersagen. Die Schätzung wird während des Projekts kontinuierlich verbessert.

Teufelsquadrat
Produktivität (Fläche) bei gegebener Organisation und Ressourceneinsatz konstant.
- Qualität
- Quantität
- Entwicklungsdauer
- Kosten

bedingen einander.



Entwicklungsdauer und Mitarbeiterzahl
Mehr Mitarbeiter erhöhen den Communication-Overhead und beschleunigen nicht zwangläufig die Entwicklungsdauer.



Frühe Methoden zur Kosten- und Terminschätzung
- Ein Entwickler schafft X Lines of Code (LOC) pro Personenmonat
- Ein Personenjahr hat 10 Personenmonate
- Schätzung der benötigten LOC für ein Projekt in Verhältnis setzen

4.2 – Schätzungsmenthoden

Top-Down & Buttom-Up
- Top-down: Schätzung gesamter Projektaufwand mithilfer mathematischer Algorithmen auf Basis funktionaler Anforderungen (pref)
- Button-Up: Einzelen Aufwandsposten ermitteln und für das Gesamtprojekt aggregieren

Vergleichsmethoden
Mit Erfahrungswerten aus Vergangenheit informelle Schätzungen treffen.

Algorithmische Methode
Aus Vergangenheitswerten mittels Regression (algorithmische Methoden/Formeln) geschlossene Formeln erstellen.
Während des Projekts durch IST-Aufwand herleiten & verfeinern.

Kennzahlen Methode
Hochrechnung von einzelnen Leistungseinheiten auf gesamtes Projekt.

4.3 – Function-Point Methode

1. Katgegorisierung der Produktanforderungen
Kategorisierung der Anforderungen aus dem Lastenheft in Kategorien (Eingabe, [einfache] Abfrage, Ausgabe [komplexer], Datenbestände [intern], Referenzdaten [extern]).

2. Klassifizierung der Produktanforderungen
Komplexitätsklassifizierung (einfach, mittel, hoch) der Anforderungen.

3. Eintrag in Berechnungsformular
Eintrag von Anforderungen und ihrer Komplexität in Berechnungsformular.

4. Bewertung der Einflussfaktoren
Bewertung der Faktoren auf den Einfluss auf die Entwicklung des Gesamtprojekts (verschiedene Bewertungsmethoden)

5. Berechnung Function Points
Berechnung der bewerteten Functionspoints:

bewertete Function Points $= E_1 \cdot (E_2/100 + 0{,}7)$

$E_1 :=$ Summe der Kategorien $\quad E_2 :=$ Summe d. Einflussfaktoren

6. Ablesen des Aufwands in PM
In auf empirischen historischen Daten basierender Kurve den Aufwand in Personenmonaten ablesen.

7. Aktualisierung der empirischen Daten
Aktualisierung der empirischen Daten, um PM/FP Kurve weiter zu verbessern und zu aktualisieren.

> Daten hinzufügen (Datenbestand wird erhöht)
> Älteste Daten durch aktuelle ersetzen (Daten passen sich aktuellem Technologieniveau an)

Diskussion
Vorteile
- Hohe Anpassbarkeit
- Produktanforderungen als Ausgangspunkt

Nachteile
- Ausschließlich Schätzung des Gesamtaufwandes
- Starke Funktionsbezogenheit
- Mischung von Projekt- und Produkteigentschaften
- Viele nicht-rationale Bewertungsschritte > personalintensiv und nicht automatisierbar

5: Konfigurationsmanagement

5.1 – Übersicht

Konfigurationsmanagement
„Ein systematischer, werkzeuggestützter Ansatz zur Kontrolle der Evolution von Software in Entwicklung und Wartung"
Gewährleistet: Identifizierbarkeit, Kontrolle, Überprüfbarkeit, Vorhersagbarkeit, Abgleich zw. Dev.

5.2 – Säulen des Konfigurationsmanagements

Version Management
Verwaltet & lenkt alle Versionen von Programmen und Dokumenten.
- Jede identifizierbare Einheit ist ein Objekt (manuell erzeugt, oder daraus generiert) und muss versioniert werden
- Speicherung der Versionen
  > Differenzen zur vorherigen Version (speichereffizient, langsam)
  > Vollständige Speicherung aller Versionen (teurer speicher, schneller Zugriff), Bsp: GIT

Build Management
Erzeugt aus Programmen ablauffähige Software aus Programmquellen.
- Workflow: Kompilieren > Java-Dokumentation erstellen > Tests ausführen > Klassen, Bibliotheken & Dokumentationen in .jar Datei > .jar Datei auf Zielserver deployen
- Tools: ant, maven, etc. mit unterschiedlichen Produktionsketten für Entwicklung & Produktion

Release Management
Jedes Software-Paket, das an den Auftraggeber ausgeliefert wird, heißt Release.
- Stückliste (ausgelieferte Elemente) + Anleitung (zur Inbetriebname) = Konfiguration
- Verschiedene Varianten (Zielumgebung, Sprache, Layouts/Logos bei mandantenfähiger Softw.)

Change Management
Workflow: Erfassung > Priorisierung > Zuordnung des Releases.

5.3 – Integration von Software- und Datenevolution

Typische Probleme
- Hohe Frequenz von Änderungen
- Hohe Abhängigkeit von anderen Systemen (speziell in verteilten Anwendungsbereichen)
- Datenkompatibilität mit neuen Versionen
- Migration von jeder Version zu jeder anderen nötig

Testen von Änderungen
Änderungen müssen produktionsnah getestet werden.

6: Technische Grundlagen von betr. IS

6.1 – Schichtenarchitekturen

Schichtenarchitekturen

Schichtenarchitekturen dienen der Partitionierung der einzelnen Komponenten und so auch der Verringerung der Komplexität.

- Innerhalb der Schichten: hohe Kohäsion
- Zwischen den Schichten: niedrige Kopplung

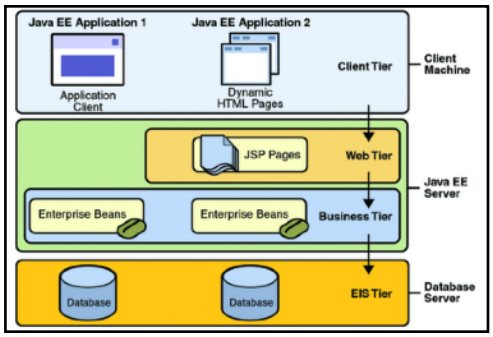
Arten von Schichtenarchitekturen

Eine Schicht darf niemals auf eine über ihr liegende Schicht zugreifen.

- strikt: eine Schicht darf nur auf die direkt unter ihr liegende Schicht zugreifen
- offen: eine Schicht darf auf alle unter ihr liegenden Schichten zugreifen

Schichtenarchitekturen in betr. Anwendungsystemen

1. Präsentationsschicht
2. Geschäftslogikschicht
3. Dartenbankschicht



Java Enterprise Edition

JEE = Web Application Techn., Enterprise Application Techn., Web Service Techn. (Middleware > RPC, etc.)

- WebClient = thin client, keine lokalen Daten/Software
- Application = thick client, lokale Software & Daten

6.2 – Verteilte Softwaresysteme

Web Server

Web Server stellen statisch/dynamisch generierte Inhalte bereit.

> Aufgaben: Ressourcen Management, Zugriffsbeschränkung, Caching, Skriptausführung

Anwendungsserver

Anwendungsserver ermöglichen Methodenaufrufe.

> Aufgaben: Nachrichtenversand (Messaging), Authtifizierung, Transaktionshandling, DB-Interfaces

Datenbankserver

- Software: Datenbanksoftware für Anfragen und Administration
- Hardware: Speicherung der relationalen Datenbank
- Client: Interface für Abfragen

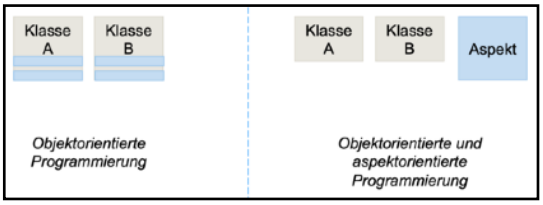
6.3 – Bibliotheken und Frameworks

Bibliothek/Library
Eine Bibliothek ist eine wiederverwendware Softwarekomponente, derren Funktionen durch das vom Programmierer definierte Programm über eine API genutzt werden können.
> API (Application Programming Interface, Anwendungsprogrammierschnittstelle)
- Programmierer nutzt Code der Bibliothek

Framework
Ein Framework ist ein halbfertiges Sofrwaresystem, das aus bereits aufeinander abgestimmten Softwarekomponenten besteht. Die Verarbeitungslogik sowie die grobe Architektur wird durch das Framework bestimmt.
- Framework benutzt Code vom Programmierer (Inversion of Control)
  > Dependencies injection: Minimierung der entstandenen statischen Abhängigkeiten via deployment description

6.4 – Aspektorientierte Programmierung

Aspektorientierte Programmierung
Bei der AOP werden generische Funktionalitäten, die über mehrere Klassen verwendet werden, einmalig in Aspekten implementiert.




```
public aspect Logging {
      pointcut toBeLogged() :
            execution (public * de.tum.sebis.*.*(..));
      before(): toBeLogged() {
            System.out.println(„Before another Method execution“);
      }
}
```

„Wann immer eine public-Methode aus dem Package de.tum.sebis aufgerufen wird, soll dieser Aufruf auf der Konsole ausgegeben werden.“
- Aspect: fasst Pointcuts und Advices zusammen.
- Advice: beinhaltet den auszuführenden Code
- Pointcut: beinhaltet einen/mehrere Joinpoints
- Interceptors: unterbrechen Programmablauf, bzw. definieren, an welcher Stelle des Pointcuts der Advice ausgeführt werden soll
- Joinpoints: gibt Stelle an, an der Interceptor ausgeführt werden soll
  > AspectJ: Methodenausführung, Konstruktoraufruf, Field Access, Exeption

Annotationen
Pattern-Matching über Pointcuts ist nicht immer mächtig genug > Annotationen als Joinpoints.
Definition von Annotations:

```
@Retention(RetentionPolicy.(RUNTIME|CLASS|SOURCE))  //wann Auswertung
@Target(ElementType.(METHOD|CLASS|FIELD))  //worauf angewendet
public @interface PrintLog {  //leerer Methodenkörper
}
```

Reflexion (Reflection)

Zugriff des Programms auf Informationen, die nicht zu den Daten, sondern zu der Struktur des Programmes gehören.

```
public class SebaExamGenerator {
	public String genExcercise() {
		return "Aufgabe 1";
	}
}
public class ReflectionTest {
	public String genExcerciseReflection() {
		SebaExamGenerator obj = new SebaExamGenerator();
		return SebaExamGenerator.class.getMethod(„genExcercise")
			.invoke(obj);
	}
}
```

Serialialisierbarkeit von Objekten

Objekt wird in Byte-Stream übersetzt, um es zwischen zwei Systemen auszutauschen. Das andere System erhält eine Kopie, keine Referenz.

- Ausgenommen von der Serialisierung sind: Thread, IOStreams, Sockets, etc.

7: Verteilung

7.1 – Überblick

Historie
- Zentraler Mainframe
- Verteiltes System

Verteilte Systeme
System mit mehreren Prozessräumen, die Informationen mit Hilfe von Nachrichten austauschen. Subsysteme kooperiern koordieniert, um gemeinsame Aufgaben zu lösen.
- Charakteristika: Ressourcenteilung, Nebenläufigkeit, Skalierbarkeit, Fehlertoleranz, Transparenz, Offene Verteilte Systeme (offene Kommunikationsprotokolle)
- Evaluation:
  +) Ressourcenteilung, Redundanz (Fehlertoleranz), Parallelisierung
  –) höhere Komplexität, Sicherheit, heterogene Soft-/Hardware Umgebung, Performance

7.2 – Kommunikationsformen

Kommunikation über gemeinsamen Speicher
- Blackboard-Architektur: gemeinsamer Speicher entspricht „schwarzem Brett“
- Speicher ist indirekte Verbindung zwischen Sender(-) und Empfänger(-prozess)

Ereignisbasierte Kommunikation
- Ereignis: autonome, asynchrone Begebenheit
  > Erreichen eines Zeitpunktes, Eintritt eines Objektes in einen spez. Zustand
- Push-Modell: Client abonniert Ereignitstypen, für die er sich interessiert

Beispiel: Datenbank-Trigger
- (Event, Condition, Action) > create/update/delete
- Anwendung: erzwingen von Integritätsbedingungen, Auditing von DB-Aktionen, Propagieren von DB-Änderungen

Kommunikation mit Messaging
- Senderprozess: erzeugt Nachricht, kodiert & adressiert, versendet
- Empfängerprozess: dekodiert Nachricht

Klassifikation
- Unicasting (Punkt-zu-Punkt), Multicasting (an abgegrenzte Gruppe), Broadcasting (an alle)
- Zuverlässig (alle Pakete kommen in richtiger Reihenfolge an), Unzuverlässig (keine Garantie)
- Paket-orientiert (Nachrichten in Paketstruktur), Bytestrom-orientiert (kontinuierlicher Bytestrom)
- Null-Kapazität (keine Nachrichten in Verbindung gepuffter, Rendesvous-Synchronisation nötig; synchron), Beschränkte Kapazität (n Nachrichten können warten; asynchron), Unbeschränkte Kommunikation (∞ Queue; asynchron)
- Direkte Adressierung (Rechnername & eindeutige Prozessnummer), Indirekte Adressierung (Kommunikation über Ports als Kommunikationsendpunkte)

`java.net.Socket` & `java.net.ServerSocket` können zur byte-orientierten Kommunikation genutzt werden.

Nachrichtenformate

- Binär-Nachricht: (+) wenig overhead, direkte codierung; (–) proprietäre/keine standardisierte codierung
- ASCII-Nachricht: (+) standardisiert, einfache Struktur; (–) kein Typsystem
- XML-Nachricht: (+) menschenlesbar, eindeutige Codierung, mächtiges Typsystem; (–) hierarchische Struktur, großer Overhead
- SOAP-Nachricht: (+) menschenlesbar, eindeutige Codierung, Unterstützung von Metainformationen, Verschlüsselung; (–) hierarchische Struktur, großer Overhead

Remote Procedure Call

Clients rufen entfernte Methoden (auf anderem Server) wie lokale Methoden auf.

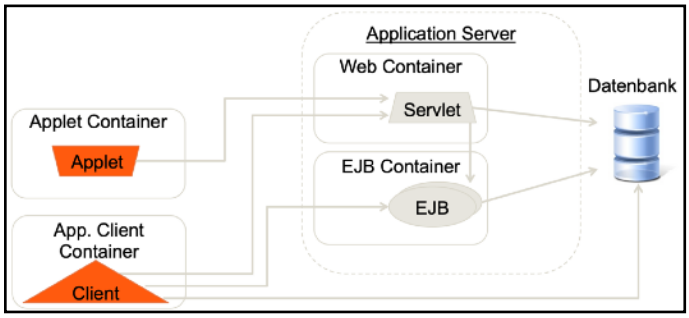
- (Un-)Marshalling: Konvertierung von/zu Objekten von/zu serialisierten Nachrichten
- Proxy und Skeleton als Interfaces, auf denen die Methoden aufgerufen werden

Aufrufsemantik

- Maybe $[1] \rightarrow [0,1]$: einmal senden
- At least one $[1,\infty)] \rightarrow [1,\infty)$: timeout, eventuell wiederholtes senden
- At most once $[1] \rightarrow [0,1]$: einmal senden
- Exactly-once $[1,\infty) \rightarrow [1]$: senden, server tracked eingehende nachrichten, timeout, wiederholtes senden (mit Prüfung nach doppelter Verarbeitung)

Kommunikation mit entfernten Komponenten

Java EE unterstützt Entwickler bei Entwicklung von mehrschichtigen, verteilten Anwendungen.



- Java EE Container (Application Server): Laufzeitumgebung für Java EE Anwendungen & mit Java EE API, Pooling von Systemressourcen, Zustandsverwaltung
- EJB Beans: Sessions Beans (synchron, asynchron)
  - > Stateful: mit Client assoziiert, innere Zustand über gesamte Client Session, Activation & Passivation
  - > Stateless: kann mehrere Client bedienen, innere Zustand über Funktionsaufruf
  - > Singleton: existiert nur einmal pro Anwendung, kann mehrere Client bedienen, innerer Zustand bis Shutdown/Crash bestehend
- EJB Beans: Message Driven Beans (asynchron): verarbeiten externe Nachrichten aus Warteschlange, kein eigener innerer Zustand (Filter im Pipe-Filter-Pattern)

Clients greifen über Schnittstelle (Interfaces, von Implementierung des Beans implementiert) des EJB-Containers auf Beans zu.

- Remote Interface: kann über RMI von überall aufgerufen werden, Parameter serializable
- Local Interface: nur von Komponenten in derselben Deployment-Einheit verwendet

```
@Stateless
@Remote{(HelloWorld.class)}
@RemoteBinding(name = „HelloWorld“)
public class HelloWorldBean implements HelloWorld {
  public String getHelloWorld() { return „Hello World“; }
}
```

Remote Interface Adressieren

Zugriff erfolgt über Java Naming & Directory Interface (& JNDI-Namen).

```
try {
  javax.naming.Context ctx = new javac.naming.InitialContext();
  HelloWorld ejb = (HelloWorld) cts.lookup(„HelloWorld“);
} catch (javax.naming.NamingException ne) { /* exception handling */ }
```

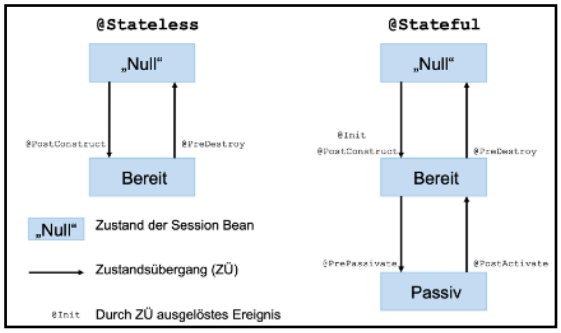
- Binding: Assoziation zwischen Objekt und Namen
- Context: Menge von Bindungen, kann Namen auflösen (→ erhalt des Objekts)
- Naming System: Menge von Context-Objekten
- Naming Service: stellt Operationen auf Naming System zur Verfügung

Verbindungsaufbau

1. Server erzeugt Interface Implementation
2. Server registriert Resource beim Naming- Service
3. Client fragt via JNDI beim Naming Service an und erhält Referenz

Lebenszyklus von Session Beans

- Stateless (für mehrere Clients): immer bereit
- Stateful (eine Client-Session): Kann Zustand ändern (Aktivierung & Passivierung)



Pooled Stateless Session Beans

Die immer bereiten Stateless Session Beans können zu Performance Problemen führen.
Lösung: Pool mit maximaler Anzahl Stateless Beans.

Namenskonventionen

Element	Name
Remote Interface	[Bean-Name]
Local Interface	[Bean-Name]Local
Bean-Klasse	[Bean-Name]Bean

7.3 – Namens- und Verzeichnisdienste

Überblick

- Namensdienst: Vergabe von Namen, Zuordnung Namen ↔ Ressourcen
- Vereinheitlichung: Gleiches Schema für lokale & entfernte Dienste
- Integration: Zusammenfügen von Namensräumen
- Namensdomäne: von einem Namensdienst verwalteter Namensraum
- Verzeichnisdienst: Auflösen von Namensreferenzen
- Verwaltung: Verteilen/Aufteilen, Replizieren

7.4 – Verteilte Architekturen

Tier Architektur

2: Client-Server 3: Präsentation, Anwendung, Datenhaltung n: verteilte Anwendung

UML zur Architekturkommunikation

- Komponentendiagramme: Komponenten, Schnittstellen, Kommunikationsbeziehungen
- Interatkionsdiagramm: Nachrichten & Kontrollfluss
- Verteilungsdiagramm (physische Struktur): phys. Geräte, Artefakte

---

7.5 – Entkopplung von Modulen mit Java

Java Messaging Services

Loose gekoppelte verteilte Kommunikation durch Austausch von Nachrichten zwischen Softwarekomponenten.

- asynchron: zeitlich versetztes Verschicken/Empfangen
- synchron: blockieren der Kommunikationspartner beim Verschicken/Empfangen von Nachrichten

Message Driven Beans (MDB)

Bean als verteiltes asynchron handelndes Objekt, das Nachrichten aus Warteschlange entnimmt und weiterverarbeitet (*keine* Remote Schnittstelle!).

- Evaluation: (+) Nebenläufigkeit, Mircoservice mit unabhängigem Deployment; (–) nur eine assoziierte Warteschlange

```
@MessageDriven(activationConfig = { @ActivationConfigProperty(
propertyName = "destinationType", propertyValue =„javax.jms.Queue"),
@ActivationConfigProperty( propertyName = "destination",
  propertyValue ="seba/jms/myQueue")
})
public class StapelDruck implements MessageListener {
  public StapelDruck() { }
  // MessageListener implementierung
  public void onMessage(Message message) {
    // process message
  }
}
```

8: Persistenz

8.1 – Persistente Datenspeicher

Persistente Datenspeicher

- Dateisystem: Daten können mittles I/O API geschrieben werden
  > (+) Verfügbarkeit; (–) Skalierbarkeit, kein paralleler Zugriff, indifferente Codierung, starr
  > XML: (+) plattformunabhängig, maschinen- & menschenlesbar, Unterstützung von DBMS, (–) im Detail komplizierter, keine Referenzen unterstützt, hierarchisch, kein paralleler Zugriff
- Relationale/Native XML/Objektorientierte/NoSQL Datenbank
  > Relational: Persistente Datenhaltung, paralleler Zugriff, Massendaten, Integrität, Recovery häufige, aber kleine Transaktionen, seltene Schreibzugriffe
  > NoSQL: nicht relationales System, verteilte & horizontale Skalierbarkeit Indexierung von großen Datenmengen, Websites mit hohem Lastaufkommen
- Content Management System: mehrere Benutzer bearbeiten und speichern gemeinsam Content, Entkopplung von Information und Repräsentation

NoSQL Systeme

Kategorisierung

- NoSQL-Kernsysteme: Wide Column Stores, Document Stores, Key/Value/Tuple Stores
- Nachgelagerte NoSQL-Systeme: Objekt/XML-/Grid-Datenbanken

Wann sollte man NoSQL-Systeme einsetzen?

- Viele Schreib- und Leseanfragen
- Schwache Konsistenzanforderungen
- Verteilte Datenbanken mit redundanter Datenhaltung (Ausfallsicherheit)

8.2 – Zugriff auf persitente Datenspeicher

Zugriffsstrategien

- Direkte SQL-Aufrufe: aus Programmiersprache heraus direkte SQL-Befehle
  > (+) günstig; (–) schwierige Entwicklung, starr (nachträgliche Änderung schwierig)
- Objektrelationales Mapping: automatisierter Zugriff auf relationalen Speicher, mapping in Objekte (javax.persistent)

Relationale Abbildung von Vererbungshierarchie

- Single Table Strategy: Eine Relation mit allen Unterklassen & null für nicht definierte Attribute
- Table per Class: für jede Unter und Oberklasse einzelne Relation mit allen Attributen
- Joined Table Strategy: für jede Unterklasse eine Relation mit zusätzlichen Attirbuten und foreign key ↔ primary key zur Oberklassenrelation mit entsprechenden Attributen

8.3 – Persistent Entities

Persistent Entities

Persistent Entities bietet objektorientierten Zugriff auf die persistenten Informationen in der Datenbank.

Entity

- Entity hat getter/setter, default Konstruktor, implementiert Serializable
- Annotation der Entity (@Entity) und des Primärschlüssels (@Id)
- Optional: Generierung des Primärschlüssels (@GeneratedValue), Auswahl Tabelle (@Table) & Spalte (@Column), Annotations am zugehörigen ‚getter'

```
@Entity @Table(name=„Pet“)
public class Pet implements java.io.Serializable {
  private int id;
  private String name;
  @Id @GeneratedValue  // jeweils an getter
  public int getId() { return this.id; }
  public void setId(int id) { this.id = id; }
  @Column(name = „tierName“)
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
}
```

- Vererbung (Strategie)

```
@Inheritance(strategy=InheritanceType
  .(SINGLE_TABLE|TABLE_PER_CLASS|JOINED))
// bei Oberklasse
```

- Assoziationen:

> One-To-One

```
@OneToOne(mappedBy=„car“)  //in Licenseplate
public Car getCar() { return this.car; }
@OneToOne(optional=false)  //in Car
public LicensePlate getLicensePlate() { return this.plate; }
```

> One-To-Many

```
@ManyToOne  //in Car
public owner getOwner() { return this.owner; }
@OneToMany(mappedBy=„car“,cascade=CascadeType.REMOVE,
  fetch=FetchType.EAGER)  //in Owner
public Collection<Car> getCars() { return this.cars; }
```

> Many-To-Many

```
@ManyToMany(mappedBy=„student“,fetchType=fetch.EAGER) //in University
public Collection<Student> getStudents() { return this.students; }
@ManyToMany(mappedBy=„university“,fetchType=fetch.LAZY) //in Student
public Collection<University> getUnis() { return this.unis; }
```

Achtung: `fetch.EAGER` lädt transitive Hülle.

EntityManager

Zum Zugriff auf die Entitites benötigt man

- EntityManager: erstellen, finden & löschen von Entities
- Getter & Setter: lesen und schreiben der Attributwerte von Entities

> Speichern & löschen von Objekten auf der Datenbank:

```
void persist(Object o)  void remove(Object o)
```

> Finden von Objekten mit Key

```
Object find(Class clazz, Object primaryKey)
```

> Objekte in Datenbank verändern

```
void merge(Object o)
```

Querying mit JPQL

Queries werden parametrisiert, um SQL-Injections zu verhindern.

```
Query q = em.createQuery(„select * from Cat c where c.lives = ?1“);
q.setParameter(1, livecount);
q.getResultList();  q.getSingleResult();
```

8.4 – Criteria API

Nachteile von JPQL

- Weitere Sprache (SQL) nötig
- Compiler kann Anfrage nicht prüfen

6 Schritte des Criteria API

1. Erstelle ein CriteriaBuilder und CriteriaQuery Objekt
```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<RetType> cquery = cb.createQuery(Type.class);
```
2. Konfiguriere die FROM Klausel
```
Root<RetType> root = cquery.from(Type.class);
```
3. Konfiguriere die SELECT Klausel
```
cquery.select(root);
```
4. Konfiguriere Prädikate
```
Predicate pLtAge = cb.lt(root.get(„age“), 10);
Predicate pGtAge = cb.gt(root.get(„age“), 2);
Predicate combined = cb.and(pGtAge, pLtAge);
```
5. Kofiguriere die WHERE Klausel aus Prädikaten
```
cquery.where(combined);
```
6. Führe die Query aus
```
Query q = em.createQuery(cquery);
List<Type> res = q.getResultList();
```

9: Gastvortrag

9.1 – Definitionen

Software Architektur
- Technisches Konzept hinter dem System
- Fundament für Projektenwicklung mit den abstrakten Entscheidungen *Rough Up-Front Design*
- Architektur ist eine These, die durch Implementierung bewiesen werden muss, besser öfter „beweisen" > Agilität

Wer braucht Software Architektur?
- Jedes System hat Architektur - also sollte man sie besser planen.

Wer ist ein Software Architekt?
- Jeder Entwickler ist ein Architekt, Architekt trägt die Verantwortung für das System

Architect as a Job – Architect as a Role – Architect as a Grid

Beyond Architectus Oryzus
- Fokus auf Koordianation der Architektur der Middleware zwischen Microservices
- Auch verteiltes Microservice System benötigt Architektur

Architekur im 21. Jahrhundert
- Agile Projekte sind deutlich erfolgreicher.
- Architektur muss DevOps Kompatible sein (IOT)
- UE ist sehr wichtig

9.2 – Architekten

Architectus Relodus
- Kontrolliert alles (Wasserfallmodell)

Architectus Oryzus
- Nicht viel Macht, aber alles Wissen > als Berater
- Unterstützt Team bei Entscheidungen, entscheidet aber nicht selbst (Agil)

Architectus Connexus

10: Betrieb & Wartung

10.1 – Einordnung im Softwarelebenszyklus

Ziele beim Betrieb und Infrastrukturmanagement

Problemsituationen durch geplantes Vorgehen beim Betrieb verhindern.

Kosten- und Nutzenaspekte im Softwarelebenszyklus

⅔ der Kosten entstehen erst während des Betriebs. Während des gesamten Einsatzes wird das System dauerhaft gewartet (normaler Deployment Zyklus). Dabei sollten die Kosten möglichst gering gehalten werden, um das Budget für Neuentwicklungen nicht zu Verdrängen.

10.2 – Ziele, Aktivitäten und Herausforderungen in der Wartung

Kategorien der Wartung

	Korrektur	Verbesserung
Reaktiv	Korrektive Wartung (Notfallwartung)	Adaptive Wartung
Proaktiv	Präventive Wartung	Perfektionierende Wartung

Verbesserung

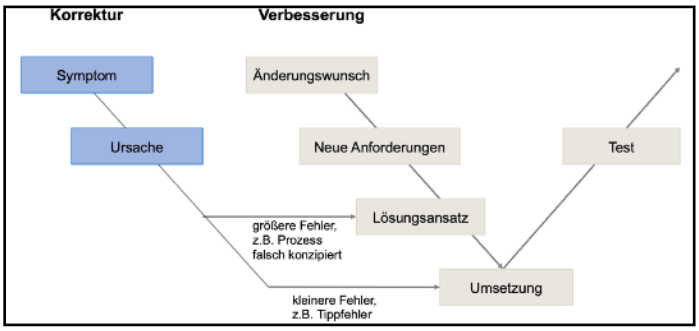
Verbesserung über formalisierte Verfahren.

- Kombinierter Ansatz: Mitarbeiter sind für Entwicklung & Wartung zuständig
  > (+) Kenntnis über die Software, sorgfältigere Entwicklung; (–) Zeitaufwändige Perfektionieru.
- Separierter Ansatz: Trennung von Entwicklungs- & Wartungsaktivitäten
  > (+) einfacheres Zeitmanagement, bessere Dokumentation; (–) Lernaufwand, Kommunikationsaufwand, gute Dokumentation nötig - sonst langsam

Softwarehaltungsaktivitäten

- Fehlerkorrektur
- Änderung: vorhandene Bausteine ändern, Architektur bleibt bestehen
- Erweiterung: neue Funktionalität schaffen, Architektur kann verändert werden
- Sanierung: groß angelegte Änderung, Wechsel von Datenbank/Programmsprache/etc.
- Optimierung: fortlaufende Verbesserung von Code & Stabilität
- Management: Changemanagement

Grundablauf des Wartungsprozesses



Wartungsprozess

- Wartungsfall erfassen: Release, Umstände, Ansprechpartner, Reproduzierbarkeit
- (Erste Klassifizierung: sicherheitsrelevant, bis zur nächsten Nachbesserung aufschiebbar, besondere Ressourcen zur Behebung des Fehlers nötig)
- Analyse der Ursache: Anforderungen beim lösen
- Analyse der Lösungsansätze: Änderung Architektur/Code, Aufwandsschätzung
  > Auswirkungen im Code: Änderung Erfolgreich? Regressionstest?
  > Auswirkungen auf Lieferung: Verzögerung bei Update/Upgrade/Vollauslieferung
- Entscheidung Change Control Board (CCB): Budget, Zeitplan, Lösungsoption, Bearbeiten?
- Entwerfen & Implementieren, Release erstellen
- Testen: Vermerk über Erfolg des Tests, Dokumentation anpassen
- Analyse der Auswirkungen: Bewertung, ob prognostizierte Auswirkungen eingetroffen sind
- Fehlermeldung ändern/schließen
- Liefern (Deployment):

10.3 – ITIL ° serviceorientierte Perspektive auf den Betrieb

IT Infrastructure Library

Sammlung vordefinierter Prozesse, Funktionen & Rollen für den Lebenszyklus von IT-Services.

Wartungsprozess in ITIL

- Erfassen: Fehler = Incident, Änderung: Change Request
- (erste Klassifizierung: nach Auswirkung (Impact) (Anzahl Betroffene, Abteilung, Ort), Dringlichkeit (Urgency) (Zeitraum, bis ein Lösung gefunden werdne muss) > 1st Level Support)
- Vorläufige Klassifizierung: Incident oder Service Request? > Incident Manager
- Lösungsansätze finden > Supporteinheit
- Entscheidung CCB > Incident Manager
- Entwerfen & Implementieren > Supporteinheit
- Fehlermeldung ändern/schließen > 1st Level Support

Service Level Agreements

Verbindliche Vereinbarungen zwischen Kunden und Servicebereicht, definiert Leistungen.
> Vertragspartner, Zeitplan, Verfügbarkeit, Verlässlichkeit, Support

10.4 – Systembetrieb

Gefahren für den Betrieb

- Mensch, Hardware, Software, äußere Einflüsse

10.5 – Neuere Einflüsse auf Betrieb & Wartung

Einflüsse

- Cloud: Verlagerung unterschiedlicher Betriebsaufgaben von eigener IT an externe Anbieter.
- Monitoring: Kontinuierliches Logging und Beobachten des Systems, passives Feedback
- CI, CD: jederzeit stabile & automatische Releases; Best Practices

11: Geschäftsprozesse

11.1 – Geschäftsprozesse

Geschäftsprozesse

Geschäftsprozesse sind Netzwerke von Aktivitäten, die nach einer definierten Reihenfolge (von unterschiedlichen Akteuren) ausgeführt werden. Dabei verarbeiten sie *Inputs* mit Hilfe bestimmter *Ressourcen* zu den gewünschten *Ouputs*.

Klassifikation von Geschäftsprozessen

	Kollaborativ	Produktion
Wert für Unternehemen	Ad hoc	Administrativ

Wiederholung

- Administrative Prozesse: einfache Koordinationsregeln, oft wiederholende und vorhersagbar
  > Budgetplanung, Genehmigung von Dokumenten
- Produktions Prozesse: Geschäftsprozesse mit komplizierter Informationsverarbeitung, komplexe Automatisierung
- Kollaborative Prozesse: selten aber wichtig
  > Software Entwicklung, Problemlösung, Anfertigung von technischen Berichten
- Ad hoc Prozesse: Koordination durch Menschen (empirisch), einmalige Prozesse

11.2 – Normative & Adaptive Prozesse

Normative Prozesse

Normative Prozesse sind starre definierte Prozesse, die sich gut in aktivitäts-zentrierten Sprachen wie BPMN und EPK modellieren lassen.

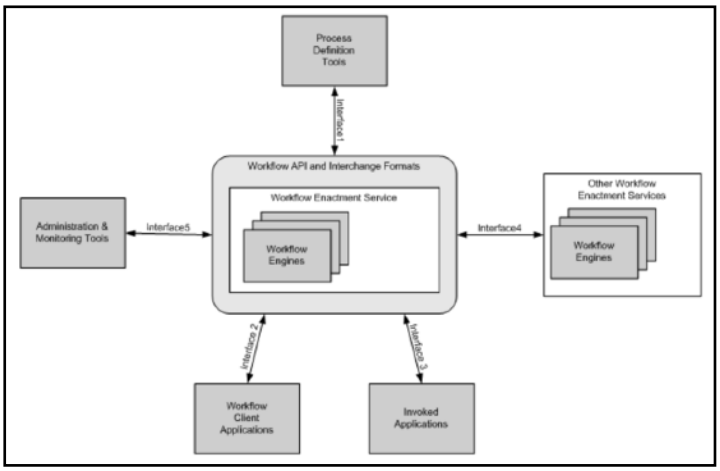
Workflows für Normative Prozesse

Explizite Repräsentation von Abläufen in Workflow Management Systems.

- Definition: legt über ein Modell fest, wie die Ausführung eines Prozesses ablaufen soll
- Instanz: konkrete Ausführung des Workflows
- Geschäftsprozess: Prozesse innerhalb von Wirtschaftseinheiten, die einen wesentlichen Beitrag zum Unternehmenserfolg leisten. I.d.R. funktions-, hierarchie- und standortübergreifend
- Workflow: teilweise automatisierte gesteuerte Gesamtheit von Aktivitäten, die sich auf organisationale Vorgänge bezieht. Sie können in Subworkflows zerlegt werden und haben einen definierten Anfang, einen organisierten Ablauf und ein definiertes Ende.
- Workflowschema: Modellierungssatz zum beschreiben von Workflows
- Workflow-Instanz: konkrete Modellierung eines Workflows
- Workflow-Anwendung: integrierte Anwendung, die durch Workflowschemata realisiert wurde

Eigenschaften von WFMS

- Definitionsfunktionalität: (meist graphische) Definition des Workflows
- Laufzeitfunktionalität: Initiierung, Kontrolle, Koordination d. Ausführung d. Workflow-Instanzen
  > Workflow Engine zur Ausführung, Analyse & Überwachung
- Schnittstellenfunktionalität: Schnittstelle für Benutzer zu Anwendungsprogrammen, mit denen konkrete Aktivitäten umgesetzt werden



Workflow Engine

- Process Definition Interface: Austausch von Workflow Definitionen
- Client Application Interface: Interaktion mit Applikationen, bei denen menschliche Interaktion notwendig ist
- Invoked Applications Interface: direkte Interaktion mit Applikationen
- Interoperability Abstract Specification: Interaktion mit anderen Workflow Engines
- Audit Data Specification: Administration, Monitoring & Analyse

Adaptive Prozesse

Adaptive Prozesse sind wissensintensiv und dynamisch in ihrer Ausführung
> Adaption zur Laufzeit nötig, *Loosely-specified Process*

11.3 – Sichten auf Prozesse im IS

Operationale Sicht

- Implementierung der zugrundeliegenden Geschäftsfunktionen (aktivitäts-zentriert)
- Formulare und Suchfelder der Prozesse (daten-zentriert)

Organisationssicht

- Zuweisungen von menschlichen Aktivitäten auf vorhandene Ressourcen (Organisationsmodell)
  > Akteure, Rollen, Organisationseinheiten

Zeitliche Sicht

- Zeitliche Regeln, die während der Prozessausführung eingehalten werden müssen

Funktionssicht

- Funktionale Bausteine, die zusammengesetzt den Prozess bilden
  > Atomare Aktivitäten: kleinstmögliche Arbeitseinheit
  > Komplexe Aktivitäten: bilden hierarchische Strukuren in Prozessen ab

Verhaltenssicht

- Betrachtung des dynamischen Verhaltens des Prozesses
  > aktivitäts-zentriert: Kontrollfluss zwischen Aktivitäten
  > daten-zentriert: Verhalten der Objekte mit ihren Interaktionen

Informationssicht

> aktivitäts-zentriert: Datenobjekte & deren Datenfluss
> daten-zentriert: Objekttypen, Attribute und Beziehungen (siehe Verhaltenssicht)

–