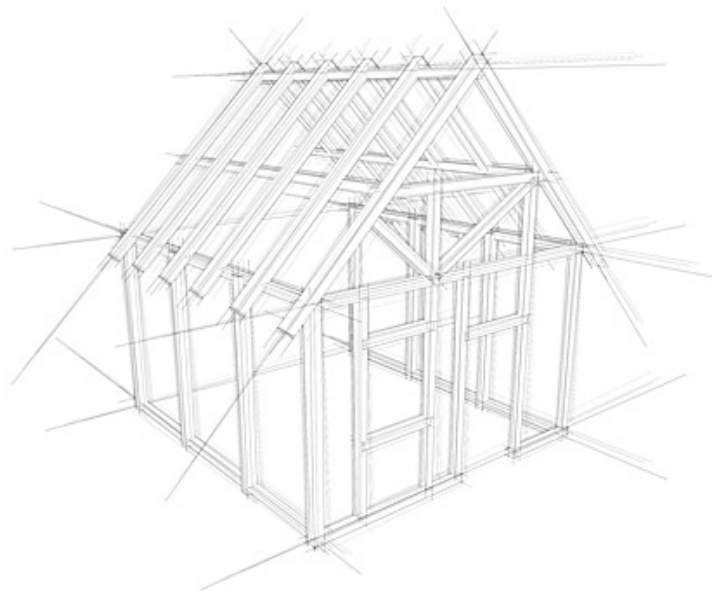


Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2018 Beuth Hochschule für Technik Berlin

ARC - Objektorientierte Architektur



Überblick und Lernziele




Lernziele

Nachdem in den vorigen Lerneinheiten UML sowie die Methodik des Analyse- und Designzykluses vorgestellt wurden, geht es nun darum dieses Wissen in einer guten Architektur zu berücksichtigen und somit ein generelles Architektur Know-how zu erlangen.

Derartiges Wissen kann überwiegend nur in der Praxis und in der Diskussion mit Kollegen, Profis oder dem Dozenten erworben werden.

Diese Lerneinheit soll ein Fundament für die Praxisarbeit und für Diskussionen legen, indem

- Techniken, Gebiete und Vorgehensweisen erläutert werden.
- Sie Begriffe kennenlernen, die in der Softwarearchitektur verwendet werden.
- Sie die Bedeutung der Entwurfsmuster Technologiearchitekturen / Architekturstile und Entwurfsmuster in der Softwarearchitektur kennenlernen. (z. B.  Enterprise Design Patterns)

Bei dieser Lerneinheit sollen Sie als angehender Architekt eine Art „Schweizer Messer“ in Form einer Checkliste erstellen. Diese Lerneinheit enthält viele Dutzend Sichtweisen, Werkzeuge, Prinzipien, Ebenen, Design Patterns, Architekturen, etc., die alle den **Fundus für ein Architekturverständnis** bilden. Jedes Einzelne für sich ist vielleicht nicht besonders spannend. Aber in seiner Gesamtheit machen alle Punkte zusammen ein **Architektur-Portfolio** aus.



Zeitbedarf und Umfang

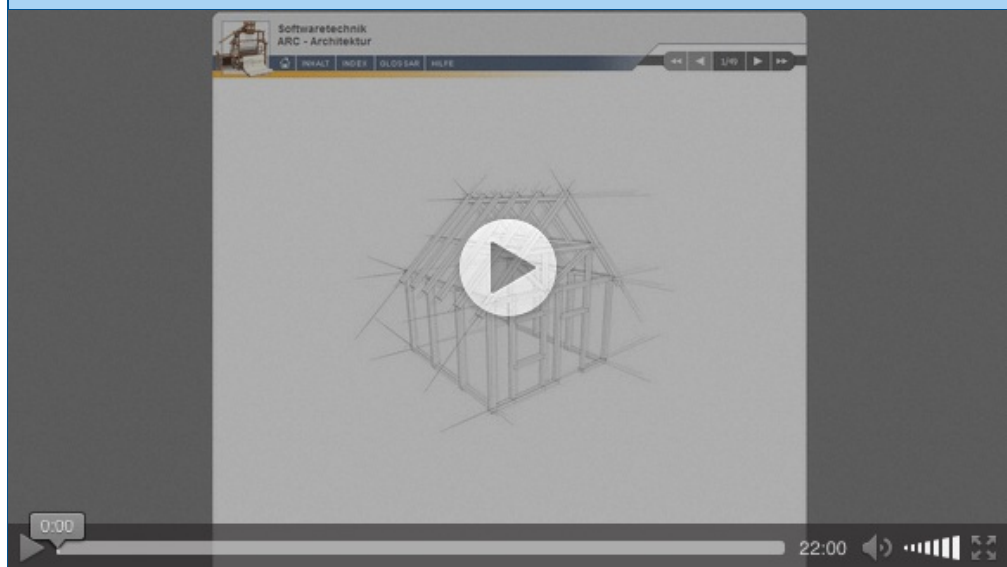
Für die Bearbeitung dieser Lerneinheit benötigen Sie 180 Minuten.

Das Entwickeln, Vorstellen und Diskutieren einer eigenen rudimentären Softwarearchitektur sind ca. 5 Stunden vorgesehen.



Film

Webkonferenz zur Lerneinheit ARC



© Beuth Hochschule Berlin - Dauer: 21:59 Min. - Streaming Media 30.1 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

Literatur zum Thema Architektur

Es gibt einige gute deutsche Bücher zu diesem Thema wie beispielsweise von **VOGEL** und von **BIEN**. Essentiell sind in dieser Kategorie aber auch (Enterprise) Pattern Bücher, da gute Architekturen derartige EE-Patterns aufweisen.

Bücher

VOGEL, ARNOLD, ET AL. (2008): Software-Architektur. Grundlagen - Konzepte - Praxis. Spektrum Akademischer Verlag, ISBN-13: 978-3827415349

RALF REUSSNER, WILHELM HASSELBRING (2006) Handbuch der Software-Architektur. dpunkt Verlag, ISBN-13: 978-3898645591

BUSCHMANN ET AL. (2007): A System of Patterns: Pattern-Oriented Software Architecture. Wiley & Sons, ISBN-13: 978-0471958697 (siehe dazu auch die Lerneinheit Enterprise Patterns)

ADAM BIEN, (2006) Enterprise Architekturen: Leitfaden für effiziente Software-Entwicklung, entwickler.press, ISBN-13: 978-3868020137

ADAM BIEN, (2007) Java EE 5 Architekturen. Java Patterns und Idiome, entwickler.press, ISBN-13: 978-3939084242

GERNOT STARKE (2008): Effektive Software-Architekturen. Ein praktischer Leitfaden. Hanser, ISBN-13: 978-3446412156

MARTIN FOWLER (2003): Patterns of Enterprise Application Architecture. Addison-Wesley Longman, ISBN-13: 978-0321127426

IAN GORTON (2006) Essential Software Architecture. Springer, ISBN-13: 978-3540287131

MARTIN VAN DEN BERG, MARLIES VAN STEENBERGEN (2006): Building an Enterprise Architecture Practice: Tools, Tips, Best Practices, Ready-to-Use Insights (Enterprise). Springer Netherlands, ISBN-13: 978-1402056055

Nicht mehr ganz aktuell aber dennoch sehr gut:

DEEPAK ALUR, DAN MALKS, JOHN CRUPI, (2001) Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall, 2003, ISBN-13: 978-0131422469

LEN BASS, PAUL CLEMENTS, RICK KAZMAN (2003) Software Architecture in Practice. Addison-Wesley Longman, ISBN-13: 978-0321154958

Links

[!\[\]\(84f47badaad7772cd95667a7c387a639_img.jpg\) Architecture Wiki](#)


[!\[\]\(28f72b996fc97883dfd9d4e8b1b16b4e_img.jpg\) .NET Architecture](#)


[!\[\]\(5d954b3e270654ad8ab0d5913161c03c_img.jpg\) Ressource Seite](#)


[!\[\]\(aff7c69c44a5e015f18c35867ef3f5c3_img.jpg\) Carnegie Mellon University Architecture Site](#)


Podcasts


General website:  <http://www.se-radio.net>

 [SE RADIO Episode 115 *Architecture Analysis*](#)


 [SE RADIO Episode 87 *Software Components*](#)


 [SE RADIO Episode 82 *Organization of Large Code Bases*](#)

 [SE RADIO Episode 74 *Enterprise Architecture II*](#)


 [SE RADIO Episode 54 *Interview Frank Buschmann*](#)


 [SE RADIO Episode 41 *Architecture Patterns \(Architecture Pt. 4\)*](#)

 [SE RADIO Episode 34 *Enterprise Architecture*](#)

 [SE RADIO Episode 30 *Architecture Pt.3*](#)

 [SE RADIO Episode 27 *Service Oriented Architecture Pt.1*](#)

 [SE RADIO Episode 25 *Architecture Pt. 2*](#)

 [SE RADIO Episode 23 *Architecture Pt. 1*](#)

1 Einführung

Stellen Sie sich bitte ein Großprojekt vor, in dem es 50-100 Domain-Klassen und Hunderte von Views im GUI gibt. Das Budget des Projektes ist viele Millionen Euro hoch. Es sitzen ca. 20 Developer vor ihnen die gleichzeitig sofort loslegen wollen und durchaus unterschiedliches Wissen und Erfahrungen haben. Warum ist das ein Problem?

Sie erahnen vielleicht, dass es neben einem guten Projektmanagement und den üblichen Spezifikationen, Analyse und Designergebnissen generell einer guten Softwarearchitektur bedarf.

Entwickler, Komponenten und Daten müssen mit vielen Informationen / Richtlinien und Visionen versehen und koordiniert werden. Passiert dies nicht, entsteht in kurzer Zeit „Spaghetticode“ - dann versteht kein Entwickler mehr den Code und das System ist nicht mehr wartbar.

Der Begriff „Architektur“ ist schwer zu fassen, da seine inhaltliche Ausgestaltung oft subjektiv ist.



Definition

Softwarearchitektur

Softwarearchitektur verleiht einem IT-System eine sinnvolle Struktur, Anordnung und einen Rahmen. Diese bestehen beispielsweise aus Regeln für das Zusammenspiel oder den Aufbau der Komponenten.

Weitere Definitionen können Sie auf der Webseite des www.sei.cmu.edu Software Engineering Institute von Carnegie Mellon finden.

Es wird bei der obigen Definition auf zwei wesentliche Elemente verwiesen:

1. **Unterteilung / Identifizierung:** Die Codeelemente müssen klar trennbar sein, aber gruppiert werden. Der Softwareentwickler muss sich quasi „reindrillen“ und den Grad der Abstraktion in der Sichtbarkeit festlegen können. Code wird unterscheidbar getrennt und / oder sinnvoll gruppiert. Beispielsweise in Klassen, Packages, Design Patterns, Deployment Views, etc.
2. Für die Interaktion der Komponenten gibt es **Regeln**. Bestimmte Zugriffe sind verboten, erlaubt oder dürfen nur auf bestimmte Weise funktionieren.

Dazu zählen Dinge wie:

- Kein Zugriff auf private Methoden
- Package **a.b.c** sollte nicht direkt auf Package **g.n.z** zugreifen
- Die Kommunikation zwischen Komponente **A** und **B** finden über einen Webservice statt
- Paket **d.e.f** greift auf Paket **d.e.g** nur über eine Fassade zu

Warum sind diese beiden Punkte so wichtig?


Für eine gute Softwarearchitektur sprechen einige gute Gründe.

- Software sollte nicht in Form eines www.bbmud.com Big Ball of Mud, also eines großen Schlammhaufens vorliegen. Die unstrukturierte Anhäufung von (oft funktionalen) Codeeinheiten wird unübersichtlich.
- Wird der Code **komplex**, kann er besser verstanden werden, wenn er strukturiert ist. Man findet sich besser zurecht.
- Code kann besser **erweitert** und gewartet werden wenn er strukturiert ist.
- Struktur im Softwaresystem spiegelt **Domänen auf Aufgabenbereiche** wieder. Diese können dann wiederum viel besser und unabhängig von den jeweiligen Spezialisten bearbeitet werden.
- Korrektheit und Qualität lassen sich in einer guten Architektur besser sicherstellen.



Hinweis

Die Ziele einer Strukturierung sind häufig auch **widersprüchlich**. Wie generell bei Design Patterns kann der Code mit Struktur oder Patterns größer werden, was oft ein Nachteil sein kann.

Architektur wird - aus naheliegenden Gründen - oft mit der Baukunst verglichen, wie das bei Design Patterns mit dem Wirken von  Christopher Alexander getan wurde. Auch in der Baukunst gibt es Strukturierungen, Anleitungen und Best Practices für den Bau eines Architekturobjektes (Haus, Wolkenkratzer, Brücke, etc.).

Wartung ist ein wesentliches Ziel der Softwarearchitektur. Durch wartbare Software wird der Aufwand für Wartung und Pflege **minimiert**. Dies kann man leider nur indirekt messen. Die Lerneinheit Softwaremetriken beschäftigt sich mit dieser **Kohäsion** von Software. Es gibt viele direkte Faktoren, die sich nicht messen lassen, die aber ein Softwareprodukt unwartbar machen (z. B. eine ungeeignete Namensgebung). Software muss schnell erlernbar und schnell änderbar sein.

Erstes Fazit

Wir halten also fest. Ziel einer guten Softwarearchitektur sind im Wesentlichen:

- **Änderbarkeit**
- **Wartbarkeit**
- **Lesbarkeit**
- **Qualität**

Diese Kriterien sind auch schon in den Lerneinheiten Refactoring, Design oder Design Patterns aufgetaucht.

2 Architektur Rahmen

Architektur umfasst verschiedenste Aufgaben und Disziplinen und ist in seiner Gesamtheit äußerst unübersichtlich. VOGEL, ARNOLD ET AL. haben versucht die Bereiche zu ordnen.

WAS

Hier geht es um die Art der Architektur: Software-Architektur, Datenarchitektur, Sicherheitsarchitektur, Enterprise-Architektur, Integrationsarchitektur, Netzwerkarchitektur, Managementarchitektur. Und natürlich um alle der im Folgenden vorgestellten Bestandteile der Architektur.

WO

- Auf welchen Software-Schichten wird gearbeitet?
- Wo befinden sich Komponenten?
- Welche Architektur Ebenen gibt es?

WARUM

- Warum legt die Problemstruktur im Großen und Kleinen diese Architektur fest?
- Warum müssen Komponenten so angeordnet sein und so kommunizieren?

WOMIT

- Welche Patterns, Frameworks, Technologiearchitekturen sind zu verwenden?

WER

Beschäftigt sich mit den beteiligten Personen. Es gibt nicht nur den Architekten. Der Architekt muss sich mit den Designern, den Anforderungsermittlern und vielen weiteren Personengruppen austauschen, um die beste Architektur zu finden.

WIE

Dieser Punkt beschäftigt sich mit der Vorgehensweise des Architekten.

All diese Punkte werden im Folgenden angesprochen.

3 Architektur Werkzeuge

In der Literatur aber auch in der Praxis gibt es viele Ebenen über die ein Architekt die Strukturen und Regeln des Systems beeinflussen kann.

Im übertragenen Sinne sind dies Werkzeuge, die dem Architekten zur Verfügung stehen, um das System nach den in der Einleitung genannten Zielen und Qualitätsmerkmalen zu gestalten - im Wesentlichen die Änderbarkeit, Wartbarkeit und Verständlichkeit.

Es sind hier keine Werkzeuge im Sinne von Tools gemeint.

3.1 Vision

Architektur ist zu einem großen Teil auch Vision und zwar nicht nur eine Vision des Gesamtsystems, sondern auch eine Vision eines Bestandteiles.

Dazu zwei Beispiele:



Beispiel

Beispiel für Architektur-Vision (1)

Es soll ein Client GUI mit einer Engine über ein Protokoll kommunizieren. Initial sind 4 Kommandos bekannt. Ein Entwickler kann die 4 Kommandos abfangen und stür implementieren. Wird aber initial gleich die Pattern Vision eines Pluggable-Selectors mitgeliefert, so denkt und implementiert der Entwickler anders.



Beispiel

Beispiel für Architektur-Vision (2)

Man denke sich eine Spezifikation:

1. Drehmoment 500NM bei 2000 Umdrehungen/Minute
2. Leistung 400 PS
3. Verbrauch 8-10l/100km bei 90 km/h
4. 8 Zylinder
5. Direkteinspritzung.

Auch hier kann der Entwickler gleich sofort beginnen und bauen. Aber was ist die Vision? Was soll das? Ist ein Rennwagen oder ein Lastwagen das Ziel? Wie man hier sieht, können Visionen viele Spezifikationen ersetzen oder ergänzen (Beispiel nach **BIEN**).

3.2 Packages / Namensgebung

Pakete (engl. Packages) sind die wichtigsten Gestaltungsmittel für die Softwarearchitektur. Es sind Namensräume die ideal bei der Strukturierung und Trennung der Komponenten helfen.

Pakete sind Hilfsmittel, um Schichten und die Sichtweisen der Entwickler oder sonstigen Personen zu strukturieren. Sie repräsentieren gleichzeitig einen jeweiligen Aspekt des zu entwickelnden IT-Systems. Z. B. könnte die Persistenzschicht im Paket `de.swt.flugbuchung.persistenz` abgebildet werden.

Packages gruppieren vertikal. Auf der horizontalen Ebene ist in diesem Bereich die **fachliche Komponente** anzusiedeln. Wie bereits im Design Teil erläutert, ist es immer sinnvoll horizontal und vertikal zu unterteilen. Also sowohl vertikal in Schichten, als auch fachlich innerhalb der Schichten. Fachliche Komponenten können Klassen sein.

Sowohl Packages als auch fachliche Komponenten trennen und gruppieren.

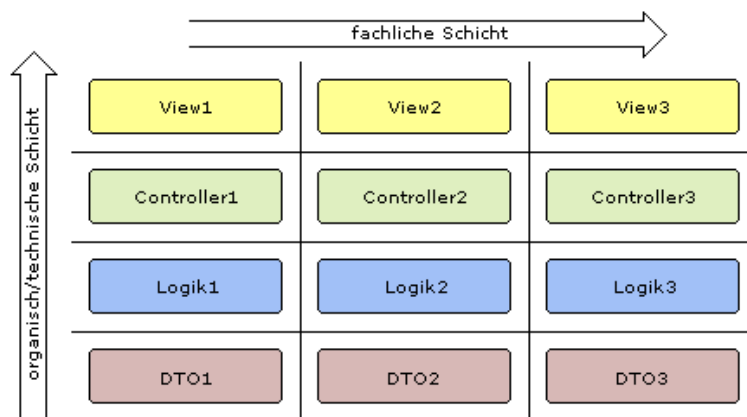


Abb.: Beispiel für Gruppierung



Hinweis

Unerfahrener Entwickler neigen dazu Dinge zusammenzufassen. Oft hat man das Gefühl, es ist leichter zu überblicken, wenn Dinge umfassend an einem Ort oder in einer Datei sind. Dieses Gefühl ergibt sich oft, wenn man in der Thematik drin ist. Jedoch ist es für andere Entwickler (oder wenn man selbst einmal länger raus ist) viel besser, Dinge klein zu halten!

Sowohl Architekt und Entwickler haben schließlich das Architekturmittel der Namensgebung. In der gesamten vertikalen und horizontalen Ebene kann man mit einer geeigneten Namensgebung dazu beitragen, dass sowohl Trennung als auch logische Zusammengehörigkeit wichtig sind.

3.3 Patterns / Kapselung / Abstraktion

Design Patterns und Enterprise Design Patterns sind ein extrem wichtiger Teil der Architektur. Ein Design Pattern **strukturiert Komponenten und orchestriert den Zugriff**. Ein Design Pattern ist damit eine Architektureinheit. Unter Orchestrieren wird hier die Steuerung des gegenseitigen Zugriffes verstanden.

Design Patterns tragen schon in der Dokumentation dazu bei, dass Entwickler das System leichter verstehen und ggf. erweitern können.



Beispiel

Beispiel für Design Pattern

Ein guter Developer wird schon eine Idee im Kopf haben wie er das System erweitert, wenn ihm der Architekt sagt, das Subsystem A ist nach einer Strategie strukturiert. Auch ohne dass der Entwickler die 5 bereits bestehenden Klassen gesehen hat weiß er, dass eine Algorithmenauswahl nach Namen vorkommt und er nur das generelle Interface zu kennen braucht!

Erstaunlicherweise kommen die meisten - auch größeren Systeme - mit relativ wenig (5-10) Patterns aus. Jedoch ist die richtige Wahl der Patterns auch maßgeblich entscheidend für die gute Architektur des Systems.

Das Prinzip der **Kapselung** beruht darauf, Dinge nach außen hin zu **verstecken**. Es ist ebenfalls ein wichtiges Merkmal der Architektur, um z. B. Spaghetticode zu vermeiden. Durch dieses Prinzip wird also die Kohäsion / Kopplung der Komponenten verringert.



Beispiel

Beispiel für Kapselung

1. Komponenten / Klassen trennen in **public & private**. Für den public Teil werden Interfaces definiert.
2. Packages sollten üblicherweise auch dafür sorgen, dass ihre Bestandteile gekapselt werden. So sollten z. B. für den Packagezugriff **Fassaden** definiert werden, die ein passendes Interface publizieren.

Ein Kernprinzip eines Architekten besteht darin, auf Abstraktion zu achten. Gibt es Konzepte für die Abstraktion? Werden Interfaces und abstrakte Klassen verwendet? Wird MDA (Model Driven Architecture) eingesetzt?

Was ist mit den drei Fragen bzw. wie sollen sie hier verstanden werden?

Eine Software, die das richtige Gleichgewicht an konkreten und abstrakten Komponenten hält, ist weniger änderungsempfindlich - also unter Umständen auch weniger stark gekoppelt.

3.4 Kohäsion / Kopplung

Die **Kohäsion** von Komponenten bestimmt Ihre Ähnlichkeit / Zusammengehörigkeit. Der Architekt richtet hier ein Auge auf die **fachliche** und nicht auf die technische Zusammengehörigkeit. Verwendet beispielsweise ein Client und ein Server die gleichen Fachklassen - wie z. B. Buch oder ISBN, so sollten diese nicht in verschiedene Packages entsendet werden.

Die **Kopplung** hat auf der anderen Seite nicht das Fachliche im Blick, sondern schaut sich nur die Benutzrelationen an. Also den Grad der Abhängigkeit zwischen Komponenten. Dies sind meistens Assoziationen. Aber auch Datenabhängigkeiten sind interessant und gehören in eine genaue Analyse.

Schwache Kopplung

Grundsätzlich ist eine schwache Kopplung erstrebenswert. Es gibt mehrere Möglichkeiten dies zu erreichen, unter anderem:

- der Einsatz von Dependency Injection Frameworks
- der Einsatz von Design Patterns wie Factories
- die Kommunikation über ein Protokoll. Z. B. durch Middleware / Messaging.

Bei letzterem werben EAI Hersteller mit z. B. Bussystemen um die Gunst der Projektarchitekten.

spröde - brittle

Systeme die zu stark gekoppelt sind, werden als spröde (brittle) bezeichnet - diese brechen bei Belastung (=Änderung). Alle Systeme sind initial recht stabil. Je größer sie werden, desto spröder werden sie in den meisten Fällen. Dann helfen die oben genannten Technologien.

3.5 Standards

Die Entscheidung der zugrundeliegenden und verwendeten Technologie durch den Softwarearchitekten - beispielsweise die Verwendung von Java EE mit JBoss - ist gleichzeitig auch die Entscheidung für einen bestimmten Standard.

Diese beispielhafte Aussage impliziert bereits einige Standards wie z. B. die Verwendung von Enterprise Java Beans oder einem Kommunikationsprotokoll wie JMS.

Tatsächlich legen die bereits bisher genannten Architekturgebiete nahe, dass Standard und Frameworks ein kleiner, wenn auch wichtiger Teil der Architekturhandlungen sind. Sie implizieren eine Architektur, die sich dann nicht mehr verändern lässt. So kann beispielsweise eine Anwendung mit „Ruby on Rails“ nicht ohne eine MVC-Architektur realisiert werden.

Weitere wichtige Standardelemente sind:

- Coding Standards
- Werkzeugstandards wie JUnit, Ant, Cruise Control, etc.

Die Wahl des zugrundeliegenden **Frameworks** ruft regelmäßig hitzige Diskussionen hervor. Besonders GUI Technologien (rich vs. thin), Datenbanken und Datenbankanmapper sind häufig Gegenstand einer - oftmals viel zu kleinen - Evaluation. Oft werden sogar eigene Frameworklösungen implementiert, obwohl Standards hier zuverlässigere Alternativen bieten.

Damit wird auch klar das hier nicht so sehr wissenschaftliche Standards oder Normierungsstandards wie z. B. H.261 gemeint sind.

3.6 Zuständigkeit / Geschwätzigkeit / Sprödigkeit (Brittleness)

Schauen wir uns nun drei weitere Architekturmittel nach A. BIEN an.

Zuständigkeit

Jede Komponente sollte ausschließlich das tun wofür sie vorgesehen / **zuständig** ist.

Wie bereits erwähnt, sollten Komponenten eher **kleine Aufgaben** erledigen - dabei sollten diese allerdings gründlich und richtig umgesetzt werden. In der Lerneinheit Refactoring haben wir viele Beispiele kennengelernt, wo man falsche Zuständigkeiten erkennt und beseitigt.



Beispiel

Zuständigkeit

- Macht eine Methode schon zu viel? Muss diese mit der **extract Method** zerschnitten werden?
- Ist eine Methode zu sehr an den Daten anderer Klassen interessiert?
- Bearbeitet eine Klasse selbst viele Interna anderer Klassen? (Z. B. eine **Person Klasse**, die bereits die Daten **Kauf** und **Bezahlung** beinhaltet!)

Entwickler neigen fast immer zu großen Komponenten, die alles auf einmal erledigen sollen. Dies mag für Insider (One-Man Developer) effizient sein. Auf lange Sicht ist ein solches System aber viel schwerer zu warten und zu verstehen.



Beispiel

Kleine Klassen und „sprechende“ Methoden

Ein bekannter Softwareentwickler berichtete einmal von einer gemeinsamen Code-Session mit **KENT BECK**. Dabei fiel auf, dass **BECK** extrem kleine Klassen und „sprechende“ **public (!)** Methoden verwendete. Der Besucher berichtete, dass sich allein daraus schon eine sehr leuchtende Systemarchitektur ergab.

Geschwätzigkeit

Unter **Geschwätzigkeit** eines Systems versteht **BIEN** den Grad der Kommunikation der nötig ist, um ein System zu gestalten. Wir haben bisher gelernt, dass es sinnvoll sein kann, ein IT-System zu entkoppeln und beispielsweise Protokollschnittstellen einzuführen (z. B. ein RMI oder ein Web-Service Protokoll). Dies wirkt sich in der Regel positiv auf den Kopplungsgrad des Systems aus. Und natürlich auch auf die „Verteilbarkeit“ des Systems.

Auf der anderen Seite können sich Systeme auch „totkommunizieren“; so wie es in Firmen auch ständig Sitzungen geben kann, ohne das inzwischen wirklich mehr gearbeitet wird. So gibt es berühmte Beispiele von Systemen, die - als Web Service designed - nicht mehr skalieren.

In einem solchen Fall muss der Architekt das Gleichgewicht zwischen Entkopplung und Anti-Geschwätzigkeit finden. Keine leichte Aufgabe!

Sprödigkeit

Der Begriff **Sprödigkeit** (Brittleness) wurde bereits bei dem Thema Kopplung diskutiert und hängt stark mit diesem zusammen.

4 Architektur Prinzipien

Die bisherigen Auflistungen von Werkzeugen und Stilmitteln die dem Architekten zur Verfügung stehen, werden von verschiedenen Autoren wie beispielsweise von **VOGEL ET AL.** variiert, zusammengefasst und erweitert.

Beispielhaft sind hier 10 wichtige Prinzipien aufgeführt.

1. Inkrementalität - Durch Prototyping und schrittweises Wachstum
2. Lose Kopplung - Durch Gesetz von Demeter und Vermeidung von Zyklen
3. Entwurf für Veränderung - Test was passiert, wenn Komponenten sich ändern oder wegfallen
4. Hohe Kohäsion - gute fachliche Gruppierungen überleben auch Technologieshifts (z. B. von Corba nach JavaEE)
5. Rückverfolgbarkeit - Änderungen, Fehler (Exceptions) und Datenflüsse transparent machen
6. Abstraktion - Interfaces als explizite Schnittstellen, Subclassing, Polymorphie
7. Information Hiding - Fassaden, Black Box Prinzip, Schichtenbildung
8. Separation of Concerns - Analyse der Zuständigkeiten
9. Modularität - Anwendung der Sprachmittel für Gruppierung
10. Selbstdokumentation - Namensgebung, interne und externe Dokumentation



Übung

Übung ARC-01

Architekturkriterien am eigenen Projekt überprüfen

Gehen Sie die Architektur Ihres Projektes - das Sie im Studienmodul begleitend durchführen - einmal auf diese Kriterien hin durch und prüfen Sie die Tragfähigkeit Ihres Ansatzes.

Bearbeitungszeit: 45 Minuten



Übung

Übung ARC-02

Checkliste Architektur

Erstellen Sie für sich eine Checkliste aus allen Punkten die in dieser Lerneinheit behandelt wurden. Welche Punkte halten Sie für wichtig und in welcher Reihenfolge? Stellen Sie sich vor, Sie sind Architekt und müssen eine solche Checkliste für das nächste Projekt parat haben.

Bearbeitungszeit: 30 Minuten



Übung

Übung ARC-03

Diskussion über Architektur

Diskutieren Sie mit Ihrem Dozenten über die Architektur-Prinzipien. Wie gewichten Sie Ihre Prinzipien als Architekt?

Stimmen Sie sich mit Ihrer Kursbetreuung ab, wie die Diskussion durchgeführt wird. Die asynchrone Kommunikation ist über das Forum, im Lernraumsystem, möglich. Eine synchrone Kommunikation hingegen ist beispielsweise über eine Audiokonferenz oder auch die Präsenzveranstaltung machbar.

Bereiten Sie ein eigenes Statement vor, welches Sie in die Diskussion einbringen.

Bearbeitungszeit: 30 Minuten

5 Architektur Ebenen

In den vorigen Kapiteln wurden die horizontalen und vertikalen Ebenen bereits angesprochen und werden hier noch einmal aufgegriffen.

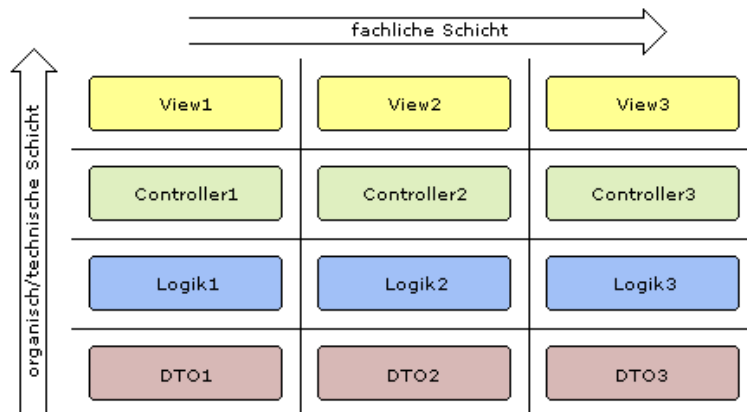


Abb.: Architektur Ebenen

Vertikale Ebene

Auf der **vertikalen Ebene** wird unterteilt, benannt, strukturiert und gruppiert. Dies können die folgenden Ebenen sein.

- **Ziel:** (Gesamtziel) Dies kann eine Anwendung für mehrere Länder und Unternehmen sein.
- **Firma:** Für eine Firma wird die Anwendung X entwickelt. Die muss aber vielleicht mit anderen Applikationen niederlassungsweit kommunizieren. Hier wäre auch ein Deployment möglich.
- **Anwendung / System:** In der Firma selbst gibt es eines oder N Systeme.
- **Software Schicht:** Ein System ist grob strukturiert in z. B. eine MVC Architektur oder in ein N-Schicht - Modell (Präsentation, Dialog-Kontrolle, Applikations-Kernel, Persistenzschicht, Datenbank)
- **Package, Komponentendiagramme:** Strukturieren die Grobschichten feiner.
- **Klassen:** **Klassen sind** die unterste Schicht einer Komponente. Darin sind viele verschiedene Dinge, wie Funktionsgruppen, Module, oder strukturierte Datenstrukturen enthalten.

Horizontale Ebene

Auf **horizontaler Ebene** finden wir die fachlichen Ausprägungen:

Einige Beispiele sind:

- View Objekte
- Controller für X, Y, Z
- Berechnung für Zinsen, Steuer, Rente oder Schachengine A, B, C
- Domain Objekte
- DTO Objekte (Data Transfer Objekte)

Dies ist aber nur eine - recht gute - Möglichkeit, die Welt zu sehen. Es gibt generell viele Sichtweisen, um ein potentielles IT-System zu betrachten.

- **Geschäftssicht:** Fachlichkeit, Organisation, Geschäftsprozesse, Interessenvertreter oder Anforderungen sind hier zu finden.
- **Logische Sicht:** Bausteine der Anwendung, Systemgrenzen, Verarbeitungsinstanzen, Schnittstellen.
- **Datensicht:** Datenbanktabellen, Ein- und Ausgabequellen, Datenflüsse.
- **Technologiesicht:** Welche Tools werden verwendet, Frameworks, IDEs, Richtlinien, Tests, Codeverwaltung, Qualitätssicherung.
- **Verteilungsschicht:** Deployment, Installation, Konfiguration, Betrieb.

Auf dieser Basis gibt es eine Vielzahl von älteren Modellen teils standardisiert oder als Vorschlag:

- RM-ODP der ISO / IEC (ISO10746) Referenzmodell für Open Distributed Processing
- Zachmann Framework (IBM)
- 4+1 Sicht von Malveau und Mowbray

6 Architekturstile

Stile und Entwurfsmuster voneinander zu trennen ist nicht ganz einfach. In diesem Teil wird unter Architekturstil eher ein Kommunikationsprinzip verstanden, mit dem Komponenten interagieren. Die nachfolgende Unterteilung geht dabei auf **SHAW** und **GARLAN** zurück.

 [SG96]

Diesen Stilen und Patterns ganz allgemein liegt die folgende Philosophie zugrunde:

- Es benötigt **Komponentengruppen** - miteinander interagierende Einheiten.
- Die **Anordnung** der Komponenten.
 - Was ist wem vorgeschaltet?
 - Wer erbringt wo welche Aufgabe?
 - Welche Topologie liegt vor?
- Es bedarf **Konnektoren** - über definierte Schnittstellen oder Kommunikationsendpunkte werden Informationen ausgetauscht oder aufgerufen.
- Meist gibt es noch **Constraints** - Randbedingungen / semantische Einschränkungen, die festlegen was nicht oder wie getan werden darf.

Ein Beispiel für Constraints wäre, das B nicht mit X kommunizieren darf oder das praktisch angewendete Prinzip des Information Hiding, das nicht gebrochen werden darf.

6.1 Ordnung der Architekturstile

Die nachfolgenden Stile sind Richtlinien für den Architekten. Er sollte sich überlegen, von welcher Klasse das Problem ist, das mit der zu entwickelnden IT gelöst werden soll. Ist es ereignisbasiert, regelbasiert, datenbasiert oder muss doch ein Programm im klassischen Sinne geschrieben werden. Diese initiale Erkenntnis legt schon ein Großteil der verwendeten Architektur oder sogar der verwendeten Produkte fest.

Kommunizierende Prozesse

- Event Systeme (impliziter Aufruf / expliziter Aufruf)

Zu dieser Klasse gehören viele Middleware Systeme oder Bussysteme, die alle Nachrichten als Event auffassen. Diese landen dann meistens in Queues und müssen weiter verarbeitet werden.

Datenorientierter Fluss

- Stapelverarbeitung (Batch Processing)
- Pipes & Filter

In der Stapelverarbeitung werden Kommandos oder Kommandogruppen abgearbeitet. Pipes und Filter sind ein wesentlicher Bestandteil in UNIX Architekturen. Hier werden Informationen kettenartig weitergereicht und jeder Filter trägt zur Verarbeitung der Information bei.

Call and Return

- Prozedurale Programmierung
- Objektorientiert
- Schichtenorientiert

Dies ist die Sichtweise, in der Entwickler normalerweise loslegen würden. In der gegebenen Programmiersprache mit den gegebenen Paradigmen Code zu erstellen und zu organisieren oder auch in Bibliotheken zu unterteilen. Dennoch muss dies nicht die ideale Vorgehensweise sein. Die anderen genannten Stile können viel besser geeignet sein.

Datenzentriert

- Repository
- Blackboard

In einem Repository werden in der Regel Objekte nach einem (oft auch hierarchischen) Schlüssel abgelegt. Anwendungen können diese nun finden und weiterverwenden. Die Objekte selbst müssen dabei nicht nur aus Daten bestehen, sondern können sogar Logik beinhalten (wie z. B. echte Java Objekte oder Stored Procedures). Kernelement sind jedoch die gespeicherten Daten.

Ein Blackboardsystem ist eher angebracht wenn es keine deterministischen Lösungsstrategien gibt. Beispiele dafür wäre ein Blackboardsystem zur Sprach- / Bilderkennung oder für die DNA-Sequenzanalyse. Hier kann es Agenten geben, die sich Teile aus dem Blackboard entnehmen oder anschauen und damit arbeiten oder lernen. Eine vollständige Analyse des Problems ist häufig nicht möglich, so dass der Lösungsraum zu groß ist. Einzelne Programme oder Agenten die das Blackboard benutzen sind darauf spezialisiert Teilprobleme zu lösen, indem sie sich der Wissensquelle Blackboard bedienen.

Virtuelle Maschine

- Interpreter
- Regelbasierte Systeme

Hierbei kann das Problem in einer Sprache - z. B. auch einer Domain Specific Language - oder in Form von Regeln vorliegen. Die virtuelle Maschine hat dabei Anfangszustände, bekommt Informationen (Input), arbeitet die Sprache oder Regeln ab und liefert ein Ergebnis. Das erinnert an die Automatentheorie. Die Methode der Abarbeitung eignet sich ideal für solch auftretende Probleme. So sind Expertensysteme oder auch Suchanfragen im Semantic Web derart gestaltet. Aber auch Datenbankabfragen in der DSL ähnlichen Sprache LINQ z. B. können initial interpretiert, dann transformiert und dann erst auf die Datenbank losgelassen werden. Auch viele Steuerungsprobleme in der Hardware sind für diese Klasse prädestiniert.

7 Technologiearchitekturen

Die vorgestellten Architekturstile sind generelle Kommunikationsarchitekturen, die an keine Technologien oder sonstige Schlagwörter gebunden sind.

Es werden nun einige dieser Stile in Ihrer konkreten technologischen Ausprägung vorgestellt. Dabei ist zu beachten, dass diese nicht disjunkt sind. Viele der unten genannten Technologien überschneiden sich. So werden beispielsweise Thin-Client Architekturen als Client-Server Architekturen realisiert und sind in der Regel auch noch N-Tier.

Die in diesem Kapitel aufgeführten Technologiearchitekturen sind keineswegs vollständig. Weitere bekannte sind:

- Content Management Systeme
- Suchmaschinen
- Web-Server (eben keine Application Server)
- Portal Frameworks
- Reporting- und Analyse-Tools
- Lasttest Frameworks
- Datenbanksysteme

Auch diese sollten in die initiale Architekturdiskussion mit eingebracht werden.

Als Antipattern für eine Architektur gilt ein **Monolith** also eigentlich keine Architektur, sondern einfach ein großer „Klumpen“. In der Regel sind solche Systeme nicht zu warten und auch nicht erweiterbar. Monolithische Strukturen sind für sehr kleine Systeme passend, an denen auch nur ein Entwickler arbeitet. Bei unseren Architekturdiskussionen gehen wir immer von größeren Architekturen aus.

7.1 Zentral-Dezentral / Client-Server

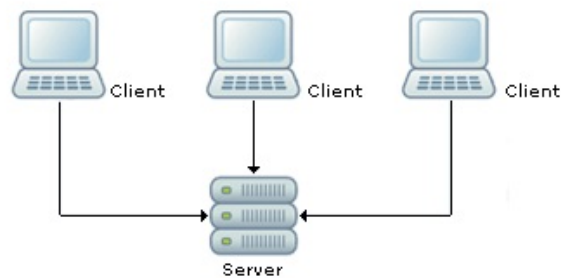
Zu den wichtigsten Anfangsüberlegungen gehört, ob das zu entwickelnde IT-System verteilt vorliegen wird oder nicht. Bei vielen großen Systemen ist dies selbstverständlich. So wird jede Anfrage an Google (je Land) auf einen anderen Server geroutet. Weiterhin gibt es weltweit viele Datenbestände, auf denen die Suchergebnisse gehostet werden.

Zu diesem Thema gehört auch das Thema der Replikation. Sollten die Datenbestände repliziert (also verteilt / kopiert) werden?

Die wichtigsten Aspekte die hier eine Rolle spielen sind natürlich die Zugriffszeit, die **Ausfallsicherheit** und die Aktualität der Datenbestände.

Die mit am meisten verbreitete Architektur im Internet, ist die Client-Server Architektur. Datenbestände liegen hierbei auf zentralen Webservern. Dabei werden zugleich Datenbestände (z. B. mit RAID) gesichert und der Wartungsaufwand reduziert. Das System ist jedoch nicht so ausfallsicher wie in einer dezentralen Architektur.

Zentral:



Dezentral:

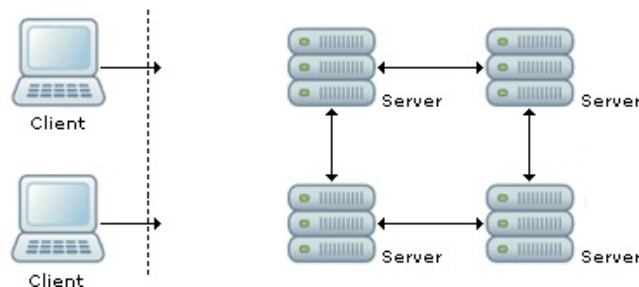


Abb.: Mehrschichtenarchitektur

Die Grafik bildet die Mehrschichtenarchitektur physisch ab. Ein Bild der logischen Architektur könnte hier ebenfalls die MVC-Elemente (Model View Controller) enthalten, die sich dann durchaus auch auf die physischen Elemente abbilden können lassen.

7.2 Rich- und Thin-Client

Bei der Unterscheidung zwischen Rich und Thin-Client, wird oberflächlich betrachtet, ob der Anwender einen Webbrowser zur Arbeit mit dem System benötigt oder nicht. Neuerdings gilt diese Möglichkeit der Unterscheidung als veraltet und nicht mehr gebräuchlich. Dadurch lassen sich mehrere Stufen erkennen, die sich beim Client eher nach dem Grad der Funktionalität ausrichten. Folglich kann somit bspw. beim User erkannt werden:

- statisches html
- dynamisches html mit `<? expr ?>` wie **JSF**, **JSP**, etc.
- noch reichere AJAX Funktionalität
- rich clients (auch im Browser) mit Silverlight, Flex / Air, Java FX
- „Echte“ rich clients wie z. B. Swing GUIs oder der Eclipse Rich-Client Platform

Letztere werden lokal am Rechner gestartet und nehmen dann mit einem Kommunikationsmechanismus (z. B. RMI), der u.U. ein zentraler Server sein kann, Kontakt auf.

Hierbei müssen initial Dinge wie Serverbelastung, Wartung oder Auslieferung geprüft und beachtet werden.

7.3 N - Tier

Unter N-Tier Architektur versteht man lediglich eine Mehrschichtenarchitektur. Nach der Lektüre der vorigen Einheiten ist es keine Überraschung mehr, das heute fast alle klassischen großen Systeme so gestaltet sind. Also gerade auch Anwendungen auf Application-Servern mit Rich / oder Thin-Architektur, an welchem ein ganzes Team von Entwicklern arbeitet.

Der Vorteil liegt hier nicht nur in der einfacheren Entwicklung durch die Trennung, sondern auch durch die Möglichkeit des einfacheren load balancing (Lastverteilung), der Ausfallsicherheit und generell einer besseren Performance. So kann auf fast allen Schichten verteilt werden. Ein http-Proxy (wieder ein Pattern) verteilt die Webanfragen auf einzelne Rechner, die Logik wird dann wiederum verteilt aufgerufen und die Daten liegen möglicherweise auf einem Datenbank-Cluster.

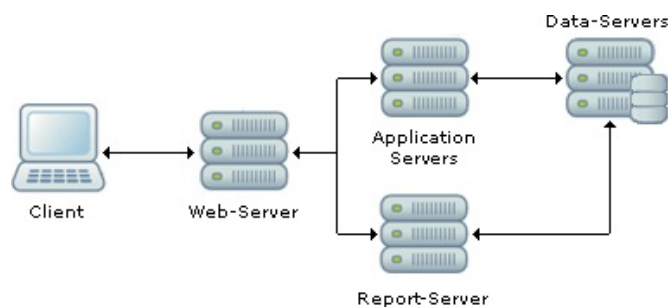


Abb: N-Tier

7.4 Container- / Komponenten-Architektur

Hierunter versteht man den Ansatz, dass ein Container-System vorliegt, welches die folgenden Eigenschaften hat:

- Es nimmt Komponenten auf (wie ein Behälter)
- Der Container übernimmt viele Querschnittsfunktionalitäten wie Security oder Lastverteilung
- Die Komponente wird vom Container in ihrem Lebenszyklus verwaltet (z. B. als Pool)

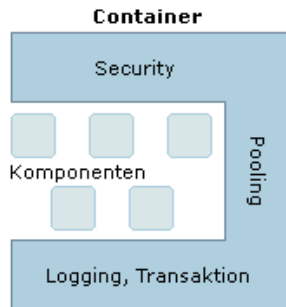


Abb: Container

Weiterhin sind Dinge wie Transaktionen, Verteilung, Persistenz, Nebenläufigkeit und Ressourcenmanagement standardisiert. Für diese Services gibt es APIs - wie die JTA (Java Transaction API) - die von den Entwicklern verwendet werden sollen.

Typische Architekturen sind die EJB Architektur oder die CCM/COM+ Architektur.

Unter einer Komponentenarchitektur versteht man lediglich die Strukturierung eines Systems in Komponenten. Hierbei ist üblich dass nicht nur der klare Servicegedanke unterstützt wird (siehe auch später bei OSGi), sondern gegebenenfalls auch auf einem System deployt wird, welches den Lebenszyklus der Komponenten überwacht.

Abschließend die Definition einer **Komponente**:



Definition

Komponente

Unter einer Komponente versteht man eine Einheit von Code, die eine definierte Schnittstelle (= Vertrag) besitzt. Sie hängt nur von sich selbst ab und kann daher beliebig mehrfach eingesetzt und erweitert werden.

7.5 Middleware Architekturen (MOM)

Unter Middleware versteht man Programme, die versuchen die Komponenten und die Kommunikationsarchitektur zu verbergen.

Middleware sorgt für die Übermittlung von komplexen / strukturierten Daten. Die wohl bekannteste Architektur in diesem Bereich war Corba. Dort wurden Objekte / Komponenten installiert, auf die dann verteilt zugegriffen werden konnte.

Middleware-Produkte gibt es von vielen Herstellern und auch im Open-Source-Bereich. Sehr bekannte Produkte sind ESB Produkte (z. B. von Apache), die später vorgestellt werden.

Middleware nimmt die Komponenten und stellt diese - oder entsprechende Daten - über das Netzwerk in irgendeiner Form (z. B. RPC) zur Verfügung. D. h. es wird meist auf TCP/IP oder http oder WebServices genutzt.

7.6 Serviceorientierte Architektur SOA

SOA ist einer der Hype-Begriffe zu Anfang des Jahrtausends. Er steht für **Serviceorientierte Architektur**, also für eine strikte Ausrichtung auf Dienste.

Die Grundidee ist, alles unter einer einheitlichen Dienstschnittstelle als Service zur Verfügung zu stellen. Man nehme z. B. eine Großbank mit Hunderten von IT-Systemen. Die Daten von all diesen Systemen zu integrieren und auszuwerten ist - wenn spät begonnen wurde - sicherlich eine harte, wenn nicht sogar unmögliche Arbeit. Mit Hilfe eines SOA Produktes könnte aber jedes System gekapselt werden. Da jedes System die Daten in einem anderen Format oder Kommunikationsmechanismus liefert, stellt das SOA Produkt meistens Adapter (wieder ein Pattern) zur Verfügung, die die Daten einheitlich anbieten können.

Mit Hilfe dieses Mechanismus kann dann auf solche „middleware“ SOA Produkte zugegriffen und die Daten einfach importiert werden. Eine Datenintegration ist so viel einfacher möglich. Besser wäre es eine solche Serviceorientierte Architektur von Anfang an durchzusetzen (wenn sie sinnvoll ist), damit keine parallelen Daten- und Systeminseln entstehen. Firmenfusionen sind beispielsweise eine ideale Voraussetzung, um auf einer SOA zu beginnen.

SOAs gehen deutlich über die reine CORBA oder RMI-Kommunikation hinaus, indem sie:

- Infrastruktur bieten
- Dienste koordinieren (z. B. zeitlich)
- Echte Enterprise Application Integration bieten
- Zentrale Anlaufstelle für middleware Produkte sind, die wiederum mit den Daten der Services arbeiten.

7.7 Enterprise Service Bus ESB




Der Enterprise Service Bus ist eine Unterkategorie der Serviceorientierten Architekturen SOA. Auch hierbei handelt es sich um Middleware.

Zentraler Punkt bei ESB ist die Vorstellung eines einheitlichen Leitungsmediums (BUS) auf dem Producer Nachrichten einstellen und Consumer diese abrufen können. Es handelt sich also im Wesentlichen um eine Messaging Infrastruktur.

Diese kann auf unterschiedlichen Kommunikationsmechanismen basieren wie z. B. :

- JMS (Java Messaging Service)
- Web-Services wie SOAP oder REST

Bekannte Vertreter findet man besonders im Open-Source-Bereich.

-  Mule
-  ServiceMix
-  Apache Active MQ

Auch hier gibt es wie bei der serviceorientierten Architektur Filter, Konverter und Adapter, um jedes nur erdenkliche System mit dem Bus zu verbinden.

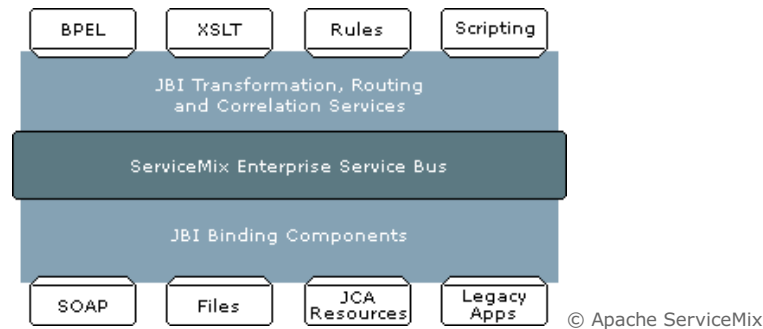


Abb.: Apache ServiceMix

7.8 Peer-To-Peer (P2P)

Als Peer-to-Peer Verbindungen bezeichnet man viele Rechnersysteme mit Komponenten, Programmen oder Agenten, die gleichwertig sind und miteinander kommunizieren. Eine Einheit kann sowohl Dienste anbieten und auch Dienste anderer nutzen. Es gibt keinen zentralen Server.

Zweifelhafte Berühmtheit bekommen Peer-to-Peer Netze durch **Computerviren**, die Rechner mit Bots infizieren und dann z. B. dazu missbrauchen Spam zu versenden. Weiterhin sind **Filesharing** Netzwerke als derartige Architekturen bekannt (Napster, Gnutella, Bearshare, BitTorrent, eDonkey oder Soulseek), deren Nutzung teilweise illegal ist.

Der Vorteil dieser P2P-Systeme besteht darin, dass kein zentraler Server vorhanden ist. Der Systemzustand ist das System selbst und daher viel schwerer zu durchschauen. Eben dies macht auch eine Strafverfolgung in diesen Filesharingsystemen viel schwerer. Für den Zugang zu diesen Systemen gibt es **Lookup**-Dienste, die den dynamischen Zustand der Entitäten (=Peers) widerspiegeln. Diese Lookup-Dienste können wiederum serverbasiert sein, so dass eine hybride Architektur vorliegt.

Peer-to-Peer Systeme haben meistens folgende Eigenschaften.

- Sie organisieren sich selbst
- Sie handeln mit ihren Ressourcen autonom
- Dienste und Ressourcen können „wandern“
- Peers können anbieten und nutzen
- Verhalten, Zustand oder Durchsatz kann nicht vorhergesagt werden und erinnert eher an einen Schwarm

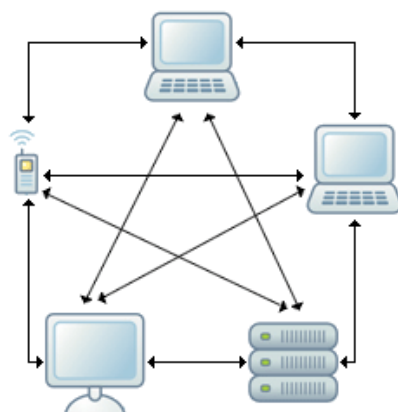


Abb.: P2P

7.9 Application Server

Application Server bündeln meist viele der bereits genannten Architekturen. Sie enthalten in der Regel einen Container, der die genannten Dienste einer Komponentenarchitektur anbietet. Weiterhin ist ein Web-Server integriert der z. B. CGI oder fastCGI anbietet.

Zusätzlich sind viele weitere standardisierte Services enthalten wie Messaging Dienste oder Komponentenbibliotheken.

Sinn dieser Applikationsserver ist:

- die Lieferung von besagten Containerdiensten (wie Transaktion, Security, etc.)
- das Angebot an weiteren Diensten (Directory Dienste, Persistenzdienste, Kommunikationsdienste, etc.)
- das eingebaute Skalieren ohne Änderung der Anwendung z. B. durch Clustering
- das Management des Lebenszyklus der Komponenten
- die Logging und Monitoring Funktionalität

Im Rahmen von Standards (wie JEE oder .NET) - auf denen Applikations-Server aufbauen - sind auch Template Sprachen verfügbar wie JSP, ASP, PHP, etc.



Beispiel

Bekannte Applikations-Server

- Apache Tomcat
- JBoss
- Glassfish
- WebLogic
- WebSphere

8 Frameworks

Jeder fortgeschrittene Architekt hat selbst Praxiserfahrung mit den bekanntesten Komponentenframeworks gemacht. Dazu gehören die in diesem Kapitel erläuterten Frameworks J2EE, .NET und auch - aus vergangenen Zeiten - Corba.

Die beiden letzten Frameworks dieses Kapitels beschäftigen sich eher mit Kommunikationsinfrastrukturen und sind daher häufig Basis für Komponentenframeworks.

▢ Java EE

▢ Net


▢ CCM Corba

▢ Web Services

▢ REST

Microservices

8.1 Java EE


Eine der bekanntesten Architekturen ist die  Java EE Architektur (vormals auch J2EE genannt). Die Java 2 Plattform, Enterprise Edition (J2EE) von SUN Microsystems umfasst eine Sammlung zusammengehöriger Spezifikationen und Programmierrichtlinien, die insgesamt die Entwicklung mehrschichtiger, verteilter Serveranwendungen erleichtern sollen. Ein Merkmal der J2EE liegt in ihrer besonders starken Ausrichtung auf das Internet:

“J2EE is a distributed, Web-centric, Java-enabled world” (SUN Microsystems)

Sie besteht aus mehreren Teilen:

J2EE-Spezifikation


Die J2EE-Spezifikation beschreibt einen einheitlichen Standard, repräsentiert durch eine Menge notwendiger APIs und Richtlinien, für Applikationsserver die durch einen Application-Server unterstützt werden müssen.

Diese Spezifikation soll es möglich machen, Anwendungskomponenten zu entwickeln, die auf jedem J2EE-kompatiblen System einzusetzen sind. Die Spezifikation steht unter  <http://java.sun.com/javaee/technologies/> zum Download bereit.

J2EE-Blueprints

Auf der Webseite von SUN ( java.sun.com/blueprints) ist die Architektur-Referenz als Online-Handbuch abrufbar - diese ist auch als Buch erhältlich. Zusätzlich werden dort Rezepte zum Entwickeln von JavaEE-Anwendungen, Design Patterns für JavaEE-Anwendungen, sowie eine Beispielanwendung, der Pet Store zur Verfügung gestellt. In der Beispielanwendung werden Änderungen der Spezifikation meist zeitnah berücksichtigt.

J2EE-Server

Die Referenzimplementierung eines Applikationsservers von SUN Microsystems für JavaEE dient zur Demonstration der Fähigkeiten der JavaEE-Plattform und kann für Testzwecke (nicht kommerziell) genutzt werden; sie ist im J2EE-SDK enthalten, das SDK kann unter  <http://java.sun.com/javaee/downloads> heruntergeladen werden.

Weiterhin gibt es noch eine JavaEE Testsuite, die zur Zertifizierung von JavaEE Produkten benötigt wird. Sie enthält zahlreiche Tests.

8.1.1 Java EE Basisdienste

Die Installation und der Betrieb komponentenbasierter, verteilter Anwendungen ist ohne die Bereitstellung bestimmter Basisdienste nicht denkbar. Dazu gehören Namensdienste zum ortstransparenten Auffinden von Objekten, Transaktionsdienste zur Abgrenzung von Anwendungsabläufen in unteilbare Arbeitsschritte, Sicherheitsmechanismen zur Authentisierung und Autorisierung, die Bereitstellung von Persistenzmechanismen zur Speicherung von Objekten, die Bereitstellung einer genormten Kommunikations-Infrastruktur sowie Dienste zur Installation, Konfiguration und Administration.

Alle JavaEE-konformen Produkte (Container, Applikationsserver) müssen solche Infrastrukturen bereitstellen.

Namensdienst

Namensdienst

Namensdienste erlauben es J2EE-Anwendungen, auf Ressourcen und Komponenten über logische Namen zuzugreifen, und tragen damit wesentlich zur Portabilität und Wartungsfreundlichkeit von Anwendungen bei. Die Standardschnittstelle für den Zugriff auf Namensdienste (und Verzeichnisdienste) in JavaEE ist das Java Naming and Directory Interface (JNDI).

Hersteller von Applikationsservern, die dem J2EE-Standard entsprechen sollen, müssen mindestens einen Namensdienst anbieten, der via JNDI angesprochen werden kann.

Transaktionen

Transaktionsdienst

Transaktionsdienste gewährleisten, dass einzelne Arbeitsschritte in Anwendungsabläufen konsistent und isoliert ablaufen können. J2EE sieht dafür sowohl ein Konzept deklarativer Transaktionsbegrenzung über Deskriptoren vor, als auch die Möglichkeit der programmatischen Transaktionssteuerung über eine Programmierschnittstelle, den Java Transaction Service (JTA).

Das JavaEE-SDK enthält den JTS als Standardimplementierung eines Transaktionsdienstes. JTS implementiert dabei auch das Java Mapping der OMG Object Transaction Service (OTS) 1.1 Spezifikation unterhalb der JTA-API und implementiert Transaktionen, die auf dem Internet Inter-ORB Protocol (IIOP) basieren.

Viele J2EE-Produkte bieten auch Möglichkeiten an, traditionelle Transaktionsmonitore wie CICS, Tuxedo oder UTM einzusetzen, die das Leistungsvermögen einfacher Transaktionsdienste weit übertreffen.

Sicherheit

Sicherheitsdienst

Applikationsserver stellen für ihre Clients eine große Anzahl schützenswerter Ressourcen zur Verfügung. Deshalb ist die Umsetzung und Einhaltung anwendungsspezifischer Sicherheitsrichtlinien von herausragender Bedeutung.

Die Zugangskontrolle bei JavaEE-Systemen erfolgt grundsätzlich in zwei Schritten: Authentisierung und Autorisierung. Vergleichbar dem Transaktionskonzept kann auch hier die Umsetzung von Sicherheitsrichtlinien, der Security Policy, deklarativ oder programmatisch erfolgen. Zur Integration bestehender oder neu einzurichtender Benutzerverwaltungen in das J2EE-Sicherheitskonzept stellt J2EE mit JAAS (Java Authentication & Authorization Service) einen normierten Standard bereit.

Persistenz

Persistenz

Zu den wichtigsten Aufgaben von Applikationsservern zählt das Speichern von Anbindung der Datenschicht, die von vielfältigen Datenbanksystemen oder Objektzuständen auch von betrieblicher Standardsoftware und hostbasierten Systemen repräsentiert wird.

Operationen zur Gewinnung und zur Manipulation von Daten aus diesen Systemen können durch EJB-Objekte abgebildet werden und müssen oft dauerhaft verfügbar sein. Die persistente Speicherung von Objekten bzw. Objektzuständen erfolgt überwiegend in relationalen

Datenbanken.

JavaEE stellt mit JDBC einen Standard zum Zugriff auf relationale Datenbanken bereit. Die Abbildung von Objekten auf eine flache relationale Struktur kann allerdings zu einer schwierigen Aufgabe werden. An dieser Stelle sei auf weitere Technologien zum Thema JPA, EJB, Hibernate, JDO verwiesen.

Kommunikation

Zur Kommunikation zwischen den Komponenten einer verteilten Anwendung sieht J2EE verschiedene Kommunikationstechniken vor. Die genaue Umsetzung der Vorgaben obliegt den Herstellern von Applikationsservern bzw. Containern. Ein J2EE-Anwendungsentwickler muss sich daher normalerweise nur wenig um Details tieferliegender Schichten kümmern. Dabei spielen Dienste wie TCP/IP, SSL, RMI und Corba/RMI-IIOP eine bedeutende Rolle.

Konfigurations- und Administrationsdienste

Sie unterstützen die flexible Anwendungen von Verpackung, Installation und Konfiguration von Komponenten sowie die Administration von Anwendungen.

Eigenschaften von Servern, Containern, Anwendungen, Komponenten und Diensten werden in XML-Dateien auf der Grundlage festgelegter XML-Schemata beschrieben (Deployment-Deskriptoren). Die Konfiguration der Persistenz- und Transaktionsanbindung und Umsetzung von Sicherheitsrichtlinien in der gegebenen Serverumgebung sind damit verbunden. JavaEE-Container müssen diese Einstellungen interpretieren, prüfen und anwenden können.

Ebenso wichtig sind einfach anzuwendende Administrations- und Kontrollmöglichkeiten (Audit) beim Betrieb von Anwendungen. Applikationsserver stellen für all diese Tätigkeiten in der Regel grafische Benutzeroberflächen zur Verfügung.

8.1.2 Java EE Anwendungsarchitektur

Beim Entwurf und der Entwicklung von Web-Anwendungen und unternehmensweiten Anwendungen haben sich in den letzten Jahren Modelle durchgesetzt, die auf mehrschichtigen Architekturen beruhen, die auch als n-tier-Architekturen bezeichnet werden. Das Konzept der Architektur von Anwendungen, wie es in der Spezifikation und auch in den J2EE-Blueprints beschrieben ist, macht sich ein solches mehrschichtiges (multi-tier) Architekturmodell zu eigen. Die Abbildung zeigt die, für die J2EE-Anwendungsarchitektur typische, Aufteilung in mehrere Schichten.

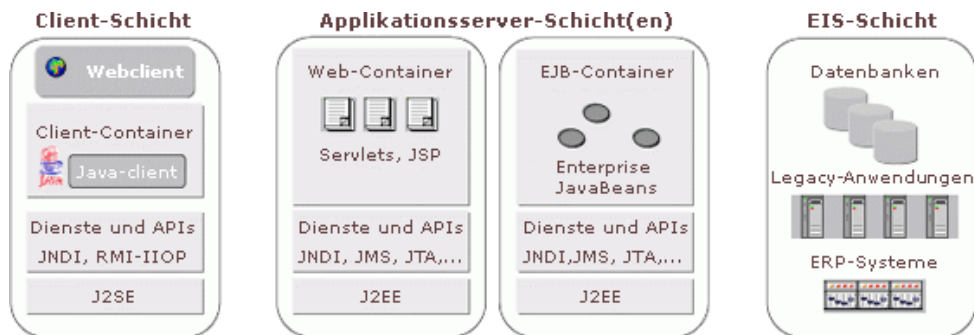


Abb.: J2EE-Anwendungsarchitektur

So kann eine Web-Anwendung in einer mehrschichtigen Anordnung aufgeteilt sein. Web-Container und EJB-Container könnten dabei auch auf verschiedenen Rechnern betrieben werden, z. B. um im Unternehmen geltende Sicherheitsrichtlinien einzuhalten. Anwendungsdesign nach mehrschichtigen Architekturmodellen kann beim Betrieb von Unternehmensanwendungen eine Reihe von Vorteilen mit sich bringen:

1. Steigerung der Skalierbarkeit
2. Lastverteilung auf mehrere Rechner
3. Erleichterte Umsetzung von Sicherheitsstrategien
4. Höhere Modularität
5. Einfachere Wartbarkeit einzelner Anwendungsteile

Desweiteren gibt es eine Vielzahl von Programmier-APIs und Bibliotheken:

- Java Naming and Directory Interface (JNDI)
- Java Message Service (JMS)
- Java Database Connectivity (JDBC)
- Java Transaction API (JTA)
- Java Connector Architecture (JCA)
- Java API for XML Processing (JAXP)
- JavaMail
- JavaBeans Activation Framework (JAF)
- Java Authentication & Authorization Service (JAAS)

Entscheidend für die JavaEE Architektur ist auch das EJB Modell (Enterprise Java Beans, siehe dazu auch nächste Grafik). Dazu gibt es:

- **EJB-Container:** Dieser bietet den installierten Komponenten eine Laufzeitumgebung. Dazu verwaltet er insbesondere die Bean-Instanzen, steuert ihren Lebenszyklus und stellt ihnen einen standardisierten Zugang zu den Enterprise Services zur Verfügung, u. a. JDBC für den Zugriff auf Datenbanken, JTA für den Zugang zum Transaktionsdienst und JMS für den Messaging-Service.

- **EJBs:** Dies sind die Server-Komponenten, in denen die Anwendungslogik gekapselt ist. Es gibt drei Ausprägungen: Session Beans bilden Geschäftsprozesse ab, Entity Beans repräsentieren Geschäftsdaten und die Message-Driven Beans verarbeiten asynchron empfangene Nachrichten.
- **Clients:** Auf EJBs können browserbasierte Clients (thin clients), Java-Clients (fat clients), Servlets, CORBA-Clients oder auch andere EJBs zugreifen. Dies geschieht durch Remote- oder lokale Methodenaufrufe bzw. – bei Message-Driven Beans – durch einen Messaging-Dienst (asynchrone, entkoppelte Kommunikation).

Abschließend sei noch darauf hingewiesen, dass die JavaEE Architektur damit auch **Rollen** in Ihrer Architektur vorsieht. In der klassischen Software-Entwicklung wird meist vorausgesetzt, dass Entwickler in unterschiedlichen Bereichen, die beispielsweise Datenbankankbindung, Transaktionsverwaltung oder Installation von Anwendungen betreffen, über gute Kenntnisse verfügen.

Die EJB-Spezifikation teilt die Entwicklungsverantwortlichkeiten auf verschiedene logische Rollen auf. Mit der Zuteilung konkreter Aufgaben wird beabsichtigt, die Entwicklung von Anwendungen für die Beteiligten zu vereinfachen, da sie sich auf ihre Kernkompetenz konzentrieren können.

Alle Tätigkeiten, die bei der Entwicklung und dem Einsatz von EJBs anfallen, werden von der EJB-Spezifikation weiterhin in verschiedene Szenarien unterteilt, die gemäß der Rollenverteilung von den daran beteiligten Parteien ausgeführt werden sollen. Die Spezifikation bezieht sich auf drei Szenarien, die in der Spanne von der Entwicklung einer EJB-Komponente bis hin zur Installation als Bestandteil einer EJB-Anwendung angesiedelt sind.

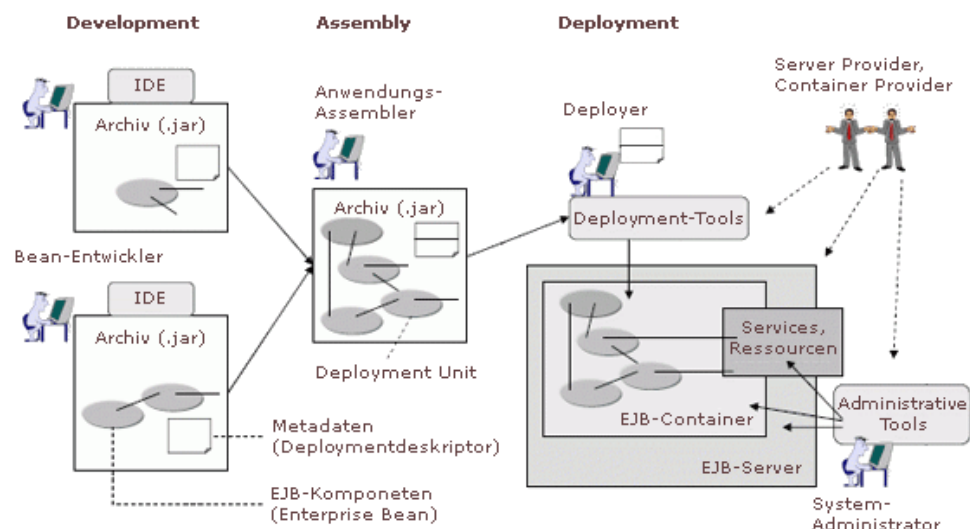


Abb.: Rollen in der JavaEE Architektur

Die in der Abbildung dargestellten Rollen sind:

- Bean-Entwickler
- Assembly-Developer
- Deployment-Manager
- Systemadministrator
- Sonstige Provider

8.2 .NET

In der Industrie nicht minder wichtig als Java EE ist die .NET Architektur von Microsoft. Sie ist in der Regel auf das Windows Betriebssystem beschränkt. Jedoch laufen dank Mono auch einige Dienste unter UNIX. Mit .NET sollte auch das COM Komponentenkonzept abgelöst werden.

Sie besteht aus den Komponenten:

1. Laufzeitumgebung
2. Klassenbibliotheken
3. Dienstprogramme

Am interessantesten ist sicherlich die **CLI** (Common Language Infrastruktur) von .NET. Hier gibt es ebenfalls eine virtual Machine wie bei Java und zusätzlich eine Universalsprache, in der alle Compiler übersetzen können.

Diese frühe Ausrichtung hat zur Folge, dass es eine Vielzahl von Sprachen für die .NET Plattform gibt. Neben den Kernsprachen VB (Visual Basic) und C# gibt es daher auch C++, J# und viele mehr. Aber auch dynamische Sprachen wie IronPython oder IronRuby sind auf der .NET Plattform verfügbar. Für das Zusammenspiel dieser Sprachen sorgt auch das **Common Type System**, welches ein vereinheitlichtes Typsystem bereitstellt.

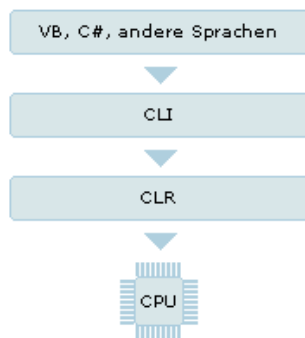


Abb: .NET

Die **CLR** ist dabei die Common Language Runtime und stellt die Laufzeitumgebung dar. Dabei sind alle Programme, die innerhalb der CLR laufen, managed. Alle anderen sind unmanaged.

Weitere wichtige Eigenschaften sind:

- **Performance:** Ziel der .NET Sprachumgebung war es auch, performanten Code bereitstellen zu können. Dafür gibt es zusätzliche Konzepte wie Just-In-Time ähnliche Compiler und die Möglichkeit native images zu erstellen. Dennoch wurde versucht auch mit Garbage Collection einen möglichst performanten Zwischencode zu erstellen.
- **Klassenbibliotheken:** Mit geliefert werden eine Vielzahl von Bibliotheken wie ADO.NET, ASP.NET, LINQ, WFORMS und viele weitere Basisklassenbibliotheken.
- **Assemblies:** Die unter .NET entwickelte Software wird in Assemblies zusammengefasst. Diese ähneln den .jar-Dateien in java.

Auffällig ist hier allerdings eine umfangreichere **manifest** Datei, die die Registrierung der **.exe** oder **.dll** Dateien sicherstellt.

8.3 CCM Corba

CORBA steht für die **Common Object Request Broker Architecture**. Corba wurde in den 90er Jahren entwickelt und ist eine relativ abstrakte Spezifikation für objektorientierte Middleware. Ziel von Corba war es eine abstrakte Schnittstelle zu schaffen, mit der Objekte aus beliebigen Sprachen in dem **Object Request Broker** (ORB) deployt und damit in heterogenen Umgebungen genutzt werden können.

Mit der Definition der **Interface Definition Language** (IDL) kann man Objekte dann mit ihren Feldern und Methodensignaturen abstrakt beschreiben und dem ORB bekannt machen. Dann muss mit dem IDL Compiler (ähnlich wie bei Java-RMI) ein Stub und ein Skeleton erstellt werden. Diese wirken wie ein Broker und erlauben es, entfernte Aufrufe wie lokale Aufrufe erscheinen zu lassen.

In der Regel gibt es Implementierungen für Java und C++. Jedoch sind auch Anbieter mit anderen Sprachinterfaces verfügbar. Kommerzielle Anbieter bieten meistens noch mehr Services wie:

- Lookup Services
- Management des Lebenszyklus
- Bus- oder Event-Driven Erweiterungen
- Relationship Services zwischen Objekten

und natürlich viel mehr Erweiterungen wie Persistenzdienste, Synchronisation, Transaktionsmonitore, etc.

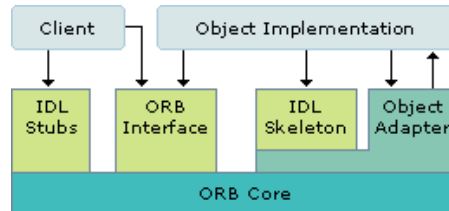


Abb.: Corba Architektur

8.4 Web Services

Unter einem Web-Service versteht man einen Dienst, mit dem Daten über eine Adresse ausgetauscht werden können. Die Adresse ist dabei eine sogenannte URI (Uniform Resource Identifier).

Dieser Dienst ist kein Komponentenframework, kann jedoch eine Basis für ein Komponentenframework sein. Meistens sind Web-Services auch Basis für SOA Architekturen / Frameworks.

In der Regel meint man mit Web Service drei Dienste, die auch standardisiert wurden:

1. Eine **Registry**: Dies ist oft UDDI.
2. Eine Sprache für **Netzwerkdienste**: Z. B. WSDL. Hier werden Datentypen, Nachrichten, Ports, Bindings und Services definiert.
3. Eine **Kommunikationsprotokoll**: Hier ist meistens SOAP oder XML-RPC gemeint. Hiermit werden die Daten selbst ausgetauscht und die entsprechenden RPCs (Remote Procedure Calls) ausgeführt.

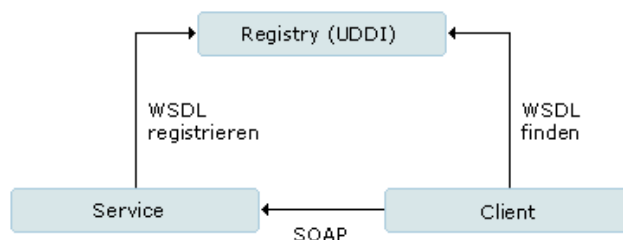


Abb: Web-Service

Da die Kommunikation textuell über den http-Port erfolgt, werden Web-Services mit WSDL und SOAP oftmals eine hohe Komplexität und eine schlechte Performance nachgesagt.

8.5 REST

REST steht für **Representational State Transfer** wird auch als Web Service bezeichnet. Auch dies ist kein Framework für Komponenten, jedoch ein Softwarearchitekturstil, der wiederum Voraussetzung für viele Softwarearchitekturen ist. Großer Vorteil gegenüber SOAP ist, dass jede Ressource (auch Dokumente, also jede Daten) ganz einfach über http mit einer URI adressiert werden können. Der Autor der Idee ist ROY FIELDING.

Über das http Protokoll kann damit jeder Anwender oder jede Software eine URI (in diesem Fall eine besondere URL) eingeben und sowohl Daten erhalten, als auch Operationen ausführen. Der ganze Dienst selbst ist zustandslos ausgelegt und wird in der Regel auch so verwendet. Zustände für Workflows lassen sich jedoch über IDs oder Passwort Hashes vom Client einführen.

Entscheidend sind die ersten vier http-Operationen auf beliebigen Daten auf dem Server:

1. **GET**: Sendet eine Datenanforderung an den Server. Entspricht dem Read aus CRUD.
2. **POST**: Sendet eine Datenänderung an den Server. Entspricht dem Update (Modify) aus CRUD.
3. **PUT**: Sendet neue Daten an den Server. Entspricht dem Create aus CRUD.
4. **DELETE**: Sendet eine Löschanforderung an den Server. Entspricht dem Delete aus CRUD.
5. **HEAD**: Fordert Metadaten vom Server zu der Ressource an.
6. **OPTIONS**: Sendet eine Anfrage, welche Methoden auf der Ressource verwendet werden kann.

Auch REST wird mittlerweile von allen großen Frameworks wie J2EE, .NET oder Ruby on Rails teils nativ oder auch mit Erweiterungsbibliotheken unterstützt.

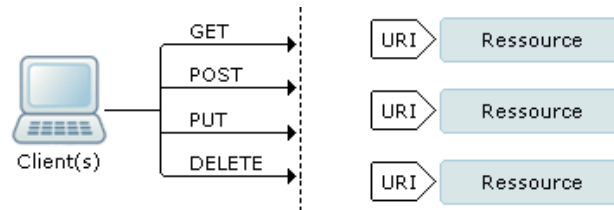


Abb: REST

Die zentrale Idee ist also, dass alles eine Ressource ist. Objekte sind mit Ihren Feldern Ressourcen oder XML Dateien oder Word Dokumente oder Bilder. Alles kann mit REST verwaltet und auch bearbeitet werden.

Vorteile

Die Vorteile sind:

- Der Aufwand auf dem Client ist minimal.
- Kann leicht skaliert werden, da nur Serverressourcen geclustert werden müssen.
- REST Daten auf einem Webserver sind komplett herstellerunabhängig.
- Benötigt keine große Registry. Eine initiale Adresse und dann die Adress-Strategie für Ressourcen müssen bekannt sein.
- Die Daten können immer gut gecached werden.

Man spricht in einer konkreten Realisierung dann von einer RESTful Anwendung.

Eine typische REST Adresse sieht dann z. B. so aus.

<http://myshop.com/resources/camcorders/sonyhdr-tg3e>

Man kann sich jetzt leicht vorstellen, wie man auf dieser Basis einen kompletten webbasierten CRUD Service aufbauen kann.

Bekannte Implementierungen sind Blogs, die mit RSS oder ATOM Daten zur Verfügung stellen. Aber auch Flickr oder Amazon S3 stellt auf diese Weise interessante Services für Grafik oder Speicherplatz zur Verfügung.

9 Referenzarchitekturen

In der Praxis sind viele unterschiedliche Referenzarchitekturen bekannt. Manche haben Standardcharakter, manche nur Empfehlungscharakter. Bei vielen gibt es Vorgaben - wie beispielsweise UML-Diagramme - aber bei einigen auch konkrete Implementierungsanweisungen und Komponentenbibliotheken.

Eines der bekanntesten Referenzmodelle ist das von SUN in den [JEE Blueprints](#).

Für viele Industriesparten haben sich auch unterschiedliche Referenzarchitekturen entwickelt. So z. B. im Bankgewerbe, der Telekommunikationsindustrie oder bei der Geräteprogrammierung (Embedded Devices).

9.1 NGOSS

NGOSS ist eine Referenzarchitektur für die Telekommunikationsindustrie und steht für „Next Generation Operation Support Systems Initiative“. Im Wesentlichen geht es dabei, neben der Schichtenarchitektur, um eine Werkzeugsammlung die wohldokumentiert ist und dadurch Kosten spart. Im Mai 2010 wurde NGOSS in Framework umbenannt.

<https://www.tmforum.org/tm-forum-framework/>

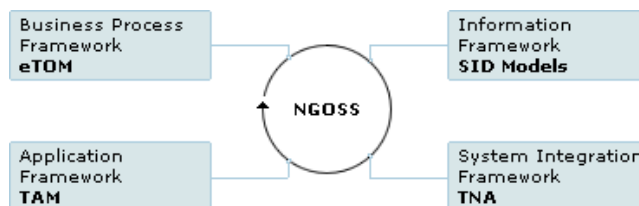


Abb: Wichtige Elemente des NGOSS

Es geht dabei um die folgenden Komponenten:

- Automatisierung von Geschäftsprozessen mit **eTOM** (enhanced Telecom Operations Map)
- Ein gemeinsames Informations- und Datenmodell **SID** (Shared Information & Data Model)
- Standardisiertes Einbinden von Systemen mit neutralen Contract Interfaces **TNA** (Technology Neutral Architecture)
- Das Anwendungsframework selbst TAM (Telecom Application Map)
- Einheitliche Teststrategien



Abb.: Hierarchie der NGOSS Teilbereiche

9.2 OSS/J

OSS/J ist eine Referenzarchitektur auf Java Basis, die von dem Java Community Process (JCP) vorgeschlagen und bearbeitet wird.

 OSS/J

 SBM-OSSJ

Es beinhaltet:

- Architektur Patterns
- OSSJ/APIs, d. h. Programmierschnittstellen
- Beispiele und Beispielcode
- Modellierungstools und Bibliotheken
- Aktivierungsdienste (z. B. beim Kauf einer SIM-Karte)
- Client / Server Adapter, um sich mit den Systemen zu verbinden
- OSS/J Erweiterungen, um verschiedene Anwendungsbereiche oder Industrieanforderungen abzudecken
- Test und Qualitätssicherungsdienste
- Abrechnungsdienste
- Trouble Ticketing als Teil einer API, um standardisiert Fehler zu verwalten
- Inventory Dienste: welche Komponenten und Systeme gibt es überhaupt?

Unter Java gibt es daher den Ansatz Java EE selbst als OSS/J Vorschlag zu etablieren, da dort bereits viele der benötigten Komponenten enthalten sind. Es ist dann nur noch aufzuzeigen, wie Java EE an den NGOSS Standard angepasst werden muss.

9.3 OSGI

OSGi ist eine „Open Services Gateway Initiative“ die versucht Komponentendienste dynamisch und hardwareunabhängig bereitzustellen. Diese Dienste sind dabei als sogenannte Bundles bekannt und können mit einer Registry gefunden werden. Auf den ersten Blick wirkt dies wie ein neuer Versuch CORBA zu reaktivieren. Besonderer Unterschied dabei ist, dass eine Java VM vorausgesetzt wird (auch wenn eine JNI Schnittstelle und C/C++ Anbindungen möglich sind).

Bekannte (Open Source) Implementierungen sind:

- **Equinox** - enthalten in der entsprechenden Eclipse Distribution
- **Felix** - von Apache
- **Knopflerfish** - von Gamespace

Die Alliance selbst wird von Großunternehmen wie Oracle, IBM, SAP, Siemens, NEC, Deutsche Telekom, Motorola oder NOKIA unterstützt und hatte ursprünglich auch einen großen Fokus im embedded Bereich, d. h. auch in der Vernetzung von Geräten. Etwas später wurde OSGi im Java Community Process (JSR 232, 246 und 248/9) übernommen. Die Bedeutung von OSGi nimmt stetig zu.

OSGi adressiert also zwei entscheidende Punkte: Modularisierung und Dynamik. In der Regel ist Modularisierung eben nicht nur mit Komponenten oder Bibliotheken zu erreichen. OSGi fokussiert daher besonders auf das Laufzeitverhalten der Komponenten. Also wie werden Komponenten:

- installiert
- aktualisiert und
- entfernt?

Komponenten können also dynamisch (zur Laufzeit!) Dienste bereitstellen, indem diese registriert, von anderen gefunden und genutzt werden. Dies fördert eine lose Kopplung zwischen den Komponenten.

OSGi sieht also schon im Kern eine Trennung von API und Realisierung der Komponente vor. Java Interfaces tun dies nicht zwingend, da die z. B. einfach nicht verwendet werden können. Es ergibt sich dadurch von vornherein eine hohe Wiederverwendbarkeit und eine sehr gute Testbarkeit.

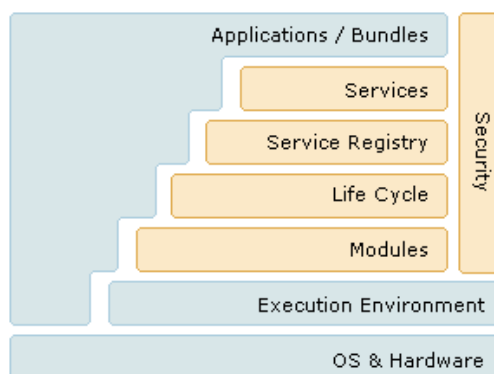


Abb.: Architektur OSGi

Wir sehen, dass Services über Bundles zur Verfügung gestellt werden. Dies sind in der Regel Java Jars mit ordentlicher Manifest Datei, in der die Schnittstelle sauber deklariert wird.

Die Registry ist insofern interessant, als dass die Dienste mit dem jeweiligen Interface als Schlüssel angemeldet werden können. Und die bekannte Java Security Architektur - mit ihren Signaturen und Policies - steht dabei voll zur Verfügung.

Danach ist dann das LifeCycle Management wichtig, in dem Services eingerichtet werden können. Aber auch eine direkte Steuerung der Services mit `start()` oder `stop()` ist möglich. Nachdem also ein einfacher Dienst mit seinem Interface registriert worden ist, können ein oder mehrere Provider als Bundle realisiert und registriert werden. Abschließend kann ein Consumer als Bundle diesen Dienst nutzen.



Beispiel

Bundles `hallo.provider` und `hallo.consumer`

Das Beispiel besteht aus zwei Bundles. Das Bundle `hallo.provider` beinhaltet die Programmlogik und bietet dafür einen OSGi-Service an. Das Bundle `hallo.consumer` benutzt es.

Um das Beispiel auszuprobieren empfiehlt sich Eclipse, welches in der Standardversion das Plug-in Development enthält. Damit steht ein guter Editor und Wizards zur Verfügung, außerdem braucht man keine Jar-Dateien zu erstellen.

Bundle Provider

Dieses Bundle beinhaltet die abrufbare Programmlogik, welche über einen OSGi-Service angeboten wird. Im Wesentlichen ist diese gekapselt über eine Schnittstelle - die Methode `sagHallo` - aufrufbar. Zusätzlich wird das Package der Schnittstelle exportiert.

Das Interface

```
package hallo.provider; // extern sichtbar, weil Export in Manifest
public interface IHalloProvider {
    String sagHallo();
}
```

Die Implementierung

```
package hallo.provider.internal; // intern, wird nicht exportiert
public class HalloProvider implements IHalloProvider {
    public String sagHallo() {
        return "Hallo Softwaretechniker!";
    }
}
```

Registrieren des Dienstes im Activator

```
package hallo.provider.internal; // intern, wird nicht exportiert
public class Activator implements BundleActivator {
    public void start(final BundleContext context) {
        context.registerService(
            IHalloProvider.class.getName(),
            new HalloProvider(), null); // Service wird unter dem
                                     // NamensString IHalloProvider
                                     // registriert.
    }
    public void stop() {}
}
```

Metadaten für OSGi: `MANIFEST.MF` Eclipse PDE-Perspective bietet hier einen komfortablen Editor an. Man muss im Reiter **Runtime** nur das Package `hallo.provider` exportieren.

Bundle Consumer

Der **Consumer** braucht das Bundle **Provider** und holt sich dessen Service, der **IHalloProvider** heißt. Bei diesem rufen wir die Methode ab. Der **Consumer** braucht einen Import vom exportierten Package des Providers, um die Abhängigkeiten aufzulösen. Mehr als den **Activator** braucht es nicht.

Start-Methode des Activators im Consumer

```
public void start(final BundleContext context) {
    ServiceTracker halloProviderTracker = new ServiceTracker(context,
                                                            IHalloProvider.class.getName(),
                                                            new ServiceTrackerCustomizer() {
        public Object addingService(final ServiceReference reference) {
            final IHalloProvider halloProvider = (IHalloProvider) context
                .getService(reference);

            System.out.println(halloProvider.sagHallo());
            return halloProvider;
        }
    });

    public void modifiedService(final ServiceReference reference,
                               final Object service) {
        // Nothing to be done!
    }

    public void removedService(final ServiceReference reference,
                               final Object service) {
        // Nothing to be done!
    }
});
halloProviderTracker.open();
}
```

Metadaten für OSGi: MANIFEST.MF



Im PDE-Editor im Reiter **Dependencies** **hallo.provider** importieren. Zusätzlich braucht es den Import von **org.osgi.util.tracker**, dieses wird einem automatisch bei den „Problems“ als Lösung angeboten.

Start des Service

Unter **Run / Run Configurations** wird eine Launch-Config für OSGi Framework angelegt. Im Reiter **Bundles** behält man seine beiden Bundles drin und nimmt ansonsten nur die Required Bundles hinzu, nicht alle. Nach dem Start sollte der String in der Eclipse Console erscheinen.

10 Enterprise Design Patterns

Das Fundament eines guten Softwarearchitekten ist die Kenntnis von Enterprise Design Patterns. Im Folgenden sollen die Kategorien dieser Patterns besprochen und einige daraus beispielhaft genannt werden.

Viele davon sind in der angegebenen Literatur (**BUSCHMANN**) und in den Werken von  Sun oder  **MARTIN FOWLER** zu finden. Diese Enterprise Patterns werden in Softwaretechnik Advanced erläutert. Aber auch im Internet findet man leicht mehr Information zu jedem der Patterns.

Strukturpatterns

Um die Menge von Komponenten zu strukturieren gibt es Patterns, von denen einige auch bereits genannt wurden:

- N-Schichtenmodell
- MVC (Model-View-Controller)
- Pipes und Filter
- Blackboard

Messaging-Patterns

Für die Verteilung von Informationen zwischen den Komponenten gibt es bekannte Muster wie:

- Publish-Subscriber
- Broker
- Message Router
- Server / Client Handler
- Requestor
- Invoker

Aufteilung von Komponenten / Interfaces

Interfaces spielen eine zentrale Rolle bei der Architektur von Softwaresystemen mit statischer Typisierung. Die Frage ist hierbei, wie die Komponenten untereinander zugreifen und voneinander Kenntnis haben.

- Interfaces
- Pluggable Selector
- Proxy
- Facade

Aufteilung der Komponenten selbst

Im vorigen Katalog ist vorwiegend die Anordnung und der Zugriff der Menge an Komponenten adressiert. In diesem Fall geht es um die Komponenten selbst, die oft auch noch einmal sinnvoll unterteilt werden können:

- Composite
- Whole-Part
- Master-Slave

Kontrollfluss

Wer steuert den Kontrollfluss zwischen den Komponenten? Auch hier gibt es Muster, die Komponenten mit einer derartigen Zuständigkeit beschreiben und den Kommunikationskontext / -abfolge festlegen oder empfehlen:

- Front-Controller
- Page-Controller
- Application-Controller
- Command Processor
- Template View
- Transform View
- Firewall Proxy
- Authorization

Nebenläufigkeit und Synchronisation

In verteilten Systemen entstehen Probleme durch nebenläufigen Zugriff auf Daten und Komponenten. Die nachfolgenden Patterns sorgen dafür, dass der Zugriff geordnet erfolgt und keine Fehler entstehen.

- Monitor Object
- Learer Follower
- Active Object
- Thread-Safe Interface
- Double Checked Locking / Two Face Commit
- Strategized / Scoped Locking

Interaktion, Adaption und Erweiterung

Zu dieser Klasse gehören viele der bereits erwähnten klassischen GoF Patterns:

- Observer
- Mediator
- Command
- Memento
- Data Transfer Object
- Bridge
- Adapter
- Interpreter
- Interceptor
- Visitor
- Decorator
- Template Method
- Strategy

Ressourcenmanagement

Eine eigene Klasse an Enterprise Patterns bildet die Verwaltung von Ressourcen. Dies können klassische programmiersprachliche Objekte sein aber auch Daten auf der Platte oder z. B. Datenbankverbindungen. Also jede Art von Ressource.

- Container
- Object Manager
- Lookup
- Task Coordinator
- Resource Pool
- Resource Cache
- Lazy / Eager / Partial Acquisition
- Garbage Collection
- Abstract / Method Factory
- Builder

Datenbankzugriff

- Überhaupt eine (abstrakte) **Datenbankzugriffsschicht** einzuführen ist ein EE-Pattern
- **Data Mapper**: Wird ein Datenbank-Mapper verwendet? Beispiel sind Hibernate, Castor, iBatis, OpenJPA, etc.
- **Row / Table Data Mapper**: Um ein Objekt auf die Datenbank zu mappen gibt es verschiedene Möglichkeiten und Tools
- **Active Record**: Ausgehend von einer Datenbanktabelle wird ein Objekt erzeugt, welches die entsprechenden Daten enthält und zusätzlich gleich alle CRUD (**C**reate, **R**ead, **U**ppdate, **D**eleate) Operationen auf der Tabelle in der Datenbank anlegt.

11 Verwandte Themen

In diesem Abschnitt werden noch einige Bereiche kurz angesprochen, die das Thema Architektur nur indirekt tangieren.

Es sind meistens Technologien oder Vorgehensweisen, die nicht initial in der Architekturdiskussion zur Sprache kommen, sondern meist im Verlauf der Architekturanalyse eine nicht unbedeutende Rolle spielen können.

Ein guter Architekt sollte daher auch zu diesen Themen Erfahrungen gesammelt haben.

11.1 MDA

Modellgetriebene Architektur nennt man MDA (Model Driven Architecture) oder auch MDSD (Model Driven Software Development). Die Idee ist, Teile der Anwendung abstrakter zu definieren und dann generieren zu lassen. Dies hat wesentliche Vorteile:

1. Das Generieren von Software ist deutlich **günstiger** als das komplette Ausprogrammieren.
2. Die abstrakte Definition der Software ist zunächst **technologieunabhängig**. Generierter Code kann später plattformabhängig generiert werden und so Portierungsaufwand gespart werden.
3. Der Code ist **weniger fehleranfällig**.
4. Es kann **schnell** generiert werden. Dagegen ist eine Eigenentwicklung langsam.

Weiterhin können abstraktere Dinge oft auch einfacher gehandhabt oder verstanden werden. Ausgangspunkt ist oft - aber nicht zwingend - UML bzw. das Austauschformat XML.

Wichtig sind Modelle und deren Tools auf zwei Ebenen:

- Platform Independent Model **PIM**: plattformunabhängiges Modell
- Platform Specific Model **PSM**: plattformabhängiges Modell für die Architektur

Dies bedeutet, dass zunächst eine plattformunabhängige Darstellung erstellt und danach auf die Zielplattform (J2EE, .NET, etc.) generiert wird.



Hinweis

Sie können sich hierzu noch einmal den Teil MDA in der Lerneinheit UML - Unified Modeling Language anschauen. Empfehlenswert ist das Buch:

ROLAND PETRASCH, OLIVER MEIMBERG (2006):

Model-Driven Architecture: Eine praxisorientierte Einführung in die MDA.

Dpunkt Verlag, ISBN-13: 978-3898643436

11.2 Aspektorientierte Programmierung

Die Aspektorientierte Programmierung (AOP) bezeichnet ein Programmierparadigma. Die grundlegende Idee ist, sogenannte Querschnittsaspekte getrennt zu entwickeln und erst später und transparent in die Anwendung zu integrieren.

Querschnittsaspekte sind Aufgaben wie Security, Logging, Transaktionalität oder Ähnliches. Dies sind Aufgaben, die unter Umständen in fast allen Komponenten verwendet werden müssen. Um nun nicht den Code „zu verschmutzen“ sollen diese möglichst transparent, also unsichtbar, in den Code eingewoben (weaving) werden.

Am bekanntesten ist wahrscheinlich **AspectJ**, das bei Xerox PARC entwickelt wurde und definierte Aspekte in den Java Bytecode nachträglich einweben kann. Dies geschieht mit einem Compiler, der üblicherweise bequem in den Buildzyklus integriert werden kann.

Aspekte am Beispiel von AspectJ:

- **inter-type declarations**

Dies sind Möglichkeiten, um Methoden oder auch Felder oder Interfaces nachträglich in den Code hineinzuwoben. So z. B. eine Loggingfunktion an bestimmten Stellen im Code.

- **pointcuts**

Diese definieren die genauen Stellen im Code, wo etwas geschehen soll. Z. B. nach oder vor einer Methode, bei einer Objektinstanziierung oder einem Feldzugriff. Die kann sehr komfortabel mit Regular Expressions definiert werden. Eine einzige Stelle wird als „**joint point**“ bezeichnet.



Beispiel

```
call(void Car.setTire(id))
call(void Car.setTire(id)) || call(void Car.motor(mid))
call(public * Car.*(..))
```

- **advice**

Dies ist der Code, der ausgeführt werden soll, wenn ein Pointcut matched, also zutrifft. Hier kann der Entwickler **before**, **around** oder **after** angeben [werden], was sich dann auf den konkreten **joint point** bezieht.



Beispiel

```
before() : getBook() {
    DB.log(book);
}
```

Viele große Softwareanwendungen - wie z. B. der Application Server JBoss - sind auf einer aspektorientierten Architektur aufgebaut.

Siehe dazu auch weiterführend: [www http://www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)

11.3 DSLs

Wie wir im Kapitel der Architekturstile gesehen haben gibt es unterschiedliche Ansätze für verschiedene Probleme. Einer dieser fünf Fälle war der, dass das Problem mit einer (virtuellen) Maschine abgearbeitet werden kann und in Form einer speziellen Sprache vorliegt. Die einzelnen Sprachelemente werden dann abgearbeitet oder interpretiert und somit ein Ergebnis oder Endzustand ermittelt.

Dies wird als domänenspezifische Sprache oder auf Englisch als **Domain-Specific Language** DSL bezeichnet. Kennzeichnend für diese Sprache ist, dass sie exakt und optimal auf das Problem angepasst ist und auch leicht von Menschen zu erstellen oder zu erlernen sein soll.

Global gesprochen sind fast alle Sprachen DSLs. Also z. B. die Farben einer Ampel an der Kreuzung oder Musiknoten. Beides sind Beispiele für eine Notation die ideal zum Problem passt und (hier nicht durch IT) weiterverarbeitet werden kann.

In Bezug auf die Informatik sind meistens Sprachen gemeint, die viel spezieller sind als die bekannten Programmiersprachen. Der Vorteil solcher Sprachen liegt darin, dass Sie eine Problemlösung kürzer beschreiben können - und zumeist lesbarer sind als es beispielsweise in Java möglich wäre. Zudem sind sie sehr leicht verständlich und lernbar, da der Sprachumfang oft geringer ist als der bei Hochsprachen.

Zu diesem Thema ist das [www DSL Buch](#) von **MARTIN FOWLER** sehr hilfreich welches er noch während des Schreibens öffentlich ins Internet stellt.

Wichtigstes Unterscheidungsmerkmal einer DSL ist:

- **Interne DSL**

Hier ist die DSL Teil einer Programmiersprache, wie dies z. B. bei [www Rake](#) oder den *Unit-Frameworks der Fall ist.

- **externe DSL**

Hier ist die DSL eine komplett neue Sprache, die nicht zwingend von einer existierenden Programmiersprache abhängt. Ein gutes Beispiele ist SQL.

Hier ein Beispiel unter [www LINQ](#):



Beispiel

```
public void Linq1() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

    var lowNums =
        from n in numbers
        where n < 5
        select n;
    Console.WriteLine("Numbers < 5:");
    foreach (var x in lowNums) {
        Console.WriteLine(x);
    }
}
```

Quelle: [www 101 LINQ Examples](#)

Hierbei sind die Kommandos **from**, **where** und **select** einfach in die Sprache hinein definiert. Damit ist es möglich Datenbank-, XML oder Collection-Abfragen in C# selbst zu machen und kein SQL mehr zu benötigen. Dennoch hat man volle C# Mächtigkeit.

Derartige DSL Sprachen lassen sich in dynamischen Sprachen wie Ruby, Python oder Groovy leicht selbst definieren, da diese Sprachen sich einfach um neue Schlüsselwörter erweitern lassen.

Auch die Buildsprache Rake ist eine Erweiterung um neue Schlüsselwörter von Ruby:



Beispiel

```
desc "Test your code"
Rake::TestTask.new('test') do |t|
  t.pattern = 'test/**/*.rb'
  t.warning = true
end
```

```
Rake::RDocTask.new do |rd|
  rd.rdoc_dir = '_doc'
  rd.main = "README.rdoc"
  rd.rdoc_files.include("README.rdoc", "lib/*.rb") # lib/**/*.rb
end
```

Hier wird einfach der Code getestet und die (Ruby) Dokumentation erzeugt.

Der große Vorteil beider Sprachen ist, dass man die volle Mächtigkeit der Programmiersprache hat und nicht wie bei ANT Dinge in Java ausprogrammieren und einbinden muss.

11.4 Architecture Definition Languages

Aufgrund des großen Interesses für Architektur ist es nicht verwunderlich, dass neben UML noch weitere Architektursprachen entstanden sind. Man bezeichnet diese als **ADLs** (Architecture Definition Languages). Dies sind meistens formale Sprachen, bei denen entsprechende Werkzeuge die Architektur auch gleich simulieren können. Dies ermöglicht im Vorfeld Probleme besser zu analysieren und erinnert an Ablaufmodellierung in der Autoindustrie. So können beispielsweise Performance Bottlenecks besser erkannt werden.

Wie UML müssen diese ADLs vom Menschen erstellt und verstanden werden können (ggf. mit Visualisierungswerkzeugen). Und natürlich soll wie bei MDA auch eine Code Generierung möglich sein.

Vorreiter - wie so häufig in der IT-Architektur - ist die Carnegie Mellon Universität mit ADLs wie **ACME**, Wright und UniCon. Jedoch gibt es noch weitere Systeme wie Aesop, Rapide, (C2) SADL, Darwin, MetaH, Koala, Weaves und SSEP.

Beispiele sind unter anderem bei [www.Rapide](#) von Stanford zu finden.

Alle Frameworks und Sprachen verwenden **Komponenten, Konnektoren und Konfigurationssprachmittel** für die Interaktion der ersten beiden Teile.

Natürlich ist die Forschung hier erst am Anfang und so entstehen erst langsam Sprachen und Architekturumgebungen, die z. B. als Eclipse Plug-In genutzt werden können.

12 Nichtfunktionale Anforderungen

Für einen Architekten ist es wichtig, sich den Anforderungen (Requirements) bewusst zu werden. Die funktionalen Anforderungen spiegeln sich unmittelbar im Design wieder. So hat eine zusätzliche funktionale Anforderung (z. B. wir behandeln auch die Abgeltungssteuer) klare und hoffentlich ausschließlich *einfache* Auswirkungen auf die Architektur, da sich z. B. einfach neue Komponenten oder Domain Objekte ergeben.

Ganz anders sieht es bei nichtfunktionalen Anforderungen aus! Auch wenn nur wenige nichtfunktionale Anforderungen vorliegen, haben diese meist *erhebliche* Auswirkungen auf die komplette Systemarchitektur.

Schauen wir uns die folgende Liste an:

1. Wartbarkeit
2. Testbarkeit
3. Installierbarkeit
4. Performance
5. Skalierbarkeit
6. Antwort- / Ausfallsicherheit
7. Erweiterbarkeit
8. Einfachheit
9. Security
10. Administrierbarkeit
11. Herstellerunabhängigkeit
12. Portierbarkeit / Plattformunabhängigkeit
13. Interoperabilität
14. Internationalisierung
15. Time to Market
16. Kosten

Hier wird schnell klar, dass Anforderungen wie Performance oder oder Testbarkeit durchaus heftige Auswirkungen auf die Architektur haben. Stellen Sie sich vor ihr Chef sagt, das System soll in 2 Wochen an den Markt gehen. Oder es muss 1000 http Anfragen pro Sekunde verarbeiten können.

Ein Architekt muss diese Punkte also sehr früh **gewichten**, um die richtigen Entscheidungen treffen zu können (falls diese noch nicht vorliegen). Beispielsweise mit einem Bewertungssystem wie ++,+,+,+,O,-,--,---.

Oft wird das Ausbalancieren von extremen Requirements und Architekturstilen als eine der entscheidenden Aufgaben des Architekten betrachtet.

13 Guter Architekt

Was zeichnet nun einen guten Architekten aus? Wir wollen nochmals alle Punkte zusammentragen:

1. Er kann **visualisieren** (z. B. mit UML) und darauf aufbauend Visionen vermitteln
2. Er **diskutiert** mit anderen Profis und seinen Entwicklern über den Architekturvorschlag
3. Er hat **Designerfahrung**, viele Entwürfe gemacht und aus Fehlern gelernt
4. Er kennt sehr viele Design Patterns und **Enterprise Design Patterns**
5. Er kann **strukturieren**, d. h. trennen und gruppieren
6. Er hat **Zuständigkeiten** im Blick

Der Punkt Visualisieren ist natürlich nicht einfach. Wie soll das geschehen?

Initial sind sicher einige große Bilder - in denen **Komponentendiagramme** oder auch Paket- und Klassendiagramme gezeigt werden - eine gute Möglichkeit. Später muss jedoch zunehmend verfeinert und alle anderen Dinge visuell kommuniziert werden wie z. B. Anforderungen oder Visionen. Generell müssen daher alle Punkte kommuniziert und visualisiert werden, die vorher angesprochen worden sind (Sichten, Stile, Ebenen, etc.).



Was tut ein Architekt also eigentlich global gesehen nacheinander?

1. Er versteht die Geschäftsanwendungsfälle
2. Er versteht, bewertet und gewichtet die Anforderungen
3. Er entwirft eine Architektur
4. Er kommuniziert die Architektur
5. Er setzt die Architektur, prüft und betreibt Risikomanagement

Insbesondere zum letzten Punkt gehört die Realisierung von Prototypen oder Skelettvorgaben, die Erstellung von Richtlinien und die Kontrolle der Architektur. Dazu gehören im Verlaufe des Projektes auch Entscheidungen, ob Software selbst gebaut wird oder gekauft wird.

Mit Hilfe dieser Punkte und den vorangegangenen Stilmitteln, Werkzeugen, Patterns, Ebenen und Beispiel-Architekturen sollte es ihnen nun möglich sein, für Softwareprojekte eine tragfähigere Architektur zu definieren.

Zusammenfassung

- Für Architekten gibt es weder eine gute Anleitung, noch klare Richtlinien. Es gibt lediglich ein Sammelsurium an guten Ideen, Verfahren oder Dingen, die man prüfen kann.
- Es gibt viele gute Bücher, weiterführende Links und Audiomaterial
- Gute Architektur ist wichtig für langfristig tragfähige Softwarelösungen.
- Wir bauen keinen Monolithen
- Gute Architektur macht änderbarer, wartbarer, lesbarer und erhöht die Qualität.
- Ein Architekt hat viele „Werkzeuge“ für seine Arbeit (Abstraktion, Kapselung, Kopplung, etc.)
- Es gibt verschiedene Architekturgrundformen / -stile die für verschiedene Probleme passen.
- Ein Architekt kennt ESB-, SOA-, N-Tier, Middleware, Peer-To-Peer- und viele weitere Architekturformen.
- Als guter Architekt hat man beispielsweise Erfahrung in .NET, JavaEE, Webservice und REST.
- Standard wie NGOSS sind Industrierelevant. Standards wie OSGi sollte man kennen.
- Enterprise Design Patterns sind das Fundament einer leistungsfähigen Architektur
- Nichtfunktionale Anforderungen sind kritisch, d. h. extrem wichtig für jede Architektur

Abschließend sei dem interessierten Studenten nochmals der Literaturteil empfohlen. So nebenbei mal einen Blog zu hören oder in der Bahn ein Buch zu lesen ist kaum aufwendig.

Wissensüberprüfung




Formulieren

Übung ARC-04

Nutzen von Softwarearchitektur

Wozu dient eine gute Softwarearchitektur hauptsächlich?

 Musterlösung (Siehe Anhang)

Bearbeitungszeit: 15 Minuten



Zuordnung

Übung ARC-05

Was ist was?

Ordnen Sie zu!

- A** In allen Softwarekomponenten die Idee vermitteln.
- B** Ich trenne und ordne die Komponenten in Gruppen.
- C** Ich versuche die Methode, das Feld, die Klasse, die Komponente Ihre Aufgabe selbst mitteilen zu lassen.
- D** Sind Basisbausteine einer guten Architektur.
- E** Nicht jeder soll sehen, was in Komponenten en Detail geschieht.
- F** Ich gruppiere gleiche Komponenten oder Domain Objekte zusammen.
- G** Ich versuche die gegenseitigen Abhängigkeiten zu vermindern.

- ☐ Kapselung
- ☐ Packages
- ☐ Vision
- ☐ Design Patterns
- ☐ Kohäsion
- ☐ Namensgebung
- ☐ Kopplung

 [Test wiederholen](#) [Test auswerten](#)



Richtig oder Falsch

Übung ARC-06

Richtig oder falsch?

Haken sie an!

	Richtig	Falsch	Auswertung
Zentral-Dezentral imd Client-Server ist das Selbe!	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
Rich-Clients sind ziemlich teuer.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
N-Tier Architekturen haben mehrere Softwareschichten.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
Ein Container nimmt nur die Komponenten aus.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
Eine Middleware Architektur beinhaltet eine Software, die Services via RPC vermittelt.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
Ein ESB ist das Pendant zu einem Company Service Train.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
In einer Peer-to-Peer Architektur kann man meist nicht deterministisch lokalisieren, wo ein Dienst angeboten wird.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
Ein Application-Server ist ein Web-Server.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
MDA ist ein Verfahren zur Verbesserung der Dokumentationsqualität.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>

[?](#) [Test wiederholen](#) [Test auswerten](#)


Richtig oder Falsch

Übung ARC-07

Aspect J

Bei Aspect J gibt es...

	Richtig	Falsch	Auswertung
...die Möglichkeit, genaue Stellen im Code zu identifizieren.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
...pointcuts.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
...einen joint.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
...separate-cuts.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>
...joint-points.	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>

[?](#) [Test wiederholen](#) [Test auswerten](#)


Freitext

Übung ARC-08

DSL

DSL ist eine Abkürzung. Wofür steht es?

[?](#) [Test wiederholen](#) [Test auswerten](#)



Zuordnung

Übung ARC-09

Werkzeuge und Stilmittel

Ordnen Sie die unten aufgeführten Werkzeuge / Stilmittel den einzelnen Sachverhalten zu.

Änderungen, Fehler (Exceptions) und Datenflüsse transparent machen:	
Durch gute fachliche Gruppierungen werden auch Technologieshifts (z.B. von Corba nach J2EE) überlebt:	
Wird durch Prototyping und schrittweises Wachstum erreicht:	
Analyse der Zuständigkeiten:	
Namensgebung, interne und externe Dokumentation:	
Test, was passiert, wenn Komponenten sich ändern oder wegfallen:	
Anwendung der Sprachmittel für Gruppierungen:	
Fassaden, Black-Box-Prinzip, Schichtenbildung:	
Wird durch das Gesetz von Demeter und der Vermeidung von Zyklen realisiert:	
Interfaces als explizite Schnittstellen, Subclassing, Polymorphie:	

Entwurf für Veränderungen

Information Hiding

Hohe Kohäsion

Rückverfolgbarkeit

Abstraktion

Selbstdokumentation

Seperation of Concerns

Inkrementalität

Lose Kopplung

Modularität

? Test wiederholen Test auswerten

Einsendeaufgaben

Bevor Sie mit der Bearbeitung der Einsendeaufgabe beginnen, besprechen Sie bitte mit Ihrer Modulbetreuung in welcher Form die Aufgaben bearbeitet werden sollen.



Einsendeaufgabe

Einsendeaufgabe ARC-E1

Vorgehen

Stellen Sie Ihren eigenen Katalog / Checkliste bzw. Ihren eigenen Vorgehensfahrplan als Architekt zusammen.

Bearbeitungszeit: 30 Minuten



Einsendeaufgabe

Einsendeaufgabe ARC-E2

Entwurf I

Entwerfen Sie eine Architektur für Ihr Thema in Softwaretechnik. Diskutieren Sie diese in der Präsenz mit dem Dozenten.

Bearbeitungszeit: 60 Minuten



Einsendeaufgabe

Einsendeaufgabe ARC-E3

Entwurf II

Entwerfen Sie eine Architektur für das neue Portal **<http://eternalyou.com>**. Diese Plattform soll in einem Jahr live gehen und eine persönliche Webseite für jede Person der Welt zur Verfügung stellen, die laut Provider (= Ihrem Arbeitgeber) unendlich lange Online ist (also auch nach ihrem Tode). Ihr Chef möchte von Ihnen, dass sie noch weitere fachliche und nicht-fachliche Anforderungen finden.

Bearbeitungszeit: 60 Minuten

Appendix

Nutzen von Softwarearchitektur

Wozu dient eine gute Softwarearchitektur hauptsächlich?

Antwort:

Komprimierbarkeit | Änderbarkeit* | Kostengünstigkeit | Wartbarkeit* | Qualität* | Deploybarkeit