Inhaltsverzeichnis

[AppArch: Application Architecture 1](#_Toc504993959)

[Abkürzungen 1](#_Toc504993960)

[Overview 1](#_Toc504993961)

[Architektur 2](#_Toc504993962)

[IT Architecture 2](#_Toc504993963)

[Architectural Analysis of Requirements 3](#_Toc504993964)

[Templates zur Erhebung von FR: functional requirements 3](#_Toc504993965)

[NFR: Non functional requirements 3](#_Toc504993966)

[Templates zur Erhebung von NFR 3](#_Toc504993967)

[Viewpoints 4](#_Toc504993968)

[ADD: Attribute-Driven Design 4](#_Toc504993969)

[Architectural Synthesis 5](#_Toc504993970)

[Solution Strategy: Making Big Decisions 5](#_Toc504993971)

[Pattern 5](#_Toc504993972)

[Why create a model 6](#_Toc504993973)

[C4 Model 6](#_Toc504993974)

[System Context Diagram (SCD) → similar to C4 Context Diagram 6](#_Toc504993975)

[Component Modeling 6](#_Toc504993976)

[Component Identification 6](#_Toc504993977)

[Component Specification: RDD (Responsibility Driven Design) 6](#_Toc504993978)

[Reference Architectures 7](#_Toc504993979)

[CID: Component Interaction Diagram 8](#_Toc504993980)

[Component Realization: Buy vs Build 8](#_Toc504993981)

[DDD: Domain Driven Design 8](#_Toc504993982)

[Tactic DDD (centers on good Aggregate design) 9](#_Toc504993983)

[DDD Layers 9](#_Toc504993984)

[Strategic DDD (combines Aggregates into larger units) 9](#_Toc504993985)

[Relationship Types 9](#_Toc504993986)

[Relationship types in Context map 9](#_Toc504993987)

[AECS: Architecturally Evident Coding Styles 9](#_Toc504993988)

[PoEAA Pattern 10](#_Toc504993989)

[Service and Integration Pattern 11](#_Toc504993990)

[SOA: Service Oriented Architecture 11](#_Toc504993991)

[Service Contract 11](#_Toc504993992)

[Microservices (SOA-substyle) 11](#_Toc504993993)

[REST: Representational State Transfer 12](#_Toc504993994)

[ESB : Enterprise Service Bus 13](#_Toc504993995)

[Logging 13](#_Toc504993996)

[EIP: Enterprise Integration Patterns 13](#_Toc504993997)

[AE: Architectural Evaluation 14](#_Toc504993998)

[AD: Architectural Decisions 14](#_Toc504993999)

[State Management 14](#_Toc504994000)

[Session State (→ Presentation Layer) 14](#_Toc504994001)

[Event Sourcing 14](#_Toc504994002)

[Coupling 14](#_Toc504994003)

[Architectural Evaluation 14](#_Toc504994004)

[DCAR: Decision-Centric Architecture Reviews 15](#_Toc504994005)

[ATAM: Architecture Tradeoff Analysis Method 15](#_Toc504994006)

[arc42 Template 15](#_Toc504994007)

|  |  |  |
| --- | --- | --- |
| AppArch: Application Architecture | | |
| Abkürzungen | | |
| Abk. | Bedeutung | Mehr dazu siehe |
| AA | Architectural Analysis | 2 |
| ACL | Anti Corruption Layer | 9 |
| AD | Architectural Decision | 14 |
| ADD | Attribute Driven Design | 4 |
| ADR | Architectural Decision Record | 14 |
| AE | Architectural Evaluation | 14 |
| AECS | Architecturally Evident Coding Styles | 9 |
| AKM | Architectural Knowledge Management | 14 |
| AOP | Aspect Oriented Programming | Dependency Injection via annotations, one form of AOP in Spring |
| AS | Architectural Synthesis | 5 |
| ASE | Architectural Significance Elements | 3 |
| ASR | Architectural Significance Requirement | 3 |
| ATAM | Architecture Tradeoff Analysis Method | 15 |
| BAC | backup capability, consistency, availability |  |
| BLL | Business Logic Layer |  |
| C4 | Context, Container, Components, Class -Diagrams | 6 |
| CAP | Consistency vs. availability vs. partition tolerance |  |
| CID | Component Interaction Diagram | 8 |
| COTS | Commercially-off-the-Shelf | 5, 8 |
| CQRS | Command Query Responsibility Segregation Pattern | 14 |
| CRC | Components, Responsibilities, Collaborators | 6 |
| CSC | Client Server Cuts | 5 |
| DAD | Disciplined Agile Delivery |  |
| DAO | Data Access Object |  |
| DCAR | Decision-Centric Architecture Reviews | 15 |
| DDD | Domain Driven Design | 8 |
| DTO | Data Transfer Object | 10, 11 |
| EAD | Enterprise Application Development | 5 |
| EAI | Enterprise Application Integration | 5 |
| EIP | Enterprise Integration Patterns | 11, 13 |
| ESB | Enterprise Service Bus | 13 |
| HatEoAS | Hypermedia As The Engine Of Application State | 12 |
| IDEAL | Isolated State, Distribution, Elasticity, Automated Management, Loose Coupling | 11 |
| JEE | Java Enterprise Edition | 7 |
| MFL | Message Format Language | 13 |
| mid-tier | Middle Tier 🡪 processing that takes place in an application server that sits between the user's machine and the DB server. | performs the business logic |
| MSA | Microservices Architecture | 11 |
| NFR | Non functional Requirements | (2), 3 |
| OHS | Open Host Service | 9 |
| OOD | Object Oriented Design | 2 |
| PoEAA | Patterns of Enterprise Application Architecture | Alle, 10  Plugin Pattern, 5  Domain Model Pattern (BLL), 5  DTO, 11  Remote Façade, 11  Client/Server/DB Session State, 14  Gateway, 10  Service Layer, 11 |
| POINT | Taxonomie für gute Microservices | 11 |
| QAS | Quality Attribute Scenario | 4 |
| QoS | Quality of Service | 3, 5, 11, 14 |
| RA | Reference Architectures | 7 |
| SCD | System Context Diagram | 6 |
| SEI | Software Engineering Institute | Urheber von u.a. Quality Utility Tree, 3,4 |
| SOA | Service oriented Architecture | 11 |
| SOAP | Simple Object Access Protocol | 9, **12**, 13 |
| WADL | Web Application Description Language | 12 |
| WSDL | Web Service Description Language | 11, **12** |
| Overview | | |
|  | | |

|  |  |
| --- | --- |
| Architektur | |
| Definition (IEEE) | “The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”  “A software system’s architecture is the set of principal design decisions made about the system.”  The fundamental organization of a system is embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. |
| Tipps | - Je nach Branche kann die Lebensdauer einer Software Architektur stark variieren - Architekturentscheide müssen immer begründet sein (Y-Statements) - Gute Fragen stellen, mit Projektteam sprechen, anderen Abteilungen - Verwende Erfahrungswerte von vorherigen Projekten - In der Architekturphase bereits den Kunden um Feedback bitten (Habe ich etwas vergessen?) |
| Architectural Drives / Forces | - Business needs vs. construction complexity  - Processing style: online (transactional) vs. offline (batch processing)  - Distribution vs. performance, security, consistency  - Software distribution cost  - Reusability vs. performance vs. complexity  - Supportability |
| Trade-offs | - Performance vs. security  - Security vs. usability  - Performance vs. supportability  - Reliability vs. simplicity  - Scalability vs. manageability |
| Agile Core Concepts | Minimize Waste |
| Timeboxing: kleine Zyklen |
| Design for Changes |
| regelmässige Resultate |
| Inkrementelles Vorgehen, wenig vorgängig planen |
| Phasen | **AA: Architectural Analysis of Requirements** 🡪 Erhebung von Qualitätsattributen  - OOA: Object-Oriented Analysis 🡪 RUP, OpenUP - QAS: Quality Attribute Scenario |
| **AS: Architectural Synthesis of Design** 🡪 Architekturentwurf - ADD: Attribute Driven Design → Create architecture to meet quality attributes of the software  - OOD: Object-Oriented Design 🡪 RUP, OpenUP |
| **AE: Architectural Evaluation** 🡪 Überprüfen des Entwurf gegenüber Anforderungen - ATAM: Architecture Tradeoff Analysis Method - DCAR: Decision Centric Architecture Review - Code Reviews |
|  |
| Vorgehen | 1. Start from functional requirements and NFRs (SMART)  2. Identify top-level components  3. Decide what is architecturally (most) significant – when drawing diagrams, when making design decisions  4. Find architectural patterns which resolve the forces underneath the NFRs  5. Make conscious pattern selection decisions and follow-on decisions about technologies and products  6. Document and model these decisions (model with a target audience in mind)  7. Component names should carry domain semantics (consistently) |
| SW Architecture Essentials |  |
| Methods for Software Architecture Design |  |
| Design Goal: Interopera-bility on Different Levels (EIP) |  |

|  |  |
| --- | --- |
| IT Architecture | |
| Architecture is about: | 1 ) Current State  - flows  - pain point  2) Define strategy  - target state  - roadmap  - agree  3) governance  - control  - execution |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.18.28.png |
| Disciplines | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.21.38.png |
| Key Questions | - Which systems interact with the data  - Do they have common interfaces  - What kind of datamodel -> different physical datamodel which is optimized for performance (denomalized)  - What kind of technologies  - Which level of security  How large the data is: do you take the data to the calculation or do you bring the calculation to the data  **Solution**:  1. what is already there?  2. what additional data do i bring in?  3. when do I bring the data?  - process things in batches.  break it into layers and try to think about the key Architecture decisions  identify the options. (is there another option)  -> provide justifications (it could turn out that one thing does not work as expected) |
| Big Data | 3 V: Volume, Velocity, Variety |
| technology that allows parallel processing of data on low cost hardware |
| is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation. |
| Hadoop is an open-source software framework that supports the processing and storage of extremely large data sets in a distributed computing environment. |
| Hadoop Cluster is a group of connected commodity server nodes. It presents unified file system, data and services. It is hugely scalable by adding nodes and provides intrinsic fault tolerance and load balancing. It is managed as a logical unit via unified console. Requires fast connections required between nodes. |
| Machine Learning | Is enabled by big data |
| Machine Learning is a subfield of Artificial Intelligence that enables computers to learn without being explicitly programmed by applying algorithms.  With modern Big Data technologies, Machine Learning algorithms can be executed efficiently on huge volumes of data to yield insights and predictions. |

|  |  |
| --- | --- |
| Architectural Analysis of Requirements | |
| Requirements | In conjunction with functional requirements, quality concerns (NFR) drive and constrain a system's architectural design and often introduce significant trade-offs that must be carefully considered and balanced. |
| ASR: Architecturally Significant Requirement | Eine Anforderung ist *architecturally significant*, wenn einer der folgenden Punkte zutrifft.  Indicators for architectural significance of *requirements*  1. High business value and/or technical risk? 2. Concern of a high stakeholder? Or it is governed by a SLA (service level agreement) 3. First-of-a-kind character? (Novelty)  4. New QoS (Quality of Service) character 5. The requirement has been troublesome and caused critical situations, budget overruns or client dissatisfaction in a previous project with a similar business/technical context |
| ASE: Architectural Significance Elements (component, connector, class, method) | His checklist for architectural significance of *elements* is:  1. associated with critical functionality of the system E.g. money transfer  2. associated with critical quality of the system E.g. performance of distributed communication  3. associated with a critical constraint on the solution E.g. access to an external system  4. incurs a particular technical risk E.g. access to a never-before-tried capability  5. presents a particular architectural challenge E.g. high transaction volume |
| Questions about Decisions | questions to classify the architectural significance of *decisions*  1. Is the decision hard to change later?  2. Is the decision expensive to implement/execute upon?  3. Are demanding, qualitative requirements stated (high security level, high availability, high performance)? 4. Are requirements di cult to map to existing (solutions, experiences)?  5. Is the experience in the solution space weak (in the team)? |
| Assessment Matrix | criteria for the significance of requirements, elements, decisions can be combined    Example: |
| Templates zur Erhebung von FR: functional requirements | |
| User Story Template | As a [role],  I would like to [goal]  so that [effect (business value, impact)]  To achieve this goal, I expect the following qualities (in descending order of priority):  • [measurable usability quality property, e.g. input steps required to complete story]  • [measurable performance quality, e.g. average and worst case story response time]  • [...] |

|  |  |
| --- | --- |
| NFR: Non functional requirements | |
| NFR | Latency: Antwortzeit für den Client  Throughput: Abarbeitungsvolumen |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 08.34.11.png |
| SMART  Taxonomie | Specific: Which feature or part of the system should satisfy the requirement  Measurable: How can testers and other stakeholders find out whether the requirement is met (or not)? Is the requirement quantified?  merke: "signifikant" entspricht einer messbaren grösse (1%, 5% → vgl Statistik)  Agreed upon (Achievable): Do all affected internal and external stakeholders agree on the “S” and the “M” wording?  Realistic (Relevant): Is it technically and economically feasible to achieve the “M” measure in the context of all features or system parts specified under “S”?  Time-bound: When should the NFR meet the “M” measure, is there a growth path from iteration to iteration? |
| FURPS +  Taxonomie | Eine Möglichkeit zur generellen Einteilung/Organisation von QAs |
| **F**unctionality  **U**sability  **R**eliability  **P**erformance  **S**upportability |
| Plus (**+**)  Design constraints  Implementation constraints Physical constraints  Interface constraints |
| Templates zur Erhebung von NFR | |
| Quality Utility Tree | A way to identify, document and prioritize quality attributes  When navigating in the tree from the root to the leaves, the NFRs/QAs get more and more precise; |
|  |
| L, M, H: Low, Medium High  (1, 2) → 1: Importance to system success, 2: Risk/difficulty in achieving  Score entspricht Step 1 von ASR Checkliste (Business Value und Technical Risk) |

|  |
| --- |
| neue Version des Trees: |

|  |  |
| --- | --- |
| QAS: Quality Attribute Scenario  An NFR Specification Template (from SEI)  Deckt sich mit S und M von SMART, hat aber noch weitere Tabellen-  Einträge | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 09.24.24.png/Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.09.34.png |
| Example:  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 09.25.03.png |
| Quality Stories  (Decoupled form Single User Story) | As a [role concerned with system quality, e.g. a leadership or maintenance role],  I would like to [achieve quality goal A]  – without changing the functional scope of the system –  so that future versions of the system can benefit from:  • [technical debt reduction effect]  • [improved service level/system property]  • [positive impact on other technical constraints and environment]  To achieve this goal, I am willing to invest/accept:  • [impact on other quality attributes, e.g. performance penalty for security feature]  • [impact on project plan (cost, timeline)]  • [impact on technical dependencies and risk] |
| As a Development and Operations (DevOps) engineer at a social network/media firm,  I would like to be able to add attributes to the database w/o having to migrate data  – without changing the functional scope of the system –  so that future versions of the system can benefit from:  • New features of the Web software can be introduced more often.  • It is no longer needed to migrate the large amount of existing data to new schemas.  • We become independent of the provider of the current RDBMS.  To achieve this goal, I am willing to accept:  • Data access and data validation logic becomes more complex.  • Approximately five developer days and additional test effort have to be invested.  • Technical feasibility and performance have to be validated in a PoC. |
| Viewpoints | |
| Viewpoint | Complexity management , Viewpoint groups related stakeholder concerns 🡪 Stakeholder focus  Design the architecture in such a way that these concerns are met  Neben 4+1 Model gibt es auch alternativen, zB zwei-dimensionale Modelle (mit Viewpoints und Cross-Cutting Concerns) von IBM und Rozanski/Woods |
| 4 + 1 View Model (RUP) | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 08.41.02.png |
| 0. **Use Case View / Scenario View**: Beschreibt wichtige Anwendungsfälle oder Anwendungsszenarien. (Functional & Non-Functional Requirements) |
| 1. **Logical View**: Beschäftigt sich mit der Funktionalität des Systems für den Endnutzer. (Use Case Realisierung) Dabei werden verschiedene UML-Diagramme zur Darstellung benutzt, darunter Klassendiagramm, Kommunikationsdiagramm, Sequenzdiagramm. |
| 2. **Implementation View / Development View**: Beschreibt das System vom Standpunkt eines Entwicklers und beschäftigt sich mit dem Softwaremanagement. Es wird als UML-Komponentendiagramm oder Paketdiagramm dargestellt (Klassen, Packages, i.e. Software Components.) |
| 3. **Process View**: Beschäftigt sich mit den dynamischen Aspekten des Systems. Er verdeutlicht die Prozesse des Systems und wie diese kommunizieren hinsichtlich des Laufzeitverhaltens. Er soll Parallelität, Verteilung, Integration, Performance und Skalierbarkeit beschreiben. Die entsprechenden UML-Diagramme beinhalten zum Beispiel das Aktivitätsdiagramm. Performance, Scalability, Concurrency (Betriebssystemprozesse, VSS) |
| 4. **Deployment View / Physical View**: Beschreibt das System vom Standpunkt des Systemarchitekten. Er beschäftigt sich mit der Verteilung der Softwarekomponenten auf physikalischer Ebene (also der Zuordnung dieser zu Hardware-Teilen) und der Kommunikation zwischen diesen Komponenten. Das zugehörige UML-Diagramm ist das Deploymentdiagramm (Physische Knoten) |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 08.42.46.png |
| ADD: Attribute-Driven Design | |
| Input and steps | guiding steps to perform design in a systematic way |
|  |
| Agile Modelling |  |
| DAD  Disciplined Agile Delivery |  |

|  |  |  |
| --- | --- | --- |
| Architectural Synthesis | | |
| Overview  EAD  Enterprise Application Development  EAI  Enterprise Application Integration | Solution Outline/Solution Strategy as first Phase in Architectural Synthesis  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 09.53.48.png | |
| Phasen | 1. Context / Scope control: functional components, external interfaces  2. “Big” decisions (the hard-to-change ones):  - Architectural style, client/server cuts (layers, tiers, containers)  - Technology selection (middleware, frameworks)  3. Adressing NFRs via tactics and patterns  4. Component identification, specification, realization | |
| Twin Peak Model | The dependencies that exist between requirements and architecture have been referred to as the twin peaks of requirements and architecture. | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 09.58.26.png | |
| Solution Strategy: Making Big Decisions | | |
| Definition | term comes from arc42.  arc42 offers a template to formulate a solution strategy in which "executive decisions" are made  executive decisions: decisions that are hard to change due to their impact on budgets and people  the focus is on the long lasting architectural decisions. | |
| Overview |  | |
| Component identification (as part of Solution Strategy) | The identification of candidate components primarily, but not only for the logical viewpoint is one of the key activities in solution strategy  - For any user story or use case (i.e. FR), ask questions like:  - Which domain model elements are required to perform the task (verb) outlined by the FR?  - Is the feature something the system can do by or does it need help from downstream backend systems?  - For any Quality Attribute Scenario (i.e. NFR), ask questions like:  - Which already identi ed logical components are a ected (the “S” in SMART)? Can they deliver all the quanti ed behavior (the ‘M‘ in SMART)?  - Are additional components needed? Which layer/tier should they go to? | |
| Decision Making Criteria | Cost (of development e ort or software procurement)  license type  coverage of FURPS requiements (or other NFR taxonmy)  supported platform  vendor credibility and market position | |
| Examples | Examples of activities in solution strategy and executive decisions include:  - Buy vs. build with options such as Do-it-Yourself (DIY) vs. Open Source Software (OSS) / participation vs. procurement of Commercially-o -the-Shelf (COTS) software  - Choice of programming language  - Layering scheme, database paradigm and vendor/product (or open source asset), client-server cut  For more recurring Architectural decisions → Tabel in "Architectural Decisions, page 14" | |
| Pattern | | |
| Popular Architecture Pattern | Client Server | |
| POSA1: Layers, Pipes and Filter, Broker | |
| MVC, Domain Model, Active Record, Table Data Gateway | |
| DDD: Bounded Context | |
| EIP: Messaging, Channel | |
| GoF: Adapter, Observer | |
| ACID | Transaktionseigenschaften: atomicity, consistency, isolation und durability | |
| IoC: Inversion of Control | Hand over flow management responsibility from a main program to a QoS-aware framework; configure it with Lambdas (functional programming), events or framework-defined interface implemented by application components (beans). | |
| Hollywood Principle – “don't call us, we'll call you” | |
| Service Locator / Registry  (Configuration Mgmt) | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 13.48.26.png | |
| Table Data Gateway | An object that acts as a Gateway (466) to a database table. One instance handles all the rows in the table. | |
| Active Record | An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data. | |
| PoEAA Plugin Pattern  (Configuration Mgmt) | Links classes during configuration rather than compilation    Separated Interface is often used when application code runs in multiple runtime environments, each requiring different implementations of particular behavior  Use Plugin whenever you have behaviors that require different implementations based on runtime environment | |
| Dependency Injection | Dependency injection means giving an object its instance variables.  Extend the Inversion of Control pattern commonly found in user interface frameworks and application servers to support constructor injection, setter injection and interface injection (via files or annotations). | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 13.40.18.png | |
| PoEAA Domain Model | Not all UML class diagrams are OOAD domain models | |
|  | **Alternativen**:  PoEAA Transaction Script: Procedural variant of Domain Model PoEAA Table Module: abstraction of Microsoft ADO Data Set Tools, COM/.NET | |
| Layers Pattern  Logical Layers | Distinguish logical and physical usage of Layers pattern  physical → tiers → pattern applied to process/deployment viewpoint  logical → layers → pattern applied to logical/implementation viewpoint | |
| Beispiel zur Anwendung:  links: Layers Pattern 2 mal angewendet: 1) physical → 3 tiers, 2) logical → presentation, logic, data | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.11.45.png |  |
| The Layers pattern takes a logical view and does not imply process/server boundary (any use of remoting is optional) | |
| CSC: Client Server Cuts  Distribution Pattern | How to partition an application into a number of client and server components so that users functional and non-functional requirements are met?  Solution: 5 possible "cuts": 3 cuts within layers, 2 cuts between layers | |
| Two Tier vs Three Tier  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 10.29.19.png | |
| Pros and Cons of CSCs 1 to 5 (Forces vs. Patterns)  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 10.31.22.png | |

|  |  |  |  |
| --- | --- | --- | --- |
| Why create a model | | | |
| Justification | 1. Modeling forces you to think deeply about the requirements 2. Models communicate (architects, team, stakeholder) 3. Models document a design and allow to automate tasks, e.g. test  case creation | | |
| C4 Model | | | |
| Offers one way to model different aspects of a software.  Analogy: World-map, Country-map, Region-map, Street-map <-> Context, Container, Components, Class  Connection to CSC: Container diagram shows tiers, component diagram shows the layers  Connection to Viewpoints:  Context <-> Scenario, Container <-> logical and physical, Component <-> logical, Classes <-> implementation  Avoid waste: Agile approach of UML / Hierarchical relationship between diagram types (zoom) | | | |
| Diagram Hierarchy | 1. **Context** SDC: System Context diagram provides a black box view (external dependencies) 🡪 Provided and consumed interfaces, users | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 11.17.33.png | Draw a diagram showing your system as a box in the center, surrounded by its users and the other systems that it interacts with  Detail isn’t important here as this is your zoomed out view showing a big picture of the system landscape.  Diagram should be understandable by non-technical people.  Legende: | |
| 2. **Container** Container diagram gives an architectural overview (on solution strategy) 🡪 Combines logical and physical view, Shows high-level technology choices (Frameworks, Middleware, DB, App Server)  zoom-in to the system boundary with a Container diagram. a container is a separately deployable unit that executes code or stores data e.g. web application, desktop application, mobile app, database, …  Show high-level shape of the software architecture, responsibilities, major technology choices, how containers communicate  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 11.18.42.png | | |
| 3. **Components** Component diagram refines the overview to meet ASRs 🡪 Logical viewpoint, component responsibilities  shows how a container is made up of a number of ”components“, what each of those components are, their responsibilities and the technology/implementation details /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 11.20.20.png | | |
| 4. **Classes / Code** Classes let design arrive at code level 🡪 implementation viewpoint | | |
| System Context Diagram (SCD) → similar to C4 Context Diagram | | | |
| Definition | - Represents system(s) to be built as black box(es)  - Depicts its interaction with external entities (systems and end users)  - Identifies the information and control flows between the system and external entities  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 10.41.35.png | | |
| Component Modeling | | | |
| Solution Strategy | first sub-stage of architectural synthesis  Four overall component modeling steps. Each step has its own notations and techniques | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 13.08.35.png | | |
| Component Identification | | | |
| CanCos: Candidate Component | Request for structure in a software architecture is met by identification of candidate components and their and continuous refinement  An architectural element in the **logical viewpoint** and used for planning, decision making, architectural prototyping. A candidate component is an architectural element in the logical viewpoint **grouping related responsibilities** that jointly **satisfy one or more (non-)functional requirements** so that design and implementation work can be planned and component realization decisions can be made.  Types of candidate components are: channel components, business logic beans, and adapter components  Examples: “Validate Address”, “Create Telephone Network”  Notation: informal rich picture, C4 Component Diagram, CRC Cards | | |
| Implementation Components | Implementation components reside in the development viewpoint. They group one or more classes and provide an interface that hides their implementation details e.g. in JEE and Spring. Their dependencies are made explicit and managed; they are versioned.  Notation: UML class or component diagrams, detailed C4 component diagrams, actual code | | |
| Deployment Units | Deployment units then bundles one or more implementation components to be hosted on one or more nodes (physical viewpoint). | | |
| Component Identification Algorithm | identify one candidate component per layer and feature/entity  **Input**:  FRs, NFRs/QAs, architectural vision statement (arc42 table, X-statement)  Plus analysis-level domain model (e.g. UML class diagram)  1. First iteration: Find Candidate Components (CanCos)  1. One *channel component* per actor/consuming external system in context diagram  2. One component per layer per feature (user story) and domain model partition  3. One adapter component per backend system appearing in context  2. CRC brainstorming or workshop (to find responsibilities, collaborators)  3. Starting in second iteration: Architectural Refactoring (of CRC cards)  - Address quality concerns such as security and management (from QAS)  - List potential realization technologies (implementation candidates)  - Run a sanity/completeness check (supported by reference architecture)  **Output**:  Refined requirements and domain model, C4 diagrams, CRC cards | | |
| Component Specification: RDD (Responsibility Driven Design) | | | |
| Roles | - Information Holder  - Structurer  - Service provider  - Controller  - Coordinator  - Interfacer | | |
| CRC Cards  Components  Responsibilities  Collaborators | Names should communicate what application/architecture is about  - Metaphors are good, but must be chosen wisely (stakeholder reaction?)  - Strong semantics preferred, e.g. “Web Browser” over “(Page) Client”  - Naming scheme: domain concept plus architectural role/pattern  Value consistency (no contradictions) over completeness (know it all)  - Good component descriptions should be SMART (like goals, but adapted)  - Each outgoing collaboration relationship must correspond to an incoming one elsewhere in system or its context (service consumer and provider)  - Sunny day and rainy days to be considered (remember the QASs?)  Model on same level of detail on all cards and find a medium ground:  - Too precise – hard to implement and to change  - Too vague – no added value, implementations hard to integrate, architecture difficult to validate  - “If in doubt leave it out”  - Might be a whole Layer! | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 11.37.12.png | | |
| Example: | | |
| Reference Architectures | | | |
| Reference architectures compile architectural patterns so that full quality attribute-driven design processes can be streamlined by knowledge and design reuse. | | | |
| Architectural Styles | - sets of principles and patterns aligned with each other to shape an application and make designs **recognizable** and design activities **repeatable**  - The principles express architectural design intent  - the patterns adhere to the principles and are commonly occurring (proven) in practice  - An architectural style improves separation of concerns and promotes design reuse by providing solutions to frequently recurring problems, offers common language 🡪 easier **communicable**  Examples: Client/Server, Layered Architecture, Pipes and Filter, Mater-Slave, … | | |
| Reference Architectures (RA) | make architectural styles more concrete, aim at accelerating design work, offer rough solution/sketch  → RAs can and have to be adopted and refined by definition | | |
| JEE: Java Enterprise Edition  (RA Example)  (JBoss, contains managed Container) | container-based RA that is implemented in application servers  The containers in JEE application servers and Spring realize the Inversion of Control  - Tradeoff: convenience vs. control  - Risk: complexity of middleware  component-based development supported by middleware - Let an application server do the hard work (i.e. managed containers) - Build in Cross Cuttings concerns (Security, Operational Mgmt., Communication) - Annotations  Grundsätze: - Separation of Concerns - Shared services, resource pooling - Information hiding through IoC and DI - Portability - Declarative configuration    JEE to Layer Mapping     |  |  |  |  | | --- | --- | --- | --- | | JDBC | Data Access Layer | DB Access (connecting to a DB directly and running SQL against it) |  | | JMS | Presentation Layer / Service Layer | asynchrone Kommunikation (MOM) → Zeitliche Entkopplung |  | | | |
| Java Technologien | **POJO Plain** Old Java Object: Ein „ganz normales“ Objekt in der Programmiersprache Java  **JavaBeans**: JavaBeans werden auch als Container zur Datenübertragung verwendet. (Public Getter/Setter, Serializable, public Constructor)  **JEE/J2EE**: Spezifikation einer Softwarearchitektur für die transaktionsbasierte Ausführung von in Java programmierten Anwendungen und insbesondere Web-Anwendungen  **Servlet**: Java-Klassen, deren Instanzen innerhalb eines Webservers Anfragen von Clients entgegennehmen und beantworten  **EJB-Container**: Der Container kümmert sich sowohl um die persistente Speicherung der Zustände, als auch um die Verfügbarkeit der EJB-Komponenten für jeden autorisierten Client. (läuft auf einem JEE-Server)  **EJB Enterprise JavaBeans**: Mit EJB können wichtige Konzepte für Unternehmensanwendungen, z. B. Transaktions-, Namens- oder Sicherheitsdienste, umgesetzt werden, die für die Geschäftslogik einer Anwendung nötig sind  **JMS Java Message Service**: API für die Ansteuerung einer MOM zum Senden und Empfangen von Nachrichten aus einem Client heraus, der in der Programmiersprache Java geschrieben ist.  **JAX-WS**: Java-API zum Erstellen von Webservices.  **JAX-RS**: Spezifikation einer Java-API, die die Verwendung des Software-Architekturstils REST im Rahmen von Webservices ermöglicht und vereinheitlicht  **JAX-P Java API for XML Processing**: leichtgewichtiges standardisiertes API zum Validieren, Parsen, Generieren und Transformieren von XML-Dokumenten  **JAX-B Java Architecture for XML Binding**: API die es ermöglicht, Daten aus einer XML-Schema-Instanz heraus automatisch an Java-Klassen zu binden, und diese Java-Klassen aus einem XML-Schema heraus zu generieren. Diesen Vorgang nennt man XML-Datenbindung.  **JDBC Java Database Connectivity**: Datenbankschnittstelle der Java-Plattform, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet und speziell auf relationale Datenbanken ausgerichtet ist.  **JPA Java Persistence API**: Objektrelationalen Abbildung, Persistierung von Java Objekten in relationalen Datenbanken  **RMI Remote Method Invocation**: Aufruf einer Methode eines entfernten Java-Objekts und realisiert die Java-eigene Art von RPC. | | |
| Microsoft Application Architecture  (RA Example) | defines several so-called application archetypes:  - Web Applications  - Rich Client Applications  - Rich Internet Applications  - Mobile Applications  - Service Applications  - Hosted and Cloud Services  - Office Business Applications  - SharePoint LOB Applications | |  |
| Containers | Containers are application-level frameworks for tier 2 of a 2-tier or 3-tier application (note the overloaded usage of the word container: application-level containers are not to be confused with container-based virtualization on the operating system level, for instance a la Docker)  They implement patterns such as Plugin, Inversion of Control and Dependency Injection  Managed containers come as application servers with start/stop commands, etc.; the main routine is inside the framework. An unmanaged container is created by the programmer (who then hands over control to the framework/container). Both implement the two container patterns (Inversion of Control, Dependency Injection) and offer shared services such as security and transaction management. | | |
| Inversion of Control Pattern |  | | |
| Dependency Injection Pattern |  | | |
| Spring  (unmanaged Container) | The Spring framework is a framework for enterprise Java development (open source with extensions for building web applications on the top of JEE). It includes abstraction layers for transactions, persistence frameworks, web application development and JDBC  Spring supports both constructor and setter injection, but its developers tend to prefer setter injection  Many Spring features come as pluggable services, using Dependency Injection to make themselves known to other services/beans  hnliches Foto  /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 13.44.23.png | | |
| CID: Component Interaction Diagram | | | |
| Runtime View  (arc42) | Components as participants (columns), just like in CRC usage  (if visualized looks similar to UML sequence diagram) | | |
| UML Sequence Diagram | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 13.25.46.png | | |
| C4 Dynamic Diagram | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 13.24.57.png | | |
| Component Realization: Buy vs Build | | | |
| Variants | COTS: Commercially Off The Shelf | | |
| DIY: Do it yourself | | |
| OSS: Open Source Software | | |
| Buy Evaluation Process | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 15.55.33.png | | |
| Weighted Scoring | Might biased, personal experience and opinion | | |
| Example Criteria | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 16.02.07.png | | |

|  |  |
| --- | --- |
| PoT: Proof-of-Technology | Platform specific-level Not client- and project context-specific PoT shows that an asset (library, framework, middleware, ...) functions according to its specification and meets basic quality properties |
| PoC: Proof-of-Concept | Similar to PoT, but more detailed. Platform independent-level Client- and project context-specific PoC applies asset in a domain-specific context and reports on experience gained T-shape or T-cut (dt. “Durchstich”): deep but not broad 🡪 Needs experience |
| SWOT Analysis | Strengths (Stärken), Weaknesses (Schwächen), Opportunities (Chancen) und Threats (Bedrohungen) |

|  |
| --- |
| DDD: Domain Driven Design |
| Refines PoEAA Domain Model Pattern 🡪 allg: proposes patterns, some of which are alternatives to each other |
|  |

|  |  |
| --- | --- |
| Tactic DDD (centers on good Aggregate design) | |
| Ubiquitous Language (allgegenwärtig) | Überall (Diskussionen & Code) dieselbe Sprache verwenden |
| Striving to use the vocabulary of a given business domain, not only in discussions about the requirements for a software product, but in discussions of design as well and all the way into the product's source code itself. |
| Entity  Value Object  Service | Distinguish domain model entities that have a global, life-long identity and mutable state from anonymous, immutable value objects. Distinguish these two types of objects that carry mutable or immutable state from stateless services. |
| Entity: Has ID, state, behaviour |
| Value Object: No behavior, no ID, immutable, Is often used to describe, quantify or measure an entity |
| Service: Klasse ohne Member, stateless |
| Aggregate | Groups entities, Validates (e.g. preconditions, invariants) of grouped entities  enforces invariants and is responsible for conceptual integrity of all contained entities by establishing a consistency boundary |
| Group entities and value objects with many relationships and close semantic proximity into aggregates. Identify a single root entity (aggregate root) per aggregate that provides temporary access to other entities as/if needed. Check the invariants (implement the business rules) on aggregate root level. |
| Best Practices:  - Only the root entity may be referenced from external dependencies  - Use asynchronous communication between Aggregates  - Give enforcement responsibilities (for invariants) to root entity  - Keep one Aggregate on one server, allow different Aggregates to be distributed among nodes  - Use the same boundaries for transactions and distribution  - Model true invariants in consistency boundaries  - Design small aggregates  - Reference other aggregates by identity  - Use eventual consistency outside the boundary |
| Repository | Define a repository for all entities that need to be stored transiently or persistently and queried directly (by id). Alternatively, CRUDS entities via root entities handling domain events.  *Note that these Repositories are on higher layers, and have nothing to do with lower data persistence.* |
| Repository: Wird abgefragt (find(), findAll(), store(), etc.) |
| Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects. |
| Factory | Define a single factory per aggregate and root entity  Factory: Pro Aggregate und Root Entity |
| Tactic DDD Meta Model | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 14.51.15.png |
| DDD Layers | |
| Two additional sub-layers in Business Layer  The patterns mostly “live” in the domain model layer (services may also appear in the application layer and in the infrastructure layer) | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 14.28.23.png | |
| Strategic DDD (combines Aggregates into larger units) | |
| Subdomain  (Problem space, top-down) | Subdomains represent a functional/logical partition in/of a complex model |
| Core domain: “Order Management” (high priority), Generic subdomain: “Billing” (less important), Supporting Subdomain |
| Module | Modules (a.k.a. packages) simply group related concepts/abstractions (should promote loose coupling) |
| Domain Event | Something that happened in the domain |
| Intention Revealing Interfaces | Use concrete concept from domain in name  Use pattern name in name to indicate arch. role and resp. |
| BC: Bounded Context  (Solution space, bottom up) | Groups multiple subdomain in one bounded context (for example all core subdomains) limit validity of model definitions |
| Each Bounded Context defines and protects its own Domain Model  One IoC container instance or .war deployment per BC; remote interface(s) |
| A bounded context is an explicit boundary within which a domain model exists. Inside the boundary, all terms and phrases of the Ubiquitous Language have specific meaning, and the model reflects the language with exactness |
| Organize the problem space (i.e., analysis-level domain model) into multiple subdomains; distinguish the core domain from supporting and generic subdomains. Separate the solution space (i.e., design-level domain model) into multiple bounded contexts and apply tactic DDD to each context-specific model |
| Why bother about boundaries?  • Same term, different meaning (homonym)  • Same concept, different use (polyseme)  • External system differences (heterogeneity)  • Scaling up the organization (multiple teams) |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 15.31.33.png  Both are partitioning strategies: top-down from business requirements (subdomains) vs. bottom up from people/team organization and system/project reality (Bounded Context) |
| Strategic DDD Meta Model | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 15.38.11.png |
| Relationship Types | |
| Context Map | Technique to identify/connect bounded contexts |
| Map subdomains to one or more bounded contexts. Make the permitted organizational relationships between bounded contexts explicit and visualize them in a context map. Assign one or more relationship types to these relations that regulate the visibility of the related parties, their right to influence each other (coupling), as well as the (a)symmetry of the relation and remote connectivity |
| Upstream = Supplier  Downstream = Consumer |
| **Three plus one basic decisions:** - Published Language (all)  - Local vs. remote?  - Visibility?  - (A)symmetry?  - Amount of control and influence for client?  - ACL as an option |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 15.47.06.png |
| Relationship types in Context map | |
| PL: Published Language | The interacting bounded contexts agree on a common a language (for example a bunch of XML schemas over an enterprise service bus) by which they can interact with each other. |
| Shared Kernel | Two bounded contexts use a common kernel of code (for example a library) as a common lingua-franca, but otherwise do their other stuff in their own specific way. |
| shared code base: Änderungen in einem context haben auch Änderungen im anderen zur folge |
| OHS: Open Host Service | A Bounded Context specifies a protocol by which any other bounded context can use its services (e.g. a RESTful HTTP service or a SOAP Web service). This protocol exposes the Published Language. |
| Public API: Einheitliches Protokoll (RESTful HTTP) |
| Customer / Supplier | One bounded context uses the services of another and is a stakeholder (customer) of that other bounded context. As such it can influence the services provided by that bounded context. |
| Conformist | One BC uses the services of another but is not a stakeholder to that other BC. As such it uses "as-is" (conforms to) the protocols or APIs provided by that bounded context. |
| Conformist ist der Slave der mit einem ACL geschützt werden kann |
| ACL: Anti-Corruption Layer | DDD pattern that encourages you to create gatekeepers that work to prevent non-domain concepts from leaking into your model. They keep the model clean. repositories are a type of ACL. They keep SQL or object-relational mapping (ORM) constructs outside of your model.  One bounded context uses the services of another and is not a stakeholder, but aims to minimize impact from changes in the bounded context it depends on by introducing a set of adapters – an anti-corruption layer. |
| Protects downstream |
| Makes upstream independent |
| Mapper zwischen JSON und interne Objekte |
| Translate between modeling context (Legacy and New) |
| Tipps, Combinations | 1. Keep the published language minimal, especially when defining an OHS.  2. Avoid shared kernels between supporting subdomains and in customer-supplier relations.  3. Add an ACL on the conformist side of an OHS-conformist relation. Consider introduction of ACLs also for customer-supplier relations |
| AECS: Architecturally Evident Coding Styles | |
| * Components and patterns should be visible in code (Pattername im Klassenname) * Startup code should be centralized (and explicitly named 🡪 XYInitialization) so that it can be located easily and tested separately * Quality-related properties should be marked as such, with names or annotations (Performance, Security) * Be intention revealing   An AECS can be promoted by applying pattern languages such as DDD – if this is done visibly and explicitly. → all DDD patterns per se qualify as elements of an AECS | |
| What is in a name | - Use a grammar construct that unveils type of artifact  - Verb for use case and methods, nouns for components and data members, etc.  - Be intention revealing  - Use concrete concept from domain in name  - Use pattern name in name to indicate arch. role and resp. |

|  |  |
| --- | --- |
| PoEAA Pattern | |
| Übersicht |  |
| Plugin | VL5 Container Patterns  Centralized runtime configuration: Files or annotations  Known Uses: Eclipse plugins, Web browsers, JEE Web container, EJB container, Spring Framework |
| Domain Model | basic patterns to structure BLL (oo) |
| Transaction Script | procedural pattern to structure BLL  *Organizes business logic by procedures where each procedure handles a single request from the presentation.* |
| Table Module | basic patterns to structure BLL  *A single instance that handles the business logic for all rows in a database table or view.*  https://martinfowler.com/eaaCatalog/tableModuleSketch.gif  A Table Module organizes domain logic with one class per table in the data-base, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with Domain Model (116) is that, if you have many orders, a Domain Model (116) will have one order object per order while a Table Module will have one object to handle all orders. |
| Remote Façade | VL9 SOA |
| Data Transfer Object Pattern | VL9 SOA |
| Gateway | *An object that encapsulates access to an external system or resource.*  https://martinfowler.com/eaaCatalog/gatewaySketch.gif  Interesting software rarely lives in isolation. Even the purest object-oriented system often has to deal with things that aren't objects, such as relational data-base tables, CICS transactions, and XML data structures.  When accessing external resources like this, you'll usually get APIs for them. However, these APIs are naturally going to be somewhat complicated because they take the nature of the resource into account. Anyone who needs to under-stand a resource needs to understand its API - whether JDBC and SQL for rela-tional databases or W3C or JDOM for XML. Not only does this make the software harder to understand, it also makes it much harder to change should you shift some data from a relational database to an XML message at some point in the future.  The answer is so common that it's hardly worth stating. Wrap all the special API code into a class whose interface looks like a regular object. Other objects access the resource through this Gateway, which translates the simple method calls into the appropriate specialized API. |
| Service Layer | *Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.*  https://martinfowler.com/eaaCatalog/ServiceLayerSketch.gif  Enterprise applications typically require different kinds of interfaces to the data they store and the logic they implement: data loaders, user interfaces, integration gateways, and others. Despite their different purposes, these interfaces often need common interactions with the application to access and manipulate its data and invoke its business logic. The interactions may be complex, involv-ing transactions across multiple resources and the coordination of several responses to an action. Encoding the logic of the interactions separately in each interface causes a lot of duplication.  A Service Layer defines an application's boundary [Cockburn PloP] and its set of available operations from the perspective of interfacing client layers. It encapsulates the application's business logic, controlling transactions and coor-dinating responses in the implementation of its operations.  Example: |
| Client/Server/ DB Session State | siehe Architectural Evaluation |

|  |  |  |  |
| --- | --- | --- | --- |
| Service and Integration Pattern | | | |
| Compo-nent | A component is used locally (think jar file, assembly, DLL, or a source import) | | |
| Service | A service will be used remotely through some remote interface, either synchronous or asynchronous (e.g. Web service, messaging system, RPC, or socket.) | | |
| - Enterprise Resource Planning (ERP)  - Customer Relationship Management (CRM)  - Document Archive  - Document generation  - Asset management | | |
| Service Layer Pattern | How to hide complex domain models from presentation layer | | |
| Define an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation | | |
| Responsibilities: Role-Based Access Control, transaction control, logging, exception handling  Eignet sich für Kommunikations-basierte Dinge (z.B. RESTful HTTP Schnittstelle, Remote Facade, …) | | |
| Remote Façade | Provides a coarse-grained façade on fine grained objects to improve efficiency over a network | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 16.47.41.png | | |
| DTO: Data Transfer Object | An object that carries data between processes in order to reduce the number of method calls | | |
|  | | |
| Integration Styles | The Enterprise Integration Patterns (EIP) book introduces four styles:  • File Transfer  • Shared Database  • Remote Procedure Invocation  • Messaging  • The Web can be seen as a 5th style. | | |
| SOA: Service Oriented Architecture | | | |
| Overview | SOA, REST and micro-services are related problem solving concepts in this integration design space (🡪Enterprise Application Integration (EAI)):  SOA and REST qualify as architectural styles specializing on integration. | | |
| Definition | Business Analyst / scenario viewpoint:  *A set of services that a business wants to expose to their customers and partners, or other portions of the organization* | | |
| IT Architect / logical, process or deployment/physical viewpoint:  *An architectural style which requires a service provider (server), a service requestor (client, consumer) and a service contract (a.k.a. client/server)  A set of architectural patterns such as service contract, enterprise service bus, service composition, and service registry, promoting principles such as modularity, layering, and loose coupling to achieve design goals such as separation of concerns, reuse, and flexibility.* | | |
| Developer, Administrator:  *A programming and deployment model realized by standards, tools and technologies such as Web services and Service Component Architecture (SCA)* | | |
| JAX-WS | Java API for XML Web Services (JAX-WS) | | |
| Routing 🡪 Wohin; Transformieren 🡪 Daten; Adaptieren 🡪 Protokoll | | | |
| Service Consumer and Provider | | | implementation details of service providers are hidden from the consumers  Consumer and provider do not have to be implemented in the same programming language; they can run on multiple hardware and operating system platforms  service consumers can share and reuse service providers freely |
| Service Registry | | | - provides information about services that can be invoked via the ESB  - It makes service contracts and service provider access information available to the ESB and to service consumers  - objective of the pattern: provide location transparency for service consumers |
| Service Contract | | | |
| Definition | | Service Contract required for REST and other SOAs (Swagger, RAML)  contract has a functional and a behavioral part:  - The functional contract is machine- readable and specifies one or more operations which comprise request and, optionally, response messages  - The behavioral part of the service contract defines the non-functional characteristics (e.g. Security policies) of the message exchange and the operation invocation semantics.  motivating principles for this pattern are modularity and platform transparency 🡪 The service contract separates interface and implementation; it is the only knowledge shared by service consumer and service provider | |
| Template | | **/Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 09.29.05.png** | |
| Example | | Web services technologies → Web Service Description Language (WSDL) defines functional service contracts | |

|  |  |  |
| --- | --- | --- |
| Microservices (SOA-substyle) | | |
| Definition | *"microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms"*  Microservices Architecture (MSA) constrains SOA to make services independently deployable and scalable  • Business alignment and loose coupling (goal: agility/ exibility)  • Independent deployability and scalability (and therefore replaceability)  • Decentral decisions (service autonomy)  • Single reason to change per service | |
| Microservice-based SOA  Seven Tenets | Kein neuer Architekturstil → Implementation approach and physical refinement of SOA | |
| 1. Fine-grained interfaces to single-responsibility units that encapsulate data and processing logic are exposed remotely to make services independently scalable, typically via RESTful HTTP resources or asynchronous message queues.  2. Business-driven development practices and pattern languages such as Domain- Driven Design (DDD) are employed to identify and conceptualize services.  3. Cloud-native application design principles are followed, e.g., as summarized in Isolated State, Distribution, Elasticity, Automated Management and Loose Coupling (IDEAL).  4. Multiple storage paradigms are leveraged (SQL and NoSQL) in a polyglot persistence strategy; each service implementation has its own data store.  5. Lightweight containers are used to deploy and scale services. (Docker)  6. Decentralized continuous delivery is practiced during service development.  7. Lean, but holistic and largely automated approaches to configuration and fault management are employed within an overarching DevOps approach. | |
| Structure |  | |
| MicroService mapped to SOA |  | |
| MicroService mapped to Viewpoints |  | |
| Best Practices for Designing a Microservices Architecture | - Create a Separate Data Store for Each Microservice  - Keep Code at a Similar Level of Maturity  - Do a Separate Build for Each Microservice  - Deploy in Containers  - Treat Servers as Stateless | |
| POINT | purposeful | |
| should expose domain objects | |
| isolated | |
| neutral: not optimized for a client | |
| T-shaped: Offer broad and deep calls | |
| Size  (not in LoC) | The size of a microservice should be chosen such that it can be  - Developed (and operated => DevOps?) by a single team  - Fully understood by each developer on the team  - Replaced by a new implementation if necessary  On the other hand, it should not be too small  - Communication and deployment overhead  - Transactions spanning multiple microservices are hard to manage  - The same is true for data consistency (consistency boundaries) | |
| Granularity | • Business granularity: semantic density (role in domain model and BPM)  - major impact on agility, flexibility, maintainability  • Technical granularity: syntactic weight and QoS entropy  - determines performance, scalability, interoperability, maintainability, flexibility  format dimension of loose coupling is directly related to service granularity. | |
| From Monolith to Microservice | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 09.10.37.png  Tradeoffs: Monolith vs MicroService  Fast delivery (of new versions) and flexibility vs. cognitive load and effort  Scalability vs. performance | |
| SOAP | Simple Object Access Protokoll: XML Sprache, nicht an Protokoll gebunden 🡪 weder Protokoll noch objektorientiert 🡪 toller Name ☺ | |
| Two communication styles: Document style, RPC style | |
| message elements: Envelope, Headers, Message Body and Faults |  |
|  | |
| WSDL | Web Services Description Language: XML Language for Service Descriptions | |
| WADL | Web Application Description Language | |
| API Gateway Pattern | Example of a Microservices Pattern  Wrap multiple microservices behind a API Gateway so that client and service can evolve independently 🡪 Façade, single entry point  Goal: Frontend-to-Backend Integration → "external ESB"  ttp://uniknow.github.io/AgileDev/site/0.1.9-SNAPSHOT/images/api-gateway.pngdefining characteristics:  - protocol adaptation  - model transformation  - routing (incl. fanning out,  which is a form of routing)  - security policy enforcement point  - mediation | |
| other used patterns | Remote Facade, Anti-Corruption-Layer (ACL), ESB | |
| REST: Representational State Transfer | | |
| REST vs SOAP/WSDL | Kann man so nicht vergleichen 🡪 REST ist ein Architkturstil  WSDL 🡪 XML Format | |
| REST | REST is an architectural style (for integration), defined via constraints (so not an API technology or protocol)  REST is an architectural style for Web-based application integration  RESTful HTTP as dominating implementation of the REST style | |
| REST Constraints | Client-server | |
| Stateless: State immer mitsenden 🡪 Server merkt sich nichts 🡪Keine Cookies 🡪 Besser skallierbar | |
| Cacheable | |
| Uniform interface (URI, HTTP) | |
| Layered system | |
| Code on demand (optional) | |
| REST key concepts | - Linked resources with URIs  - Representations (external view)  - Unified method set (nicht nur HTTP möglich) - self- descriptive | |
| URI | URI-reference = [ absoluteURI | relativeURI ] [ "#" fragment ] | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 17.12.51.png | |
| HTTP Verbs | /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 17.13.42.png | |
| Maturity Level | Level 0: POX (Plain Old XML) Sumpf | |
| Level 1: Jede Ressource hat eine eigene URI | |
| Level 2: Korrekt verwendete HTTP Verben (POST, GET, PUT, DELETE, …) 🡪 HTTP API oder Web API | |
| Level 3: Hypermedia Controls 🡪 **HATEOAS**: Hypermedia As The Engine Of Application State (RESTful / Hypermedia API) | |
| HatEoAS | Client discovers possible interactions with resources in responses | |
| HatEoAS = Hypermedia As The Engine Of Application State  H: Typed links (hypertext) as identifiers of AS  E: Custom Media Types (CMTs) as carriers of AS  AS: Application State, i.e. State of processing, Preconditions, post conditions, enterprise resources | |
| Media Type (MIME Type)  🡪 HatEoAS, Engine | - application/xml  - application/atom+xml  - text/plain  - image/png - Your custom media type  Selecting an existing media type or creating a new, domain-specific Custom Media Type (CMT) is an important design decision when crafting RESTful HTTTP APIs | |
| JAX-RS | Java API for RESTful Web Services (Java EE) | |
| HATEOS Response | Response contains all valid next step options | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-30 um 17.21.14.png | |
| Swagger / OpenAPI | Swagger is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services. (API description language.)  Support for REST level 1 and 2, incomplete 3.  Concepts: Path, schema, operation, parameters, error codes | |
| REST Anti Pattern | • Tunneling everything through GET  • Tunneling everything through POST  • Ignoring caching  • Ignoring response codes  • Misusing cookies  • Forgetting hypermedia  • Ignoring MIME types  • Breaking self-descriptiveness | |

|  |  |
| --- | --- |
| ESB : Enterprise Service Bus | |
| Definition | ESB pattern is the SOA-specific refinement of the general broker pattern  An enterprise service bus (ESB) is a software architecture model used for designing and implementing the interaction and communication between interacting software applications in service-oriented architecture (SOA)  primary responsibility of an ESB is to route request and response messages  The term ESB is used for the architecture pattern and the product category (for enterprise application integration in a SOA landscape)  An ESB is an implementation of the Anti Corruption Layer (ACL) |
| Ohne ESB | Punkt zu Punkt Verbindungen sind schlecht wartbar |
| - Logical coupling: One service uses the function of another service.  - Physical coupling: Implementation of a logical coupling.  - Services use different communication technologies such as JMS, SOAP, DB, RESTful HTTP, SFTP, FTP, File, Socket, UDP, Custom, etc.  - Different services use different data formats and encodings.  - no central way to monitor or troubleshoot.  - Reinventing the wheel over and over |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 08.16.54.png |
| Mit ESB | - Zentraler Bus, der die Interaktion zwischen heterogenen Services steuert - An ESB decouples systems from each other, allowing them to communicate without dependency or knowledge of other systems on the bus. |
| Core Functionality: Routing, Message transformation, Protocol transformation |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 08.20.59.png |
| Features | - The ESB provides Adapters to various communication techniques:  - SOAP, JMS, SFTP, REST, DB-protocols, SMTP, (RPC, SAP)  - Routing from messages from the sender to the receiver service  - Fixed routing (EIP: Recipient List)  - Content-based routing (EIP: Content-Based Router)  - Transformation  - XML documents (XSLT, XQuery)  - Binary documents (e.g. MFL Message Format Language)  - XML to JSON  - ESBs may also offer:  - Orchestration define the sequence of services called to fulfill a task.  - Security: Mediation between different security protocols |
| various functions that are assigned to an ESB ("blueprint for an enterprise service bus") |
| Produkte | IBM WebSphere Enterprise Service Bus, Oracle Service Bus, Microsoft Biztalk Server, Mule, Open ESB |
| XQuery | Sprache für das Mappings zwischen zwei Schemata |
| MFL | Message Format Language: Proprietary Oracle format to define rules to transform formatted binary data into XML |
| Process Big Files | One transaction for download and xquery and one transaction for upload |
| iPaaS | Integration Platform as a Service 🡪 Cloud |
| Restricted Resources | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 09.03.48.png |
| Flooding | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 09.03.15.png |
| Logging | |
| Correlation ID | Write message ID in filename, if you cannot touch message content |
| Splunk | - Effiziente Suche in distributed Log Files  - Standartisierte Log Meldungen ermöglichen übersichtliche Darstellung |
| Distributed Logging | - *Stakeholders & Concerns*: System administrators and support sta : logging needed for toubleshooting, performance measurement and improvement. It is also essential for/in certain agile practices and pproaches such as DevOps and microservices.  - *difficult in practice*: Message identifiers need to be consistent and transported so that correlation is posible and the ”big picture” can be seen.  - *Implementation*: Correlation Identi er and all EIPs in systems management category; Splunk or combination of Elastic- search, Logstash, and Kibana |
| EIP: Enterprise Integration Patterns | |
|  | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 08.23.24.png | |
| Kategorien | Message Endpoints: Transactional Client |
| Message Constrcution: |
| Message Channels: Channel Adapter |
| Message Routing: Content-based routing |
| Systems Management: |
| Message Transformation: |
| Typical scenario | Adapter 🡪 Router 🡪 Transformer |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 08.37.35.png |
| Versioning | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 08.40.11.png |
| Other things ESB can do | Security checks, logging/monitoring (used for billing), support versioning by routing requests from certain clients to old provider version or by transforming messages |
| -ilities that can be achieved | Interoperability is promoted by adapting proprietary protocols to standard ones.  Reliability is promoted if the ESB is message queue-based.  Changeability is promoted if ESB supports versioning and compatibility modes |

|  |  |
| --- | --- |
| AE: Architectural Evaluation | |
| AD: Architectural Decisions | |
| Definition | The design decisions that are costly to change  Architectural decisions **capture key design issues** and the **rationale behind chosen solutions**. They are conscious design decisions concerning a software-intensive system as a whole or one or more of its core components and connectors in any given view. The outcome of architectural decisions influences the system’s nonfunctional characteristics including its software quality attributes |
| Recurring Ads  (wieder-kehrende ADs) | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.12.47.png |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.12.40.png |
| ADR Architectural Decision Records | AD Records (ADRs) capture the rationale justifying a design, in addition to the design itself. they answer “why” questions. (🡪 Y Template)  purpose of AD work product  • Provide a single place to find important architectural decisions  • Make explicit the rationale and justification of architectural decisions  • Preserve design integrity in the provision of functionality and its allocation to system components  • Ensure that the architecture is extensible and can support an evolving system  • Provide a reference of documented decisions for new people who join the project  • Avoid unnecessary reconsideration of the same issues |
| AKM: Architectural Knowledge Management | It is not practical to record every architecture decision about a system |
| - Decisions regarding architecturally significant requirements;  - Decisions needing a major investment of effort or time to make, implement or enforce;  - Decisions affecting key stakeholders or a number of stakeholders;  - Decisions necessitating intricate or non-obvious reasoning;  - Decisions that are highly sensitive to changes;  - Decisions that could be costly to change; |
| Decision Log Template  Example of ADR regarding *Integration Style* | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 10.45.56.png |
| Y-Template | - Link to (non-)functional requirements and design context  - Tradeoffs between quality attributes |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 10.46.31.png  Example:  *In the context of* the order management scenario at T,  *facing* the need to process customer orders synchronously, without loosing any messages,  *we decided to* apply the Messaging pattern and the RPC pattern  *and neglected* File Transfer, Shared Database, no physical distribution (local calls)  *to achieve* guaranteed delivery and request buffering when dealing with unreliable data sources  *accepting* that follow-on detailed design work has be performed and that we need to select, install, and configure a message-oriented middleware provider.  Im Kontext …,  konfrontiert mit der Notwendigkeit …,  haben wir uns für … entschieden und gegen …,  um eine flexible, … zu erreichen;  wir nehmen die daraus resultierende … in Kauf. |
| State Management | |
| Stateless | Statelessness (of interactions) helps to achieve loose coupling (to scale out, to deploy rapidly and flexibly); avoid server session state  state management is a key Architecturally Significant Requirement (ASR) |
| Application State | information about where you are in the interaction (e.g. login status)  Application state should live on the client and should be supplied with each client request  can also be seen as a general term for session state (in presentation layer) and ressource state (in BLL) |
| Resource State | state that needs to be persistent and survivable even after client disconnect/restart/session/end  (e.g. uplodaed img, order content, purchase history, all DDD entities)  🡪 Business Logic Layer |
| Session State (→ Presentation Layer) | |
| Client Session State | - Scales well, but has security and possibly performance problems  - HTML/HTTP: cookie, hidden field, URL rewrite (REST: no cookies!) |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 10.13.20.png |
| Server Session State | - Uses main memory or proprietary data stores in an application server (e.g. HTTP session API in JEE servlet container)  - Persistent HTTP sessions no longer recommended when deploying to a cloud due to scalability and reliability concerns  - smallest development effort  - does not go well with clustered deployments |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 10.13.27.png |
| Database Session State | - Is well supported in many clouds, e.g. via highly scalable key-value storage (a type of NoSQL database) such as Redis |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 10.13.33.png |
| Event Sourcing | |
| Event Sourcing | Let components communicate with each other by sending events when their internal state changes. Capture all changes to application state as a sequence (or stream) of events. Allow state transitions to be triggered by incoming events only. Provide replay capabilities and (optionally) temporal query interfaces. |
| Charac-teristics | Different approach of session managment Start from empty application state and use change log to find current state (event replay)  All changes come in as events and are handled via event processing  Events are never deleted, but additional compensating events are sent Event sourcing goes well with functional programming and cloud computing |
| Event | Something that happened that affects application/resource state Content must be complete enough to indicate update needs (Id, Timestamp, Entity reference (id), Attributes often codified as self-descriptive markup (JSON, XML)) |
| Usage Scenario | - Complete rebuilds, temporal query, event replay (e.g. Audit Log, Version Control)  - used to implement asynchronous, eventually consistent updates across DDD aggregates/bounded contexts (services as event handlers) |
| Coupling | pattern supports time, reference, and platform autonomy  The degree of format autonomy depends on the event content. |
| **CQRS: Command Query Responsibility Segregation Pattern** | |
| Definition | Use a different (domain, communication) model to update information than the model you use to read information |
| Separate the query processing from the create, update, delete business logic (two models). Optionally, use two data stores as well, a query cache and a transaction log (archive). Update the cache straight from the entity modifiers and synch. the two databases via message queues or database replication. |
| Vorraus-setzung | Daten werden ganz selten geschrieben, aber ganz oft gelesen |
| Nachteil | Schwierig, die zwei Modelle konsistent zu halten |
|  | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 10.31.26.png |
| Coupling | |
| Granularity | Loose coupling between few coarse-grained services is easier to achieve than loose coupling between many fine-grained ones – sharing and delegating responsibilities increases coupling |
| Business granularity: semantic density (role in domain model and BPM)  Technical granularity: syntactic weight and QoS entropy |
| Dimensionen | 1. Zeitliche Autonomie: Server und Client müssen nicht zur gleichen Zeit aktiv sein (Message Queues) |
| 2. Referenz/Adresse Autonomie: Client und Server kennen sich nicht |
| 3. Platform Autonomie: Unterschiedliche Teilnehmer können in unterschiedliche Umgebungen und in unterschiedlichen Programmiersprachen programmiert sein |
| 4. Format Autonomie: Client und Server müssen sich nicht auf ein Datenmodell einigen |
| Architectural Evaluation | |
| Goal | Assess and communicate whether the design is able to address the requirements (NFRs in particular) with the objective to reduce risk of budget overruns (e.g., time, money): ”Am I doing the right things? Am I doing things right?” (in terms of decision making) |

|  |  |  |
| --- | --- | --- |
| DCAR: Decision-Centric Architecture Reviews | | |
|  | /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.04.48.png | uncover and evaluate the rationale behind the most important architecture decisions  identify problems or risks early enough so that they can be more easily xed or mitigated  decision-centric method |
|  |  | |
| |  |  | | --- | --- | | Step 1: Preparation | - invite stakeholders to DCAR session  - lead architect prepares a presentation: most important architectural require- ments, high-level views of the ar- chitecture, the approaches used (such as patterns or styles), and the technologies used (such as database management systems or middleware servers)  - The management/customer representative prepares a presentation describing the software product and its domain, the business environment, market differentiators, and driving business requirements and con- straints. | | Step 2: Introduction to DCAR | - introductory presentation of the DCAR method | | Step 3: Management Presentation | - The management/customer representative presents slide from step 1  - review team notes any potential forces during the presentation | | Step 4: Architecture Presentation | - lead architect presents slide from step 1  - reviewers revise and complete the list of architecture decisions they identified as preparation in Step 1 | | Step 5: Forces and Decision Completion | - reviewers have assembled a preliminary list of architecture decisions and decision forces (in step 1-4)  - goal: clarify the architecture decisions and their relationships, complete and verify the forces relevant to these decisions  - One of the reviewers creates a decisions relationship diagram | | Step 6: Decision Prioritization | - the number of decisions elicited in the previous steps is too large 🡪 stakeholders will have to negotiate which decisions to review in the following steps (should include mission-critical decisions, decisions known to bear risks, and decisions causing high costs.) | | Step 7: Decision Documentation | - document the set of decisions that received the highest ratings in the previous step  - describing the applied architectural solution, the problem or issue it solves, known alternative solutions, and the forces that must be considered to evaluate the decision 🡪 siehe table in next column | | Step 8: Decision Evaluation | - challenge the described decision by identifying additional forces against the chosen solution  - Finally, all participants decide by voting whether the decision is good, acceptable, or has to be reconsidered | | Step 9: Retrospectives and Reporting | - After all of the selected decisions are evaluated, the review team collects the notes and artifacts created dur- ing the session 🡪 input for the evaluation report | | |
| ATAM: Architecture Tradeoff Analysis Method | | |
|  | oftware architecture evaluation method to systematically uncover architectural problems, scenario-based  /Users/Michi/Desktop/Bildschirmfoto 2017-12-31 um 11.06.27.png | |
| - Which architectural decisions were made to satisfy quality attribute scenario X?  - Which architectural approach supports the satisfaction of scenario X  - Which compromises (tradeoffs) were made along with this decision?  - Which other quality characteristics or architectural goals does this decision influence?  - Which risks stem from this decision or approach?  - Which risks exist that could prevent the scenario and the related quality  requirements from being satisfied??  - Which analyses, investigations or prototypes back this decision? | |

|  |  |
| --- | --- |
| arc42 Template | |
| Übersicht |  |
| Zweck | Dieses Template (bzw. das daraus erzeugte Dokument) gibt Ihnen eine standardisierte Gliederung für Software-Architekturbeschreibungen vor. Das aus diesem Template entstehende Dokument stellt die umfassende und vollständige Beschreibung der Architektur eines IT- Systems dar. |
| Struktur |  |
| 4+1 Mapping | Scenario: Section 1, 2, 3;  Logical: Building Block View (level 1, 2);  Implementation: Building Block View (level 3);  Process: Building Block View (level 2) and Runtime View;  Deployment: Deployment View. All other template elements can have content that crosses viewpoints or belong to several of them. |
| C4 Mapping | The C4 model is compatible with the arc42 documentation template as follows:  Context and Scope => System Context diagram,  Building Block View (level 1) => Container diagram,  Building Block View (level 2) => Component diagram,  Building Block View (level 3) => Class diagram”. |
| Lecture Mapping |  |