Traditionelle Vorgehensmodelle und OO-Konzepte

Traditionelle Vorgehensmodelle

- Wasserfall-Modell
  - sequentielle Durchführung sämtlicher Arbeitsschritte mit Rückschritten zur vorherigen Phase
- V-Modell
  - Entwicklungsstandard für IT-Systeme des Bundes, Schwerpunkt auf Tests
- Spiral-Modell
  - evolutionäres, risikoorientiertes Vorgehensmodell mit Zyklen
- Prototyping
  - horizontal: aus der Gesamtmenge werden alle/viele Funktionen unvollständig implementiert
  - vertikal: aus der Gesamtmenge aller Funktionen werden einzelne Funktionen vollständig implementiert
  - Formen: Demonstrations-, Anforderungs-, Wegwerf-Prototyping

Objektorientierte Konzepte

- Klassen/Objekte
  - Klasse: Struktur + Verhalten einer Gruppe gleichartiger Objekte
  - Objekt: konkrete Instanz; besitzt eine Identität, einen Zustand und ein Verhalten
- Kapselung
  - Information Hiding, Zustandsänderung nur durch Operationen, Kommunikation durch Nachrichten
- Schnittstellen
  - Vertrag zwischen Klassen und Benutzer, repräsentiert Teilmenge der Objekt-Funktionalität
- Vererbung
  - Generalisierung/Spezialisierung, Mehrfachvererbung, abstrakte Klassen
- Assoziation/Aggregation
  - Verweis- und Enthaltenseinbeziehung, Komposition als existenzabhängige Aggregation
- Polymorphismus
  - je nach Kontext (Klasse) verhält sich eine Operation unterschiedlich
- Binden/Bindung
  - early binding/statische Bindung: Festlegung der Einsprungsadresse zur Compilezeit
  - late binding/dynamische Bindung: Ermittlung der korrekten Einsprungsadresse erst zur Laufzeit, besonders für Polymorphismus benötigt
- Events
  - Ereignisse, die in der Umgebung von Objekten geschehen
- Ausnahmebehandlung
  - Entdeckung und Behandlung von unvorhergesehenen Zuständen
- Pakete
  - Zusammenfassung von Komponenten zu einer Einheit mit gleichem Namensraum
- objektorientierte Analyse
  - Identifikation der Stakeholder, Erhebung der Anforderungen, Erstellung des OOA-Modells
- objektorientiertes Design
  - Weiterentwicklung des OOA-Modells, Erstellung des OO-Systemmodells, nächste Phase: OOP

UML

Diagrammtypen

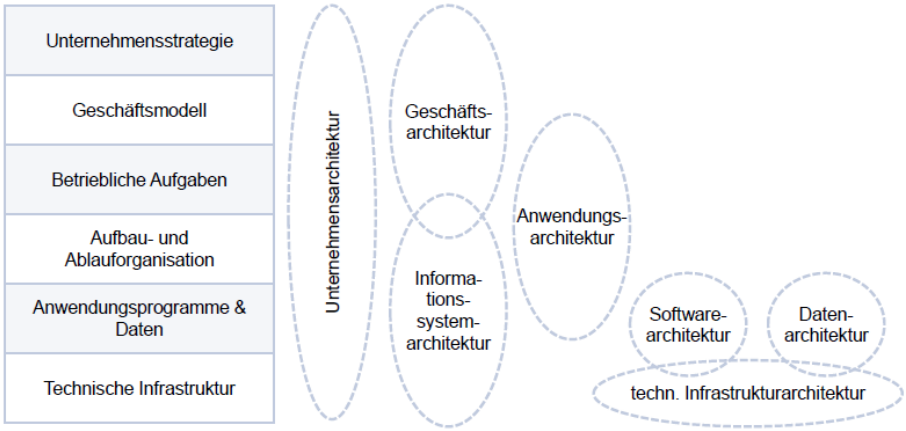
- Anwendungsfalldiagramm
  - zeigt Akteure, Anwendungsfälle und ihre Beziehungen
  - Schwerpunkt liegt bei der Ausführung auf dem „was“ und nicht auf dem „wie“
- Klassendiagramm
  - zeigt Klassen und ihre Beziehungen zueinander
  - Sichtweisen: konzeptionell (ohne Details), spezifizierend, implementierend
- Implementierungsdiagramme
  - Komponentendiagramm
    - zeigt Komponenten und ihre Beziehungen
    - Komponente repräsentiert ein physikalisches Quelltextmodul
  - Verteilungsdiagramm
    - zeigt Komponenten, Knoten und ihre Beziehungen
    - zeigt die realen Beziehungen zwischen SW-/HW-Komponenten im ausgelieferten System
- Verhaltensdiagramme
  - Aktivitätsdiagramm, Objektflussdiagramm
    - zeigt Aktivitäten, Objektzustände, Zustände, Zustandsübergänge und Ereignisse
    - beschreiben eine Reihenfolge von Aktivitäten
  - Kollaborationsdiagramm
    - zeigt Objekte und ihre Beziehungen inkl. ihres räuml. geordneten Nachrichtenaustauschs
  - Sequenzdiagramm
    - zeigt Objekte und ihre Beziehungen inkl. ihres zeitl. geordneten Nachrichtenaustauschs
  - Zustandsdiagramm
    - zeigt Zustände, Zustandsübergänge und Ereignisse
    - beschreibt das Verhalten eines Systems

Software-Architekturen

Einleitung

- Definition
  - strukturierte/hierarchische Anordnung der Systemkomponenten sowie deren Beschreibung
- Eigenschaften
  - früheste SW-Design-Entscheidung
  - kritischste bzw. wichtigste Phase
  - Änderung der Architektur sind in der späteren Entwicklungsphase nur mit hohen Kosten und Aufwand zu realisieren
- Ziele
  - Unterstützung der SE durch die Repräsentation eine Frameworks
  - schnelles Verständnis des Gesamtsystems
  - Kommunikationshilfe für die unterschiedlichen Stakeholder
  - Vereinfachung der Wartbarkeit
  - einfache Erweiterbarkeit des Systems
  - Vereinfachung der Kosten- und Zeitabschätzung
  - Wiederverwendung (der einzelnen Komponenten oder der gesamten Architektur)

Architektur-Typen

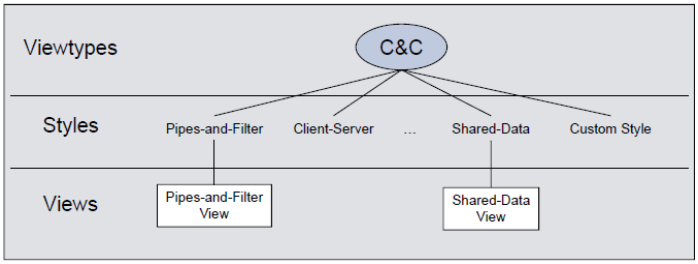


- Unternehmensarchitektur
  - besteht aus zusammenhängenden Modellen, die Struktur und Funktionen eines Unternehmens beschreiben
    - Ziele und Pläne des Unternehmens
    - Prozesse und Organisation des Unternehmens
    - Systeme und Daten des Unternehmens
    - die verwendete Technologie
- Geschäftsarchitektur
  - Beschreibung der Organisation eines Unternehmens, ihre wesentlichen Komponenten, Ressourcen und deren Beziehungen sowie Austauschbeziehungen des Unternehmens mit seiner Umwelt
  - Beschreibung umfasst die zentralen Kundengruppen und Leistungen des Unternehmens, die betrieblichen Aufgaben und Aufgabenträger, Lieferanten, Wettbewerber und Kooperationspartner sowie die Informationsflüsse und Leistungsstrukturen
- Anwendungsarchitektur
  - Beschreibung der Funktionalität und Zusammenhänge der Anwendungssysteme
- Informationssystemarchitektur
  - Beschreibung der informationstechnischen Infrastruktur, der Daten und Anwendungsprogramme, der unterstützten Aufgaben und der benötigten Aufbau- und Ablauforganisation
- Softwarearchitektur
  - Beschreibung der Komponenten eines SW-Systems sowie deren Beziehungen und Schnittstellen
- Datenarchitektur
  - Beschreibung der wesentlichen Daten und ihre strukturelle Beziehung, der DBs und DW-Systeme
- technische Infrastrukturarchitektur
  - Beschreibung der technischen Komponenten (z.B. HW, OS) und deren Beziehungen

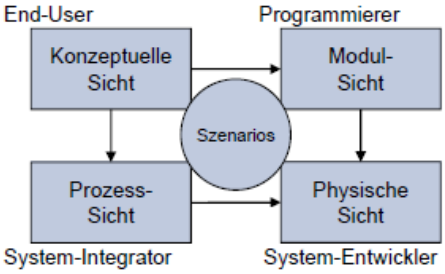
Architektur-Sichten

- beschreiben den Blickwinkel auf die Architektur
- Architektursichten (nach SEI)
  - Viewtype: definiert Element- und Beziehungstypen zur Beschreibung einer Architektur aus einer bestimmten Perspektive (unterteilt in Module-, Component-and-Connector- und Allocation-Viewtype)

- Style: Spezialisierung von Element- und Beziehungstypen, beinhaltet zusätzlich eine Menge von Bedingungen
- View: repräsentiert einen Typ, der an ein bestimmtes System gebunden ist



- Architektursichten (nach Kruchten)
  - konzeptuelle (logische) Sicht
    - zeigt die wichtigsten funktionalen Komponenten und deren Beziehungen in einem System
    - funktionale Anforderungen
    - keine technischen Details der Implementierung
  - Prozess-Sicht
    - Aufteilung der Operationen auf Tasks/Threads
    - dynamische Aspekte
  - Entwicklungs-Sicht
    - Organisation der SW-Module
    - Zusammenfassung zu Libraries oder Subsystemen
  - physische (technische) Sicht
    - Abbildung der SW auf die HW
    - Betrachtung der nichtfunktionalen Anforderungen (z.B. Verfügbarkeit, Performanz)
  - Szenarios
    - „+1“-Sicht, welche sich über alle 4 Sichten erstreckt
    - demonstrieren die (reibungslose) Zusammenarbeit der 4 Sichten



Architektur-Stile

- „an architectural style is a description of element and relation types together with a set of constraints on how they may be used”
- Bestandteile
  - Komponenten(-typen)
  - Anordnung der Komponenten
  - semantische Einschränkungen
  - Konnektoren
- Architekturstile
  - Client/Server-Architektur
    - Host = Rechner der Dienste (Server) anbietet
    - Clients fordern Dienste bzw. Daten eines Servers an
  - Peer-To-Peer-Systeme
    - Gegensatz zum Client/Server-Prinzip
    - jeder Host (Peer) kann gleichzeitig Server und Client sein
    - Anwendungsgebiete: IM, VoIP
  - Distributed Computing

- eventbasierte Systeme
- Blackboard-Systeme
  - zentrale Datenverwaltung
  - unabhängige Komponenten (Agenten)
  - Blackboard sendet Nachrichten an Agenten, wenn sich entsprechende Daten ändern
- Monolithische Systeme
- Web Service Architektur
- N-Schichten-Architektur
- Dreischichtenarchitektur
- Service Oriented Architecture (SOA)
- Representational State Transfer (REST)
- Pipes & Filters
  - Schwerpunkt auf Datenfluss
  - Output einer Komponente ist wiederum der Input einer anderen Komponente
  - Pipes: Output eines Filters bzw. Input eines anderen Filters
  - Filter: unabhängige Komponenten, die Input- in Output-Datenströme transformieren
  - Vorteile: Austauschbarkeit, Wiederverwendbarkeit, Wartbarkeit
  - Nachteile: ungeeignet für Interaktionen, gemeinsames Datenformat nötig, Filter müssen Datenströme komplett einlesen

Architektur-Beschreibungssprachen

- grundsätzlich können Architekturen als Menge von Komponenten und deren Verbindungen beschrieben werden
- je nach Domäne und Sicht sind unterschiedliche Elemente zur Beschreibung sinnvoll
- es existieren zahlreiche ADLs (Architectural Description Languages) zur Beschreibung von Architekturen (sowohl formal, als auch graphisch)

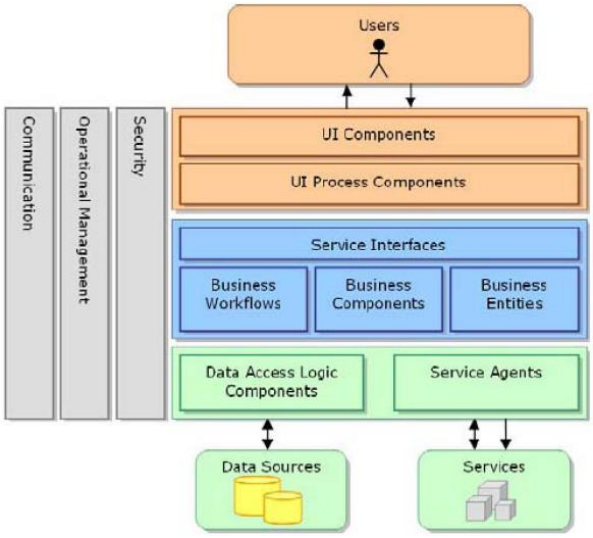
Beispiel: Dreischichtenarchitektur

- logische Struktur: Komponenten werden ihren Aufgaben entsprechend logischen Schichten zugeordnet
  - fokussiert die Wiederverwendung von Komponenten oder Subsystemen
- Struktur der Schichtenarchitektur entspricht einem Stack
  - Komponenten verwenden die Funktionalität, die von anderen Komponenten der gleichen Schicht oder auf unteren Schichten zur Verfügung gestellt werden
- ermöglicht weitere Unterteilung der Schichten in Subschichten
- Aufbau
  - Präsentationsschicht
    - beinhaltet alle Komponenten, die für die Interaktion zwischen Benutzer und Anwendung erforderlich sind
    - Komponenten zur Datendarstellung
    - Komponenten, die die Eingabe und Validierung von Daten ermöglichen
    - Steuerung des Benutzerverhaltens
    - Benutzerschnittstelle abhängig von Endgeräten
  - Anwendungsschicht
    - umfasst alle Komponenten, die die Kernfunktionalität des Anwendungssystems zur Verfügung stellen

- Kernfunktionalitäten erstrecken sich von der Bearbeitung einfacher Aufgaben bis hin zur Durchführung komplexer GPs oder Transaktionen
- Geschäftskomponenten werden in ihrer Gesamtheit als Geschäftslogik bezeichnet

- Datenschicht
  - umfasst Komponenten für den Zugriff auf Datenquellen
  - Komponenten stellen Methoden für die Abfrage und Manipulation von Daten zur Verfügung

- Komponenten einer Dreischichtenarchitektur



  - UI Components
    - ermöglichen die Interaktion des Benutzers mit dem System
    - Darstellung und Formatierung von Daten
    - Erfassen und Validieren von Daten
  - User Process Components
    - steuern vorhersehbare Prozesse, die aus Benutzerinteraktionen mit dem System resultieren
    - Synchronisierung und Orchestrierung von Benutzerinteraktionen
    - erfassen Daten während der Interaktion
    - Verwendung gesonderter Prozesskomponenten (zur Wiederverwendung)
  - Business Workflows
    - erhalten die gesammelten Daten der User Process Components
    - definieren und koordinieren lang anhaltende, mehrstufige GPs
    - verantwortlich für die korrekte Abwicklung (Reihenfolge, Orchestrierung) der GPs
    - Management der Einzelschritte eines GPs
  - Business Components
    - implementieren die Geschäftslogik des Systems
    - beinhalten Geschäftsregeln und -funktionen
  - Service Agents (Service Gateways)
    - kapseln die Kommunikation mit externen Diensten
    - verantwortlich für Kommunikationsdetails
  - Service Interface
    - stellen die Geschäftslogik als Service zur Verfügung
    - unterstützen die für die Kommunikation erforderlichen Verträge (Protokolle, Formate)

        - werden auch als Business Facades bezeichnet
    - Data Access Logic Components
        - Zugriff auf Datenquellen
        - Abfrage und Manipulation von Daten
        - Schaffung einer zentralen Datenzugriffsfunktionalität
    - Business Entity Components
        - repräsentieren reale Geschäftsobjekte
        - dienen dem Austausch von Daten zwischen Komponenten (Weiterreichen von Daten der Datenschicht an die Präsentationsschicht)
        - unterschiedliche Datenstrukturen (XML, DataSets, Records, OO-Klassen)
    - Security, Operational Management, Communication
        - Security: Authentification, Authorization, Secure Comm., Auditing, Profile Mgmt.
        - Operational Mgmt: Exception Mgmt, Monitoring, Metadata, Configuration
        - Communication: Synchronicity, Format, Protocol
- allgemeine Entwurfsempfehlungen
    - frühzeitige Entscheidung für die Art der Schichtung (z.B. strikte Ordnung)
    - Identifizierung aller Komponenten, die für das System benötigt werden
    - möglichst lose gekoppelte Komponenten
    - Verwendung konsistenter Datenformate
    - saubere Trennung von Policies und Geschäftslogik

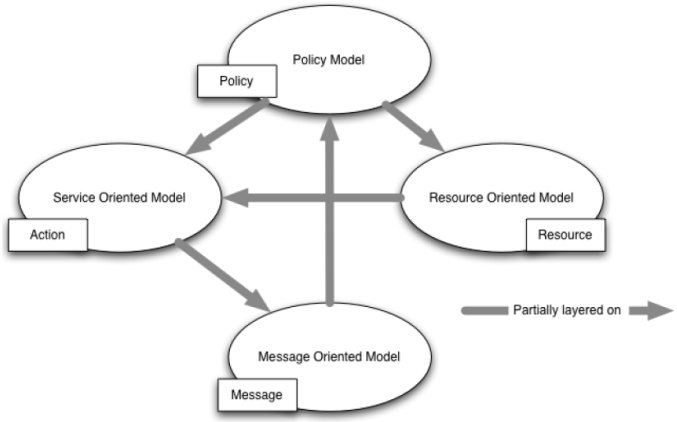
Web Service Architektur

Einleitung

- konzeptuelles Modell
- beschreibt die Zusammenhänge zwischen den Komponenten des Modells
- beschreibt die erforderlichen globalen Elemente, um die Interoperabilität zwischen Webservices sicherzustellen
- „A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Modelle der Architektur

- die Architektur besteht aus 4 Modellen, die die Schlüsselkonzepte beschreiben



- Message Oriented Model (MOM)
  - fokussiert die Aspekte der Architektur in Bezug auf Nachrichten und deren Austausch/Verarbeitung
    - Nachrichtenstruktur
    - Beziehungen zwischen Sender und Empfänger
    - Nachrichtenübertragung
- Service Oriented Model (SOM)
  - fokussiert die Aspekte der Architektur in Bezug auf Dienste und Aktionen
- Resource Oriented Model (ROM)
  - fokussiert Aspekte der Architektur in Bezug auf Ressourcen
    - Ressourcen als fundamentales Konzept
    - Besitz von Ressourcen
    - Policies in Verbindung mit Ressourcen
- Policy Model (PM)
  - fokussiert die Aspekte der Architektur in Bezug auf Policies
  - stellt ein Framework zur Verfügung, mit dem Aspekte wie Sicherheit oder QoS realisiert werden können

Perspektiven

- Web Service Management
  - Monitoring, Controlling, Reporting von Web Services
  - Service Qualität (Verfügbarkeit, Performance, Erreichbarkeit)
  - Service-Verwendung (Häufigkeit, Dauer, Gültigkeitsbereich, Funktionsumfang, Zugriffsber.)
- Web Service Semantics
  - Metadaten: Steigerung der Interoperabilität durch erhöhte Präzision bei der Dokumentation von Web Services
  - höherer Grad an Automation bei der Entwicklung von Web Services
  - geeignete Technologie: OWL als standardisierte Ontologie-Sprache
- Web Service Security
  - Bedrohungen auf Nachrichtenebene (z.B. Man-in-the-Middle, DoS, Spoofing)
  - Sicherheitsanforderungen (z.B. Autorisierung, Authentifizierung, Integrität, Vertraulichkeit)
  - Sicherheitsbetrachtungen (z.B. Cross-Domain-Identities, Trust-Policies, Secure Messaging)
- Peer-to-Peer Interaction
  - Unterstützung von Peer-to-Peer-Nachrichtenaustausch
  - persistente IDs für Web Services
  - Beschreibung der Fähigkeiten eines Peers
- Web Service Reliability
  - Steigerung der Verlässlichkeit beim Nachrichtenaustausch
  - Fragestellungen beim Nachrichtenaustausch
    - Ist eine Nachrichten beim beabsichtigten Empfänger (nur einmal) angekommen?
    - Entspricht die Nachricht dem im Vertrag spezifizierten Format und den Geschäftsregeln des Empfängers?
  - Transaktionsmechanismus

Serviceorientierte Architektur (SOA)

Warum SOA?

- hoher Grad an Komplexität durch schnell wachsende heterogene Systemlandschaften → traditionelle Architekturen stoßen an ihre Grenzen
- Forderung von Unternehmen IT-Kosten zu senken, aber gleichzeitig schnell und flexibel reagieren zu können
- Forderung nach schnellen Implementierungen neuer Systeme
- Integration von Geschäftspartnern und Kunden
- Framework wird benötigt, das Komponenten und Dienste miteinander verbindet und schnelle und dynamische Lösungen liefert
- technische Probleme/Herausforderungen
  - endlose Vielfalt an HW, OS, Middleware, Sprachen, Datenspeicher
  - redundante und nicht wieder verwendbare Programmierung
  - n * (n-1) Integrationsproblem (Integration Killer)
- CIO-Priorität Nr 1: Application Integration
- Ziel
  - IT unterstützt die Geschäftsvorfälle, nicht umgekehrt
  - Wettbewerbsvorteil durch die Möglichkeit, schnell neue (optimierte) GPs zu realisieren
  - „Echtzeit-Unternehmen“ realisieren
    - zu jeder Zeit vollständigen Überblick über relevante Daten
    - frühzeitiges und gezieltes Reagieren auf Veränderungen des Marktes

Einleitung

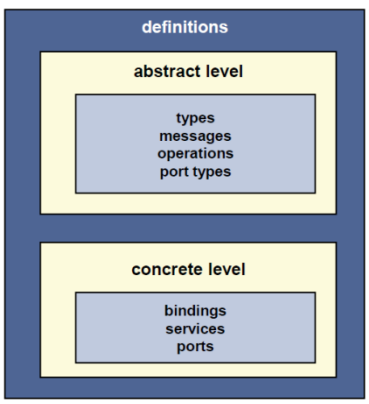
- SOA = unabhängiges Architektur-Paradigma
- beschreibt Systemarchitekturen, die sich hauptsächlich aus Diensten zusammen setzen
- SW-Architektur (allgemein) = Konfiguration von Komponenten und Konnektoren
- bei SOA
  - Komponenten = Dienste, Konnektoren = Interaktionen zwischen den Diensten
- wichtige Eigenschaften
  - Transparenz
    - Realisierung getrennt von der Beschreibung eines Dienstes
    - zu jedem Dienst existiert eine Schnittstelle (service interface)
    - Folge: homogenes und interoperables Gesamtsystem
  - Geschäftsprozesse stehen im Vordergrund
    - Technik rückt in den Hintergrund
  - dynamische Konfigurierbarkeit
- SOA und Web Services
  - Web Service-Spezifikation definieren die Details, welche zur Implementierung und Interaktion von Diensten benötigt werden
  - SOA ist ein Ansatz für verteilte Systeme, die ihre Anwendungsfunktionalität zu Endbenutzer-Anwendungen übertragen
  - eine SOA kann mit Web Services umgesetzt werden, es können aber auch andere Technologien verwendet werden

- Dienste (Services)
  - Anwendungslogik in einer wieder verwendbaren Komponente gekapselt zur Verwendung in Geschäftsprozessen
- lose Kopplung
  - Dienstkonsument hat keinerlei Kenntnisse über technische Details der Dienstimplementierung des Anbieters
  - Aufruf (und Antwort) der Dienstmethoden über Nachrichten statt über APIs
- zustandsloses Design
  - Dienste sollten unabhängig und in sich geschlossen sein
  - sie sollten keine (Status)-Informationen anderer Dienste benötigen
    - Abhängigkeit von Informationen anderer Dienste bzw. Requests erhöht die Komplexität der Implementierung
- Granularität der Dienste
  - wichtiger Design-Faktor
  - grobe Granularität bei externen Schnittstellen
    - Nachricht beinhaltet alle Informationen (z.B. SubmitPurchaseOrder)
  - feine Granularität bei internen Schnittstellen
    - flexibler, auf der Aufwand für den Anbieter ist größer (z.B. SetShippingAddress, AddItem)
  - Verwendung von Choreographie zur Erstellung von grobgranularen Schnittstellen eines GPs, der aus fein-granularen Operationen besteht

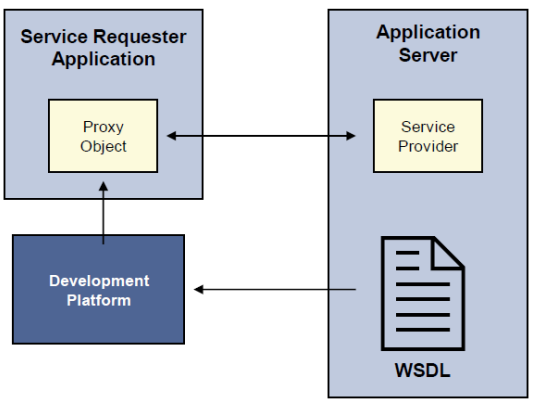
Service-Beschreibung

- beschreibt Details, welche der Service-Konsument zur Herstellung der Verbindung zum Service-Anbieter benötigt
- enthält keine technologischen Details
- Beispiel: Web Service Description Language (WSDL)
  - XML-basierte Beschreibungssprache
  - die WSDL-Spezifikation beschreibt ein Vokabular zur Erstellung eines Vertrages zwischen Dienst-Konsument und Dienst-Anbieter
  - enthält
    - Format der Anfragen(-nachrichten)
    - Format der Antwort(-nachrichten)
    - Ort der Nachrichtenempfänger

WSDL-Aufbau

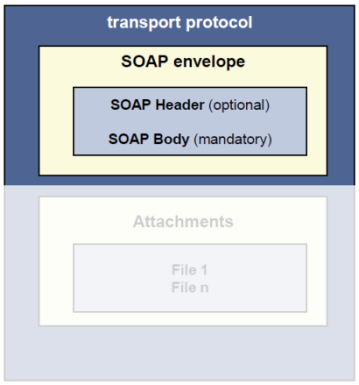


Verwendung von WSDL-Dokumenten



Aufruf von Diensten

- Beispiel SOAP
  - Nachrichten im XML-Format zur Kommunikation
  - prinzipiell: zustandsloser one-way-Nachrichtenaustausch
  - Kommunikation über http(s)
  - Methodenaufruf wird innerhalb eines sog. SOAP-Envelopes an den Anbieter geschickt (Antwort an den Konsumenten auch)

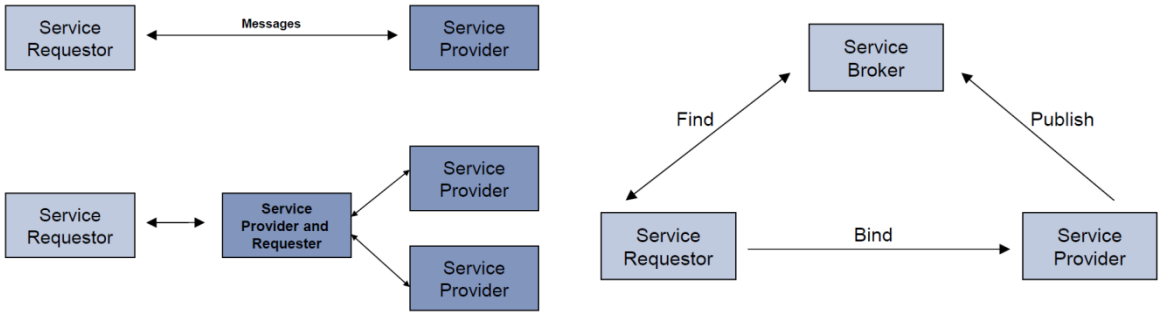


- Formen des Nachrichtenaustauschs (synchron)
  - Request-Response
    - aus Gründen der Performance, des einfacheren Designs und den Vorteilen der losen Kopplung sollten grob-granulare Schnittstellen entworfen werden, um die Anzahl der Nachrichtenaustausche (pro Transaktion) zu minimieren
  - One-Way-Message
    - Anfragen, die keine Antwort erwarten
  - Notification
    - One-Way-Message aus Richtung des Anbieters
  - Solicit Response
    - umgekehrt zur Request-Response
- Formen des Nachrichtenaustauschs (asynchron)
  - Reply-to Addresses
    - aus der WS-Addressing-Specification
    - Erweiterung des SOAP-Envelope-Header-Elements
    - Konsument überträgt seine Anfrage an den (in der WSDL angegebenen Port), wartet aber nicht auf eine Antwort
    - der Anbieter sendet die Antwort an die „Reply-to Address“ aus dem SOAP-Header der Anfragenachricht zurück
  - Multiple Response
    - eine Anfrage, mehrere Antworten (z.B. Kurs-Abfrage)

Rollen

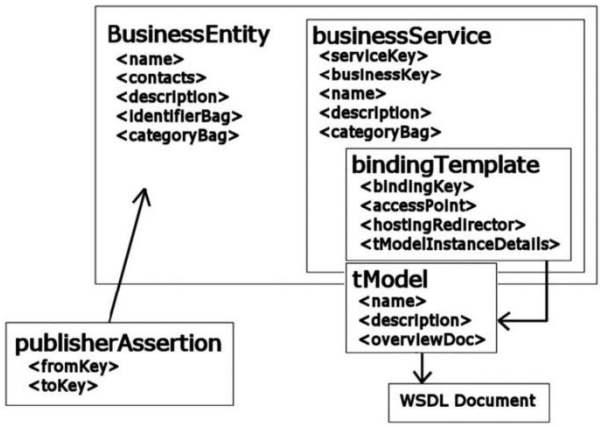
- SOA-Komponenten (Rollen)
  - Service-Anbieter (Service Providers)
  - Service-Verzeichnisse (Service Registries)
  - Service-Konsumenten (Service Requesters)
- SOA-Basisinteraktionen
  - Veröffentlichung eines Dienstes (register)
  - Suchen und Finden eines Dienstes (find)
  - Verbindungsherstellung zu einem Dienst (bind)
  - Anfrage an einen Dienst stellen (execute)

- Rollen einer SOA



Auffinden von Diensten (Service Discovery)

- optionaler Bestandteil einer SOA (im Gegensatz zu Service Invocation und Service Description)
- Beispiel Universal Description, Discovery and Integration (UDDI)
  - Verzeichnisdienst zur Veröffentlichung und Suche von Diensten
  - übersetzt SOAP-Requests in Daten-Management-Funktionen für die zu Grunde liegende DB
  - Service Registry = Service Broker
- UDDI-Informationsarten
  - White Pages
    - Namensregister aller Anbieter
    - Detailinformationen (Telefon, Fax, …)
  - Yellow Pages
    - Branchenverzeichnis mit Verweis auf White Pages
    - spezifische Suche
  - Green Pages
    - Informationen über Geschäftsmodell/-prozesse eines Unternehmens
    - technische Details über angeboten Web Services
- UDDI-Datenstruktur



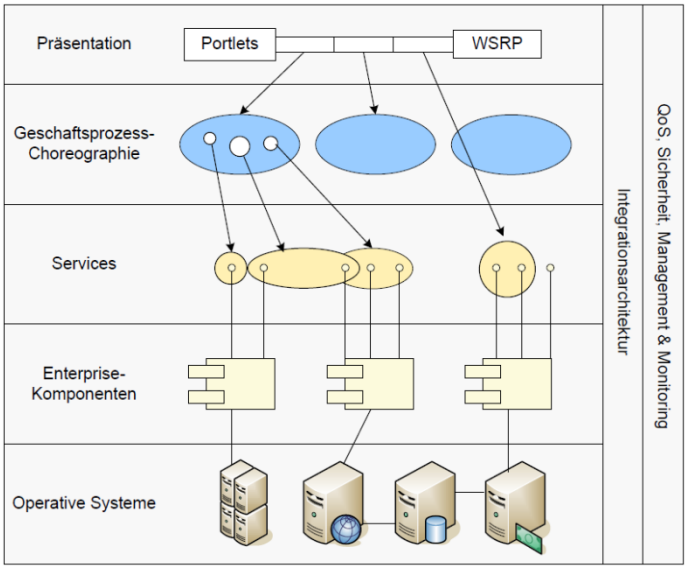
  - businessEntity
    - Informationen über das Unternehmen, welches Dienste publiziert
  - businessService
    - Informationen über einen bestimmten Dienst (Kategorie, …)
  - bindingTemplate
    - technische Details, Referenz auf zugehöriges tModel
  - tModel
    - beschreibt die Schnittstelle des Dienstes (z.B. Referenz auf WSDL-Datei)

- UDDI Verwendung
  - einmaliger UDDI-Zugriff
    - Anbieter veröffentlicht Informationen
    - Dienstnutzer sucht/findet den Dienst
    - Dienstnutzer speichert Spezifikation und Dienstendpunkt-Information
    - Dienstnutzer verwendet in Applikation den Dienst (über die öffentliche Schnittstelle)
  - mehrfacher UDDI-Zugriff
    - Applikation greift zur Laufzeit auf das UDDI-Verzeichnis zu
    - Applikation ermittelt über SOAP-Schnittstelle die Bindungsinformationen des gewünschten Dienstes

Sicherheit

- Sicherheitsanforderungen (an die Kommunikation mit einer Service-Komponente)
  - Vertraulichkeit, Berechtigung, (Daten-)Konsistenz, Glaubwürdigkeit, Verbindlichkeit
- Verfahren
  - Verschlüsselung
    - gewährleistet Vertraulichkeit
    - symmetrisch und asymmetrische Verschlüsselung
  - digitale Signaturen
    - gewährleisten Konsistenz, Glaubwürdigkeit und Verbindlichkeit
    - kurz: mit privatem Schlüssel verschlüsselter Hashwert
- Sicherheit auf XML-Schicht
  - Verschlüsselung
    - XML Encryption (Verschlüsselung von XML-Dokumenten, Elementen und deren Inhalt)
  - digitale Signaturen
    - XML Signature (Erstellung digitaler Signaturen zu XML-Dokumenten und Elementen)

Ebenen einer SOA



mit manchen Tools kann aus EPK/BPEL automatisch BPEL/XPDL erzeugt werden, so dass nicht mehrmals modelliert werden muss

- Ebene 1: Operative Systeme
  - bestehende, mit herkömmlichen Technologien entwickelte Systeme (CRM, ERP, BI, …)
- Ebene 2: Enterprise-Komponenten
  - Realisierung der Funktionalität von Services

  - verantwortlich für die QoS der bereitgestellten Services
  - typischerweise Container-basierte Technologien (z.B. Application Server wie JEE)
- Ebene 3: Services
  - einzelne oder zusammengefasste Dienste
  - dynamisches Auffinden oder statisches Binden von Services
  - Bereitstellung von Komponentenschnittstellen in Form von Dienstbeschreibungen
  - Prozess-Engine zum Ausführen von Kontrollflüssen
- Ebene 4: GP-Choreographie
  - Zusammenstellung von Diensten der 3. Ebene
  - Erstellung von Geschäftsprozessen und -flüssen durch Orchestrierung/Choreographie
  - Unterstützung spezifischer Anwendungsfälle (Use Cases) und Geschäftsprozesse
  - Entwurf mithilfe sog. „Visual Flow Composition Tools“ (z.B. ARIS, BPMN)
  - mögliche Ebenen: orchestrated services → entity services → utility services
- Ebene 5: Präsentation
  - Bereitstellung interaktiver Benutzerschnittstellen für Web Services (z.B. Plug-Ins für Portale)
- Ebene 6: Integrationsarchitektur
  - stellt Mechanismen für die Integration von Diensten zur Verfügung (Intelligent Routing, Protocol Mediation, Transformation)
- Ebene 7: Quality of Service
  - Monitoring, Management und Erhaltung der QoS (z.B. Sicherheit, Verfügbarkeit)
  - Sense-and-Response-Mechanismen (z.B. Implementierung des WS-Management-Standards)

Orchestrierung und Choreographie

- Orchestrierung bezeichnet die Interaktionen von Services mit anderen internen und externen Services auf Nachrichtenebene. Diese Interaktionen sind häufig transaktionsbezogen und langlebig. Innerhalb einer Orchestrierung wird die Prozesskontrolle aus der Perspektive eines der Geschäftsbeteiligten gesehen.
- Choreographie bezeichnet die globale Sicht auf einen GP und beinhaltet die Rollen der einzelnen Transaktionspartner. Choreographie wird mit öffentlichen Prozessen assoziiert im Gegensatz zur Orchestrierung, die eher die privaten Prozesse beschreibt.

Business Process Execution Language for Web Services (BPEL4WS)

- XML-basierte Grammatik zur Beschreibung von Kontroll-Logik
- Koordination und Integration von Web Services zu einem GP
- stark an das WSDL-Protokoll gekoppelt
- Unterstützung von privaten als auch öffentlichen Prozessen
- BPEL ist selber ein Webservice, so dass es flexibel kombiniert werden kann
- Modellierung eines Prozessmodells
  - Partner
    - Systeme oder Web Services, die Funktionen des GPs nutzen wollen. Innerhalb dieses Elements werden die Rollen der Partner zueinander definiert.
  - Messages
    - Nachrichten, die die Partner untereinander austauschen
  - Container
    - speichert globale Nachrichtendaten, die typischerweise WSDLMessageType-Elementen entsprechen

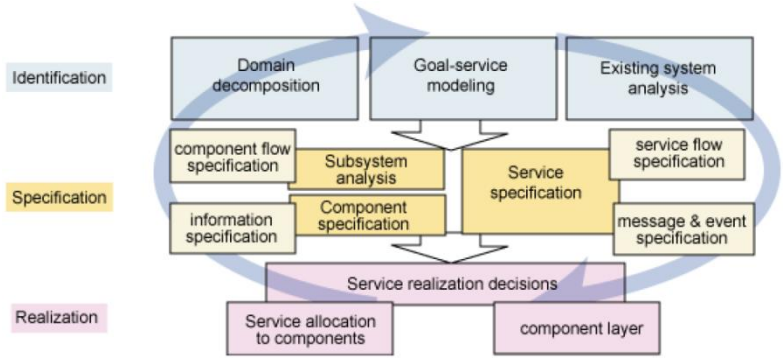
- Activities
  - einzelne Schritte innerhalb eines BPEL-Prozesses, unterschieden in Basis- und strukturelle Aktivitäten
- Scopes
  - Definieren eine Gruppen von Aktivitäten, der spezielle Eigenschaften zugeordnet werden können. Z.B. kann eine Fehlerkomponente durchgeführt werden, wenn eine Transaktion innerhalb eines Scope-Elements nicht durchgeführt werden kann.

SOA-Modellierung: Analyse und Design

- Aktivitäten von Service Provider und Service Consumer

Role	Activities in this Role				
Consumer View	Service identification	Service categorization	Service exposure decisions	Choreography or composition	Quality of Service
Provider View	Component identification	Component specification	Service realization	Service management	Standards implementation
	Service allocation to components	Layering the SOA	Technical prototyping	Product selection	Architectural decisions (state, flow, dependencies)

- Prozess bei der serviceorientierten Modellierung



- Service-Identifizierung (Kombination versch. Sichten)
  - top-down-Sicht
    - Entwurf von Anwendungsfällen, die Spezifikationen für die Business Services zur Verfügung stellen
    - Aufschlüsselung von Geschäftsdomänen in funktionale Bereiche und Subsystemen
    - Aufschlüsselung von Geschäftsflüssen und -prozessen innerhalb der Domänen in Prozesse, Sub-Prozesse und Anwendungsfälle
  - bottom-up-Sicht
    - Analyse der bestehenden Systeme
    - Auswahl der Systeme, die die Implementierung der Service-Funktionalität und die GPs unterstützen
    - Analyse und Einsatz von APIs, Transaktionen, Modulen
    - Modularisierung oder Wrapping von Legacy-Systemen zur Unterstützung der Service-Funktionalität
  - middle-out-Sicht
    - goal-service Modeling
    - Identifizierung und Validierung weiterer Dienste, die durch die anderen Ansätze nicht erfasst wurden

- Service-Klassifizierung oder -Kategorisierung
  - Klassifizierung der identifizierten Services in eine Service-Hierarchie
  - Komposition fein-granularer Komponenten und Services
  - bessere Koordination bei der Erstellung und Steuerung von einander abhängigen Services
  - „Service-Verwilderung“ wird vermieden
- Analyse der Sub-Systeme
  - Spezifizierung der Abhängigkeiten und Flüsse zwischen den zuvor identifizierten Sub-Systemen
  - Services werden Sub-Systemen zugeordnet
  - Erstellen der Objekt-Modelle
- Spezifikation der Komponenten
  - Details der Komponenten werden spezifiziert
- Spezifikation der Services
  - Flüsse zwischen den Services werden spezifiziert
  - Spezifikation von Nachrichten und Ereignissen
- Zuordnung der Services
  - Services werden den Sub-Systemen und den entsprechenden Komponenten zugeordnet
  - Strukturierung der Komponenten unter Verwendung von Patterns (z.B. Mediator, Facades)
- Service-Realisierung
  - Auswahl oder Erstellung der Software (Module), auf die die Services aufsetzen
  - Integration, Transformation, Outsourcing einzelner Teile der Funktionalität unter Verwendung von Web Services
  - Wiederverwendung oder Neuentwicklung von Modulen bei der Realisierung
  - Realisierung der Sicherheit, des Managements und des Monitoring von Services

Representational State Transfer (REST)

Kontext

- Probleme
  - steigende Komplexität verteilter heterogener Systeme
  - Skalierbarkeit von Server-Anwendungen
  - Performanz von netzwerkbasierten Anwendungen
  - Transparenz bei der Software-Entwicklung
- Lösungen
  - Schaffung klarer Regeln sowie Gebrauch verbreiteter und allseits bekannter Technologien
  - Last verteilen bzw. minimieren
  - einheitliches Interface

Definition

- „REST is an architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.”
- Was ist REST?
  - Architekturstil für verteilte (und hypermediale) Anwendungen bzw. Systeme
  - Essenz des Zusammenwirkens verschiedener netzwerkbasierter Architekturstile

  - eine Menge von Regeln (Constraints) zur Gestaltung von verteilten Anwendungen (→ RESTful)
  - *„Best Practices“, Verhaltenskodex, Prinzipiensammlung*
  - pragmatisch: ein Lösungsansatz zur Begegnung heutiger Herausforderungen
- Was ist REST nicht?
  - Neuerfindung (i.e.S.), Methode oder Werkzeug, Programmiersprache, …

Prinzipien

- Resources and Resource Identifiers
  - die Orientierung an Ressourcen und die (weltweit) eindeutige Identifikation jeder Ressource ist eine essenzielle und grundlegende Idee hinter dem REST-Konzept
  - eine Ressource ist jede Information, die anderen zur Verfügung gestellt werden soll
  - Resource Identifiers ermöglichen
    - die (weltweit) eindeutige Identifikation jeder Ressource
    - eine einfache und effiziente Art Ressourcen wiederzufinden, zu verteilen und zu kommunizieren
    - Beispiel: http://www.softec.wiwi.uni-due.de/team/stefan-eicker/
  - entsprechen im Internet der Unifom Resource Identifier (URI) → Schaffung eines Global NS
- Uniform Interface
  - bezeichnet die Verwendung wohldefinierter Operationen und einheitlicher Schnittstellen
  - dadurch Reduzierung der Komplexität und des Entwicklungsaufwands, Steigerung der Effizienz, Erhöhung der Produktivität (und Kreativität)
    - GET: requests a specific representation of a resource
    - PUT: create or update a resource with the supplied representation
    - DELETE: deletes the specified resource
    - POST: submits data to be processed by the identified resource
  - fokussiert die von HTTP bereitgestellten Methoden
- Statelessness
  - Zustandslosigkeit ist ein grundlegendes Prinzip von HTTP und „dem Web“
  - bedeutet, dass eine eingehende Anfrage einzig und allein basierend auf der Anfrage selbst behandelt werden kann (es sind keine weiteren Informationen erforderlich)
  - eine Anfrage muss daher alle notwendigen Informationen enthalten
  - Auslagerung der Sitzungsverwaltung auf die jeweiligen Clients
  - Anmerkung: Zustandslosigkeit bezieht sich nur auf die Kommunikation, Ressourcen- und Anwendungszustand im verteilten System existieren weiterhin
- Hypermedia
  - zusammengesetztes Wort aus Hypertext und Multimedia
  - Kernidee: link things together
  - Links ermöglichen die Verknüpfung von Ressourcen über Anwendungs-, Server- und Unternehmensgrenzen hinweg → weltweite Verknüpfung
- assoziierende Prinzipien
  - Caching: Speichern von als „cacheable“ gekennzeichneten Antworten und Wiederverwendung dieser Antworten bei gleichen Anfragen
  - Representations: Darstellungsform einer Ressource, multimedial
  - Separation of Concerns: u.a. Trennung von user interface concerns & data storage concerns

Technologien

- REST ist prinzipiell Technologie-unabhängig – aber bestimmte Technologien tauchen häufig im Kontext von REST auf
  - HTTP!, XML, JSON, …
- auch Frameworks unterstützen den REST-Ansatz
  - WCF, JAX-WS, …

REST im SOA-Kontext

- Grundlage
  - SOA als Paradigma für verteilte Systeme, das in der Praxis häufig mit Hilfe von Web Services realisiert wird
  - REST als „Verhaltenskodex“ für die Konzeption und Entwicklung von verteilten Systemen
- aktuelle Diskussionen in Praxis und Literatur
  - REST vs. SOA, REST vs. SOAP, Resource Oriented Architecture (ROA), REST vs. WS-*
- Pro- und Contra-Argumente (i.w.S.)
  - SOA ist zu komplex, REST ist einfacher (zu skalieren, zu entwickeln, handzuhaben, …)
  - REST ist nicht mächtig genug, um den Anforderungen im (über-)betrieblichen Einsatz gerecht zu werden und versagt in Punkten wie Sicherheit und Anwendbarkeit
- aber: Integrationsmöglichkeiten (i.e.S.)
  - die Gestaltung und Bereitstellung von Services innerhalb einer SOA kann unter Berücksichtigung der von REST vorgeschlagenen Prinzipien erfolgen
  - sogenannte „RESTful Web Services“ als Resultat
  - spezielle Diskussion: REST vs. SOAP

Patterns

Einführung

- Prinzip: Wiederverwendung von Lösungen, die sich für bestimmte Problembereiche bewährt haben
- Darstellung als (Entwurfs-)Muster → (Design-)Pattern
  - Entwurfsmuster sind Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.
- grundlegende Elemente von Patterns
  - Patternname
    - Benennung des Entwurfsproblems und der Lösungen (1-2 Worte)
    - Erweiterung des Entwurfsvokabulars
    - erleichtert das Handling des Patterns (Dokumentation, …)
  - Problemabschnitt
    - Wann ist das Pattern anzuwenden?
    - Welches Problem in welchem Kontext wird adressiert?
    - beschreibt spezifische Entwurfsprobleme, Bedingungen, …
  - Lösungsabschnitt
    - Bestandteile (Elemente) des Entwurfs
    - wichtig ist die Bewertung von Entwurfsalternativen
    - Lösung als Schablone zur Anwendung in unterschiedlichen Situationen

- Konsequenzabschnitt
  - Vor- und Nachteile des Entwurfs
  - wichtig für die Bewertung von Entwurfsalternativen
  - betrifft z.B. Speicherplatz, Ausführungszeit, Sprach-Implementierungsaspekte, Flexibilität

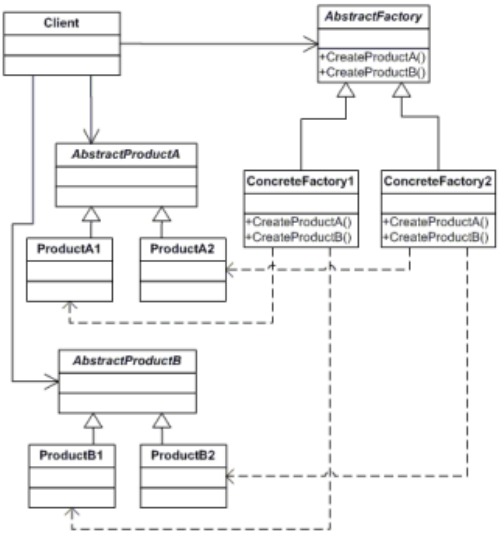
Design Patterns

- werden in der Entwurfsphase des SW-Lebenszyklus eingesetzt
- unterstützen die Erstellung eines Entwurfsmodells
- Bindeglied zwischen Analyse und Implementierung
- Beschreibung
  - Name: Pattern-Name und Klassifikation
  - Zweck: Prinzip und Zweck des Patterns, Fragestellungen/Probleme, die behandelt werden
  - auch bekannt als: andere (bekannte) Namen für das Pattern
  - Motivation: beschreibt ein Szenario, einfaches Verständnis der abstrakten Beschreibungen
  - Anwendbarkeit: relevante Situationen, Problemsituationen
  - Struktur + Teilnehmer: grafische Darstellung der Klassen (UML), Interaktionsdiagramme
  - Interaktionen: Zusammenarbeit der Teilnehmer
  - Konsequenzen: Vor- und Nachteile, zu erwartende Probleme
  - Implementierung + Beispielcode: beschreibt Fallen, Tipps und Techniken der Implementierung des Patterns, sprachspezifische Aspekte
  - bekannte Verwendungen: Beispiele für die Verwendung des Patterns in realen Systemen
  - verwandte Muster: Bezug zu anderen Patterns, Zusammenarbeit mehrerer Patterns
- Design Patterns unterstützen die
  - Erfassung von (passenden) Objekten
    - Objekte führen zu Klassen mit unterschiedlichem Abstraktionsgrad
    - Design Patterns helfen bei der Identifizierung nicht direkt offensichtlicher Abstraktionen
    - z.B. die Patterns Composite, Strategy oder State
  - Bestimmung der Objektgranularität
    - Objekte variieren in Größe und Anzahl
    - Subsysteme können in Objekte gekapselt werden
    - z.B. die Patterns Facade, Flyweight oder Abstract Factory
  - Definition der Schnittstellen
    - Identifikation von Elementen und Daten der Schnittstelle
    - Beziehungen zwischen Schnittstellen
    - z.B. die Patterns Memento, Decorator, Proxy oder Visitor
  - Objektimplementierung
    - Klassen-, Schnittstellenvererbung
  - Wiederverwendung
    - Vererbung, Komposition
  - Vermeidung späterer Veränderung von Entwürfen (Entwurfsrevision)
    - Objekterzeugung, Änderung von Klassen, lose Kopplung
    - Abhängigkeit von spez. Operationen und von HW/SW vermeiden

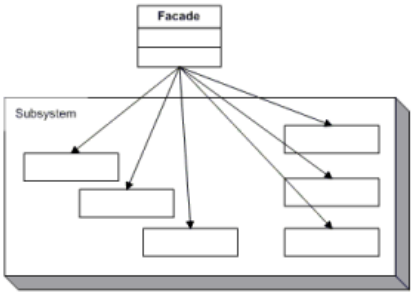
- Klassifizierung

		Aufgabe		
		Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Gültigkeitsbereich	Klassenbasiert	Factory Method	Adapter (klassenbasiert)	Interpreter Template Method
	objektbasiert	Abstract Factory Builder Prototype Singleton	Adapter (objektbasiert) Bridge Composite Decorator Facade Flyweight Proxy	Command Observer Visitor Iterator Memento Strategy Mediator State Chain of Resp.

- Erzeugungsmuster
  - dienen der Objekterzeugung
  - verstecken den Erzeugungsprozess
  - kapseln das Wissen um die konkreten (vom System verwendeten) Klassen
  - verstecken, wie Exemplare dieser Klassen erzeugt und zusammengefügt werden
  - Beispiel Abstract Factory
    - Zweck: Schnittstelle zur Erzeugung von Familien verwandter/abhängiger Objekte, ohne eine konkrete Benennung der Klassen
    - auch bekannt als: Kit
    - Teilnehmer: Abstract Factory, Concrete Factory, Abstract Product, Concrete Product, Client

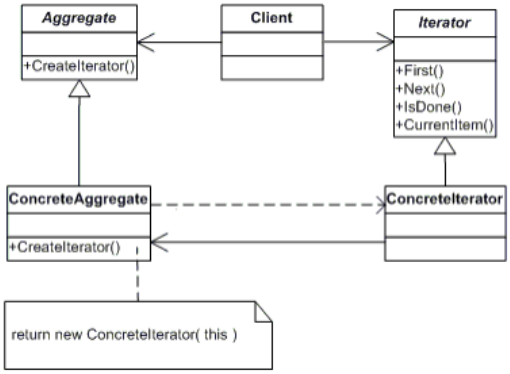


- Strukturmuster
  - Komposition von Objekten und Klassen zur Bildung größerer Strukturen
  - klassenbasiert: Zusammenführung von Schnittstellen/Implementierungen mittels Vererbung
  - objektbasiert: Zusammenführung von Objekten zur Erzeugung neuer Funktionalität
  - Beispiel: Facade
    - Zweck: Schnittstelle zu einer Menge von Subsystem-Schnittstellen zur Vereinfachung der Nutzung des Subsystems
    - Teilnehmer: Facade, Subsystem



- Verhaltensmuster
  - Algorithmen
  - Zuweisung von Zuständigkeiten zu Objekten
  - Behandlung der Interaktion zwischen Klassen/Objekten
  - klassenbasiert: Vererbung, zur Verteilung des Verhaltens unter den Klassen
  - objektbasiert: (Objekt-)Komposition statt Vererbung
  - Beispiel: Iterator
    - Zweck: ermöglicht sequentiellen Zugriff auf Elemente eines zusammengesetzten Objekts, ohne eine Offenlegung der zugrunde liegenden Repräsentation

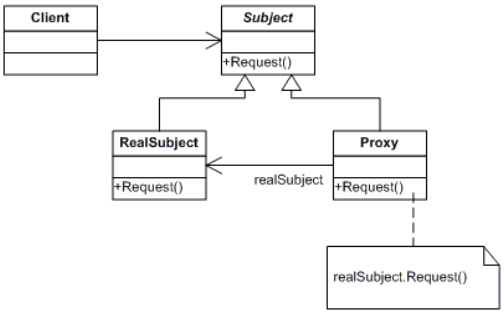
- auch bekannt als: Cursor
- Teilnehmer: Iterator, Concrete Iterator, Aggregate, Concrete Aggregate



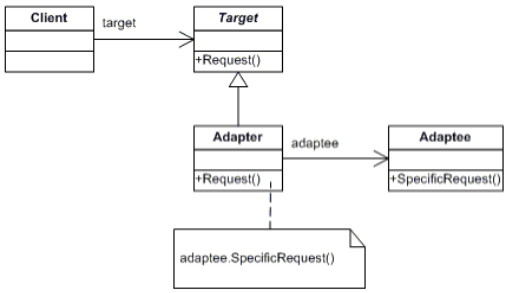
- Auswahl eines geeigneten Patterns
  - Kenntnisse über die Arbeitsweise von Design Patterns
  - „Zweck"-Bereiche der einzelnen Patterns studieren
  - Beziehungen der Patterns beachten
  - Analyse der Patterns, die die gleichen Aufgaben haben
  - beachten von Entwurfsrevisionen
  - Analyse der variablen Elemente eines Entwurfs
- Verwendung des ausgewählten Patterns
  - nach der Auswahl
    1) Vergewisserung, dass das Pattern für die Problemstellung das richtige ist
    2) Struktur, Teilnehmer und Interaktionen des Patterns verstehen
    3) Beispielcode betrachten
    4) Namen für Klassen/Objekte festlegen
    5) Definition der Klassen (Schnittstellen, Vererbung, …)
    6) Namen für Operationen festlegen
    7) Implementierung der Operationen

Auszug aus GoF-Patterns

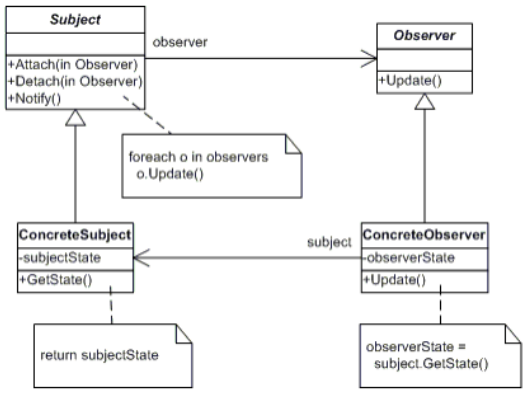
- Proxy
  - objektbasiertes Strukturmuster
  - Zugriffskontrolle eines Objekts mit Hilfe eines vorgelagerten Stellvertreter-Objekts
  - Teilnehmer: Proxy, Subjekt, eigentliches Subjekt



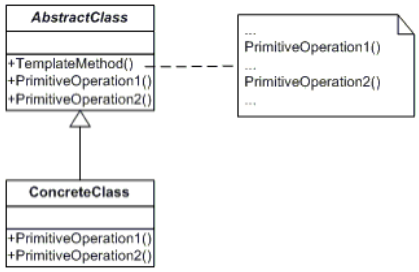
- Adapter
  - klassen-/objektbasiertes Strukturmuster
  - ermöglicht die Zusammenarbeit von Klassen, die inkompatible Schnittstellen haben
  - konvertiert die Schnittstelle einer Klasse in eine andere, vom Client erwartete Schnittstelle
  - Teilnehmer: Client, Ziel, Adapter, adaptierte Klasse



- Observer
  - objektbasiertes Verhaltensmuster
  - 1-n-Abhängigkeit zwischen Objekten
  - Änderung des Zustands eines Objekts führt zur Benachrichtigung/Aktualisierung aller abhängigen Objekte
  - Teilnehmer: Subjekt, konkretes Subjekt, Beobachter, konkreter Beobachter



- Template Method
  - klassenbasiertes Verhaltensmuster
  - die Verwendung einer Template Method ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern
  - Teilnehmer: Abstract Class (definiert abstrakte primitive Operationen, implementiert die Template Method als Skelett eines Algorithmus), Concrete Class (implementiert die primitiven Operationen, welche den Algorithmus ausführen



Enterprise Application Architecture (EAA) Patterns

- Domain Logic Patterns
  - Transaction Script, Domain Model, Table Model, Service Layer
- Data Source Architectural Patterns
  - Table Data Gateway, Row Data Gateway, Active Record, Data Mapper
- Object-Relational Behavioral Patterns
  - Unit of Work, Identity Map, Lazy Load
- Object-Relational Structural Patterns
  - Identity Field, Foreign Key Mapping, Association Table Mapping, Dependent Mapping, Embedded Value, Serialized LOB, Single Table Inheritance, Class Table Inheritance, Concrete Table Inheritance, Inheritance Mappers
- Object-Relational Metadata Mapping Patterns
  - Metadata Mapping, Query Object, Repository
- Web Presentation Patterns
  - Model View Controller, Page Controller, Front Controller, Template View, Transform View, Two-Step View, Application Controller
- Distribution Patterns
  - Remote Facade, Data Transfer Object
- Offline Concurrency Patterns
  - Optimistic Offline Lock, Pessimistic Offline Lock, Coarse Grained Lock, Implicit Lock
- Session State Patterns
  - Client Session State, Server Session State, Database Session State
- Base Patterns
  - Gateway, Mapper, Layer Supertype, Separated Interface, Registry, Value Object, Money, Special Case, Plugin, Service Stub, Record Set

Microsoft Patterns

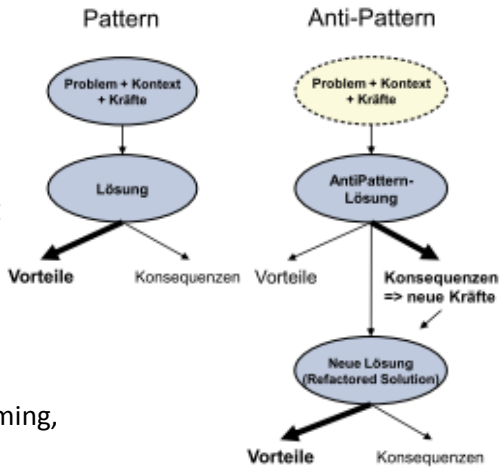
- Abstraktionsebenen:
  - Architecture, Design, Implementation
- Viewpoints
  - Database, Application, Deployment, Infrastructure
- Web Presentation Patterns, Deployment Patterns, Distributed Systems Patterns, Performance and Reliability Patterns, Services

Core J2EE Patterns

- Presentation Tier
  - Interception Filter, Front Controller, Application Controller, Context Object, View Helper, Composite View, Dispatcher View, Service To Worker
- Business Tier
  - Business Delegate, Service Locator, Session Facade, Application Service, Business Object, Composite Entity, Transfer Object, Transfer Object Assembler, Value List Handler
- Integration Tier
  - Data Access Object, Service Activator, Domain Store, Web Service Broker

Anti-Patterns

- umgekehrter Weg der Patterns
- identifizieren (häufig vorkommende) schlechte Lösungen
- (negative) Konsequenzen übertreffen Vorteile
- Grund: Übersehen wichtiger Kräfte
- Umwandlung der Anti-Pattern-Lösung in eine neue Lösung
- Ebenen von Anti-Patterns
  - Development Anti-Patterns, Architectural Anti-Patterns, Management Anti-Patterns
- Entwicklung von Anti-Patterns
  - double-checked looking, onion, copy + paste programming, lava flow, magic values



Agile Softwareentwicklung

Übersicht und Motivation

- Risiken bei der SW-Entwicklung
  - Terminverzögerungen, Projektabbrüche, Fehlerraten, System wird unrentabel, falsches Verständnis/Änderungen von GPs, Personalwechsel, …
- Oberbegriff für SW-Entwicklungsprozesse (Vorgehensmodelle), die sich durch geringen bürokratischen Aufwand und wenige, flexible Regeln und Rollen auszeichnen
- Reduzierung der Entwurfsphase auf ein Mindestmaß
- schneller Erhalt ausführbarer SW
- flexible Reaktion auf Kundenwünsche/Veränderungen
- Änderungskosten steigen nicht bzw. nur minimal mit Entwicklungszeit
- Notwendigkeit der Planung während des gesamten Entwicklungsprozesses
- neben den klassischen Methoden eine Art der methodischen SW-Entwicklung

Manifest für agile SW-Entwicklung

▪ Menschen und Zusammenarbeit ▪ funktionierende SW ▪ Zusammenarbeit mit den Kunden ▪ Reaktion auf Veränderungen	vor	▪ Prozesse und Werkzeuge ▪ umfassender Dokumentation ▪ vertraglicher Verhandlung ▪ Einhaltung eines Plans

Agile SW-Entwicklungsmethoden

- XP, SCRUM, Crystal, Lean Development, Adaptive Software Development, Feature-Driven Development, Pragmatic Development, Dynamic System Development Method, Software-Expedition

Extreme Programming

Übersicht und Motivation

- XP ist ein leichtgewichtiger agiler SW-Entwicklungsprozess für kleine Teams
- ermöglicht die Reaktion auf schnell ändernde Anforderungen
- Einsatz ideal bei riskanten Projekten mit dynamischen Anforderungen
- behandelt Prinzipien und Praktiken mit dem Schwerpunkt auf der Verantwortlichkeit der einzelnen Individuen
- Ziele
  - Anpassung des Entwicklungsprozesses an örtl. Begebenheiten
  - fortlaufende Verbesserung
- Variablen
  - Kosten, Qualität, Zeit, Umfang
  - Zusammenhang: stehen drei Faktoren fest, steht auch der vierte Faktor fest → Kunde kann nur 3 Faktoren bestimmen

die vier Werte des XP (core values)

- Kommunikation
  - SW-Entwicklung abhängig von einem effizienten Austausch von Informationen
  - in XP ist die Kommunikation möglichst zeitnah und direkt
  - Gespräche von Angesicht zu Angesicht (synchrone statt asynchrone Kommunikation)
- Einfachheit
  - do the simplest thing that could possibly work
  - Einfachheit ist die Kunst, Dinge nicht zu tun, die nicht unbedingt nötig sind
  - das XP-Team ist für die Umsetzung in Bezug auf den Entwicklungsprozess, als auch auf die zu entwickelnde SW verantwortlich
  - die Konzentration auf Einfachheit führt zu Lösungen, die leicht verständlich, schnell umsetzbar und leicht adaptierbar (agil) sind
- Feedback
  - wesentlicher Bestandteil der SW-Entwicklung besteht aus Lernen (Kunde & Entwickler)
  - Umsetzung mithilfe von kurzen Feedback-Zyklen
- Mut
  - zur radikalen Änderung (z.B. Refactoring)
  - zur Ehrlichkeit bei der Planung
  - zur Änderung von Anforderungen oder Prioritäten
  - wird bestärkt durch automatisierte Tests (→ Feedback)

XP-Prinzipien (core practices)

- Planungsspiel (Planning Game)
  - Entwicklungsplan wird von Programmieren und Kunden erstellt/angepasst
- kurze Release-Zyklen (Small Releases)
  - frühes ROI und Feedback der Anwender
- System-Metaphern (Metaphor)
  - gemeinsame Sprache zwischen Kunden und Entwicklern
  - Vision für die Systemarchitektur
- einfaches Design
  - ermöglicht leicht verständlichen und anpassbaren Code
- Tests (Testing)
  - Komponententests (Entwickler), Akzeptanztests (Kunde)
- fortlaufende Integration (Continual Integration) → siehe Continuous Integration
- Programmieren in Paaren (Pair Programming)
  - Erhöhung der Qualität durch regen Informationsfluss zwischen den Entwicklern
- gemeinsame Verantwortlichkeit (Collective Ownership)
  - „der Code gehört dem Team“ (→ „Truck-Faktor“)
- Refaktorisierung (Refactoring)
  - ständige Veränderung/Verbesserung des Codes
- ausdauerndes/nachhaltendes Tempo (Sustainable Pace)
  - „Marathon statt Sprint“
  - zu hohe Arbeitsbelastung führt zur Verringerung der Entwicklungsgeschwindigkeit
- ein Team (On Site Customer)
  - Kunde und Entwickler sind ein Team
- Programmierstandards (Shared Coding Standards)
  - ermöglicht effiziente Entwicklung im Team

Rollen in einem XP-Projekt

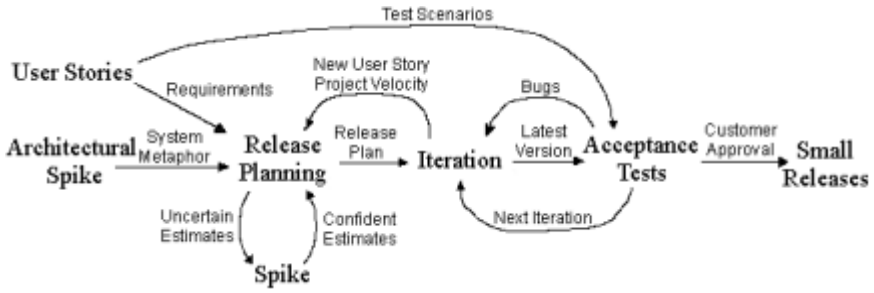
- Kunde, Entwickler, Coach (sorgt für die Einhaltung der XP-Prinzipien), Terminmanager (Tracker), Projektmanager, Tester, Berater (wird selten benötigt)

Techniken

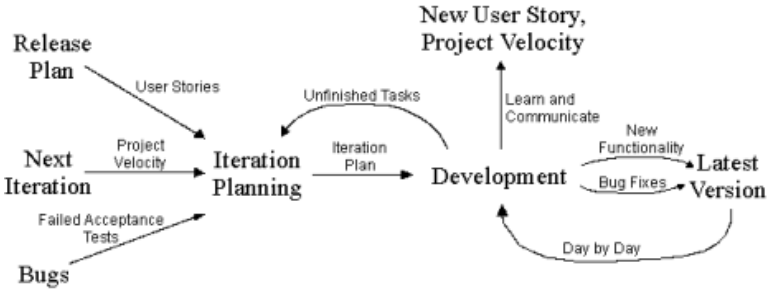
- für ein XP-Team
  - offene Arbeitsumgebung: räumlich enge Zusammenarbeit des Teams
  - kurze Iterationen: Zeitraum von 1-3 Wochen für eine Entwicklungsperiode
  - gemeinsame Sprache: gemeinsames Vokabular zur Diskussion
  - Retroperspektiven: nach jeder Iteration wird ein Rückblick durchgeführt
  - tägliches Standup-Meeting: kurze Besprechung der aktuellen Aufgaben
- für die Kunden
  - Benutzergeschichten: Anforderungen in Form einfacher Geschichten (Story Cards)
  - Iterationsplanung: Diskussion der Anforderungen, Aufwandsabschätzung, Auswahl der „Karten“
  - Akzeptanztests: Spezifikation funktionaler Abnahmekriterien
  - kurze Releasezyklen: Auslieferung nach 1-3 Monaten, um wichtiges Feedback zu erhalten
- für die Entwicklung
  - Pair Programming: paarweise Arbeiten am Code und wechseln der Partner

  - gemeinsame Verantwortlichkeit: der Code gehört zum Team
  - erst Testen (Durchführung von Unit-Tests)
  - Design für heute → YAGNI
  - Refactoring: ständige Verbesserung des Designs/Veränderung des Codes
  - fortlaufende Integration: automatischer Build-Prozess → Continous Integration
- für das Management
  - akzeptierte Verantwortung: Verantwortung liegt bei Programmierern und Kunden
  - Information durch Metriken: Darstellung des Fortschritts und zu lösender Probleme
  - ausdauerndes Tempo

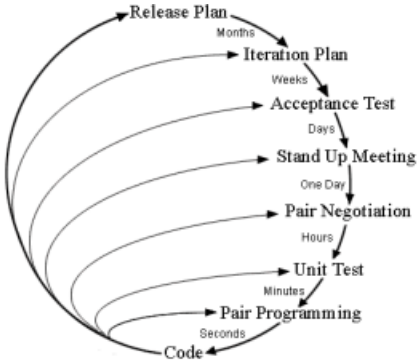
Extreme Programming Project



- Iteration



- Planning/Feedback Loops



YAGNI

- "You Aren't Gonna Need It"
- Funktionalität wird erst dann implementiert, wenn diese auch tatsächlich benötigt wird
- bei „Nicht-XP" werden häufig mögliche Funktionen schon bei der Implementierung berücksichtigt, was zu erhöhtem (häufig überflüssigem) Aufwand führt

Testdriven Development

- testgetriebene SW-Entwicklung
- traditionelle SE: erst Anforderungen umsetzen, dann testen

- TDD: erst testen, dann die Anforderungen umsetzen
- TDD führt zu einer benutzerorientierten Programmierung
  - es wird erst darüber nachgedacht, wie eine bestimmte Funktionalität verwendet wird, bevor diese implementiert wird
- Mantra: „Rot, Grün, Refactor“
  - Rot: schreibe einen Test, der fehlschlägt
  - Grün: sorge dafür, dass der Test läuft („egal wir, aber dafür schnell“)
  - Refactor: Code verbessern/aufräumen (Wiederholungen entfernen, etc.)

Contionous Integration

- Ziel: Vermeiden von Integrationsfehlern bei der Entwicklung im Team
- Grundvoraussetzung
  - vollständige Automatisierung des Build-Prozesses
    - Interaktion mit der Versionsverwaltung
    - Automatisierung der Compile-, Link- und Package-Vorgänge sowie aller Testläufe
    - Benachrichtigung aller beteiligten Personen über Build-Resultate
  - Daily Builds & Smoke Tests

Erfahrungsberichte

- Release Planning, Simplicity, System Metaphor, Pair Programming, Integrate Often, Unit Tests, Acceptance Tests

Rational Unified Process

Übersicht und Motivation

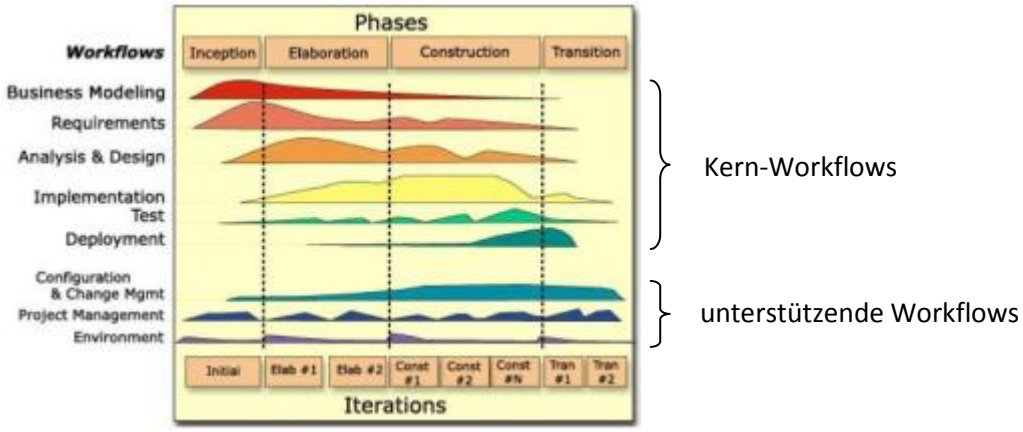
- iteratives Vorgehensmodell zur SW-Entwicklung
- beschreibt, wie mit bewährten Techniken SW effektiv erstellt werden kann
- regelt geordneten Umgang mit Aufgaben und Verantwortlichkeiten in einer Gruppe von Entwicklern

Workflow-Elemente

- Rolle: einer Rolle sind bestimmte Tätigkeiten und Artefakte zugewiesen
- Aktivitäten: Arbeitseinheit, die von einem Beteiligten in einer best. Rolle ausgeführt wird
- Artefakte: Ergebnis einer Aktivität (z.B. Dokumente, Modelle, Modellelemente)

Architektur

- die von dem Prozessmodell beschriebenen Workflows sind mittels UML-Diagramm definiert
- der Prozess beinhaltet 2 Dimensionen
  - horizontale Dimension: Zeit, Lebenszyklus (Phasen mit Iterationen)
  - vertikale Dimensionen: Kernprozesse (Workflows)
- Phasen
  - Inception (Anfang), Elaboration (Ausarbeitung), Construction, Transition
  - jede Phase besteht aus 1 bis n Iterationen
  - Iteration ähnlich dem Wasserfallmodell zusammengesetzt
  - jede Phase wird mit einem Meilenstein abgeschlossen



Best Practices

- Develop Software Iteratively
  - verwenden von kontrollierten Iterationen (siehe XP)
  - Idealfall: jede Iteration schließt mit einem ausführbaren Release ab
  - Ziele: Risiko minimieren, Reaktionsfähigkeit erhöhen, Problemverständnis schaffen
- Manage Requirements
  - Notwendigkeit eines Dokumentationsframeworks bei komplexen Projekten
  - RUP beschreibt die Dokumentation von Funktionalität, Bedingungen, Designentscheidungen, Business-Anforderungen
- Use Component-Based Architectures
  - Verwendung von komponentenbasierten Architekturen (Erweiterbarkeit, Wiederverwendung, häufig mit OOP verbunden)
  - RUP unterstützt systematische Verfahren zur Erstellung komponentenbas. Architekturen
  - Schwerpunkt auf eine früh ausführbare Architektur
- Visually Model Software
  - Verwendung von graphischen Modellen zur Abstraktion
  - ermöglicht bessere Gesamtsicht über das Projekt
  - Verbindung zwischen GPs und Code durch IT
  - RUP verwendet UML
- Continuously Verify Software Quality
  - Qualitätssicherung wird häufig vernachlässigt
  - RUP unterstützt die Planung von QS (Produkt-, Prozessqualität)
  - Einbeziehung aller Team-Mitglieder
- Control Changes To Software
  - Änderungen von SW können unterschiedliche Auswirkungen haben
  - RUP beschreibt Methoden zur Nachverfolgung und Aufzeichnung von Änderungen
  - RUP definiert „secure workspaces“, die einem Programmierer garantieren, dass Änderungen in anderen Systemen sein System nicht beeinflussen

RUP ist …

- architekturzentriert
  - Unterteilung der Gesamtsicht in fünf Sichten: logische Sicht (Endanwender), Implementierungssicht (Programmierer), Prozess-Sicht (System-Integrator), Verteilungssicht (System-Ingenieur), Anwendungsfall-Sicht (Analyst/Tester)

- anwendungsfallgesteuert (use case driven)
  - GPs werden durch Anwendungsfälle modelliert (UML)
  - Anwendungsfall-Modell fasst alle Anwendungsfälle zusammen und beschreibt die Funktionalität des Systems

benötigte Aktivitäten für RUP im Unternehmen

1) Ziele und Erwartungen festlegen (1 Tag)
   - Ergebnis: Vision, die Elemente/Aktivitäten beinhalten, die in (2) geplant werden
2) Auslieferung planen (1 Tag)
   - Ergebnis: Plan, wann welche Elemente/Aktivitäten wo vorkommen, Verantwortliche, Kosten
3) RUP instanziieren (2-10 Tage)
   - Ergebnis: projektspezifische Konfiguration des RUP (Development Case) als Dokument
4) Projektteam schulen (2,5 Tage)
   - Ergebnis: Team ist vorbereitet, RUP zu benutzen und ist vertraut mit dem Development Case
5) Projektteam beraten (i.d.R. 1 Tag/Woche in den ersten 6 Wochen, dann weniger)
   - Ergebnis: unabhängige Verwendung des Prozesses/Tool durch das Projektteam
6) Fortschritt evaluieren (2,5 – 3,5 Tage)
   - Ergebnis: Beurteilungsberichte für die key stakeholders

RUP und XP

- Vorgehen
  - inkrementelle, iterative, anwendungsfall-getriebene Vorgehen sowohl bei RUP als auch bei XP, jedoch
    - XP: jederzeit Änderungen der Anforderungen möglich
    - RUP: frühe Definitionen der Anforderungen
  - Arbeitsergebnisse
    - XP: Code und Tests
    - RUP: alle definierten Artefakte dauerhaft abgelegt
  - Designstrategie
    - XP: evolutionär
    - RUP: möglichst vollständiges Design direkt zu Beginn
  - unterschiedliche Rahmenbedingungen
    - XP: nur mittlere Projektgröße, Kunde vor Ort, Kommunikation der Entwickler
    - RUP: mittlere und große Projekte, kein Kunde vor Ort nötig

Scrum

Übersicht

- leichtgewichtiger agiles SE-Management-Prozess
- Begriff aus dem Rugby („als Team Boden gut machen“)
- besonders geeignet für SW in einer chaotischen, sich schnell ändernden Umgebung
- Scrum ermöglicht es, schnell und in regelmäßigen Abschnitten (2 – 4 Wochen) tatsächlich lauffähige SW zu inspizieren
- das Business setzt die Prioritäten, selbst-organisierende Entwicklungsteam legen das beste Vorgehen zur Auslieferung der höchstprioren Features fest
- geeignet für kleine Teams

Funktionsweise

Rollen

- Product Owner
  - legt das zu erreichende (gemeinsame) Ziel fest
  - Budget, Prioritäten
- Team
  - Aufwandschätzung der Backlog-Elemente
  - Implementierung
- Scrum Master
  - Überwachung der Rollen-/Rechteaufteilung

Artefakte

- Product Backlog
  - Features des Produktes, Funktionen und techn. Abhängigkeiten
  - detaillierte Beschreibung wichtiger Features

Wichtigkeit	Nr.	Beschreibung	Aufw.	Von
Sehr hoch				
	1	LDAP-Authentifizierung	20	MB
	2	Rollen-Verwaltung	18	TS

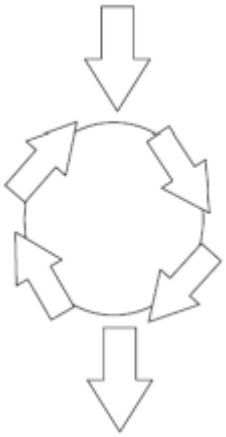
- Sprint Backlog
  - notwendige Aufgaben zur Erfüllung des Sprints
- Burndown Chart
  - tägliches Diagramm des Restaufwandes eines Sprints
- Impediment Backlog
  - enthält Hindernisse des Projektes

Funktionsweise

- Aufgaben stehen im Product Backlog
- 30-tägige Iterationszyklen (Sprints)
- Sprint Planning Meeting bei Beginn jedes Sprints
  - Product Owner priorisiert die Elemente des Product Backlogs
  - Scrum Team legt Aufgaben für bevorstehenden Sprint fest (Sprint Backlog)
- tägliches Meeting (15 min) während eines Sprints (Daily Scrum Meeting), geleitet vom Scrum Master
- Sprint Review Meeting am Ende jedes Sprints

Methodology

- Pregame
  - Planning
  - System Architecture/High Level Design
- Game
  - Sprints (Concurrent Engineering)
  - Develop (Analysis, Design, Develop)
  - Wrap
  - Review
  - Adjust
- Postgame
  - Closure



Scrum & XP

- Scrum betrachtet die Managementebene, XP die Entwicklungspraktiken
  - Kombination aus Scrum und XP möglich (z.B. User Stories aus XP werden als Backlog-Elemente bei Scrum verwendet)
  - Kombinationsprojekt xp@Scrum