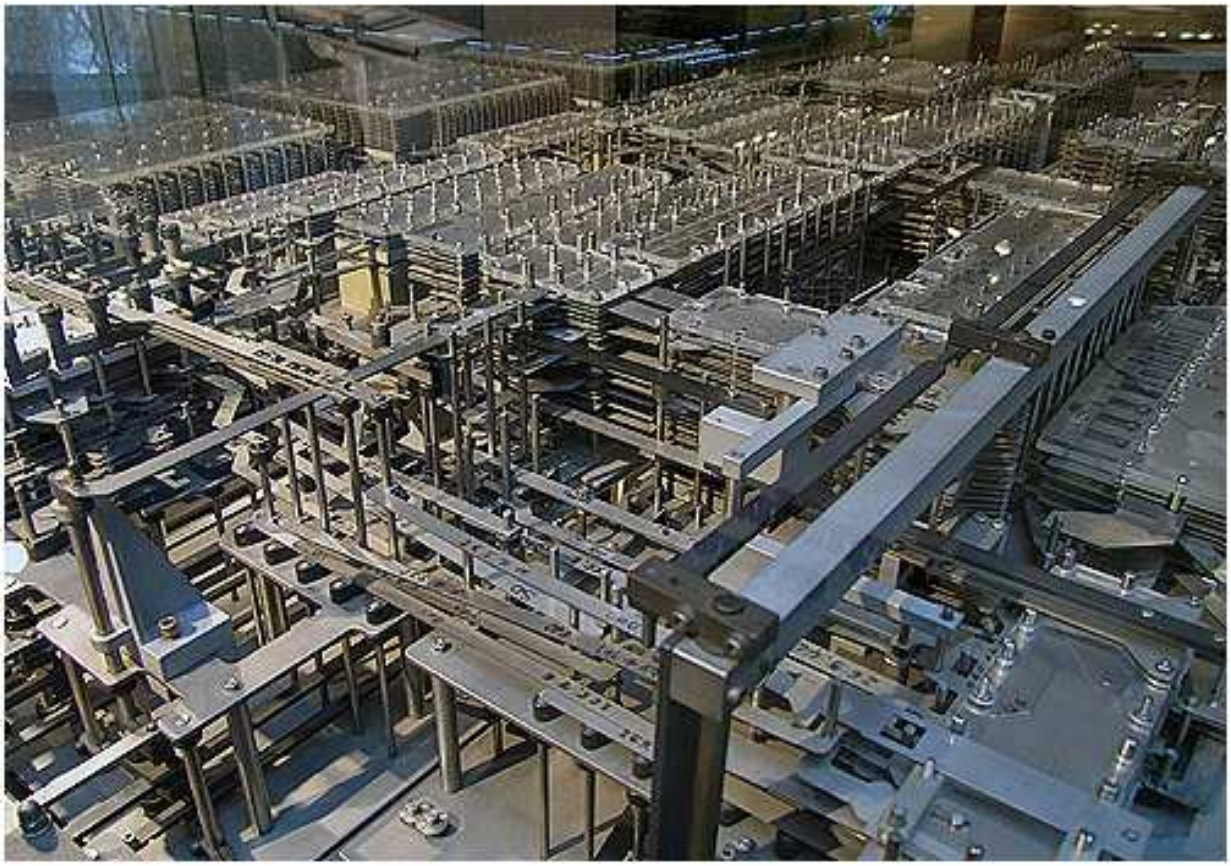
Zürcher Hochschule
für Angewandte Wissenschaften

Software Architektur - Basics



Modul Systemarchitektur - Software Architektur
14.& 21.9.2015
Daniel Liebhart

Software Architektur - Grundlagen

Daniel Liebhart, 14. & 21.9.2015

Verfasser: Daniel Liebhart
Lektorat: Bruno Holliger (2. Auflage), Jürg Fuchs (3. Auflage)
11. Auflage: 2015
Version 11.0

Inhalt

Referenzen und Abkürzungen

Referenzen

[Booch et al. 2005]	G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide, Addison-Wesley Professional, 2005
[Brooks 1987]	F.P. Brooks, Jr.: No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, Vol. 20, No. 4, April 1987
[Brooks 2007]	F.P. Brooks, Jr.: The Mythical Man-Month, Addison-Wesley, August 2007
[Buschmann et al. 1996]	F. Buschmann, R. Meunier, H.Rohnert, P. Sommerlad, M. Stal; A System of Patterns, John Wyley & Sons Ltd. 1996
[Demarco, Plauger, 1979]	T. Demarco, P.J. Plauger: Structured Analysis and System Specification, Prentice-Hall, 1979
[Dustdar et al. 2003]	S. Dustdar, H. Gall, M. Hauswirth: Software-Architekturen für Verteilte Systeme, Springer Verlag, Juli 2003
[Gottlob et al 1990]	G. Gottlob, T. Frühwirt, W. Horn: Expertensysteme, Springer Verlag, 1990
[Hoare 1978]	C.A.R. Hoare: Communicating Sequential Processes, Communications of the ACM 2, 8, August 1978
[Hruschka, Starke 2012]	P. Hruschka, G. Starke: Knigge für Softwarearchitekten, Entwickler.Press, März 2012
[IEEE610 2002]	IEEE Std 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology, IEEE 2002
[ISO/IEC/IEEE 2011]	ISO/IEC/IEEE-42010-2011: ISO/IEC/IEEE Systems and software engineering -- Architecture description, 24.11.2011
[ISO 7498-1 1994]	ISO 7498-1, DIN ISO 7498: Information technology - Open Systems Interconnection - Basic Reference Model: The basic model.
[Krasner, Pope 1988]	G. E. Krasner, S. T. Pope: A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming, Aug-Sept 1988
[Meijer, Schulte 2003]	E. Mejier, W. Schulte: Unifying Tables, Objects and Documents, Microsoft Corporation
[Monroe et al. 1997]	R.T. Monroe, A. Kompanek, R. Melton, D. Garlan: Architectural Styles, Design Patterns and Objects, IEEE Software, 1997
[Perry, Wolf 1992]	D. E. Perry, A. L. Wolf: Foundation for the Study of Software Architecture, ACM Software Engineering Notes Vol 17 No 4, Oct 1992
[Shaw, Garlan 1996]	M. Shaw und D. Garlan: Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall, 1996
[Srinivasan 1995]	S. Srinivasan: RPC: Remote Procedure Call Protocol Specification Version 2, RFC 1831, Netowrk Working Group, Aug 1995
[Venners 2000]	B. Venners: Inside the Java Virtual Machine, McGraw-Hill, 2000
[Yourdon 1997]	E. Yourdon: Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice Hall, 1979
[Zachmann 2005]	J. A. Zachman: The Framework For Enterprise Architecture: Background, Description and Utility 2005

Abkürzungen

ANSI	American National Standards Institute
API	Application Programming Interface
CSP	Communicating Sequential Processes
DMS	Document Management System
EAI	Enterprise Application Integration
GPU	General Processing Unit
ISA	Information Systems Architecture
IEEE	Institute of Electrical & Electronics Engineers
IEC	International Engineering Consortium
ISO	International Organization for Standardization
JAAS	Java Authentication & Authorization Service
LDAP	Lightweight Directory Access Protocol
MS	Microsoft
MVC	Model View Controller
OLAP	Online Analytical Processing
ONC	Open Network Computing
OSI	Open Systems Interconnection
OCS	Oracle Collaboration Suite
OID	Oracle Internet Directory
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SSO	Single Sign On
SASD	Structured Analysis - Structured Design
SQL	Structured Query Language
UML	Unified Modelling Language

Prinzipien

2.1 Einleitung

Die Architektur von IT-Systemen ist im Gegensatz zur klassischen Architektur eine junge Wissenschaft. Allgemein anerkannte Prinzipien zur standardisierten Konstruktion und die normierte Validierung einmal fertig gestellter Systeme gibt es noch nicht, respektive sie werden in der industriellen Praxis kaum verwendet. Dennoch sind in einem gewissen Sinn die Prinzipien der Gebäudearchitektur auf die Informatikarchitektur übertragbar. Gemeinsam sind der Architektur und der Informatik die planerische Zusammenstellung des gewünschten Ganzen, die Gestaltung eines "Lebensraums" für Menschen sowie die Einhaltung von Grundprinzipien, die gute von schlechter Architektur unterscheiden. Unterschiede zwischen klassischer Architektur und Informatik ist das Fehlen der Regeln der Statik in der Informationstechnologie und die fehlende Gesamtdarstellung aller Aspekte einer Konstruktion in einem geometrischen Plan.

Eine gute Architektur basiert auf einer Reihe von grundlegenden Prinzipien, beeinflusst die allgemeinen Systemeigenschaften und erlaubt einen geregelten und kommunizierbaren Entwicklungsprozess. Sie ermöglicht vor allem, dass Software-Systeme miteinander und nebeneinander existieren können, ohne dass sie sich gegenseitig stören.

2.2 Architektur und Systemeigenschaften

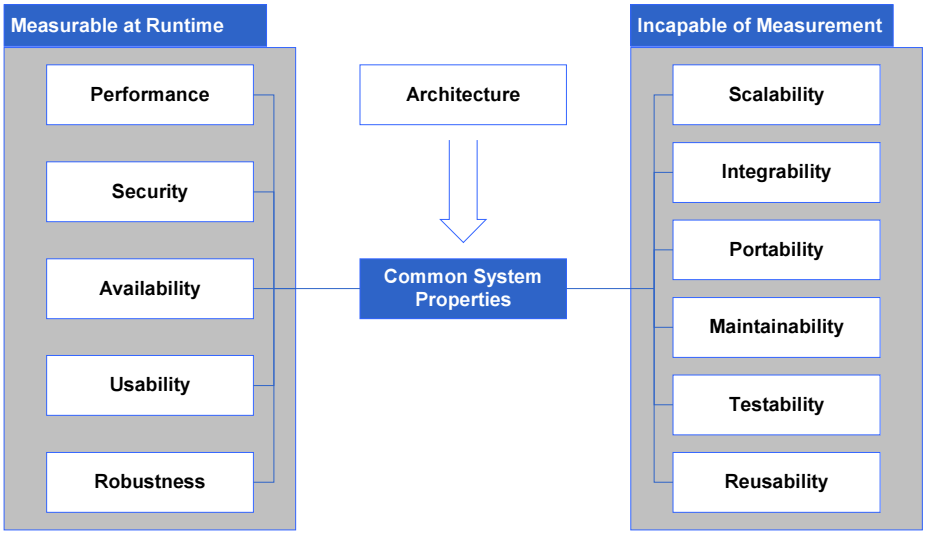


Abbildung 1: Der Einfluss der Architektur auf die allgemeinen Systemeigenschaften

Die Qualität einer Software-Architektur hat einen Einfluss auf die „allgemeinen Systemeigenschaften“ eines mittels der gewählten Architektur umgesetzten Systems. Diese Systemeigenschaften werden wiederum in zwei Gruppen aufgeteilt. Eigenschaften, die zur Laufzeit eines Systems gemessen werden können und Eigenschaften, die nur indirekt gemessen werden können [Dustdar et al. 2003]. Zur Laufzeit gemessene Eigenschaften geben Auskunft über das allgemeine Verhalten des Systems, wie die Reaktionszeiten und die Ergonomie. Nicht messbare Eigenschaften

werden erst in einer späteren Phase des Systems sichtbar, wie beispielsweise die Änderungsfreundlichkeit und die Testbarkeit. Allen diesen allgemeinen Systemeigenschaften ist gemeinsam, dass sie vor den Bau eines Systems bekannt sein sollten, was in der Realität selten der Fall ist. Was im konkreten Fall sinnvoller weise anzuwenden ist, d.h. welche Systemeigenschaften wie genau spezifiziert werden sollen, ist abzuschätzen und im Idealfall auf den Einfluss auf die Gestehungskosten und die Entwicklungszeit hin zu untersuchen.

In vielen Informatikprojekten sind nur wenige allgemeine Systemeigenschaften bekannt. Üblicherweise ist die Anzahl der User, die Menge der zu verarbeitenden Daten sowie die geforderte Reaktionszeit und Verfügbarkeit eines Systems bereits vorgängig spezifiziert. Die Architektur als Gesamtaufbau eines Systems beeinflusst diese Faktoren. Die Architektur eines Systems für 10 User sieht vollständig anders aus als diejenige eines Systems für 10'000 User.

Zur Laufzeit messbare Eigenschaften:

- **Performance (Reaktionszeit / Durchsatz):** Ein System muss die geforderten Antwortzeiten garantieren.
- **Security (Sicherheit):** Ein System muss sicher vor unautorisiertem Zugriff und vor mutwilliger Zerstörung sein.
- **Availability (Verfügbarkeit):** Ein System muss zur Verfügung stehen und dabei definierte Mindestanforderungen erfüllen.
- **Usability (Verwendbarkeit):** Ein System muss für den vorgesehen Zweck eingesetzt werden können.
- **Robustness (Stabilität):** Ein System muss stabil laufen und darf unter Last nicht seinen Dienst teilweise oder ganz einstellen.

Nicht zur Laufzeit messbare Eigenschaften sind:

- **Scalability (Skalierbarkeit):** Ein System muss ausgebaut werden können.
- **Integrability (Integrierbarkeit):** Ein System muss sich nahtlos in eine existierende Umgebung einfügen lassen.
- **Portability (Portabilität):** Ein System muss verschiedene Plattformen unterstützen.
- **Maintainability (Wartbarkeit):** Ein System muss definierte Wartungsschnittstellen aufweisen und klar und übersichtlich strukturiert sein.
- **Testability (Testbarkeit):** Ein System muss als Ganzes und in seinen Einzelkomponenten testbar sein. Das Testen des Systems muss vom System selbst durch Hilfsmittel (Logs, Traces) unterstützt werden.
- **Reusability (Widerverwendbarkeit):** Systemteile müssen sich für andere Systeme wieder verwenden lassen.

2.3 Definitionen

2.3.1 Die Aufgabe des Softwarearchitekten

Der Knigge für Softwarearchitekten umschreibt die Aufgaben des Softwarearchitekten so [Hruschka, Starke 2012]:

> **„Wir sehen als wesentliches Merkmal guter Architekten, dass er (oder sie) unter den jeweiligen Umständen die bestmöglichen Systeme konstruieren und deren Entwicklung begleiten. Systeme, die verständlich, langlebig, wartbar, funktional, performant und sicher sind. Systeme, die robust auf Fehler reagieren und ihre jeweiligen Stakeholder positiv erstaunen, statt zu nerven. Kurz gesagt: Gute Architekten liefern gute Qualität.“**

2.3.2 Zachmann

J.A. Zachmann, der Erfinder der ISA (Information Systems Architecture) definiert IT Architektur [Zachmann 2005] folgendermassen:

> **„Eine Architektur ist eine Menge von Gestaltungsgrundsätzen oder beschreibenden Darstellungen, die relevant sind zur Beschreibung eines Objektes, sodass es gemäss den Anforderungen produziert werden kann und während seines Lebenszyklus unterhalten werden kann.“**

2.3.3 IEEE

Der IEEE Standard 610.12 (IEEE Standard Glossary of Software Engineering Terminology) definiert Architektur [IEEE610 2002]:

> **Architektur: Die organisierte Struktur eines Systems oder einer Komponente.**

Der ISO/IEC/IEEE-42010 Standard 1471-2000 “Recommended Practice for Architectural Description of Software-Intensive Systems definiert [ISO/IEC/IEEE 2011]:

> **Architektur ist definiert als die grundlegende Organisation eines Systems, eingebettet in seinen Komponenten, deren Beziehungen unter einander und der Umgebung, und als die Prinzipien die das Design und die Evolution des Systems beherrschen.**

2.3.4 Booch, Rumbaugh & Jacobson

Die Autoren des Buches “The UML Modeling Language User Guide” (Grady Booch, James Rumbaugh und Ivar Jacobson [Booch et al. 2005]) definieren Architektur wie folgt:

> **„Eine Architektur ist eine Menge von signifikanten Entscheidungen über die Organsation eines Software Systems, der Auswahl der einzelnen Strukturelemente und deren Schnittstellen sowie deren Verhalten.“**

2.3.5 Formale Definition

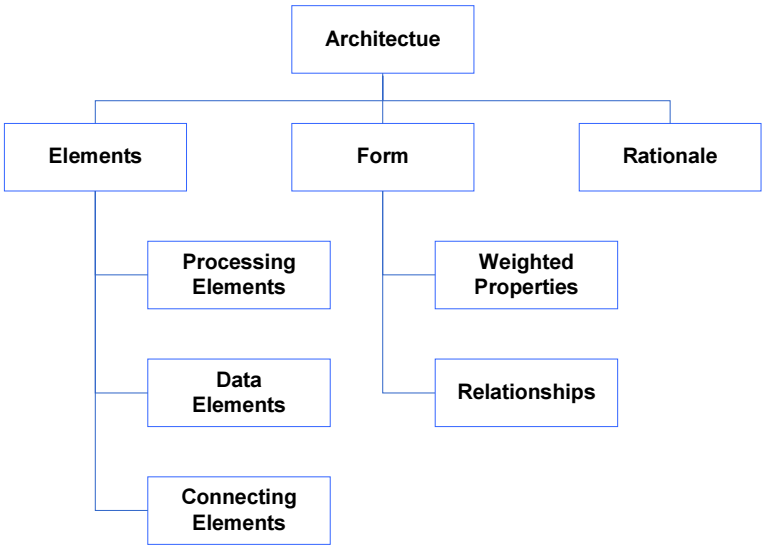


Abbildung 2: Die Elemente einer Architektur.

Eine mögliche formale Definition von Architektur gemäss Dewayne E. Perry und Alexander E. Wolf ist [Perry, Wolf 1992]:

> **Architecture = {Elements, Form, Rationale}**

- **Elements:** Die Elemente einer Architektur sind entweder Processing Elements oder Data Elements oder Connecting Elements.
- **Processing Elements:** Diese Elemente führen Transformationen auf Data Elements aus.
- **Data Elements:** Sämtliche Daten eines Systems werden als Data Elements einer Architektur beschrieben.
- **Connection Elements:** Connection Elements können entweder Processing Elements oder Data Elements oder beides sein. Beispiele sind Procedure Calls, Shared Data und Messages.

- **Form:** Die Form einer Architektur wird beschrieben durch Wheighted Properties und Relationships.
- **Weighted Properties:** Die Gewichtung von Eigenschaften eines Elements erlaubt es einer Architektur zwischen zentralen und dekorativen Eigenschaften zu unterscheiden. Die Properties sind in einer Architektur ein Mittel zur Definition der Rahmenbedingungen eines Elements der Architektur. Sie definieren die minimalen Anforderungen, die ein Element erfüllen muss.
- **Relationships:** Relationships werden zur Definition der Platzierung eines Elementes in einem bestimmten Kontext von Elementen verwendet.
- **Rationale:** Dahinter verbirgt sich die Motivation zur Auswahl bestimmter Architektur-Elemente. Sie ist Ausdruck der Abbildung der Systemanforderungen, die von funktionalen hin zu allgemeinen Systemsanforderungen reicht.

2.3.6 Allgemeine Definition

Die verallgemeinerte Definition von Architektur ist:

> **Allgemeiner ist Architektur die Kunst oder Wissenschaft des planvollen Entwurfs der menschlichen Umwelt.**

Der Begriff Architektur wird auch in anderen Gebieten für geplante, komplexe Strukturen und deren Konzeption und Entwurf verwendet. Dabei ist eine Hierarchisierung in Unterstrukturen charakteristisch. Diese Unterstrukturen werden auch Konstruktionselemente genannt. So werden heute Häuser in Elementbauweise gebaut. Die einzelnen Elemente werden aufgrund ihrer allgemeinen Systemeigenschaften ausgesucht und auf einander abgestimmt. So ist die Belastbarkeit einer stützenden Wand abhängig von der maximalen Tragfähigkeit eines Bodens und der Anzahl der eingesetzten stützenden Wände. Die stützende Wand wiederum besteht aus einer Reihe von Konstruktionselementen wie beispielsweise eines Fensters oder einer Türe. Daneben sind Materialeigenschaften der Mauer zu betrachten, um die konkreten Dimensionen zu definieren. Das Fenster wird ebenfalls aus einer Reihe von Elementen zusammengesetzt, die wiederum allgemeine Systemeigenschaften aufweisen. Diese Hierarchie von Elementen kann als Schichtung der Konstruktion angesehen werden, was wiederum eine wesentliches Eigenschaft der Systemplanung im Bereich der Informatik darstellt, da auch hier ein Problem zu seiner Lösung in mehrere Niveaus (Ebenen) gegliedert wird.

2.4 Architektur und Moduleigenschaften

Eine gute Architektur ist in einem konkreten Software Design sichtbar. Ein gutes Software Design basiert immer auf einer Reihe von klar definierten Prinzipien, die die Gesamteigenschaften des Zielsystems massgeblich beeinflussen. Die Prinzipien des Software Designs beziehen sich immer auf einzelne Module (Klassen, Pakete, Komponenten, Subsysteme) und auf das Verhältnis dieser Module zueinander.

2.4.1 Modularity (Modularität)

Modularität (Modularity) bedeutet, dass Komponenten eines Designs aus leicht austauschbaren, verständlichen und in sich geschlossenen Teilen bestehen sollten. Dieses Prinzip unterstützt sowohl die Arbeitsteilung bei der Entwicklung also auch die Wartung eines Systems. Je grösser ein System ist, desto wichtiger ist die Modularität. Typische Module eines Designs sind: Access Control, Database Access, Rule Engine, Validator.

2.4.2 Portability (Portabilität)

Portabilität ist dann gegeben, wenn eine Software oder Teile davon so gestaltet werden, dass sie auch in anderen Umgebungen lauffähig sind. Dieses Prinzip hat in zweierlei Hinsicht Konsequenzen. Einerseits sind portable von nicht portablen Modulen zu trennen, andererseits wird durch Portabilität in einem System eine Reduktion erreicht. Die Trennung von portablen und nicht portablen Modulen erfolgt in vielen Fällen über eine Zusammenfassung umgebungsabhängiger Funktionen in zentrale Module, die gegenüber den portablen Modulen ein und dieselbe Schnittstelle aufweisen. Die Reduktion durch Portabilität hat zur Folge, dass Umgebungsspezifika weniger stark in einem Design berücksichtigt werden, das System also robuster gegenüber Veränderung der Umgebung wird.

2.4.3 Changeability (Formbarkeit)

Jedes System erfährt im Laufe seiner Existenz eine Reihe von Änderungen und Anpassungen. Je formbarer ein System ist, desto leichter sind diese Änderungen durchzuführen. Dieses Prinzip ist nicht zu verwechseln mit Generi-

zität also mit der Generalisierung von Funktionalität. Nur weil ein System generisch gestaltet worden ist, heisst das noch lange nicht, dass es auch leicht geändert werden kann. Formbarkeit kann durch die Trennung von „domain specifics“ (fachspezifische Funktionalität) und „crosscutting concerns“ (allgemeine Systemfunktionen) erreicht werden.

2.4.4 Conceptual Integrity (Konzeptionelle Integrität)

Diejenigen Teile eines Systems, die ähnliche Funktionen beinhalten, sollten auch ähnlich gestaltet werden. Oder auch „Was zusammen gehört, muss zusammen wachsen“. Dieses Prinzip bedeutet im System Design, dass bestimmte Komponenten, wie beispielsweise eine Workflow Engine, so aufgebaut werden sollen, wie eine Workflow Engine gemäss dem Stand der Entwicklung auch aufgebaut werden muss. Für einen System Designer bedeutet dies, dass die Erkenntnisse aus der Forschung und aus der Praxis („Industriestandard“) unbedingt in eine Gestaltung eines Systems einfliessen sollten. Im Minimum sollte reflektiert werden, warum ein bestimmter Standard-Ansatz nicht verfolgt wird.

2.4.5 Intellectual Control (Intellektuelle Kontrolle)

Ein Software Design sollte von den Zuständigen hinsichtlich Form, Inhalt und Komplexität detailliert verstanden werden. Auf der Ebene eines Gesamtdesigns bedeutet dies, dass ein Software Design dann unter intellektueller Kontrolle ist, wenn alle beteiligten Designer von der Qualität der Lösung überzeugt sind. In Bezug auf einzelne Teile eines Systems bedeutet dies, dass alle für das entsprechende Modul zuständigen Personen von der Richtigkeit und von der Qualität des Moduls überzeugt sind und dessen Form (Schnittstelle), Inhalt (Funktionalität) und Komplexität (Umfang) verstehen.

2.4.6 Buildability (Herstellbarkeit)

Ein Software Design muss ein Zielsystem so spezifizieren, dass es von einem gegebenen Team in einer gegeben Zeit realisiert werden kann. Ein gutes Design antizipiert Änderungen an der Spezifikation während der Erstellung des Systems. Der Einsatz bestimmter Technologien muss auf das Know-how des Teams, welches das System realisieren soll, abgestimmt sein.

2.4.7 Coupling and Cohesion (Kopplung und Kohäsion)

Kopplung und Kohäsion beschreiben die Eigenschaften der Gruppierung von Modulen eines Systems. Kopplung beschreibt das Mass des notwendigen Verständnisses eines anderen Moduls, das notwendig ist, um ein gegebenes Modul zu entwickeln oder zu verändern. Je mehr Verständnis für andere Module notwendig ist, desto höher ist die Kopplung zwischen dem gegebenen und dem anderen Modul. Die Kopplung wird bestimmt durch die Abhängigkeit zwischen Modulen, der Komplexität der Schnittstelle und die Art des Informationsflusses. Kohäsion beschreibt im Gegensatz zur Kopplung die Wechselbeziehungen zwischen einzelnen Elementen innerhalb eines Moduls. Es gilt:

Je höher die Kohäsion individueller Module desto geringer die Kopplung.

Varianten der Kopplung

- **Data Coupling:** Daten werden zwischen Modulen eines Systems ausgetauscht.
- **Stamp Coupling:** Datenstrukturen werden zwischen Modulen eines Systems ausgetauscht.
- **Control Coupling:** Der Austausch der Daten zwischen Modulen steuert den Kontrollfluss.
- **Common Coupling:** Verschiedene Module greifen auf dieselben Daten zu (Shared Data).
- **Content Coupling:** Ein Modul verändert die internen Daten eines anderen Moduls

Varianten der Kohäsion

- **Coincidential Cohesion:** Die Gruppierung der Funktionalität eines Modules erfolgt durch Zufall.
- **Logical Cohesion:** Die Funktionalität ist zwar in einem Modul zusammengefasst, beziehen sich jedoch nicht aufeinander.
- **Temporal Cohesion:** Der Zeitpunkt der Verwendung bestimmt die Gruppierung der Funktionen.
- **Procedural Cohesion:** Die Aufrufabfolge der Funktionen bestimmt die Gruppierung.
- **Communications Cohesion:** Die Gruppierung der Funktionalität wird durch den gemeinsamen I/O bestimmt.

- **Sequential Cohesion:** Die Abfolge der Datenbearbeitung bestimmt die Gruppierung. Funktionen, deren Output gleichzeitig zum Input für andere Funktionen wird, werden in einem Modul zusammengefasst.
- **Functional Cohesion:** Diese Gruppierung hat zum Ziel, dass ein Modul Logik und Daten lokal halten kann (Information Hiding).

Idealerweise erreicht das System Design eine lose Kopplung zwischen den Modulen und eine hohe Kohäsion innerhalb eines bestimmten Moduls. Lose Kopplung wird durch Design-Unabhängigkeit, schmale Schnittstellen und wenig Schnittstellenverkehr erreicht, während hohe Kohäsion durch Einheit und Kapselung gefördert wird.

- **Design-Unabhängigkeit (Independence of Design):** Jedes Modul kann unabhängig von anderen Modulen entworfen werden und spätere Änderungen finden nur und ausschliesslich in einem Modul statt. Voraussetzung für die Design-Unabhängigkeit ist eine konstant gehaltene Definition der Schnittstellen sowie ein „einfrieren“ der Modulspezifikation.
- **Schmale Schnittstellen (Small Interfaces):** Die Anzahl der Meldungen, die über Schnittstellen zwischen Modulen ausgetauscht wird, ist klein.
- **Wenig Schnittstellenverkehr (Low Interface Traffic):** Die Häufigkeit des Informationsaustausches zwischen verschiedenen Modulen ist gering.
- **Einheit (Unity):** Ähnliche Problemstellungen und Anforderungen werden ähnlich umgesetzt. Das heisst, das System Design gruppiert Anforderungen aufgrund von Klassierungen, sodass eine klare Zuordnung stattfinden kann.
- **Kapselung (Encapsulation):** Abhängige Module werden zu grösseren Einheiten zusammengefasst.

2.4.8 Design for Change

Design for Change versucht die zu erwartenden Änderungen eines Systems zu antizipieren und die Gesamtlösung so zu gestalten, dass diese Änderungen möglichst einfach umgesetzt werden können. Dabei werden die zu erwartenden Änderungen gruppiert in Domain-Specific Changes, Analytical Changes und Downsizing Changes.

- **Domain-Specific Changes:** Änderungen, die in ähnlichen System bereits realisiert werden mussten. Erfahrungsgemäss ändern sich bestimmte Systeme aufgrund ihres Kontextes. Die meisten Änderungen werden aufgrund von geänderten betrieblicher oder auch fachlicher Anforderungen durchgeführt. Ist also ein System für einen Bestimmten Zweck im Einsatz, so ist eine Reihe von Änderungen absehbar. So wird beispielsweise praktisch immer etwas am Workflow geändert. Auch sicher sind Änderungen an der Benützerstruktur und der entsprechenden Vergabe der Rechte, da sich die Arbeitsteilung in einer Organisation ständig ändern.
- **Analytical Changes:** Die meisten Pflichtenhefte sind entweder ungenau oder stellenweise sogar falsch. Mehrdeutigkeiten und ungenaue Definitionen sind sehr häufig. In diesen Fällen sind Änderungen am System voraussehbar, da in vielen Fällen nicht jedes Detail der Spezifikation verifiziert werden kann.
- **Downsizing Changes:** Diese Änderungen erfolgen aufgrund von Budgetbeschränkungen und führen dazu, dass bestimmte Funktionalität weggelassen werden muss.

2.5 Schwierigkeiten des Software Designs

Software zu bauen ist und bleibt eine anspruchsvolle Aufgabe. Seit 1975 hat sich die wissenschaftliche Disziplin "Software Engineering" als eigenständiges Forschungsgebiet entwickelt. Eine Reihe von Forschern hat sich mit den grundlegenden Schwierigkeiten des Softwarebaus auseinandergesetzt, und versucht eine Antwort auf die Frage zu finden, wie die Produktivität signifikant gesteigert werden kann. Allen voran der Autor des 1975 geschriebenen Buches "The Mythical Man Month" [Brooks 2007] Frederik. P, Brooks JR, ehemaliger Leiter des Department of Computer Science der University of North Carolina. Er hat in diesem Buch die Höhen und Tiefen der Entwicklung des Operating Systems OS/390 von IBM reflektiert, dessen Entwicklung er jahrelang geleitet hatte. Er definiert darin eine Reihe von Thesen die nichts an Aktualität eingebüsst haben, zum Beispiel:

> **"Adding more people to a late projects makes it later"**
> **"Plan to throw one away; you will, anyhow"**

In seinem 1987 veröffentlichten Artikel "No Silver Bullet" versucht Professor Brooks die Gründe für die Schwierigkeiten des Softwarebaus aufzulisten [Brooks 1987]:

- **Komplexität (Complexity):** Software ist im Verhältnis zu seiner Grösse wahrscheinlich das Komplexeste, was von Menschenhand überhaupt erschaffen wird. Computer sind bereits komplexer als die meisten anderen Dinge auf dieser Welt. Sie kennen sehr viele Zustände. Software kennt noch viel mehr Zustände. F. P. Brooks beschreibt die Komplexität als essentielle Eigenschaft von Software, nicht als eine zufällige Eigenschaft. Wird durch Abstraktion ein vereinfachtes Modell eines komplexen Phänomens erzeugt, so ist für das Modell wesentlich, dass in ihm die wichtigsten Eigenschaften des Phänomens abgebildet werden können. Die anderen Eigenschaften können weggelassen werden. Dies ist jedoch bei Software nicht der Fall. Es können keine Eigenschaften weggelassen werden, um die Komplexität in einfachere Modelle abzubilden.
- **Konformität (Conformity):** Während Naturwissenschaften, wie beispielsweise die Physik auf einfachen und universell gültigen Basisregeln aufbauen können, um komplexe System zu modellieren, ist das für die Softwaretechnologie nicht der Fall. Das Umfeld eines Systems, also die Schnittstellen des Systems zu anderen Systemen variieren in dem Masse, wie die Komplexität dieser Umsysteme variieren.
- **Formbarkeit (Changeability):** Während andere Industrieprodukte, wie Autos oder Kühlschränke nur dadurch geändert werden, dass sie durch neue Modelle ersetzt werden, unterliegt Software immerwährenden Veränderungen. Da Software so einfach geändert werden kann, werden auch sehr viele Änderungen durchgeführt.
- **Unsichtbarkeit (Invisibility):** Software ist unsichtbar und nicht direkt visualisierbar. Es existiert keine geometrische Repräsentation von Software. Die Visualisierung von Software ist immer eine Überlagerung von verschiedenen Sichten (Control-Flow, Data-Flow, etc).

2.5.1 Brooks Lösungen

Brook hat bereits 1987 eine klare Vorstellung, wie die Schwierigkeiten des Softwarebaus bekämpft werden könnten. Seine Mittel sind Hochsprachen, Objektorientierte Programmierung, Künstliche Intelligenz, Expertensysteme sowie Automatische und Graphische Programmierung. Sämtliche Lösungsvorschläge von Brook haben bis heute die Produktivität der Software-Entwicklung nicht im selben Mass gesteigert, wie die Leistungsfähigkeit der Hardware gesteigert werden konnte.

2.6 Vorteile einer Software Architektur

Gemäss der Arbeitsgruppe für Software Architektur der Universität Köln hat die explizite Berücksichtigung der Aspekte der Architektur während der Softwareentwicklung folgende Vorteile:

- **Software Architektur** kann **als Grundlage für die Kommunikation** der an einem Softwareprojekt beteiligten Personen (Domänenexperten, Entwicklern, Benutzern etc.) untereinander dienen. Eine gute Software-Architektur umfasst sowohl alle Sichten, um den unterschiedlichen Anforderungen dieser Personengruppen gerecht zu werden, als auch den notwendigen "Glue", um diese Sichten in Beziehung zueinander zu setzen.
- **Software Architektur** kann **als** die **treibende Kraft des System-Designs** fungieren. Eine geeignete Systemarchitektur ist entscheidend für die Dekomposition des Systems und die Wahl geeigneter Abstraktionen, insbesondere weil Mängel der Architektur gravierende Auswirkungen auf das Gesamtsystem und den gesamten Entwicklungsprozess haben. Fehler in der Architektur eines Systems zu beheben ist aufwendig und entsprechend teuer. Die Wahl einer geeigneten Architektur ist außerdem entscheidend für die Einhaltung gesetzter nichtfunktionaler Eigenschaften.
- Eine **Software Architektur steckt den Rahmen für die Wiederverwendung einzelner Software-Artefakte ab**. Wieder verwendbare Softwareelemente können nur dann sinnvoll in einem System eingesetzt - d.h. wieder verwendet - werden, wenn sie sich nahtlos in die Architektur des Systems einfügen. Die Architektur eines Systems beschreibt somit Einschränkungen an die Menge einsetzbarer wieder verwendbarer Elemente. Sie schränkt dadurch den Suchraum für geeignete Elemente ein und erleichtert das Auffinden wieder zu verwendender Elemente.
- **Software Architektur ist der "Träger" von Qualitätscharakteristika und nichtfunktionaler Eigenschaften**, die nicht an einer bestimmten Systemkomponente festgemacht werden können.
- **Software Architektur kann** als Invariante über mehrere Softwareprojekte mit minimal unterschiedlichen Systemanforderungen gelten und somit **den Entwicklungsprozess** für eine Serie ähnlicher Projekte **vereinfachen**.
- **Software Architektur** ist die Basis für eine umfassende Betrachtung einer Software und **erlaubt die Analyse bestimmter Systemeigenschaften** bereits zu einem sehr frühen Entwicklungszeitpunkt des Systems.

Architekturbeispiele

3.1 Ein System zur Bekämpfung der Geldfälscher

3.1.1 Betriebliches Umfeld

Die Bekämpfung der Geldfälscher ist eine wichtige Aufgabe jeder Zentralbank, da Falschgeld die Integrität jeder Währung bedroht. Jede Zentralbank unterhält ein Labor zur Untersuchung der einmal aufgefundenen falschen Noten und Münzen. Eine Schwierigkeit der der Bekämpfung ist jedoch das frühzeitige Erkennen des Auftauchens und der Ausbreitung von Falschgeld. Aufgrund der staatlichen Organisation der Polizeiorgane ist ein rasches Erkennen von Fälschungen auf internationaler Ebene oft sehr schwierig und sehr langwierig. Mit den Internet und seinen Community Möglichkeiten stehen heute Mittel zur Verfügung gezielt Interessengruppen zu erreichen und eine rasche Kommunikation über Landergrenzen hinweg zu erlauben. Dieses Mittel soll ausgenutzt werden, um eine rasche Erkennung von Falschgeld zu unterstützen.

3.1.2 Anforderungen

Es soll ein System geschaffen werden, welches eine einfache Prüfung von Geldscheinen aufgrund ihrer Sicherheitsmerkmale erlaubt und die Meldung potentieller Fälschungen vereinfacht.

- Das System soll Banken, Wechselstuben und anderen Geldannahmestellen zur Verfügung gestellt werden.
- Das System muss Webfähig sein und gleichzeitig Daten mit dem Laborsystem der Zentralbank austauschen können.
- Das System muss mehrsprachig und mandantenfähig sein.

3.1.3 Logische Zielarchitektur

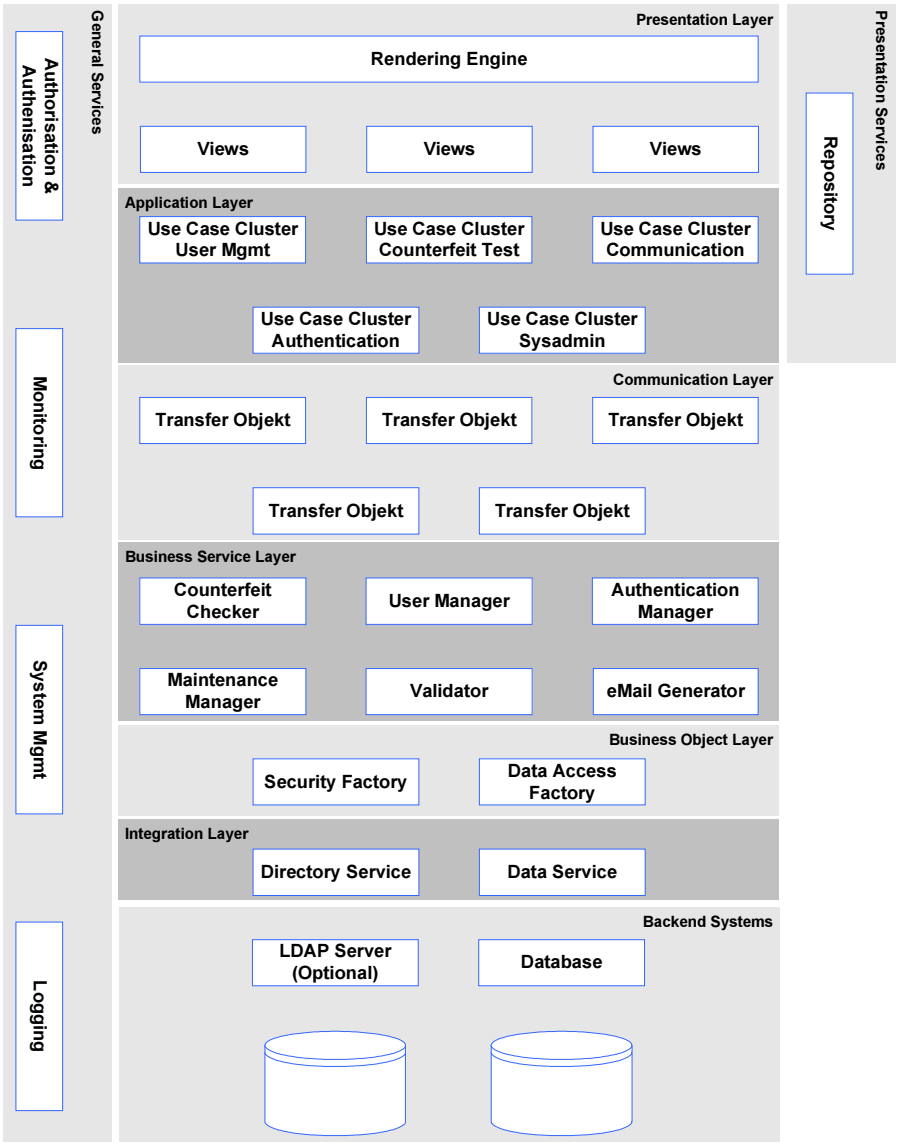


Abbildung 3: Logische Zielarchitektur

Die logische Zielarchitektur besteht aus 7 Schichten:

- **Presentation Layer:** Die Prüfung und die Erfassung der gefälschten Geldscheine erfolgt in einem Web Browser.
- **Application Layer:** Sämtliche Use Cases werden in dieser Schicht umgesetzt. Jeder Use Case wird mittels zwei Objekten realisiert, einem „Control“ und einem „Context“. Das „Control“ Objekt ist verantwortlich für die korrekte Verarbeitung des einzelnen Geschäftsfalles, während das „Context“ Objekt die verschiedenen Möglichen Stati jedes einzelnen Use Cases hält.
- **Communication Layer:** Alle Objekte, die mit dem Backend kommunizieren, sind in dieser Schicht definiert.
- **Business Service Layer:** Diese Schicht enthält sämtliche funktionalen Einheiten der Anwendung. Jede funktionale Einheit repräsentiert eine bestimmte Menge von Systemfunktionen.
- **Business Object Layer:** Der Datenzugriff (Data Access Factory) und die Sicherheitsmechanismen (Security Factory) werden in dieser Schicht realisiert.
- **Integration Layer:** Der Directory Service und der Data Service kapseln die Protokoll-Details der Implementation der beiden Dienste.
- **Backend Systems:** Es werden zwei Backend Systeme benutzt; Eine Oracle Datenbank und ein LDAP Server.

- **General Services:** Diese Dienste warden vom gesamten System benutzt; Authentication & Authorization Service (JAAS), Monitoring Service (Schnittstellen zu Tivoli & HP Open View), System Management Service (restart scripts, database management toolbox und andere) und der Logging Service (Log4J, DB Logging).
- **Presentation Services:** Auf dieser Ebene wird die Mehrsprachigkeit und die Mandantenfähigkeit der Anwendung realisiert.

3.1.4 Lösung: System Architektur

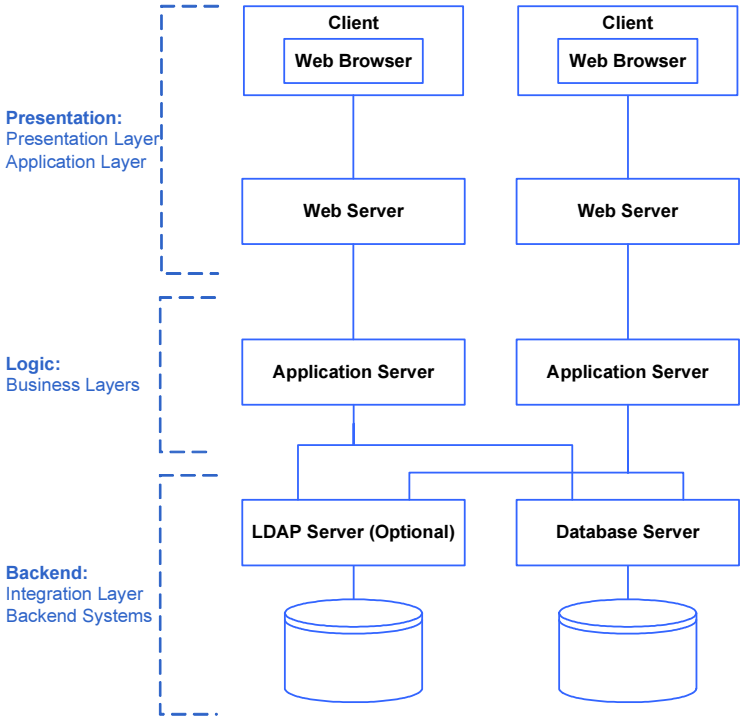


Abbildung 4: Beispiel einer Systemarchitektur

Die Systemarchitektur wird als 3-Tier Lösung umgesetzt:

- Der **Presentation Tier** enthält den Presentation Layer und den Application Layer.
- Der **Business Tier** enthält den Business Service und den Business Object Layer.
- Der **Backend Tier** enthält den Integration Layer und die Backend Systems.

3.1.5 Detail der Lösung: Testing mittels „Inversion of Control“

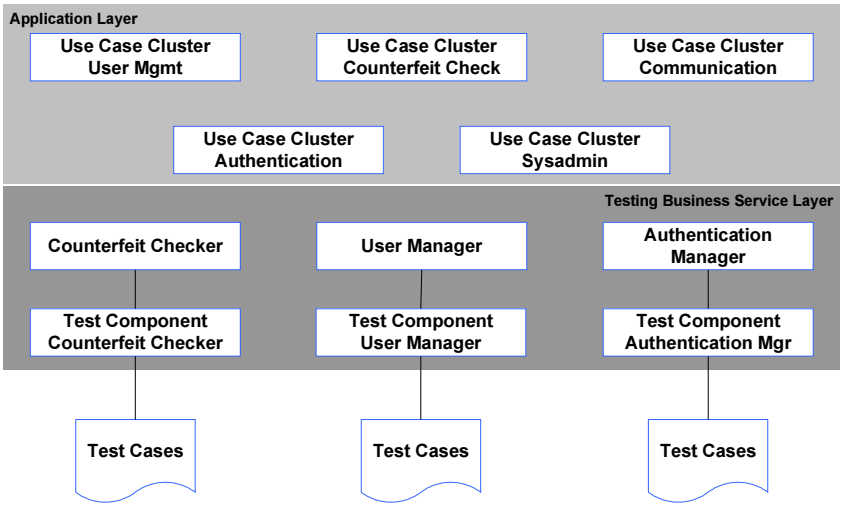


Abbildung 5a: Testen der Backend Services

Um die Realisierung des System zu vereinfachen und die Qualität der Lösung zu verbessern, wird ein „Inversion of Control“ Mechanismus eingesetzt, um Komponenten und Schichten zu teilen. Sämtliche Komponenten des Application Layers können implementiert und getestet werden, ohne dass eine Abhängigkeit zu der Entwicklung der Business Service Layer Komponenten besteht. Dasselbe gilt auch umgekehrt.

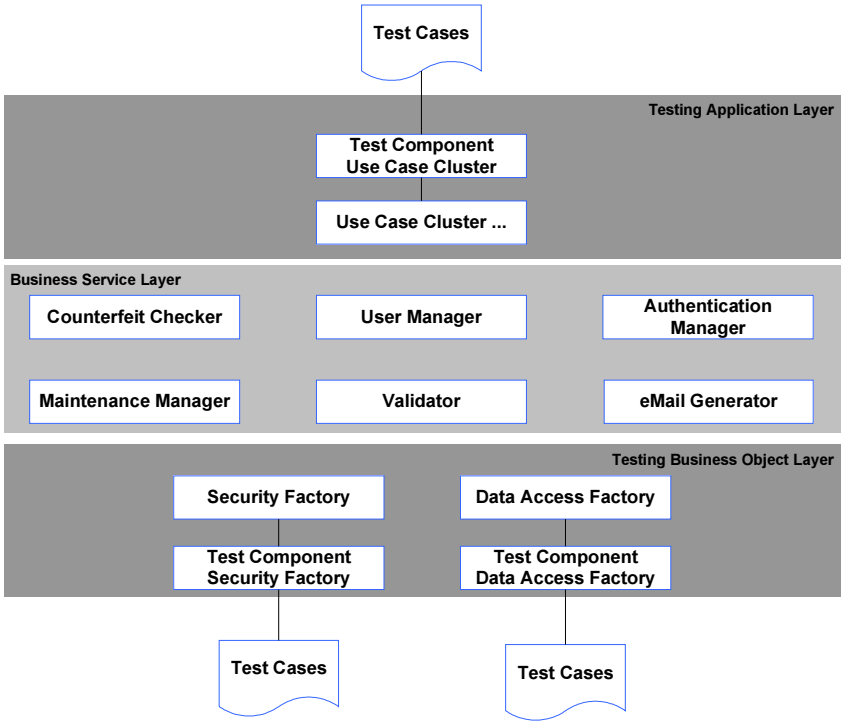


Abbildung 5b: Testen des Business Service Layer

3.1.6 Detail der Lösung: Technische Architektur

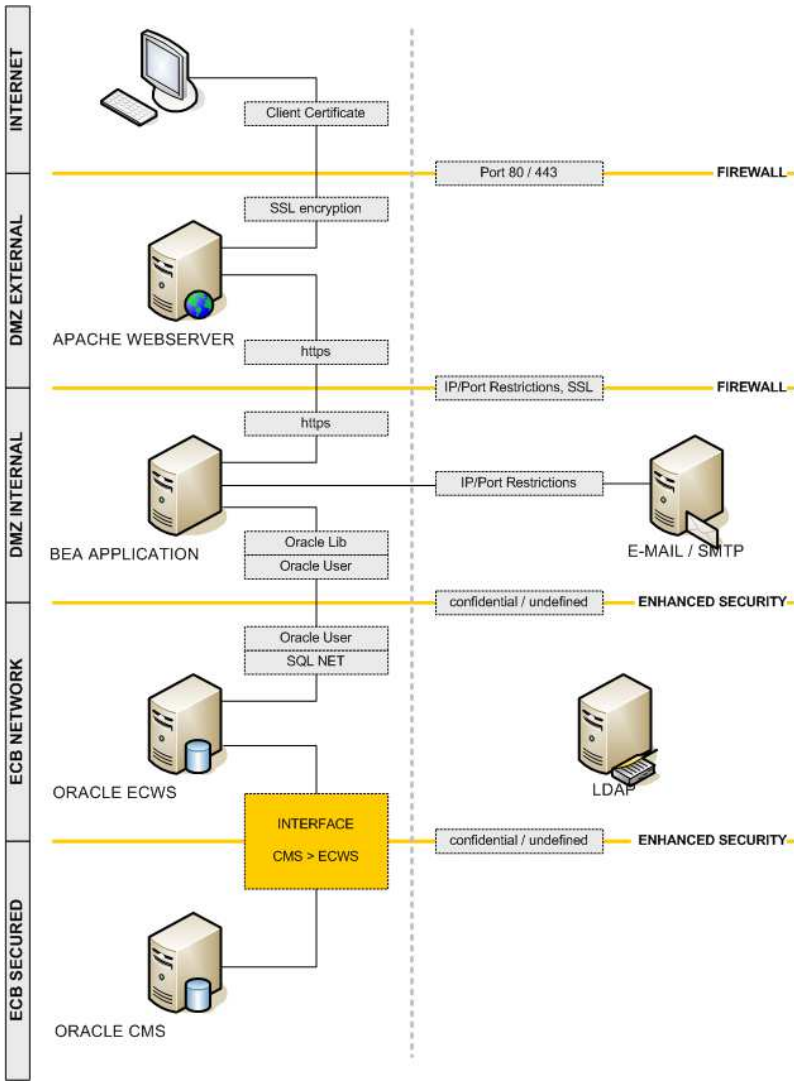


Abbildung 6: Beispiel einer Technischen Architektur

- **Sicherheitszonen:** Die technische Architektur sieht eine strenge Trennung in verschiedene Sicherheitszonen vor.
- **Eingesetzte Produkte:** Oracle Database Version 9.2.05, Sun Solaris Version 8, Bea Weblogic Version 8.1, Apache Server Version 2.0.51, Open LDAP
- **Hardware:** Als Web Server wird eine SUN E280 mit einen Dualprozessor und 2 GB Memory eingesetzt, als Application Server eine SUN E280 mit einen Dualprozessor und 8 GB Memory, als Datenbankserver eine SUN V480 mit 8 GB Memory.
- **Erwartete Anzahl User:** Das System soll maximal 10'000 User bedienen können. Es werden maximal 50 parallele Sessions erwartet.

3.2 Erweiterung einer Produktarchitektur

3.2.1 Ausgangslage

Eine öffentliche Verwaltung setzt als Kommunikationsplattform ein Gesamtsystem ein, welches auf einem elektronischen Arbeitsplatz basiert. Die Kommunikationsplattform dient der verwaltungsinternen Zusammenarbeit von Departementen und Teams. Dabei werden Dokumente, eMails und Termine ausgetauscht. Es sind ca. 30'000 Personen in der Verwaltung beschäftigt.

3.2.2 Anforderungen

Funktion	Bemerkung
Dokumentenablage	Dokumente können in einer geordneten Form abgelegt werden. Mit den vier verschiedenen Behälterobjekten Schrank, Fach, Ordner und Register können Ablagesysteme strukturiert werden. Die Dokumentenablage wird in den Amtstellen sehr unterschiedlich gehandhabt. Einzelne Dienststellen haben kollektive Ablagen für das gesamte Arbeitsspektrum eingerichtet, in anderen stehen individuelle Ablagen im Vordergrund. Führend ist die Papiergebundene Ablage. Geschäftsrelevante Dokumente werden ausgedruckt und in Papierform abgelegt.
Groupware	Sowohl an einzelnen Dateien als auch an Behälterobjekten können beliebige Anwender beteiligt werden. Mittels Verweise auf Objekte kann eine persönliche Sicht auf den interessierenden Dokumentenbestand erstellt werden. Das referenzierte Objekt befindet sich auf einem zentralen Server, wodurch ein leistungsfähigerer Betrieb und eine hohe Sicherheit garantiert werden können. Zugriffsrechte können differenziert direkt durch die Anwender geregelt werden.
Mail	Das Gesamtvolumen des Mailverkehrs beträgt ca. 100’000 Objekte (E-Mails und Verweise) täglich.
Termin- und Kontaktmanagement	Das Planen und Verwalten von Terminen, Fristen, Aufgaben und sonstigen Aktivitäten.
Workflow	Einerseits ist die durchgängige Prozessorientierung in der Verwaltung erst im Aufbau begriffen andererseits ist die Implementierung und Handhabung der integrierten Geschäftsablauffunktionalitäten sehr aufwendig. Mit der Einführung von New Public Management und e-Government erhalten die Ablaufaspekte zukünftig grösseres Gewicht.

3.2.3 Übersicht

Im Rahmen der Realisierung einer Kommunikations-Plattform wurde eine Realisierung basierend auf dem Produkt Oracle Collaboration Suite (OCS) gewählt.
Die Gesamtlösung basiert auf der Oracle Collaboration Suite (OCS). Sie wurde in zwei Versionen realisiert:

- Version 1 Intranet: **Web Client**. Diese Version setzt OCS als Standardprodukt ohne Änderungen ein („out-of-the-box“).
- Version 2 Lokale Adaption: **Lokaler Client**. Der Lokale Client erweitert bestehende Applikationen (MS Outlook, MS Explorer, MS Office, Windows Login) mit OCS-Serverfunktionalitäten zu einem System, welches alle Anforderungen des Pflichtenheftes vollständig erfüllt.

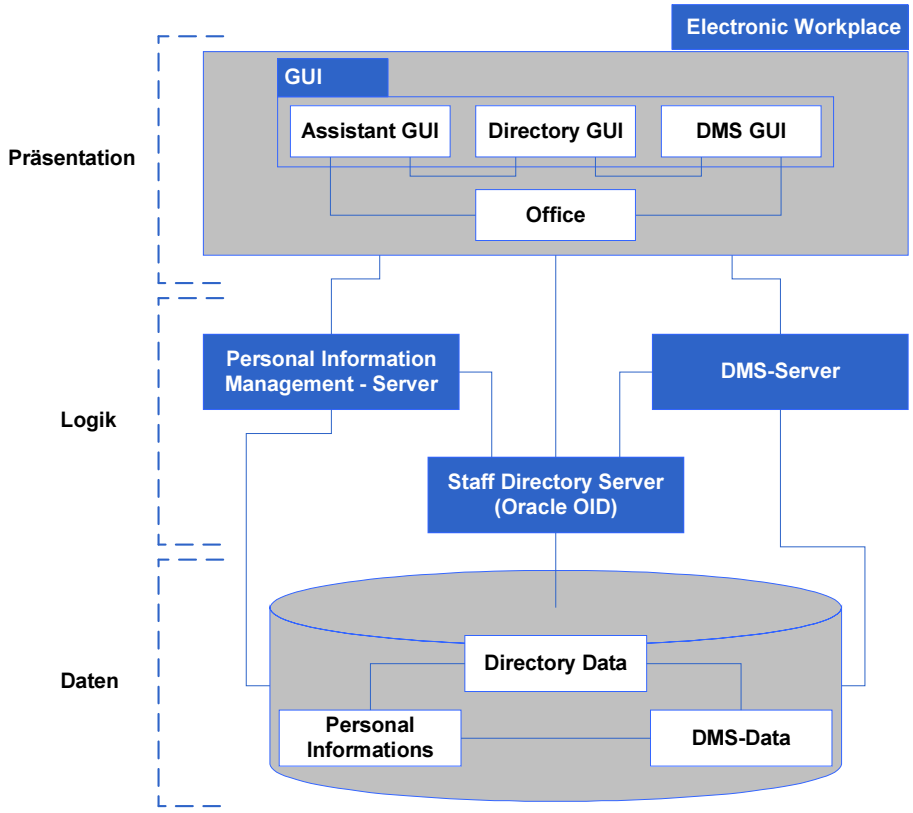


Abbildung 7: Beispiel einer erweiterten Produktarchitektur Übersicht

Die Lösung wird als mehrschichtige Architektur realisiert. Die Grundfunktionalitäten Personal Assistant, Directory und DMS werden zudem getrennt umgesetzt, sie interagieren jedoch und basieren alle auf OCS. Schichten der Software sind:

- **Präsentation:** Die User Interface laufen auf den Clients. Sie sind entweder als lokal installierte Komponenten oder aber als Web Clients realisiert.
- **Logik:** Die Applikationslogik aller drei Systeme (Personal Assistant, Directory und DMS) sind in Applikations-Servern abgebildet.
- **Daten:** Die Daten werden in einer zentralen Datenbank gehalten.

3.2.4 Adaption für das Intranet

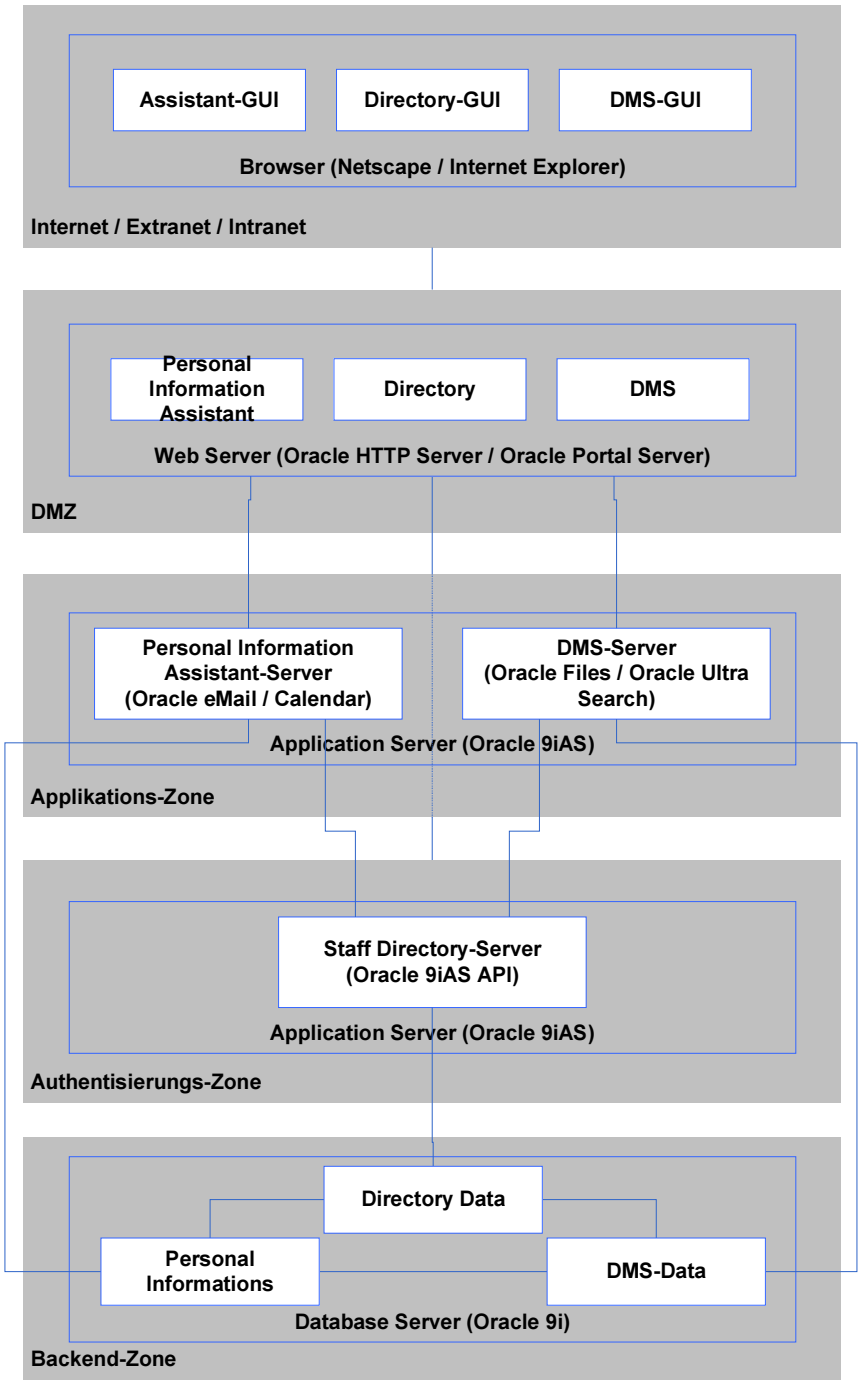


Abbildung 8: Beispiel einer erweiterten Produktarchitektur – Intranet

Der Web Client verwendet Komponenten der Oracle Collaboration Suite für die Umsetzung sämtlicher Funktionalitäten:

- **Personal Assistant** wird mittels **OCS eMail** und **OCS Calendar** abgebildet.
- **Directory** wird mittels **OCS SSO (Single Sign On)** und **OCS OID** abgebildet.
- **DMS** wird mittels **OCS Files** und **OCS Ultra Search** abgebildet.

3.2.5 Adaption an lokale Realisierung

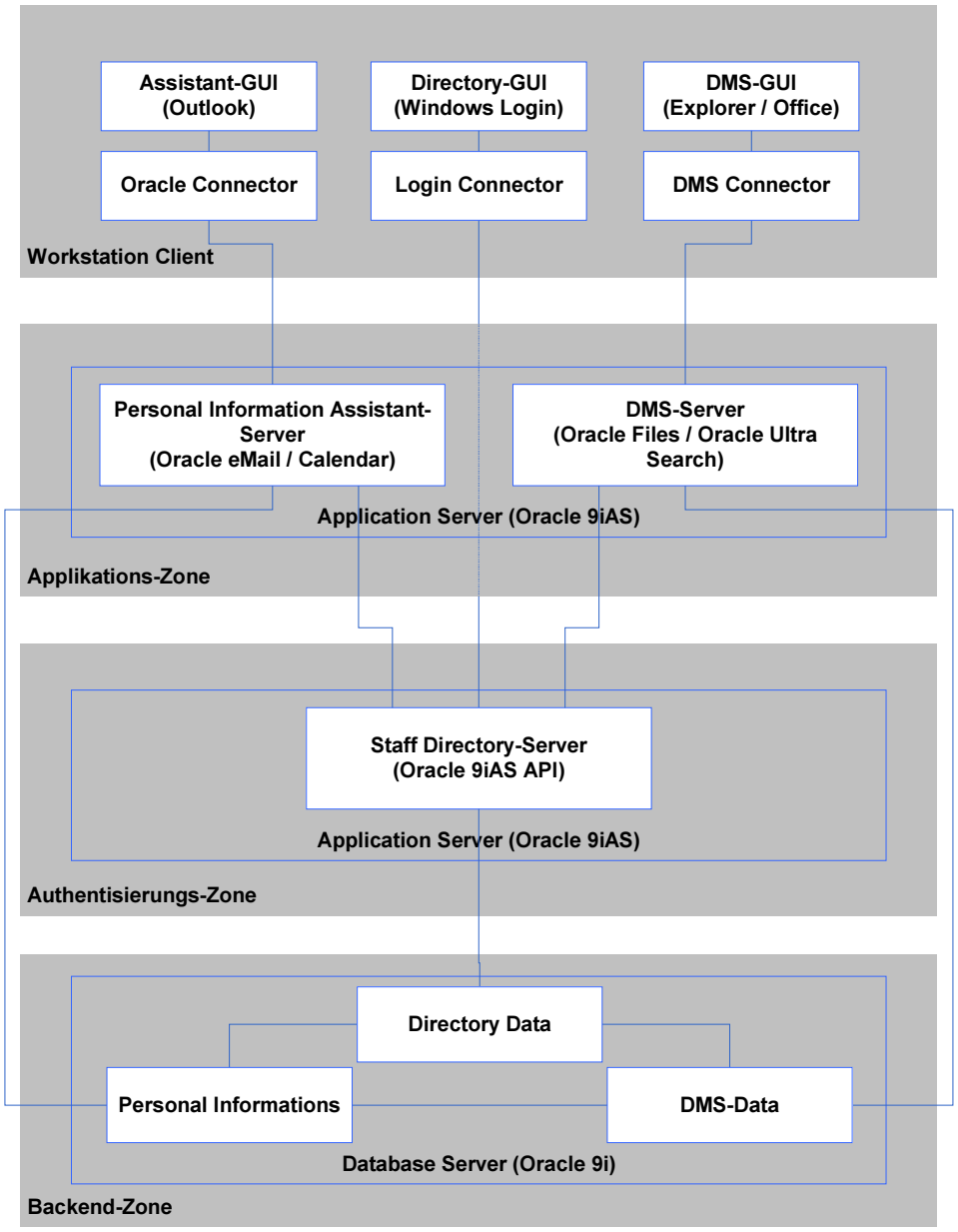


Abbildung 9: Beispiel einer erweiterten Produktarchitektur – Lokale Realisierung

Der Lokale Client verwendet lokal installierte Komponenten für die Umsetzung sämtlicher Funktionalitäten:

- **Personal Assistant** wird mittels **MS Outlook** abgebildet.
- **Directory** wird mittels des Standard **Windows Login** abgebildet.
- **DMS** wird mittels **Explorer** und **Office** abgebildet.

Um die Funktionalität der Produkte zu erweitern werden so genannte Connectoren eingesetzt. Während der Outlook Connector Bestandteil von OCS ist, sind der Login Connector und der DMS Connector im Rahmen des Projektes zu realisieren.

Architektur Style

4.1 Einleitung

Architektur Stile beschreiben eine Familie von Software-Systemen, die aufgrund ihrer Struktur und ihrer Semantik verwandt sind. Sie sind jedoch auch im Kontext einer bestimmten Zeit zu verstehen. In der groben Aufteilung Batch-Processing (60er Jahre), Call and Return (70er Jahre), Structured Decomposition (80er Jahre), Object Decomposition (90er Jahre) und Service Oriented Architecture (heute) sind verschiedene Architektur Stile zu finden. Versuche, Architektur Stile zu definieren und zu klassifizieren, fanden jedoch erst Ende der 90er Jahre statt.

4.2 Definitionen

Die vier Wissenschaftler Robert T. Monroe, Andrew Kompanek, Ralph Melton und Professor David Garlan haben 1997 versucht, architektonische Stile, Objekt Orientiertes Design und Design Patterns zuzuordnen [Monroe et al. 1997]. Zentrale Fragestellung für die Wissenschaftler der Carnegie Mellon University war, wie können bereits gemachte Erfahrungen so aufbereitet werden, dass sie zu besserem Software Design führen. Ihrer Ansicht nach kann ein Systemverhalten durch Architektur Stilelemente wesentlich einfacher beschrieben werden als beispielsweise durch Patterns.

Ein architektonischer Stil charakterisiert eine Klasse / Familie von Systemen und stellt dem Architekten die vier Bestandteile Vocabulary, Rules and Constraints, Semantics und Analytics zur Beschreibung eines bestimmten Systems zur Verfügung.

- **Vocabulary:** Ein Vokabular von Design-Elementen bestehend aus Komponenten und Verbindungselementen. Komponenten können beispielsweise Server, Parser, Datenbanken und Filter sein. Verbindungselemente können beispielsweise RPC (Remote Procedure Call), Data Streams oder Sockets sein.
- **Rules and Constraints:** Ein Architektur Stil unterliegt bestimmten Design-Regeln und Design-Einschränkungen, die definieren welche Elemente wie zusammengestellt werden können. Diese Regeln vermeiden grundlegende Fehler in einem System Design.
- **Semantics:** Die Bedeutung der verschiedenen Design-Elemente des Stils ist eindeutig. Somit können semantische Regeln zur Interpretation einer Komposition von Elementen herangezogen werden.
- **Analytics:** Analytische Instrument zur Prüfung des Vokabulars, der Regeln und Einschränkungen eines bestimmten Architektur Stils, respektive zur Analyse des konkret realisierten Systems. Mit diesen Instrumenten können Kriterien wie die „Schedulability“ eines Real-Time Systems oder auch „Deadlock Prevention“ eines Message-Passing Systems

Dewayne E. Perry, AT&T Bell Laboratories, und Alexander L. Wolf, Boulder University definieren in ihrem Artikel „Foundation fort he Study of Software Architecture“ den Stil einer Architektur als Rahmen für die Realisierung einer bestimmten Architektur [Perry, Wolf 1992]:

Falls Architektur ein formales Arrangement von architektonischen Elementen ist, so wird eine architektonischer Stil dadurch definiert, dass er Elemente und formale Aspekte verschiedener spezifischer Ar-

chitekturen abstrahiert. Ein architektonischer Stil ist weniger eingeschränkt und weniger vollständig als eine definierte Architektur.

4.3 Übersicht Architektur Stile

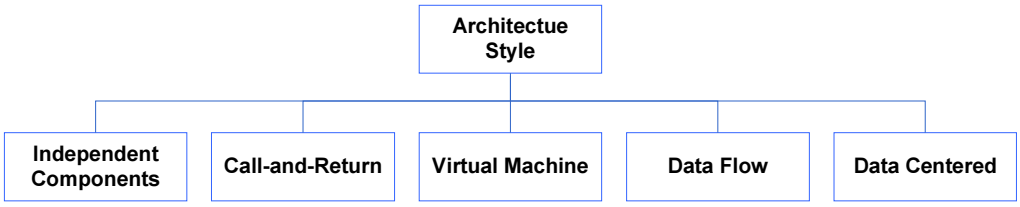


Abbildung 11: Architektur Stile gemäss Shaw und Garlan [Shaw, Garlan 1996].

- **Independent Components:** Eine Architektur unabhängiger Komponenten besteht aus unabhängig ablaufenden Elementen, die über Nachrichten miteinander interagieren.
- **Call-and-Return:** Call-and-Return Architekturen sind durch einen fixen Kommunikations-Mechanismus zwischen aufrufendem Element und aufgerufenen Element definiert.
- **Virtual Machine:** Eine Architektur, die auf Virtuelle Maschinen basiert, erlaubt die Realisierung portabler und interpretierbarer Systeme.
- **Data Flow:** Eine Datenfluss Architektur definiert ein System als Abfolge von Datenbezogenen Transformationen.
- **Data Centered:** Eine Datenzentrierte Architektur beschreibt Systeme, deren wesentliche Aufgabe der Zugriff und die Aktualisierung von Daten eines Repositories ist.

4.3.1 Vorteile und Nachteile

Architektur Stil	Anwendung	Vorteil	Nachteil
Independent Components			
Communicating Processes	Parallelverarbeitung	Einfache Modellierung, Skalierbarkeit	Komplexität der einzelnen Elemente
Event Systems	GUI's, Real Time Systems	Unabhängigkeit der Elemente, Änderungs-Freundlichkeit	Non-Deterministisches Verhalten der Elemente
Call-and-Return			
Main Program & Subroutine	Structured Programming, Client-Server (RPC)	Definierter Kontrollfluss	Skalierbarkeit, Erweiterbarkeit
Object Oriented	Allgemeines Design, Client-Server	Universell	Komplexität, Anwendungsfreiheit
Layered	SOA, Multi-Tier Architectures	Konzeptionelle Integrität, Lokalität der Änderungen	Performance, Komplexität
Virtual Machine			
Interpreter	Prozessor- und Betriebssystem-Simulation	Portabilität, Flexibilität	Performance
Rule-Based Systems	Expertensysteme	Flexibilität durch Regelwerk	Komplexität, Performance
Data Flow			
Batch Sequential	Host Systeme	Datensteuerung	Flexibilität, Interaktion
Pipes and Filters	Software Converter, Compiler	Flexibilität, Verteilung	Komplexität
Data Centered			
Repository	Stammdatenverwaltungen	Einfach	Single Point of Failure
Blackboard	Datengesteuerte Kontrollsysteme	Skalierbar	Anwendung eingeschränkt

4.4 Independent Components

Eine Architektur unabhängiger Komponenten besteht aus unabhängig ablaufenden Elementen (Independent Components – im Normalfall sind das Prozesse), die über Nachrichten miteinander interagieren. Diese Nachrichten können entweder deterministisch, als in gesteuerter Abfolge des Programmablaufs, oder nicht-deterministisch über Ereignisse ausgetauscht werden. Die meisten Architekturen, die unter dem Begriff Distributed Systems zusammengefasst werden, basieren auf diesem Architektur Stil.

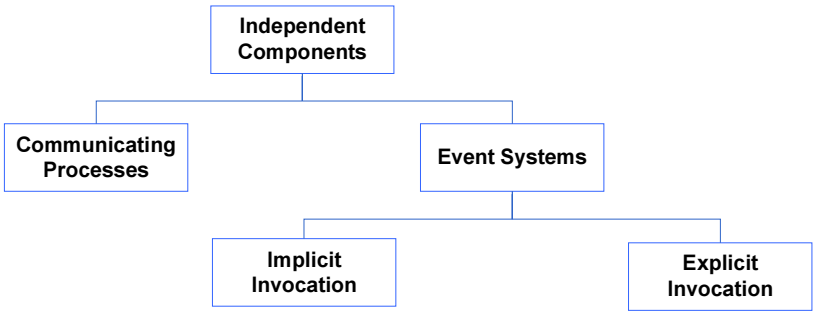


Abbildung 12: Independet Components Architektur Stil.

- **Communicating Processes:** Dieser Architektur Stil beschreibt Architekturen, die aus einer beliebigen Anzahl miteinander kommunizierender Elemente bestehen. Die Synchronisation erfolgt ausschliesslich über die Kommunikationskanäle zwischen den Elementen und kann sowohl synchron als auch asynchron erfolgen.
- **Event Systems:** Dieser Ereignisgesteuerte Architektur Stil definiert Architekturen, deren Prozesse über Events miteinander kommunizieren.
- **Implicit Invocation:** Sämtliche Events werden an einen so genannten Event Space gesendet, der dann die Verteilung der Events an die betroffenen Prozesse übernimmt.
- **Explicit Invocation:** Die Events werden direkt zwischen den einzelnen Prozessen ausgetauscht.

4.4.1 Communicating Processes

Es existieren eine Vielzahl von Architekturen, die auf dem Architektur Stil Communicating Processes beruhen. Viele Rechenmodelle und Systeme vor allem für High End Computing werden basierend auf diesen Architekturen realisiert. Ein im Bereich Parallel Computing sehr verbreitetes Paradigma zur Beschreibung von Communicating Processes ist CSP (Communicating Sequential Processes) von Tony Hoare, der lange Zeit als Professor in Oxford arbeitete und heute als Senior Researcher bei Microsoft tätig ist [Hoare 1978]. Sein gleichnamiges Buch gehört zu den am meisten zitierten Werken der Informatik. Er beweist darin, dass CSP zur Modellierung von Communicating Processes verwendet werden kann. Die wichtigsten Elemente sind ein Prozess und eine Kommunikationskanal. Damit lassen sich dynamische Abläufe definieren, die sich in jedem Fall deterministisch verhalten. Um den Architektur Stil Communicating Processes zu illustrieren, ist CSP ein geeignetes Mittel, auch wenn andere Darstellungsmöglichkeiten denselben Stil umsetzten könnten.

4.4.2 Beispiel: Eine parallele Lösung des Springerproblems

Das Problem

Ein Springer soll nacheinander alle Felder eines Schachbrettes beliebiger Breite (resp. Länge) unter Einhalten der Schachregeln besuchen ohne auch nur ein einziges Feld zweimal zu betreten. Diese Aufgabenstellung wurde vom Basler Mathematiker Leonhard Euler vor über 200 Jahren in seinem Buch "Memoires de Berlin" zum ersten Mal beschrieben. Eine Überprüfung aller möglichen Züge, die so genannte erschöpfende Suche, ist aufgrund der Anzahl Möglichkeiten (Dimension: n*n ergibt n! Möglichkeiten) zu aufwendig.

Backtracking

Ausgangspunkt ist das Backtrackingverfahren, welches durch Probieren die möglichen Lösungen aus den bereits gefundenen Teillösungen zusammensetzt. Im Fall des Springer Problems probiert das Backtracking Schritt für Schritt ausgehend von einem bestimmten Schachbrettfeld alle legalen Züge bis es auf eine Sackgasse oder auf eine Lösung trifft. Dabei wird der Suchbaum aller möglichen Züge durchlaufen ohne eine Variante zweimal durchzurechnen.

Hintergrund
Das Springerproblem kann als klassisches Weg-Problem in einem Graphen angesehen werden. So stellen die Knoten des Graphen die Felder des Brettes und die Kanten die Züge des Springers dar. Für solche Probleme gibt es verschiedene grundsätzliche Lösungsstrategien, wie zum Beispiel die erschöpfende Suche (testen aller Möglichkeiten) oder das obenbeschriebene Backtracking. Daneben existieren noch weitere Lösungsansätze, zum Beispiel die bewertende Suche und das Zerlegungsverfahren. Die bewertende Suche, ein heuristisches Verfahren, welches bereits 1823 von H.C. Warnsdorff entwickelt wurde, wählt aus den möglichen Zügen eines Springers die günstigste, das heisst die Erfolgversprechendste aus. Bewertungskriterien sind beispielsweise die momentane Position auf dem Brett oder die minimale Anzahl Nachbarn. Das Zerlegungsverfahren unterteilt ein Schachbrett in gleich grosse Teile. Die Teillösung wird mit dem Backtrackverfahren berechnet. Durch Drehen und Spiegeln wird diese Lösung auf das gesamte Brett übertragen.
Parallelisierung
Die Umsetzung der parallelisierten Lösung aus der sequentiellen erfolgt über eine Veränderung der Abbruchbedingung des Backtrack Algorithmus, sodass von einem beliebigen Niveau im Suchbaum aus weitergerechnet werden kann bis alle Lösungen des Teilbaumes gefunden wurden.

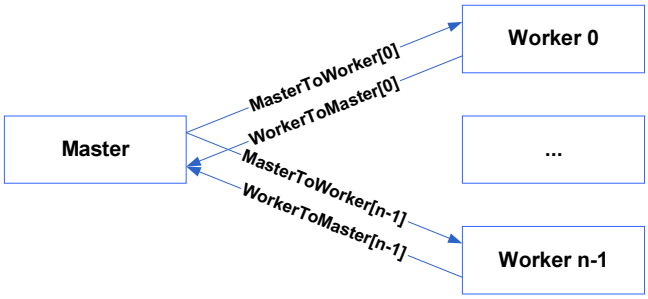


Abbildung 13: Prozessverteilung

Ein Master Prozess bedient einen bis eine beliebige Anzahl Worker Prozesse mit Aufgaben. Die Arbeitsteilung und Organisation erfolgt über ein Hin- und Herschicken von Teilbäumen zwischen den arbeitenden Prozessen (Workers). Die Worker sind dabei als Farm organisiert. Der Austausch der Teillösungen sowie die Darstellung der gefundenen Lösungen übernimmt der Master Prozess.

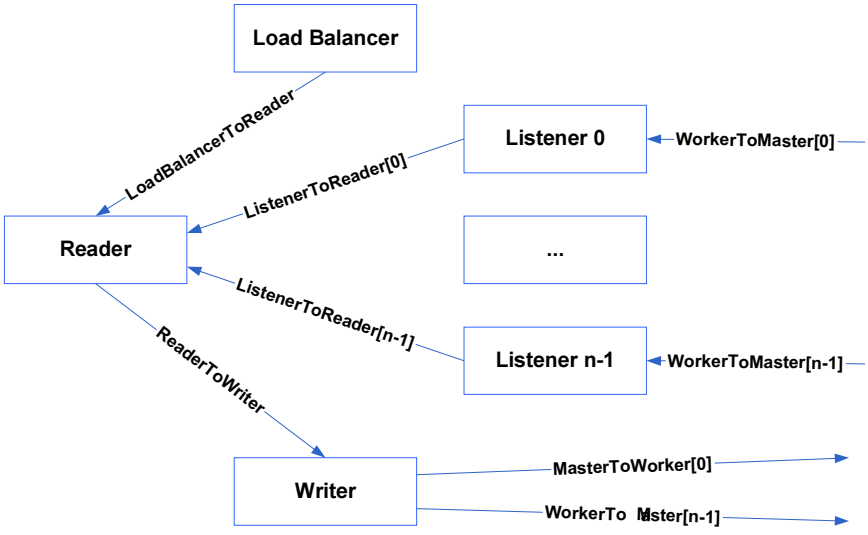


Abbildung 14: Der Master Prozess

Der Master Prozess verteilt die Arbeit über Pipes an die Worker, deren Anzahl variabel zur Ausführungszeit bestimmt werden kann. Sämtliche Ein- und Ausgaben erfolgen über ihn. Er leitet ausserdem unter Verwendung eines Reader Prozesses die von den Worker kommenden Teilaufgaben an die freien Worker weiter und behält die Übersicht über diese. Er teilt den arbeitenden Workern mit, wenn ein anderer frei ist, sodass der Arbeitende eine Teilaufgabe abspalten und an den Master weitergeben kann. Sämtliche Meldungen nimmt der Master über die Listener Prozesse

entgegen, welche diese Informationen an den Reader weitergeben. Der wiederum sortiert die Lösungen aus und zeigt sie an. Die Teilaufgaben sowie die Informationen, ob ein Worker frei ist, gibt er an den Writer weiter.

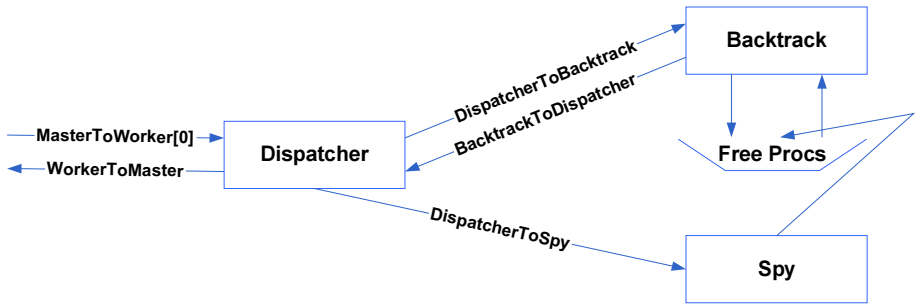


Abbildung 15: Der Worker Prozess

Der Worker sucht mit dem Backtrackverfahren den Baum der möglichen Sprünge nach Lösungen ab und schickt diese an den Master zurück. Er überprüft ausserdem über den Spy Prozess, ob ein anderer Worker frei ist, sodass ein Teilbaum abgespalten und an den Master gesendet werden kann. Der Dispatcher liest und schreibt sämtliche Meldungen auf die Pipes.

4.4.3 Event Systems

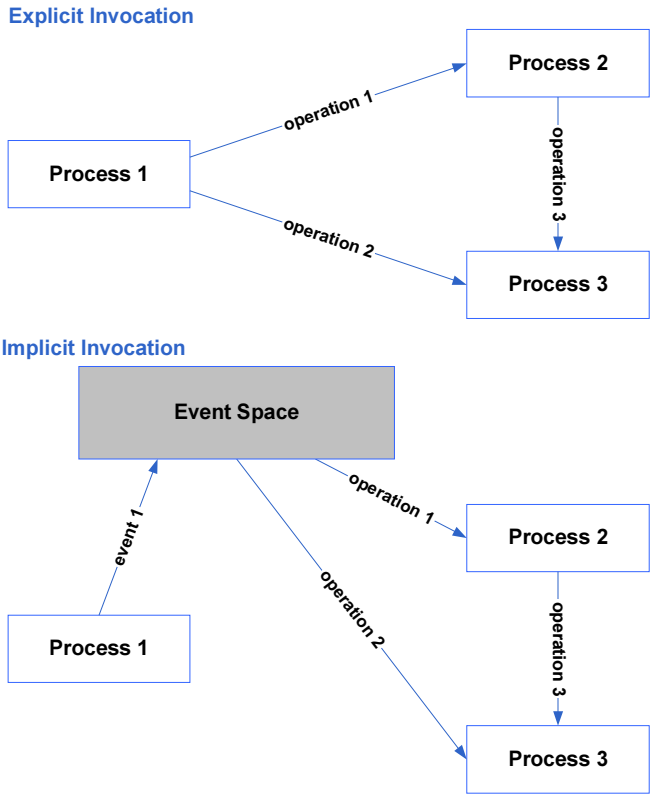


Abbildung 16: Expliziter und Impliziter Aufruf von Events

Ereignisgesteuerte Systeme bestehen aus zwei Arten von Elementen; Publisher geben Informationen bekannt, Subscriber interessieren sich für bestimmte Informationen. Die Elemente kennen sich nicht gegenseitig. Das Event System selbst erlaubt die Kommunikation zwischen den einzelnen Komponenten, in dem es allen interessierten Elementen die Nachricht weiterleitet. Die wichtigsten Eigenschaften von Event Systemen sind die Trennung der Art und Weise der Interaktion von den Interagierenden Elementen, die Auflösung statischer Abhängigkeiten durch beschränkte Namensräume sowie die mögliche Parallelisierung. Das klassische Beispiel eines Event Systems ist ein Message Broker, wie er in verschiedenen EAI Plattformen eingesetzt wird.

4.4.4 Beispiel: MVC (Model View Controller) als Event System

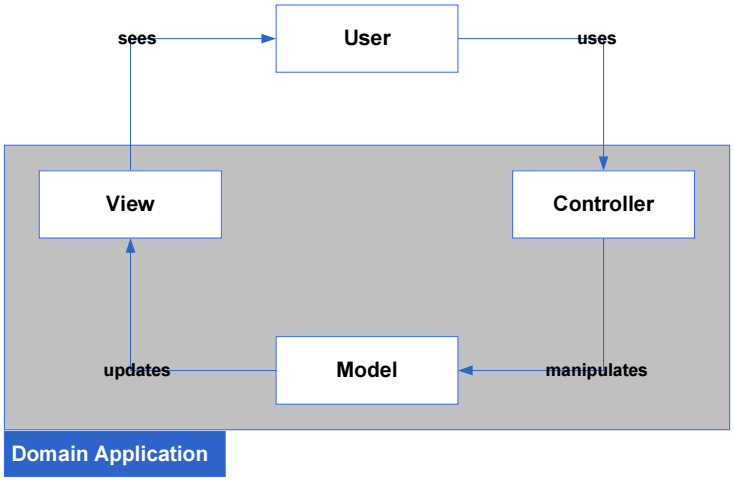


Abbildung 17: MVC

Model View Controller ist die heute übliche Art und Weise, wie moderne GUI's mit einem System interagieren. Als Event System sind einerseits die Events, die vom Controller (Mouse Move, Mouse Over, etc) erzeugt werden. Auf diese Events reagiert eines oder mehrere Model Komponenten, die wiederum alle Views (dargestellte User Interface Komponenten) nachführen. Das Modell wurde 1988 für Smalltalk-80 von G.E. Krasner und S.T. Pope beschrieben, ist jedoch heute aus keinem Software Design Pattern Buch mehr wegzudenken [Krasner, Pope 1988].

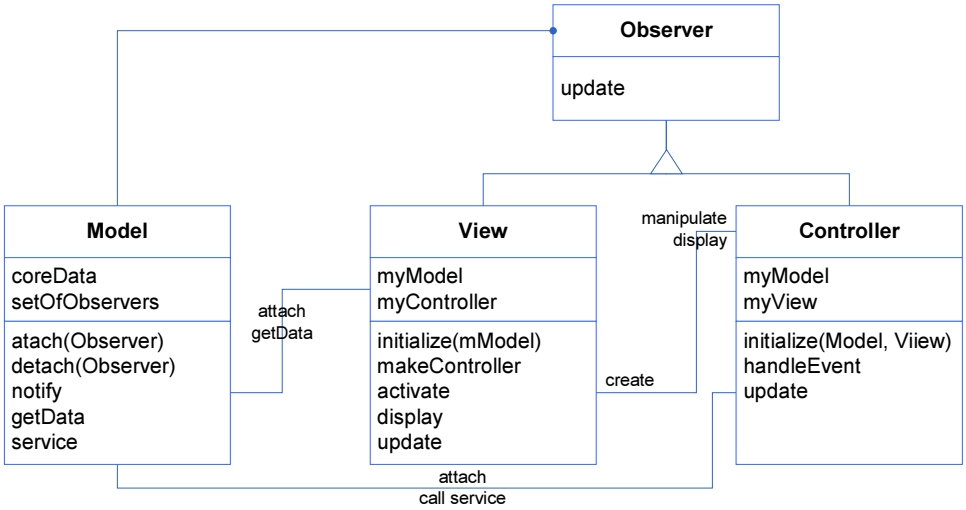


Abbildung 18: MVC als Objekt Orientierte Implementation [Buschmann et al. 1996].

- **Model:** Das Herzstück der Applikation. Es enthält die Daten und die Zustände, die ein System zu einem bestimmten Zeitpunkt darstellen. Sobald Änderungen am Model erfolgen, werden sämtliche Views, die ein Model ganz oder teilweise darstellen nachgeführt.
- **View:** Das User Interface, welches die Informationen über ein Model für die Benützer / Benützerinnen der Applikation darstellt.
- **Controller:** Die Steuerung der Interaktion durch den User erfolgt durch den Controller. Der Controller akzeptiert User Inputs als Events

4.5 Call-and-Return

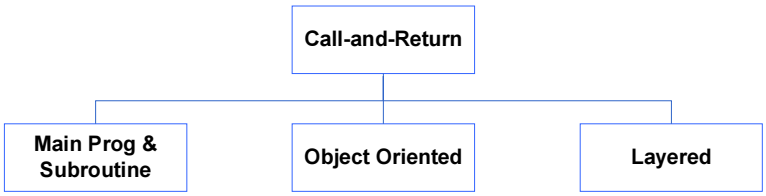


Abbildung 19: Call-and-Return Architektur Stile

Der Call-and-Return Architektur Stil ist durch einen fixen Kommunikations-Mechanismus zwischen aufrufendem Element und aufgerufenen Element definiert.

- **Main Programm & Subroutine:** Dieser Architektur Stil repräsentiert das klassische Programmierparadigma der SASD (Structured Analysis - Structured Design) Welt [Demarco, Plauger, 1979], [Yourdon 1997]. Ein eindeutiger Kontrollfluss durchläuft einen Baum bestehend aus Hauptprogrammen und Unterprogrammen, wobei die Kontrolle dem gerade ausführenden Teil des Systems übertragen wird.
- **Object Oriented:** Der Object Oriented Call-and-Return Stil folgt den Prinzipien des Main Programm & Subroutine Stils, also der Organisation des Kontrollflusses über die jeweils aufgerufenen Methode. Die Zugriffe auf Objekte erfolgt über Schnittstellen. Fokus des Stils ist das Information Hiding, die Kapselung von Daten.
- **Layered:** Sämtliche Elemente dieses Architektur Stils werden einzelnen Schichten zugeordnet, die hierarchisch geordnet sind. Jede dieser Schichten erfüllt eine klar definierte Aufgabe. Die Interaktion zwischen einzelnen Schichten erfolgt über definierte Protokolle.

4.5.1 Main Program & Subroutine / Object Oriented Call & Return

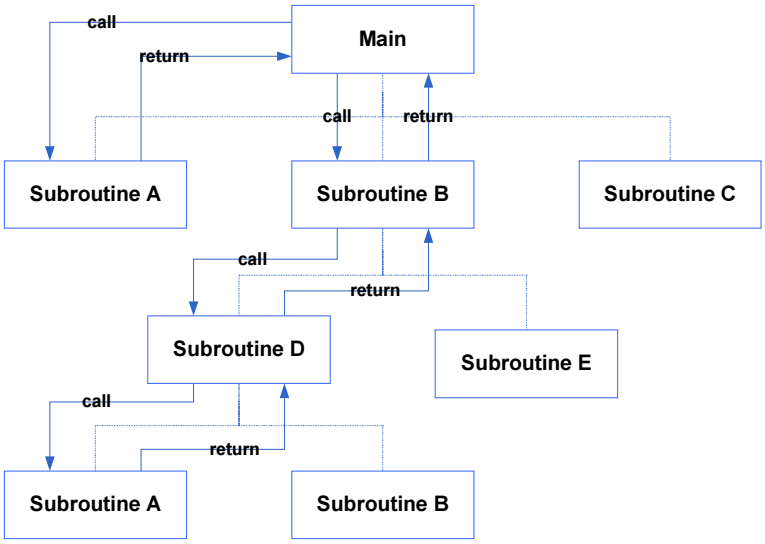


Abbildung 20: Main Program and Subroutine

Zentrale Eigenschaft des Main Program & Subroutine respektive des Object Oriented Call-and-Return Stils ist der eindeutige Kontrollfluss eines Programms. Der Ablauf eines Programms und damit die Dynamik des Gesamtsystems wird damit deterministisch. Sämtliche strukturiert aufgebauten System sowie die meisten Client-Server Systeme basieren auf diesen Architektur Stil.
In den meisten Fällen ist die Kommunikation zwischen verschiedenen Elementen dieses Architektur Stil synchron. Es existiert jedoch ein Pattern für einen Asynchronen Call-and-Return Mechanismus, der über einen Callback funktioniert. Dabei wird beim Aufruf einer Subroutine oder einer Methode ein Function Pointer oder eine Instanz auf einen Callback-Handler mitgegeben. Der wird von der Subroutine aufgerufen, wenn die Arbeit abgeschlossen ist. Dieses Pattern wird im Rahmen des Microsoft Research Projects für die Sprache Cω beschrieben [Meijer, Schulte 2003]

4.5.2 Beispiel: RPC (Remote-Procedure-Call)

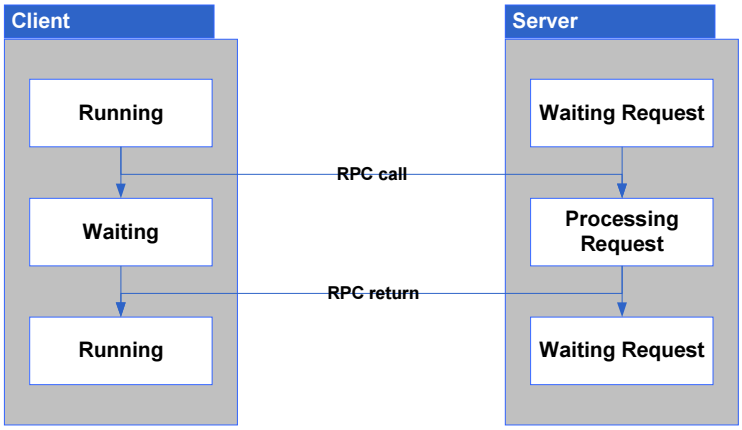


Abbildung 21: Remote Procedure Call

Ein typischer Vertreter des Main Program & Subroutine ist der Remote-Procedure-Call (RPC), der in vielen Systemen eingesetzt wird. Der Remote Procedure Call (RPC) ist ein Protokoll, das die Implementierung verteilter Anwendungen vereinfacht. Ein Programm kann eine Funktion eines Programms, das auf einem anderen Rechner läuft, nutzen, ohne sich um die zu Grunde liegenden Netzwerkdetails kümmern zu müssen. RPC arbeitet nach dem Client-Server-Modell. Ein RPC Service wird durch die Programmnummer, Versionsnummer und das Transportprotokoll eindeutig beschrieben. Die meisten heute eingesetzten RPC Dienste arbeiten mit einem Port Mapper, damit ein Dienst keine fixe Portnummer vergeben muss. RPC ist als RFC 1831 definiert, der den so genannten ONC (Open Network Computing) RPC beschreibt [Srinivasan 1995].

4.5.3 Layered Architecture Style

Geschichtete Architekturen sind heute in allen modernen Architekturen zu finden. Die typische Schichtung ist Graphical User Interface – Presentation Layer – Business Functionality – Data Access Layer – Data Layer. Der Begriff jedoch wurde in der Telekommunikation durch das OSI-Modell etabliert [ISO 7498-1 1994]. Alle Schichtenmodelle sind hierarchisch gegliedert und teilen die Aufgaben eines Gesamtsystems in verschieden Schichten auf. Damit wird auch das Spektrum der Elemente, die einer bestimmten Schicht zugeordnet werden können, eingeschränkt. Eine besondere Stärke dieses Architektur Stils ist die Lokalität der Änderungen. Änderungen betreffen maximal die geänderte Schicht sowie die untenliegende und die oben liegende Schicht. Falls an den Schnittstellen zwischen den Schichten nichts geändert wird, so ist sogar lediglich eine Schicht betroffen.

4.5.4 Beispiel: 3-Tier Architecture for a Risk Management System

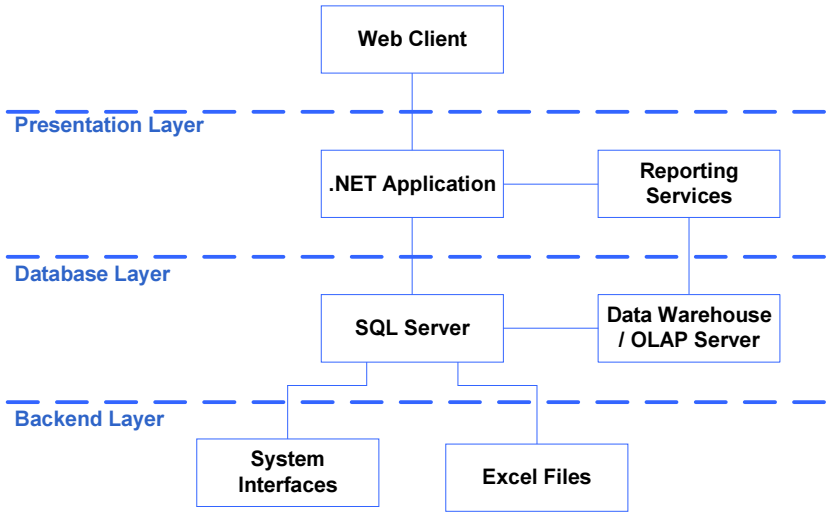


Abbildung 22: Risk Management System

Ein Risk Management System soll operative und geschäftliche Risiken, die bei der Produktion von Konsumgütern entstehen können, isolieren und bewerten. Zu diesem Zweck werden in einem Konzern die weltweiten Produktions- und Vertriebsdaten konsolidiert und von diesen Systemen in vor verdichteter Form als Excel Files oder in Form von Host Files bereitgestellt. Sämtliche Daten werden in einem SQL Server geladen und gehalten. Anschliessend werden sie in einem Data Warehouse konsolidiert und mittels OLAP Server aufbereitet. Die Risiken können nach verschiedensten Kriterien interaktiv über einen Risk Reporting Service ausgewertet und als Reports abgelegt werden. Jeder Layer des Systems entspricht auch einem Tier.

- Der **Presentation Layer** enthält die .NET Applikation sowie die Reporting Service Komponente zur Definition, Abfrage und Anzeige und Aufbereitung von Reports.
- Der **Database Layer** besteht aus einem MS SQL Server, der die operativen Daten aus dem Backend Layer täglich bezieht und dem Data Warehouse, welches die täglichen Daten konsolidiert und als OLAP-Qubes aufbereitet.
- Der **Backend Layer** enthält die Daten, die von den operativen System als Excel Files oder aber als Host Files bereitgestellt werden.

4.6 Virtual Machine

Abbildung 23: Virtual Machine Architecture Style

Eine virtuelle Maschine ist ein hypothetischer Computer. Dies ist jedoch eine generalisierte Sichtweise auf den Begriff. Als Architektur Stil eingesetzt, werden damit Architekturen für Interpreter, die abstrakte Maschinen umsetzten, und für Interpreter, die Betriebssysteme simulieren, verwendet. Darüber hinaus werden Regel-Basierte Systeme, deren Kern eine Interference Engine darstellt, mit Architekturen realisiert, die auf diesem Architektur Stil basieren. Gemeinsam ist diesen Systemen, dass sie unabhängig von der konkreten Hardware oder vom konkreten Betriebssystem Funktionalität simulieren, die einer bestimmten Spezifikation einer abstrakten Maschine folgen. Diese abstrakte Maschine kann ein Prozessor, ein Betriebssystem oder eine durch Regeln definierte generalisierte Maschine sein.

- **Interpreter:** Dieser Architektur Stil beschreibt eine abstrakte Maschine, und definierte Elemente für deren inneren Aufbau.
- **Rule-Based System:** Dieser Stil wird für die Architektur von Regelbasierten Systemen verwendet, die eine Generalisierung der Interpreter darstellen. Zentrale Eigenschaft dieser Systeme ist die Trennung zwischen der Maschine an sich und den Regeln, die die Maschine beschreiben.

4.6.1 Interpreter

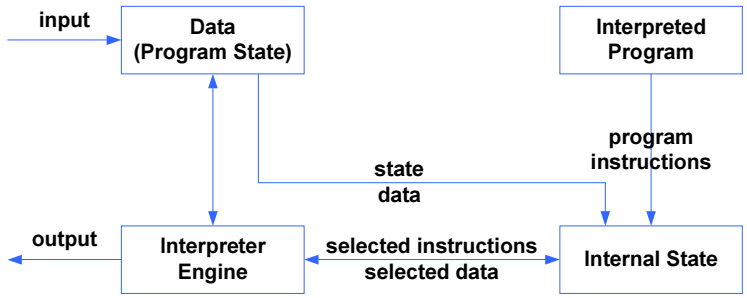


Abbildung 24: Aufbau eines Interpreters

Auf Basis des Interpreter Architektur Stil werden Architekturen für abstrakte Maschinen spezifiziert. Interpreter arbeiten mit einem definierten Aufbau und bilden in dem meisten Fällen einen statischen virtuellen Rechner ab, dessen Rechenregeln nicht verändert werden können. Die Interpreter Engine arbeitet die Anweisungen des zu interpretierenden Programms schrittweise ab. Dabei wird abhängig von der einzelnen Anweisung, die den internen Zustand des Interpreters definieren, der Zustand der Daten nachgeführt. Ein interpretiertes Programm kann angehalten werden und ggf. können zur Laufzeit Anweisungen und Daten geändert werden. Diese Flexibilität wirkt sich nachteilig auf die Performance eines solchen Systems aus.

4.6.2 Beispiel: Innerer Aufbau der Java VM

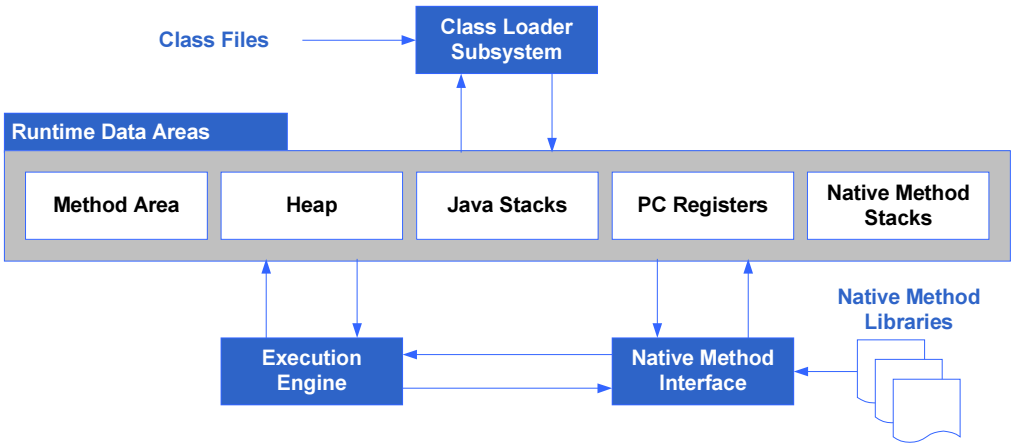


Abbildung 25: Java VM [Venners 2000]

Neben dem Einsatz von Interpretern für die Simulation von Embedded Systems ist die Java VM heute die am meisten eingesetzte virtuelle Maschine überhaupt. Die Java VM simuliert einen einfachen Prozessor, der als kleinsten gemeinsamen Nenner aller GPU's (General Processing Unit) angesehen werden kann.

- **Class Loader Subsystem:** Diese Komponente lädt Typen (Classes und Interfaces) abhängig von ihrem Full Qualified Name (Absoluter Namen). Ausserdem prüft es die Korrektheit aller importierten Klassen. Dabei werden immer die drei Schritte "Loading" (auffinden und importieren der binären Daten eines Typs), "Linking" (Prüfung der importierten Typen und optionale Auflösung symbolischer Referenzen) und "Initialization" (Initialisierung der Klassen Variablen) vorgegangen.
- **Execution Engine:** Das Ausführen der einzelnen Instruktionen erfolgt durch die Execution Engine. Die Execution Engine ist ähnlich wie ein Prozessor durch ein Instruction Set definiert, es ist also spezifiziert, welcher Befehlssatz eine Execution Engine versteht. Der Java Bytecode ist eine Sequenz von Instruktionen für die Java VM. Jede Instruktion besteht aus einem Opcode und optionalen N Operanden.
- **Runtime Data Area:** Der Speicherbereich der Java VM speichert alle für eine Ausführung notwendigen Daten. Bytecodes der geladenen Klassen, Objekte, die ein Programm instanziert, Rückgabewerte, lokale Variablen und Zwischenresultate werden in der Runtime Data Area gespeichert.
- **Method Area:** Die Method Area enthält alle Klassen eines Java Programms, das heisst alle Java Types sind in diesem Bereich gespeichert.
- **Heap:** Der Heap enthält alle instanzierten Objekte des Programms.
- **PC Register:** Jedem Java Thread wird ein eigenes Program Counter Register und ein eigener Java Stack zugewiesen.
- **Java Stack:** Der Java Stack ist in verschiedene Stack Frames organisiert, die jede für sich den Status eines Methodenaufrufs hält. Ruft ein Java Thread eine Methode auf, so kreiert die Java VM ein Stack Frame und löst ein Push des Frames auf den Java Stack aus. Ist die Methode abgearbeitet, so löst die Java VM ein Pop des Stack Frames aus und löscht den Java Stack Frame. Alle Zwischenresultate werden im Java Stack gespeichert. Ein zusätzliches Register für diese Resultate existiert nicht.
- **Native Method Stacks:** Der Status einer Native Method wird im Native Method Stack gespeichert.

- **Native Method Interface:** Diese Schnittstelle erlaubt den Aufruf von Klassen und Methoden, die in Maschinensprache in den Native Method Libraries gespeichert sind.

4.6.3 Rule-Based Systems

Auf Basis des Rule-Based Systems Architektur Stil werden Architekturen für abstrakte Maschinen spezifiziert. Diese Maschinen sind jedoch flexibel, da die Regeln ausserhalb der virtuellen Maschine definiert werden. Dadurch wird die virtuelle Maschine selbst zwar flexibel, die Komplexität der Realisierung steigt jedoch, da die Regeln in einer wohldefinierten Sprache formuliert werden müssen. Diese Sprache jedoch muss klar und widerspruchsfrei spezifiziert werden.

4.6.4 Beispiel: Inference Engine

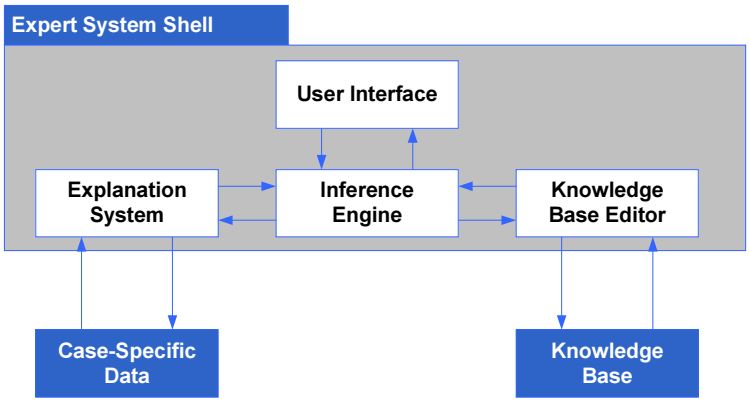


Abbildung 26: Aufbau eines Experten Systems [Gottlob et al 1990]

Die Inference Engine ist der zentrale Bestandteil eines Expertensystems. Sie ist für die schrittweise Abarbeitung der Regeln zuständig, die das Wissen eines Expertensystems darstellen. Es existieren zwei häufig eingesetzte Inference Methoden:

- **Backward-Chaining:** Die Inference Engine vermutet eine Schlussfolgerung aufgrund einer gegebenen Regel und versucht anschliessend zu beweisen, dass die Vermutung korrekt ist.
- **Forward-Chaining:** Die Inference Engine vergleicht die Fakten (vorangegangene Regeln) mit dem IF Teil der Regel, um anschliessend eine Schlussfolgerung ausgehend vom THEN Teil der Regel zu "feuern".

Der Knowlege Base Editor ist für die Verwaltung der Wissensbasis des Expertensystems zuständig. Das Explanation System übernimmt die "Erklärung" der Schlussfolgerungen aufgrund der Case-Specific Data, versucht also die gefundene Expertise in einen Kontext zu setzten, um sie verständlich darzustellen.

4.7 Data Flow

Abbildung 27: Data Flow Architecture Stile

Der Data Flow Architektur Stil umfasst Architekturen, die ein System als eine Abfolge von Transformationen auf sequentiellem Dateninput modellieren. Aktiviert und gesteuert wird das System durch vorliegende Daten. Diese Daten werden Schritt für Schritt durch das System geleitet, werden dabei verändert und schliesslich als ausgehende Daten bereitgestellt.

- **Batch Sequential:** Die Transformationen auf den Daten erfolgen durch voneinander unabhängige Elemente dieses Architektur Stils. Dies bedeutet, dass die Daten als ganzes eingelesen werden, um dann transformiert und anschliessend wieder als Ganzes abgelegt zu werden.
- **Pipes and Filters:** Die Transformationen auf den Daten erfolgen lokal (Filters) mittels Streaming (Pipes). Dies bedeutet, dass ein eingehender Data Stream laufend in einen ausgehenden Data Stream umgewandelt wird.

4.7.1 Batch Sequential

Der Architektur Stil Batch Sequential ist das Paradigma der Grossrechnerwelt. Batch Processing, also die meist Zeitgesteuerte Durchführung von einzelnen Transformationen im Sinne der Veränderung von Daten durch Programmlogik ist die standardisierte Arbeitsweise eines Hosts. Dabei wird das Prinzip EVA wörtlich umgesetzt. Zentrale Eigenschaft dieser Art der Verarbeitung ist die fehlende Interaktivität. Die Ausgabe ist in den meisten Fällen dreigeteilt: Die Datenausgabe, die Ausgabe allfälliger Fehlermeldungen und die Ausgabe eines Reports über Durchlaufmenge, Durchlaufzeit und Ressourcen-Nutzung.

4.7.2 Pipes and Filters

Prominentester Vertreter des Pipes and Filters Architektur Stil sind die so genannten UNIX Pipes, die eine flexible Verarbeitung von Input durch die sequentielle Kombination einzelner UNIX Tools erlauben. Ein Eingabestrom von Daten wird dabei laufend eingelesen, lokal transformiert und anschliessend laufend auf einen Ausgabestrom geschrieben. Ein Filter übernimmt drei Aufgaben, die Anreicherung von Daten durch Berechnung oder durch Hinzufügen von Informationen, die Verfeinerung von Daten durch Verdichtung oder durch Extraktion von Informationen sowie die Transformation von Daten durch Veränderung der Repräsentation.

4.8 Data Centered

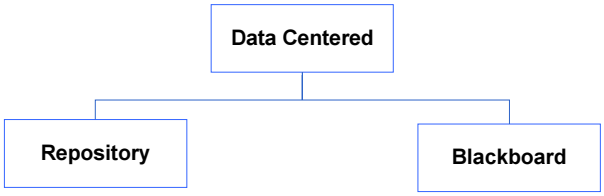


Abbildung 28: Data Centered Architecture Stile

Der Datenzentrierte Architektur Stil umfasst Architekturen, die Systeme beschreiben, deren zentrale Funktion der Zugriff auf Daten eines Repositories ist. Dieses Repository ist entweder passiv (Datenbank) oder aber aktiv (Blackboard). Ein Blackboard System sendet Nachrichten über geänderte Daten an interessierte Subscriber des Systems. Dieser Mechanismus ist sehr flexibel und skalierbar, da Blackboard Systems die Datenhaltung von den angeschlossenen Clients trennen. Die Anzahl der Clients kann sehr einfach verändert werden. Ein einfaches Subscribe eines neuen möglichen Empfängers ist in diesem Fall notwendig. Die Trennung zwischen passiven Datenzentrierten Systemen und aktiven Datenzentrierten Systemen ist fliessend, da heute jede Datenbank mit Triggern sehr einfach in ein Blackboard System umgewandelt werden kann.

Anhänge

5.1 Anhang A: The Art and Science of Software Architecture