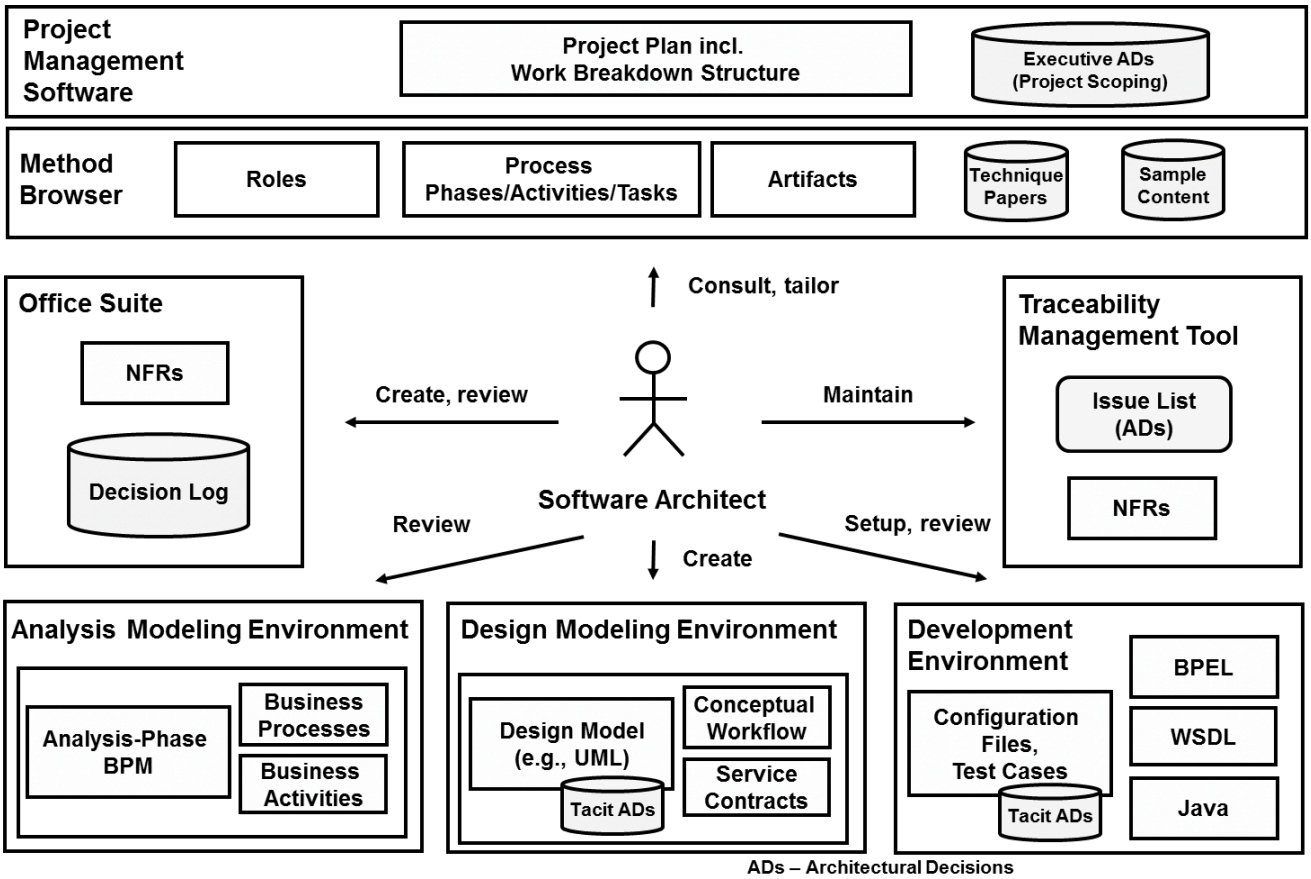
Application Architecture

Zusammenfassung von Jürg Schleutermann



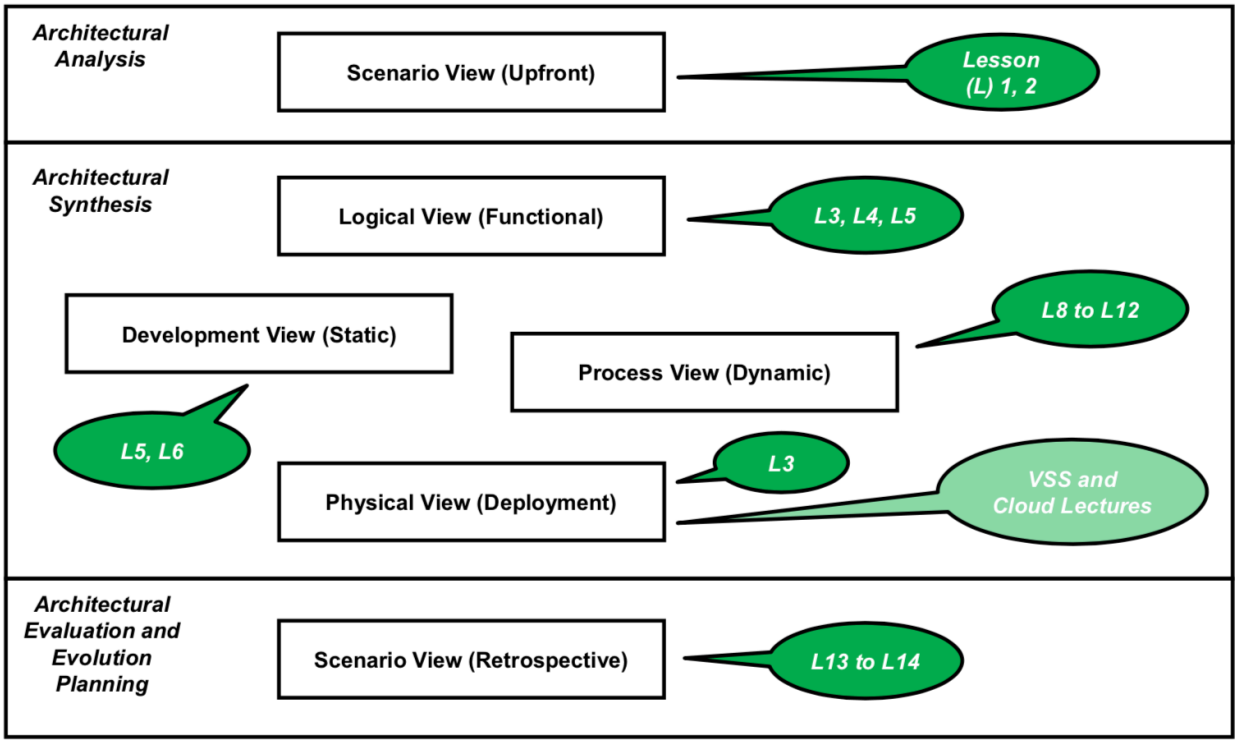
1 Abbreviations

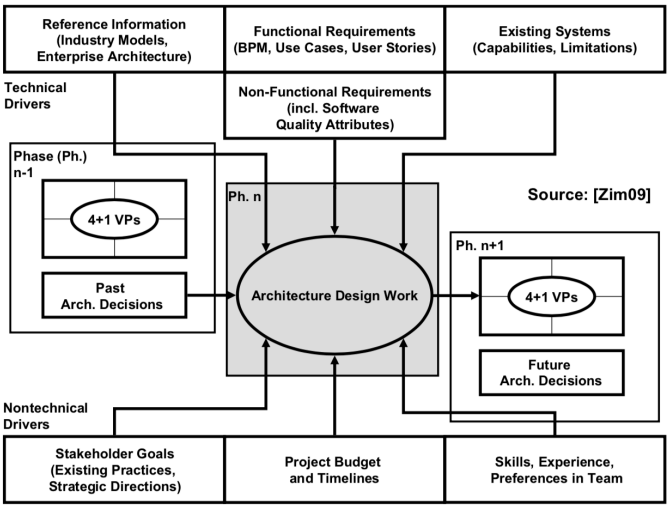
AA	Architectural Analysis
ACL	Anti-Corruption Layer
AD	Architectural Decision
ADD	Attribute-Driven Design
ADR	Architectural Decision Record
AE	Architectural Evaluation
AECS	Architecturally Evident Coding Style
AM	Agile Modeling
AMQP	Advanced Message Queuing Protocol
AS	Architectural Synthesis
ASR	Architecturally Significant Requirement
ATAM	Architecture Tradeoff Analysis Method
BA	Business Activity (z.B. Create Order)
BC	Bounded Context
BLL	Business Logic Layer
BPMN	Business Process Model and Notation
BR	Business Rule (z.B. Order Number)
CBM	Component Business Model
CMT	Custom Media Type
COTS	Commercially-of-the-Shelf
CQRS	Command Query Responsibility Segregation
DAD	Disciplined Agile Delivery
DCAR	Decision-Centric Architecture Review
DCS	Distributed Control System
DDD	Domain-Driven Design
DI	Dependency Injection (Pattern)
DIY	Do-it-Yourself
EAD	Enterprise Application Development
EIA	Enterprise Application Integration
EIP	Enterprise Integration Pattern
ER	Enterprise Resource (z.B. Order)
ESB	Enterprise Service Bus
HATEOAS	Hypermedia as the Engine of Application State
IDEAL	Isolated State, Distribution, Elasticity, Automated Management and Loose Coupling
IoC	Inversion of Control (Pattern)
JEE	Java Enterprise Edition
JMS	Java Message Service
MOM	Message-Oriented Middleware
MSA	Microservices Architecture
MVC	Model View Controller (Pattern)
NFR	Non-Functional Requirement
OHS	Open Host Service
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OpenUP	Open Unified Process
OSS	Open Source Software
PL	Published Language
POC	Proof of Concept
PoEAA	Patterns of Enterprise Application Architecture
POT	Proof of Technology

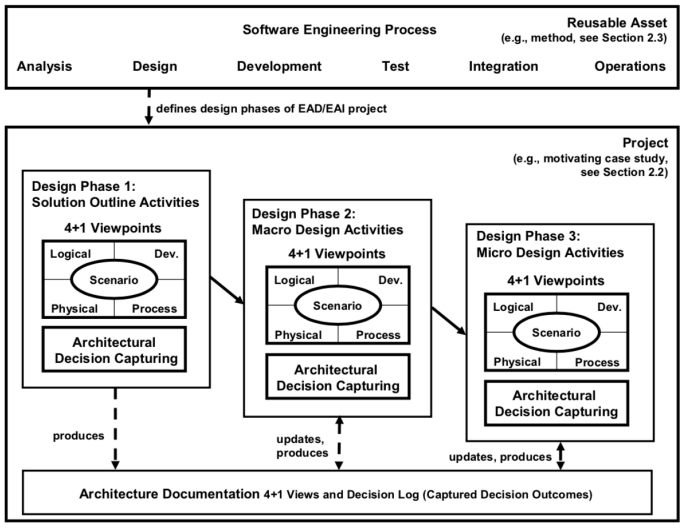
QA	Quality Attribute
QAS	Quality Attribute Scenario
QAW	Quality Attribute Workshop
QoS	Quality-of-Service
QoS	Quality of Service
RA	Reference Architecture
RBAC	Role-Based Access Control
RCDA	Risk- and Cost-Driven Architecture
RDBMS	Relational Database Management System
RDD	Responsibility-Driven Design
REST	Representational State Transfer
RPC	Remote Procedure Call
RUP	Rational Unified Process (IBM)
SAFe	Scaled Agile Framework
SCD	System Context Diagram
SOA	Service-Oriented Architecture
SWOT	Strengths, Weaknesses, Opportunities, Threats
URI	Uniform Resource Identifier
WADE	Web API Design and Evolution
XP	Extreme Programming

2 "Big Picture"

The lecture is structured according to the 3 phases of architecturing as well as the 4 + 1 viewpoints concept.







3 Architecture in General

3.1 Definition

"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." [ISO/IEC/IEEE 42010:2011]

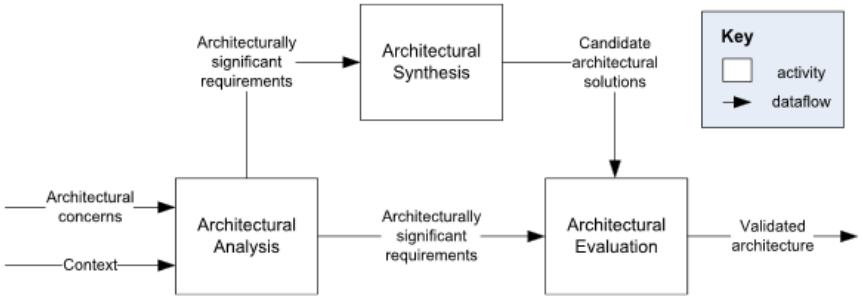
"A software system's architecture is the **set of principal design decisions** made about the system." [JansenBosch05]

3.2 3 Phases

1. Architectural **Analysis** (What?)
   Example Methods: QAS, OOA (from RUP / OpenUP)

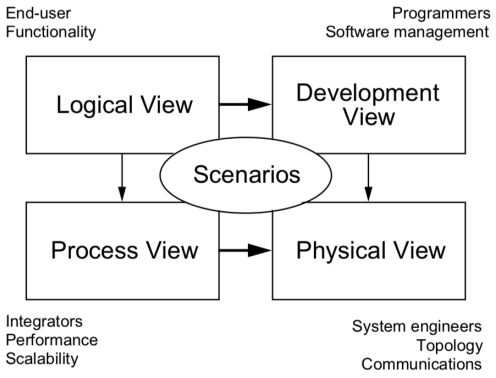
2. Architectural **Synthesis** (How?)
   Example Methods: ADD, OOD (from RUP / OpenUP)

3. Architectural **Evaluation** (How good?)
   Example Methods: ATAM, DCAR



3.3 4 + 1 Viewpoints

"A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views. Views and viewpoints have an instance-class relationship."



- The **logical** view, which is the object model of the design (when an object-oriented design method is used)
- The **process** view, which captures the concurrency and synchronization aspects of the design
- The **physical** view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect
- The **development** view, which describes the static organization of the software in its development environment.

Original definition by P. Kruchten:

	Logical View	***Process View***	***Development / Implementation View***	***Physical / Deployment View***	***Scenarios / Use-Cases View***
Components	Class	Task	Module, Subsystem	Node	Step, Scripts
Connectors	association, inheritance, containment	Rendez-vous, Message, broadcast, RPC, etc.	compilation dependency, “with” clause, “include”	Communication medium, LAN, WAN, bus, etc.	
Containers	Class category	Process	Subsystem (library)	Physical subsystem	Web
Stakeholders	End-user	System designer, integrator	Developer, manager	System designer	End-user, developer
Concerns	Functionality	Performance, availability, S/W fault- tolerance, integrity	Organization, reuse, portability, line- of-product	Scalability, performance,av ailability	Understandability

Note: Term "Class" doesn't necessarily mean a C# or Java class (more like a candidate)!

Classification from lecture:

UML	Class / State	Activity / Sequence	Component / Package	Deployment	
C4 Model	Container / Component diagram		Classes	Container	Context
arc42 Template (Section)	• Building Block View (Level 1) • Building Block View (Level 2)	• Building Block View (Level 2) • Runtime View	• Building Block View (Level 3)	• Deployment View	• Introduction and Goals • Architecture Constraints • System Scope and Context
Component	Candidate Component		Implementation Component	Deployment Unit	
OOA/D	Classes (OOD)		Classes (OOP)		Classes (OOA)

The scenarios are in some sense an **abstraction of the most important requirements**. This view is redundant with the other ones (hence the “+1”), but it serves two main purposes:

- as a driver to discover the architectural elements during the architecture design
- as a validation and illustration role after this architecture design is complete, both on paper and as the starting point for the tests of an architectural prototype

3.4 Agile vs. Architecture

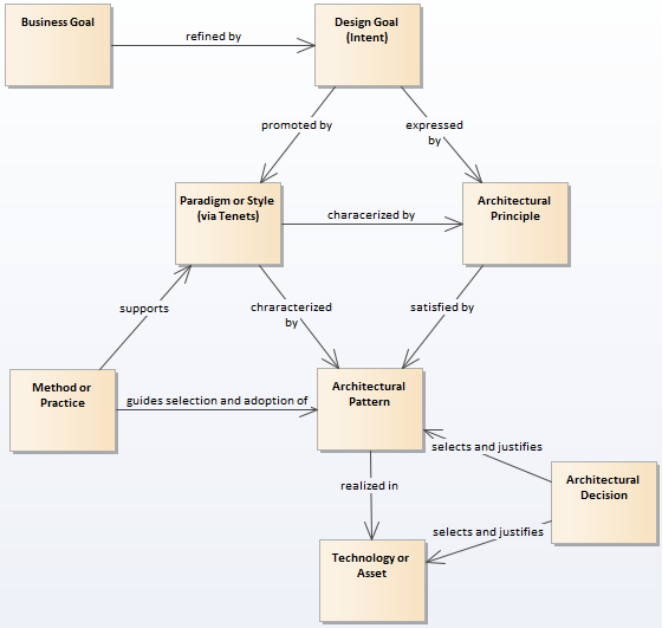
Agile Principles [AgileManifesto]:

1. Our highest priority is to satisfy the customer through **early and continuous delivery** of valuable software.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.

3. **Deliver** working software **frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. **Business** people and **developers** must **work together daily** throughout the project.
5. Build projects around motivated individuals. Give them the **environment** and support they need, and **trust** them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.
7. Working **software** is the **primary measure** of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to **maintain a constant pace** indefinitely.
9. Continuous **attention** to **technical excellence** and **good design** enhances agility.
10. **Simplicity**--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from **self-organizing teams**.
12. At regular intervals, the **team reflects** on how to become more effective, then **tunes and adjusts** its behavior accordingly.

Architecture is required in agile software development. Someone has to keep an eye on critical requirements and pay attention to technical excellence and design!

3.5 Essentials (Principles, Patterns, Decisions)



3.6 Tips and Tricks

Approach software architecture design as a **lean process**:

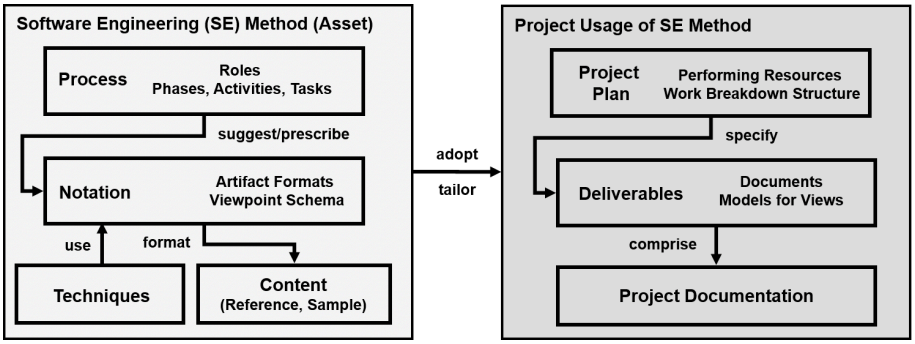
- Start from functional requirements and NFRs, particularly SMART software quality attributes (quality attribute scenarios, quality stories)
- Identify top-level components
- Continue to decide what is architecturally (most) significant – when drawing diagrams, when making design decisions
- Find architectural patterns which resolve the forces underneath the NFRs

- Make conscious pattern selection decisions and follow-on decisions about technologies and products – and document these decisions

- Document and model with a sense of pragmatism - If in doubt, leave it out (or: always model with a target audience in mind)

- Component names should carry domain semantics (and be consistent throughout viewpoints and models)

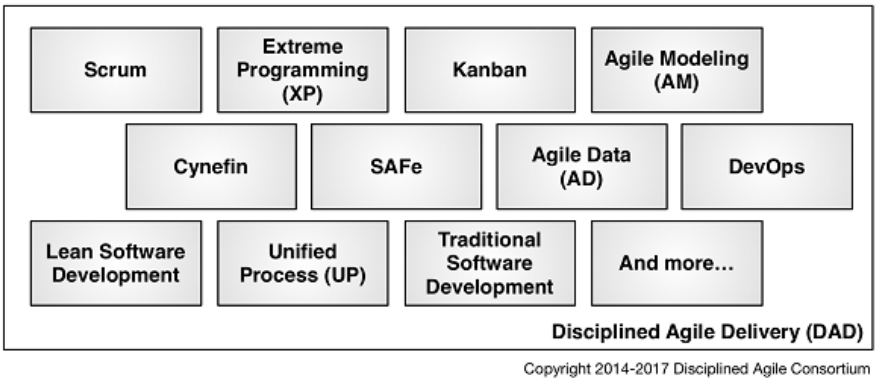
4 Methods



4.1 General Design

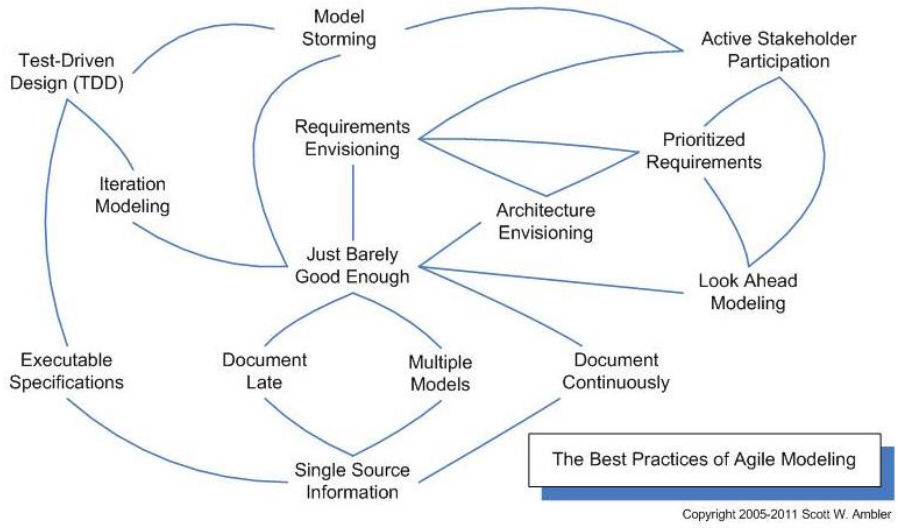
4.1.1 Disciplined Agile Delivery (DAD)

"Umbrella Method" or toolbox with agile disciplines and methods.



4.1.2 Agile Modeling (AM)

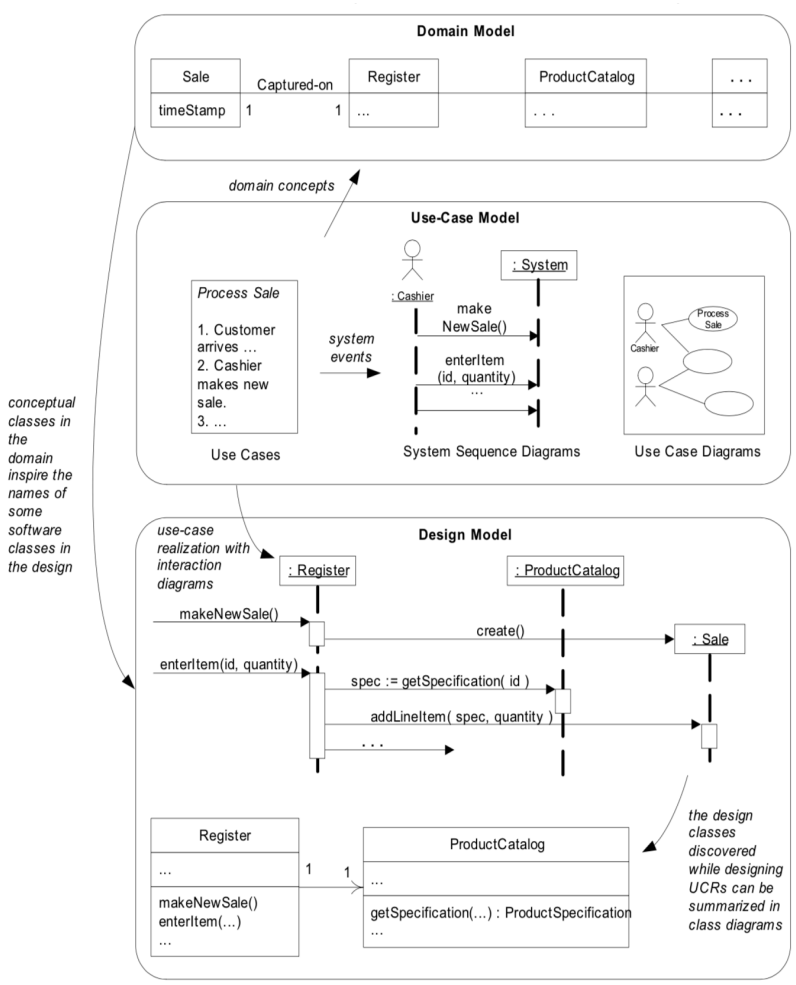
AM is a component and at the same time the predecessor of DAD.

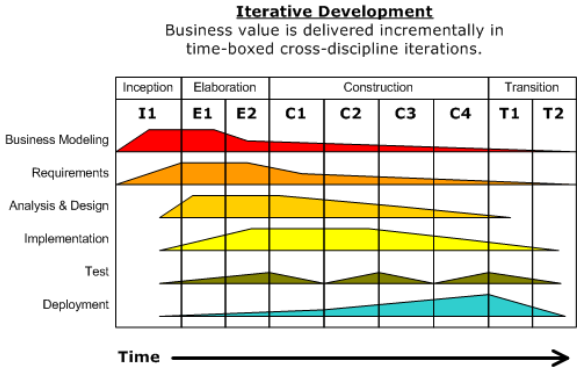


Important thought: Let's keep the modeling baby but throw out the bureaucracy bathwater. ;-)

4.1.3 Object-Oriented Analysis and Design (OOAD)

OOAD is part of the Unified Process (RUP / OpenUP).





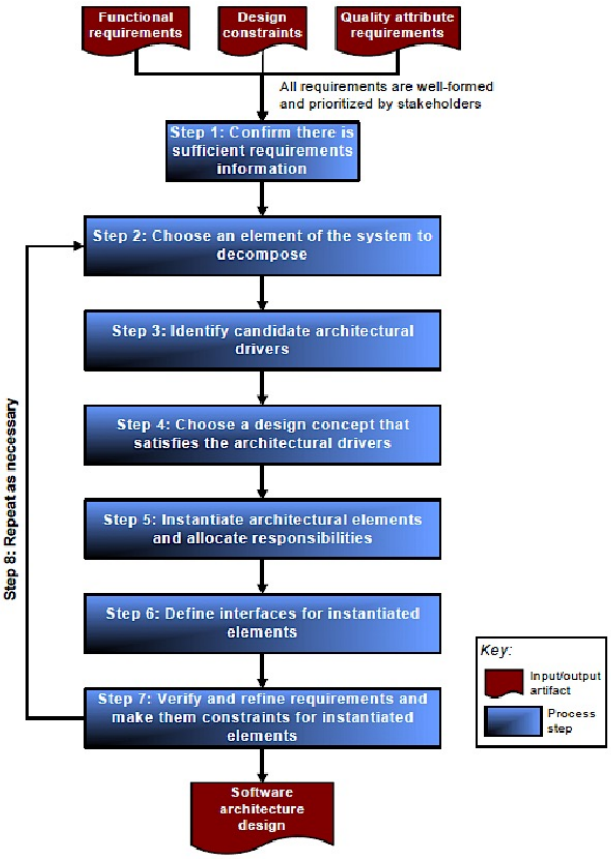
4.2 Architecture Design

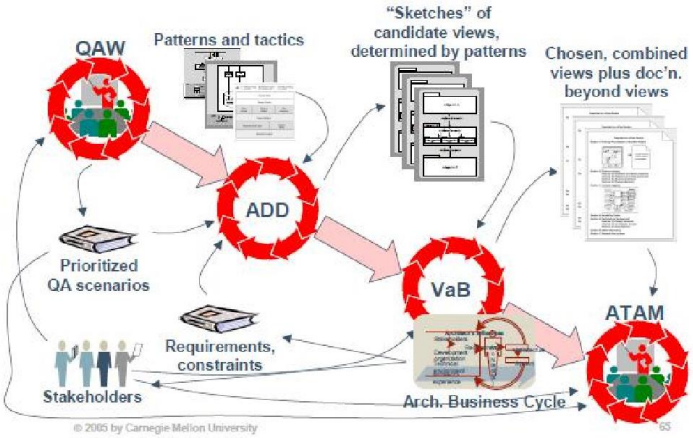
4.2.1 Agile Architecting

- Specify **ASRs** in a **SMART** way, e.g., in the form of **quality stories**.
- Make conscious **decisions** and provide **rationale**, e.g., in **Y-statements**.
- Model **context**, **containers**, **components**, **classes** (**C4**) – but not more.
- Apply an Architecturally Evident Coding Style (**AECS**).
- Architect the time dimension, e.g. in the form of event tables and roadmaps.
- Practice **architectural refactoring**, revisiting the team's decisions along the way.
- Consider the **SOA style** and its **microservices implementation** approach.

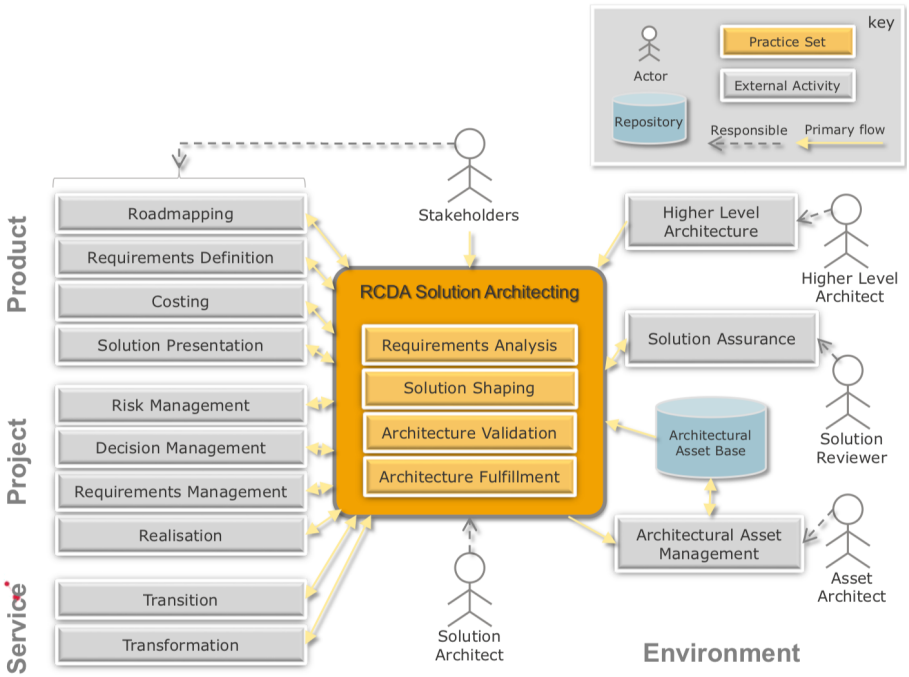
4.2.2 Attribute-Driven Design (ADD)

QAs are identified, defined and prioritized during the initial Quality Attribute Workshop (QAW) in collaboration with stakeholders of the project. QAs are the main driver of the architecture.





4.2.3 Risk- and Cost-Driven Architecture (RCDA)



5 Analysis: Non-Functional Requirements (NFRs)

5.1 SMART

- **S**pecific: Which feature or part of the system should satisfy the requirement?
- **M**easurable: How can testers and other stakeholders find out whether the requirement is met (or not)? Is the requirement quantified?
- **A**greed: Do all affected internal and external stakeholders agree on the “S” and the “M” wording?
- Realistic: Is it technically and economically feasible to achieve the “M” measure in the context of all features or system parts specified under “S”?
- **T**ime-bound: When should the NFR meet the “M” measure, is there a growth path from iteration to iteration?

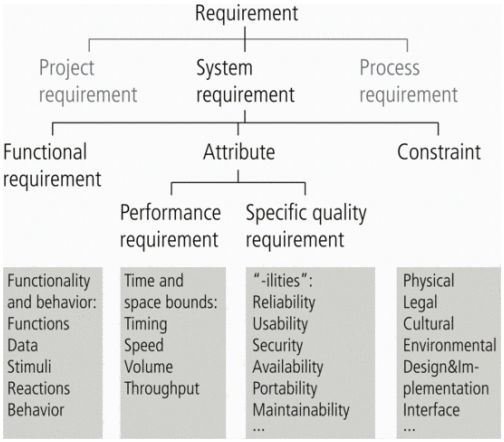
"M" and "S" are especially relevant when defining NFRs.

5.1.1 Templates / Examples

ASR	Specific (Y/N)?	Measurable (Y/N)?	Rationale for Answers	Improvement (if needed)
R1	[Y/N]	[Y/N]	(answer to question from above)	(required change or n/a)
Rn	[Y/N]	[Y/N]	...	...

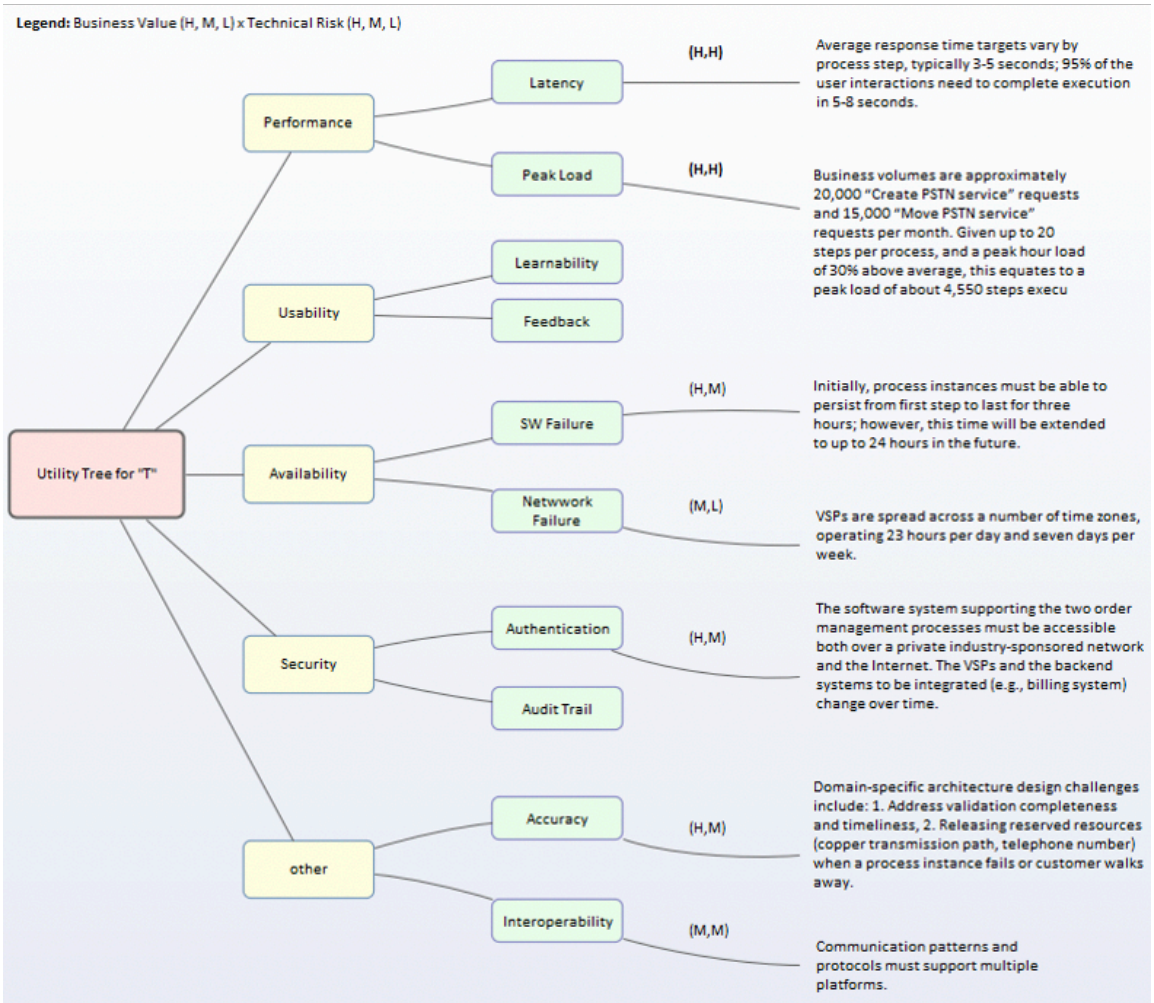
5.2 FURPS (+)

• **F**unctionality • **U**sability • **R**eliability • **P**erformance • **S**upportability	Plus (+): • Design constraints • Implementation constraints • Physical constraints • Interface constraints



5.3 Quality Utility Tree

The utility tree offers a top-down approach to **refine** and **prioritize** QAs.



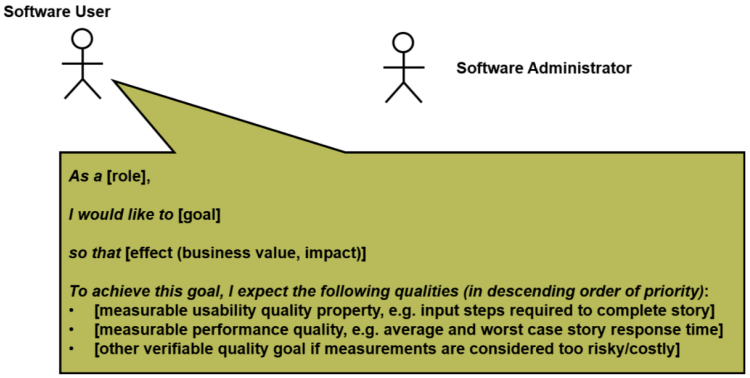
5.4 Quality Attribute Scenario (QAS)

Note that we are still in analysis mode when writing QASs. The **response** should therefore not describe or even prescribe a design, but specify a requirement (**postcondition**). The response measure brings the "M" in SMART. The "S" is achieved by the stimulus and the stimulus source.

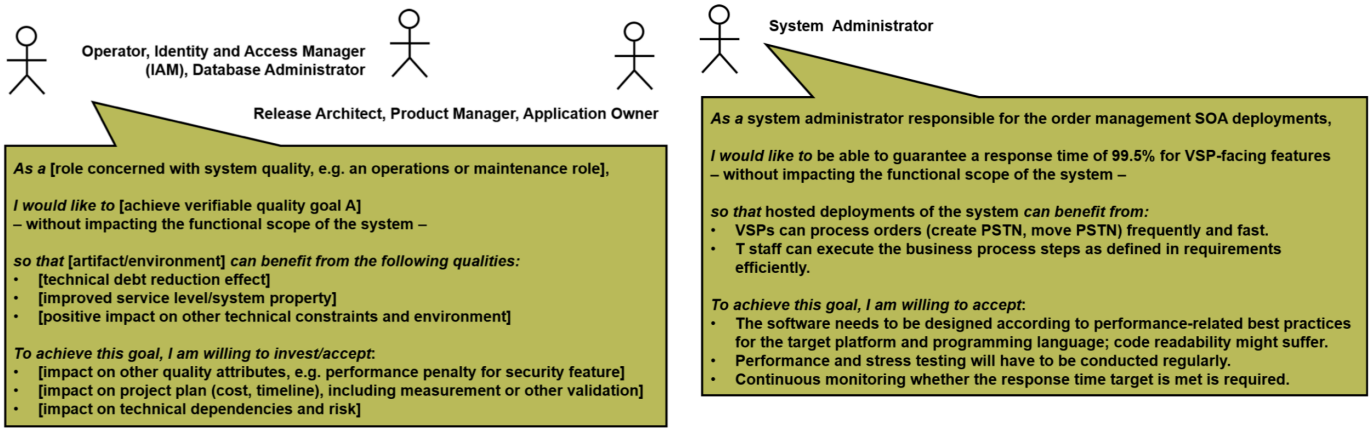
Scenario for Order Management SOA at "T" (Business-to-Business), NFR #5		
Scenario(s)		Response Time in Web Channel (serving VSPs)
Business Goals		Drive down cost of operations by interacting with VSPs efficiently
Relevant Quality Attributes		Performance (response time), scalability
Scenario Components	**Stimulus**	Order placed by VSP (system or end user)
	Stimulus Source	External to system (see functional requirements and architecture overview diagram: "Create PSTN", "Move PSTN")
	Environment (Worst Case)	Normal operation, at runtime
	Artifact (If Known)	Entire system, all logical layers and all physical tiers (including customer database in the backend)
	Response	Orders are accepted immediately, order number (identifier) is returned
	Response Measure	Order acknowledgment is sent in 3 seconds (or less). This performance is desired but not guaranteed to VSPs in any Service Level Agreement (SLA),
Questions		Can errors be identified and handled properly in this timeframe?
Issues		Backend systems might not have sufficient processing power

Scenario Portion	[Wie heisst das Szenario, um welche Art NFR geht es?]
Source	• Wer startet das Szenario (Quelle)? Woher kommt die Anforderung? [Bsp. Benutzer/Aktor, systeminterner Trigger]
Stimulus	• Was macht die Source, um das Szenario zu starten (Ereignis, Trigger/Auslöser)? [Use Case oder Aktivität im Geschäftsprozess]
Artifact	• Wo findet das beschriebene Response-Verhalten, das von der Source stimuliert worden ist, statt? Welcher Systemteil ist betroffen? [Ganze Anwendung oder einzelne Komponente]
Environment	• Wann kann das spezifizierte Response-Verhalten beobachtet werden? Um welche Umgebung/welchen Systemzustand geht es? [Laufzeit oder Phase im Software Engineering/Wartung]
Response	• Wie reagiert die Anwendung qualitativ auf den Stimulus? Was soll gemessen werden? [nach aussen sichtbares Verhalten, Messgrösse, noch kein Design]
RespMeasure	• Wie kann die Response quantifiziert werden, um sie überprüfbar zu machen? Welches Messergebnis wird angestrebt? [Messeinheit, konkrete Zahlenangabe im Kontext der vorherigen QAS-Template-Elemente wie Source, Stimulus, usw.]

5.5 Quality User Story



Alternative template with example:



5.6 Tips and Tricks

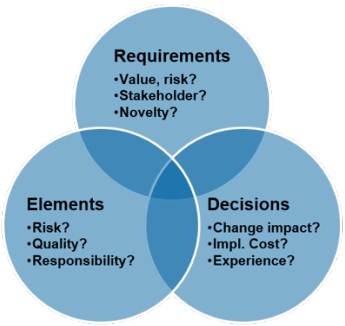
- Make **assumptions** (or use numbers from previous projects as starting point)
- Wrong numbers are better than none (can be corrected if needed)
- Work with **relative statements** (e.g. "usability should not be worse than that of the system to be replaced")
- **Ask** and **challenge** responses from stakeholders
- "Not doing" it should be a conscious decision with rationale

6 Analysis: Architecturally Significant Requirements (ASRs)

6.1 Motivation

- Scope work
- Stay focused
- Meet the NFRs
- "Avoid waste"

6.2 Identification



6.2.1 Requirement Criteria

1. The requirement is directly associated with **high business value** and/or **technical risk**.
2. The requirement is a concern of a **particularly important stakeholder** (for instance, an influential external stakeholder such as project sponsor or compliance auditor); it is governed by an official Service Level Agreement (SLA).
3. The requirement has a **first-of-a-kind character**: already existing components have different responsibilities and therefore do not address it; the **team has never built** a component that satisfies this particular requirement.
4. The requirement has **Quality-of-Service** (QoS) **characteristics that deviate** from those already satisfied by the **evolving** architecture substantially (e.g., by an order of magnitude).
5. The requirement has **caused critical situations**, **budget overruns** or **client dissatisfaction** in a previous project with a similar business/technical context.

6.2.2 Element Checklist

1. The element relates to **some critical functionality** of the system, e.g., monetary transactions
2. The element relates to **some critical property** of the system, e.g., reliability
3. The element relates to a particular **architectural challenge**, e.g., external system integration
4. The element is associated with a particular **technical risk**
5. The element relates to a **capability** that is considered to be **unstable**
6. The element relates to some **key element** of the solution, e.g., login mechanism

6.2.3 Decision Questions

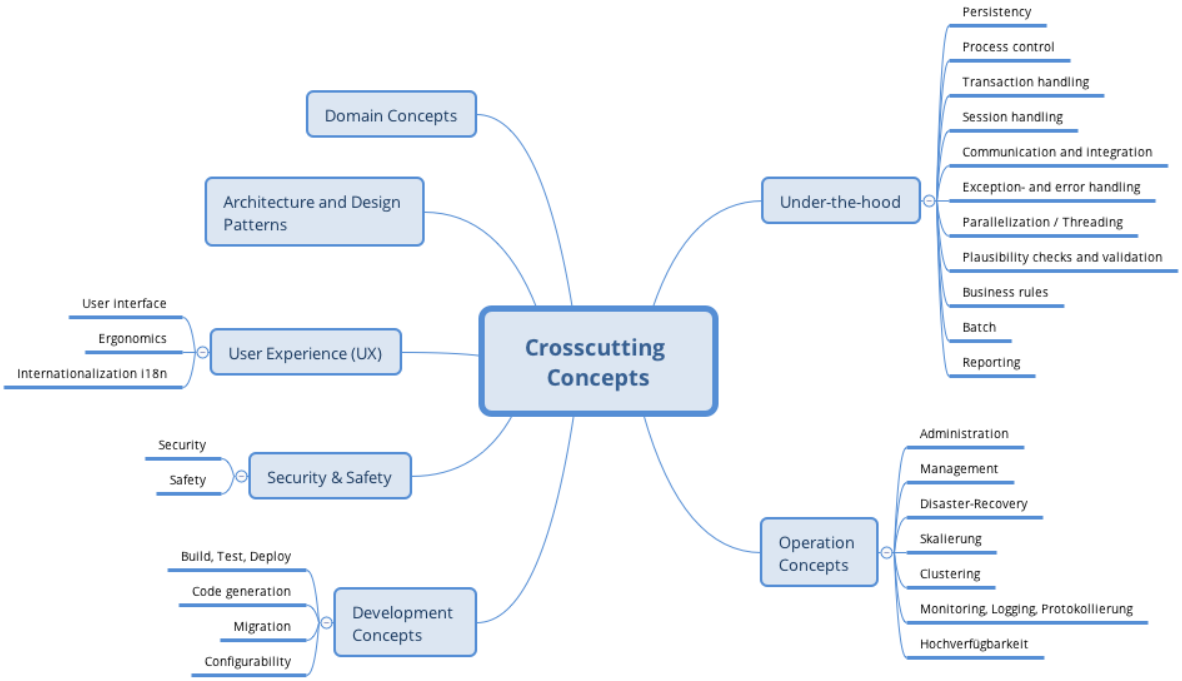
1. Is the decision **hard to change** later?
2. Is the decision **expensive** to implement/execute upon?
3. Are **demanding**, **qualitative requirements** stated (high security level, high availability, high performance)?
4. Are requirements **difficult to map** to existing (solutions, experiences)?
5. Is the **experience** in the solution space **weak** (in the team)?

6.3 Templates / Examples

	low difficulty/novelty (internal concerns)	*high difficulty/novelty (internal concerns)*
high cost/risk (external stakeholder concern)	medium to high significance	high significance
low cost/risk (external stakeholder concern)	low significance	low to medium significance

Requirement	Score	Explanation (Rationale)
"Data retention policy of 10 years required to achieve regulatory compliance"	*high*	Violation of this requirement would lead to fines; redesign might require change of database technology and hosting model
"Technical constraint to prefer a particular messaging middleware and backend API"	*medium-high*	If a different architectural element is chosen, additional licensing and training cost arise; standardized APIs promise interoperability (which has to be proven)
"Deployment pipeline automation"	*low-medium*	Needed to be able to test, deploy, release often, but not something end users and project sponsors are willing to care about and pay for usually (so team-internal concern)
"Name of Java class wrapping access to backend"	*low*	Decision that is not visible to external stakeholders; simple to change in IDEs that support refactoring

7 Synthesis: Solution Strategy ("Making Big Decisions")



7.1 Typical Decisions

- **Buy vs. build** with options such as Do-it-Yourself (DIY) vs. Open Source Software (OSS) participation vs. procurement of Commercially-of-the-Shelf (COTS) software
- **Client-server cuts** and architectural style(s); inclusion or exclusion of external interfaces to be consumed
- Choice of **programming language** (even in agile age, with autonomous teams and polyglot programing)
- Choice of **database** vendor (and other complex **middleware**)
- Choice of **method** or method mix (practice collection)

Artifact	Decision Topic	Recurring Issues (Decisions Required)
Enterprise architecture documentation	IT strategy	Buy vs. build strategy, open source policy
	Governance	Methods (processes, notations), tools, reference architectures, coding guidelines, naming standards, asset ownership
System context	Project scope	External interfaces, incoming and outgoing calls (protocols, formats, identifiers), service level agreements, billing
Other viewpoints	Development process	Configuration management, test cases, build/test/production environment staging
	Physical tiers	Locations, security zones, nodes, load balancing, failover, storage placement
	Data management	Data model reach (enterprise-wide?), synchronization/replication, backup strategy
Architecture overview diagram	Logical layers	Coupling and cohesion principles, functional decomposition (partitioning)
	Physical tiers	Locations, security zones, nodes, load balancing, failover, storage placement
	Data management	Data model reach (enterprise-wide?), synchronization/replication, backup strategy
Architecture overview diagram	Presentation layer	Rich vs. thin client, multi-channel design, client conversations, session management
	Domain layer (process control flow)	How to ensure process and resource integrity, business and system transactionality
	Domain layer (remote interfaces)	Remote contract design (interfaces, protocols, formats, timeout management)
	Domain layer (component-based development)	Interface contract language, parameter validation, Application Programming Interface (API) design, domain model
	Resource (data) access layer	Connection pooling, concurrency (auto commit?), information integration, caching
	Integration	Hub-and-spoke vs. direct, synchrony, message queuing, data formats, registration

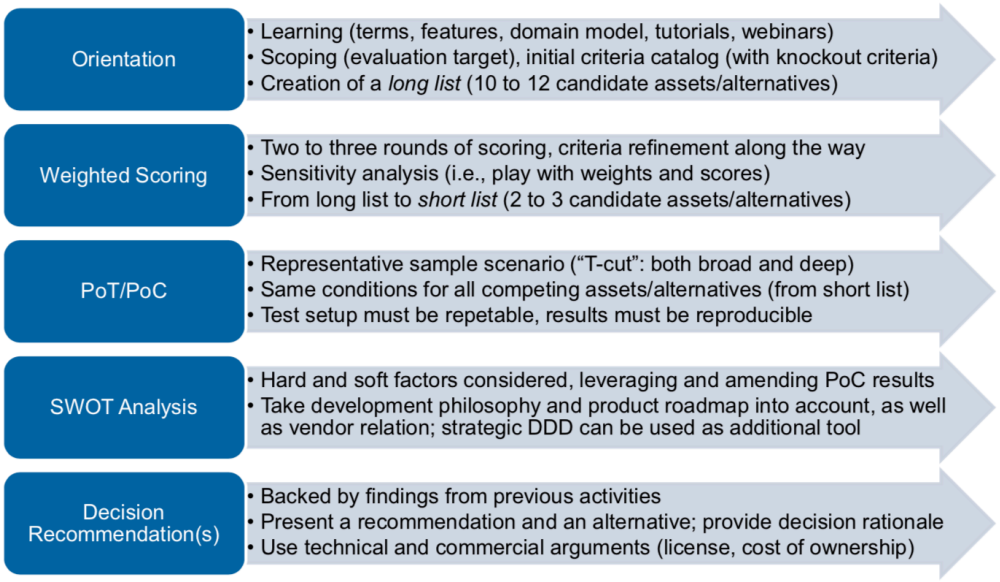
Artifact	Decision Topic	Recurring Issues (Decisions Required)
Logical component	Security	Authentication, authorization, confidentiality, integrity, non-repudiation, tenancy
	Systems management	Fault, configuration, accounting, performance, and security management
	Lifecycle management	Lookup, creation, static vs. dynamic activation, instance pooling, housekeeping
	Logging	Log source and sink, protocol, format, level of detail (verbosity levels)
	Error handling	Error logging, reporting, propagation, display, analysis, recovery
Components and connectors	Implementation technology	Technology standard version and profile to use, deployment descriptor settings (QoS)
	Deployment	Collocation, standalone vs. clustered
Physical node	Capacity planning	Hardware and software sizing, topologies
	Systems management	Monitoring concept, backup procedures, update management, disaster recovery

7.2 Popular Patterns and Styles

- Client / Server
- Layers, Pipes and Filters, Broker
- Patterns of Enterprise Application Architecture (PoEAA)
- Domain-Driven Design (DDD)
- Enterprise Integration Patterns (EIP) such as Messaging, Channel
- Adapter, Observer and other design patterns (GoF)

Comprehensive collection of patterns (description, illustration, sample code): http://java-design-patterns.com

7.3 Evaluation Process (Buy vs. Build)



Be aware that bias always exists, use PoC and/or qualitative SWOT analysis next to plain feature scoring!

7.3.1 SWOT Analysis

7.3.2 PoT vs. PoC

PoC has a more specific scope than **PoT** and investigates deeper. They reside on different levels of abstraction and refinement (i.e. platform-independent level vs. platform-specific level). PoC scoping always is a challenge and requires advise and expertise because you have to have a **representative subset of use cases that unveil ASRs**.

7.3.3 DDD as Support

Context Mapping from Tactic DDD can support

- **COTS Evaluation**
- **Buy vs. Build Decisions**

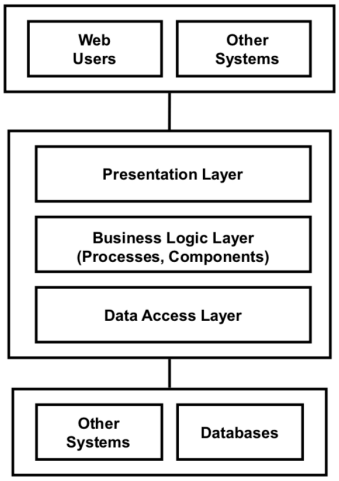
It can help to focus the evaluation effort on the **Core Domain**, and getting information about the functional coverage and the domain model of the candidates.

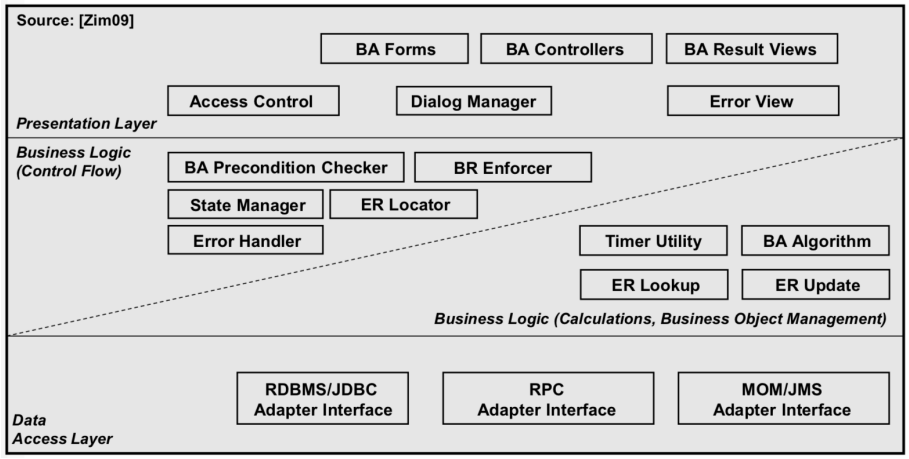
8 Synthesis: Architectural Style

Architectural styles are **sets of principles and patterns** aligned with each other to **shape an application** and make designs recognizable and design activities repeatable:

- **principles** express architectural design intent
- **patterns** adhere to the principles and are commonly occurring (proven) in practice

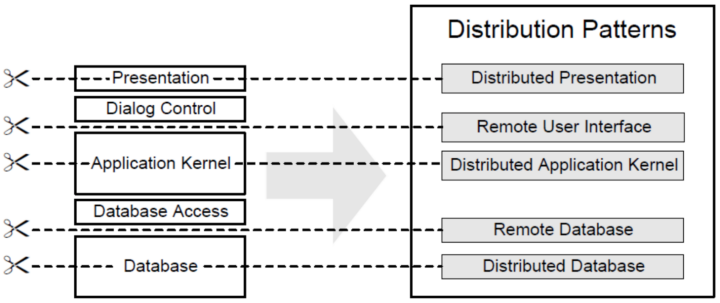
8.1 Style: Layers

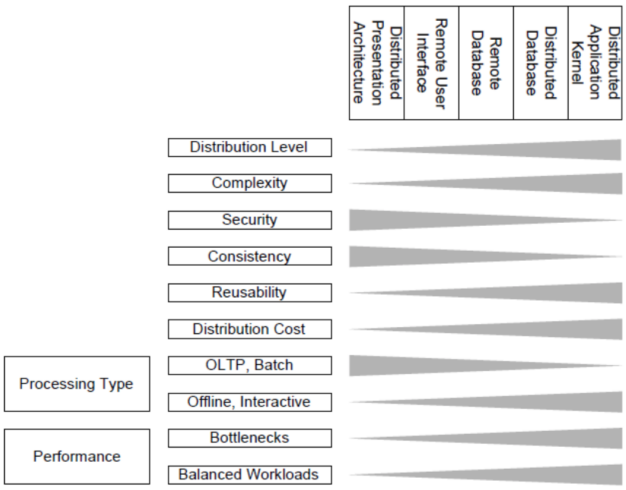




8.2 Style: Client / Server

8.2.1 Cuts





5 cuts to chose from, many architectural drivers:

- 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

Examples for different cuts:

Distributed Presentation: Thin client on tablet or mobile phone, for instance developed with a cross- platform tool based on HTML.

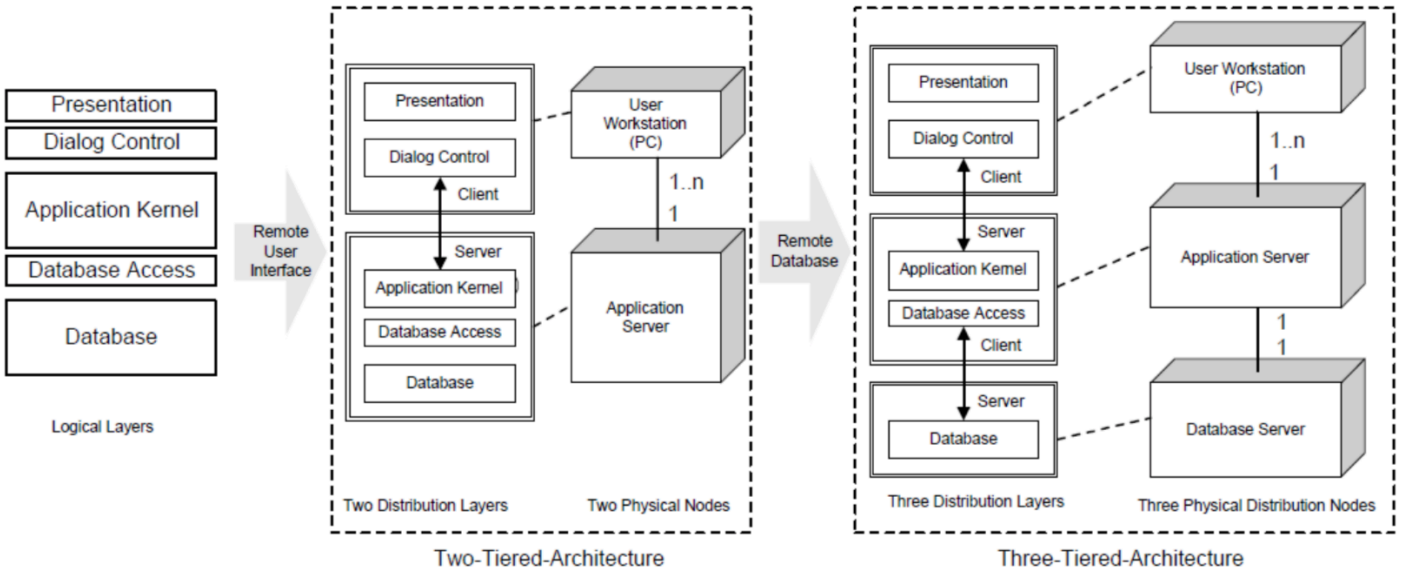
Remote User Interface: App on mobile phone talking to a data-oriented server backend (via HTTP and CRUD resources)

Distributed Application Kernel: App on mobile phone talking to a function- or object-oriented server backend (via RESTful HTTP interfaces to compute resources)

Remote Database: Remote JDBC

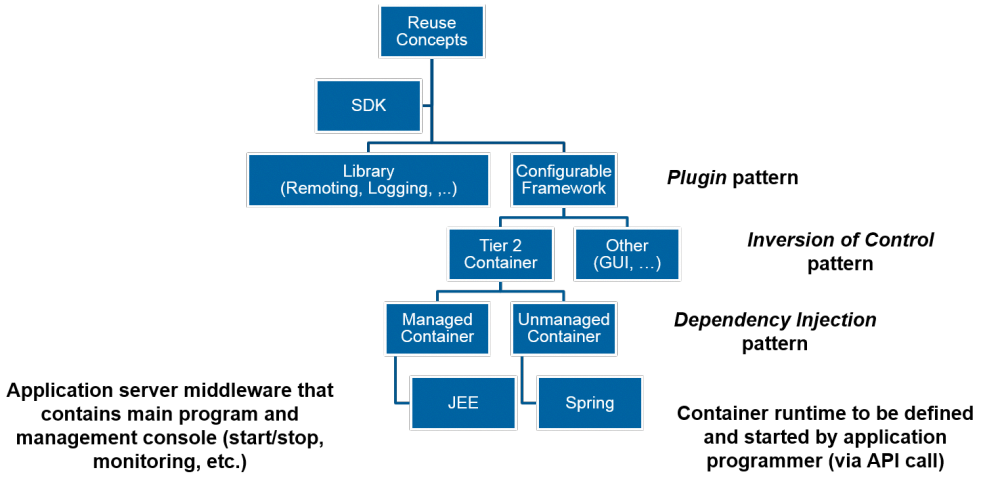
Distributed Database: Cassandra and other highly distributed databases supporting eventual consistency

8.2.2 Two-Tier vs. Three-Tier

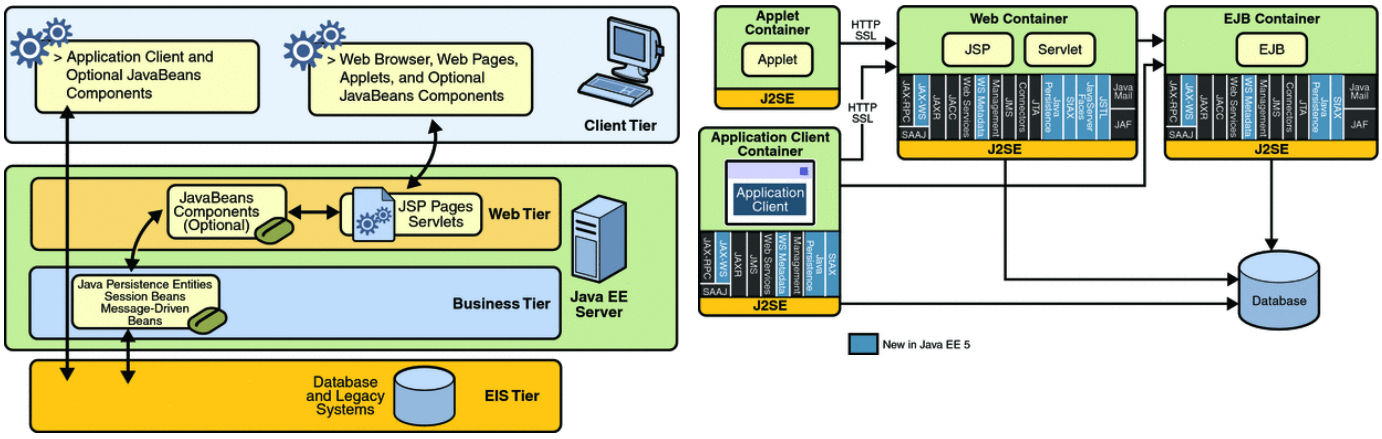


8.4 Reference Architecture (RA)

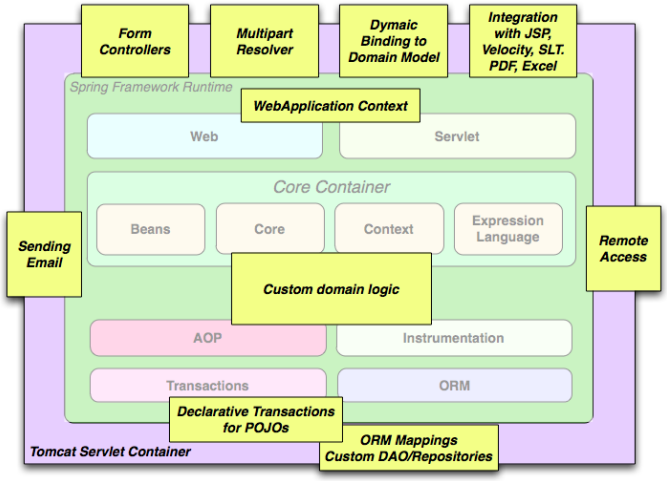
Accelerate design, but have to be refined and adopted to bring them into project context.



8.4.1 Managed Container: Java Enterprise Edition (JEE)

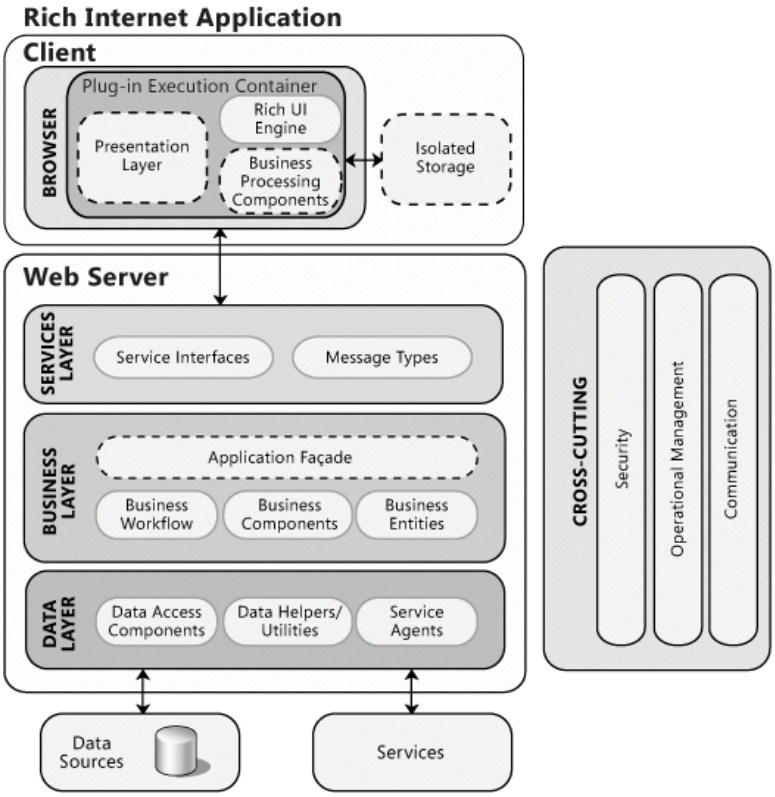


8.4.2 Unmanaged Container: Spring Framework



Spring Annotation	Meaning
`@SpringBootApplication`	Convenience annotation, bundles @Configuration, @EnableAutoConfiguration and @ComponentScan
`@Component`	Classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
`@Autowired`	Same as @Inject in CDI (JEE); replaces need for getters/setters
`@Controller`	To indicate C in MVC pattern (presentation layer, dialog control)
`@RequestMapping`	URI pattern for of RESTful HTTP resources (JAX-RS)
`@Transactional`	System transaction (ACID), container is transaction manager
`@EnableCaching`	Responsible for registering the necessary Spring components that power annotation-driven cache management
`@Table`	Spring Data JPA, steers O/R mapping
`@Entity`	Spring Data JPA, steers O/R mapping
`@Query`	Spring Data JPA, associated query with method
`@Profile`	Conditional bean registration (OS level, development stage, etc.)

8.4.3 Microsoft Application Architecture Guide



9 Synthesis: C4 Model

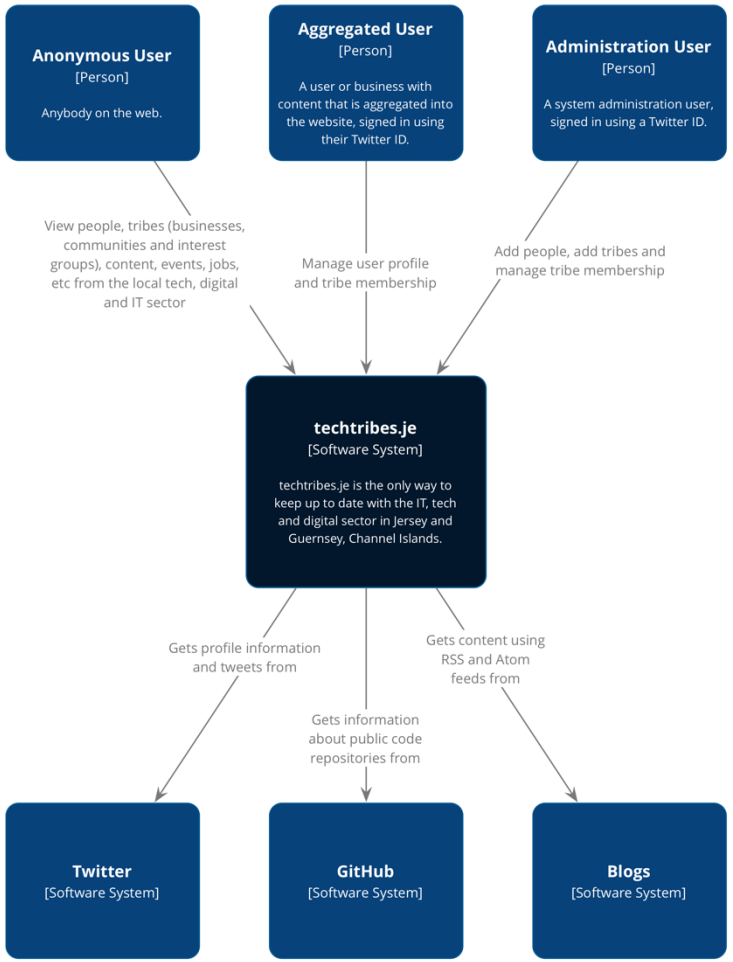
C4 model only defines a diagram hierarchy, **not a process or technique**.

9.1 Classes

Let design arrive at **code level** (in a diagram or directly in code).

9.2 (System) Context Diagram

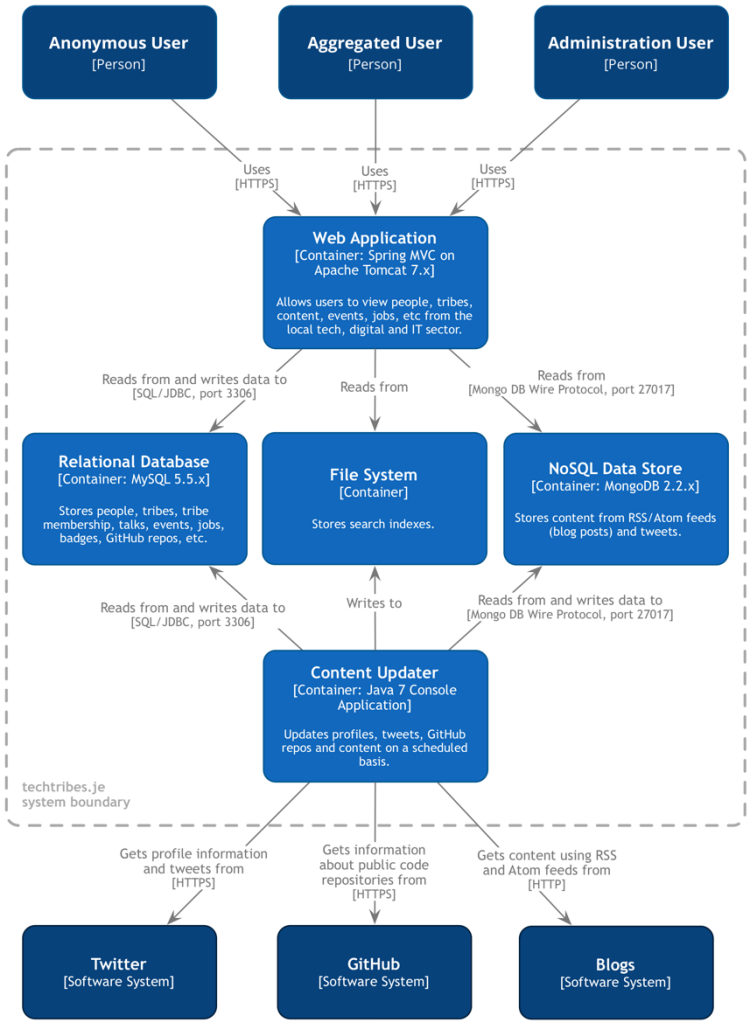
Provides a black box view (external dependencies) and shows **information provided and consumed** from/to **interfaces** and **users.**



This diagram is important for scope control!

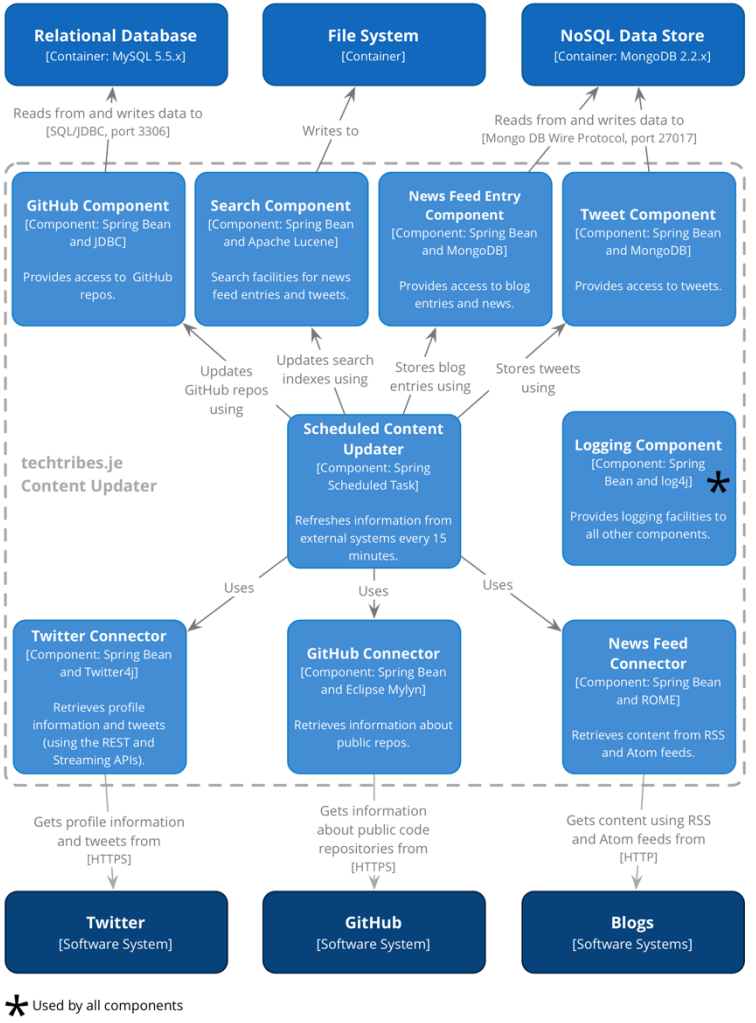
9.3 Containers Diagram

Gives an architectural overview (on solution strategy) and shows **high-level technology choices**, for instance about middleware (frameworks, application servers, databases).



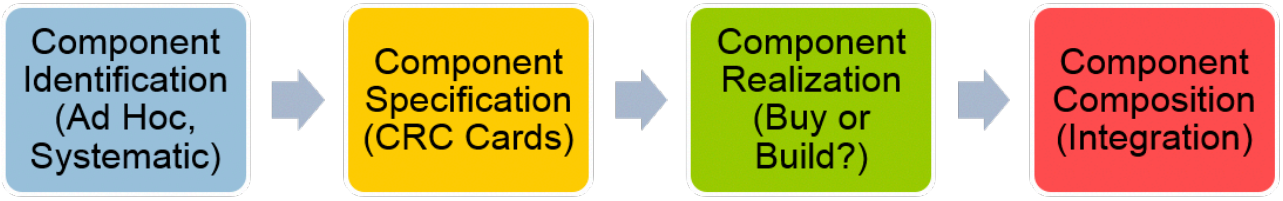
9.4 Components Diagram

Refines the overview to meet ASRs by showing **structural building blocks** and their **interactions** (including **responsibilities** like "holding" or "doing").



10 Synthesis: Component Modeling

Components in solution strategy are **candidate components** only. They are subject to **continuous refinement** and **consolidation** efforts (also known as architectural refactoring).



Candidate Component	Architectrual element 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.
Implementation Component	Groups one or more classes and provides an interface that hides their implementation details.
Deployment Unit	Bundles one or more implementation component(s) to be hosted on one or more nodes.

10.1 Component Identification

- **Input:**
  - FRs, NFRs/QAs, architectural vision statement (arc42 table, X-statement)
    - Plus analysis-level domain model (e.g. UML class diagram) as taught in SE1/SE2
1. **First iteration: Find *Candidate Components* (CanCos)**
   1. One *channel component* per actor/consuming external system in context
   2. One *component per layer* per feature (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

10.2 Component Specification

Responsibility-Driven Design (RDD) provides useful **role stereotypes**:

- **Information Holder**: knows and provides information
- **Structurer**: maintains relationships between objects and information about those relationships
- **Service Provider**: performs work and in general offers services
- **Controller**: makes decisions and closely directs others' actions
- **Coordinator**: reacts to events by delegating tasks to others
- **Interfacer**: transforms information and requests between distinct parts of a system.
  User interfacers translate requests from the user to the system (and vice versa)
  External interfacers usually "wrap" other system APIs

10.2.1 CRC Cards

Used for **specification** and **refinement** of candidate components:

- **C**lasses
- **R**esponsibilities
- **C**ollaborators

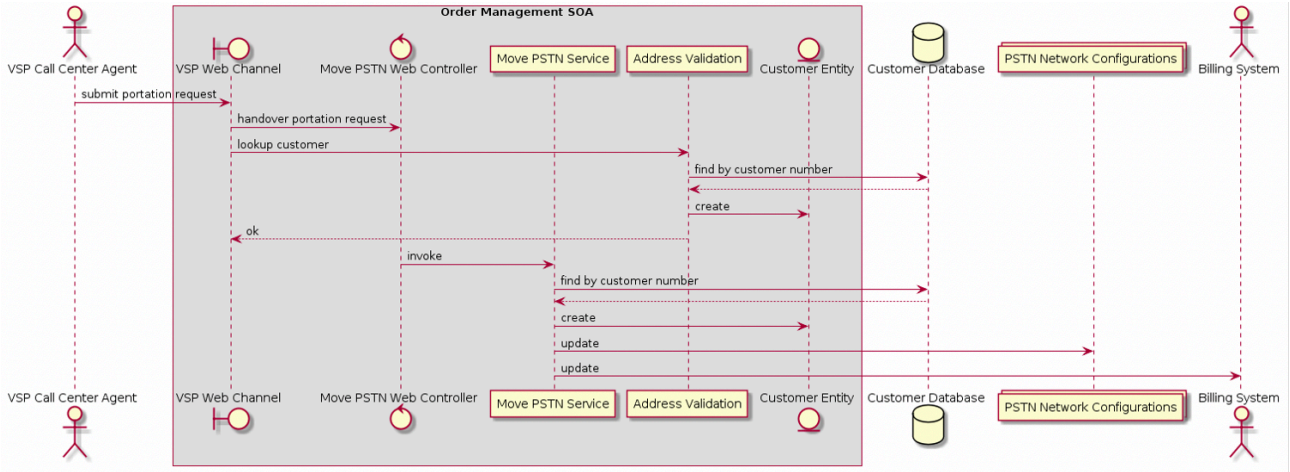
Component: **[Name of Component (with optional Role Stereotype)]**	
Responsibilities: • [What is this component capable of doing (services provided)?] • [Which data does it deal with?] • [How does it do it job in terms of key system qualities?]	*Collaborators (Interfaces to/from):* • [Who invokes this component (service consumers)?] • [Whom does this component call to fulfill its responsibilities (service providers)?] • [Any external connections (both active and passive)?]
Candidate implementations technologies (and known uses): • [Which technologies, products (commercial, open source) and internal assets can realize the outlined component functionality (responsibilities)?]	

Component: **Business Logic Layer (RDD Role: Controller, Coordinator)**	
Responsibilities: • Receives service requests • Authenticates request originator, authorizes service request • Keeps track of processing state (workflow, not page flow!), manages transaction boundaries • Accesses business objects • Performs calculations • Updates business objects and application state • Sends response • Logs layer activities in audit trail	*Collaborators (Interfaces to/from):* • Presentation layer • Data access layer • Cross-cutting infrastructure features (e.g., security subsystem) • Component container (Inversion of Control)
Candidate implementations technologies (and known uses): • Service layer, process layer in "T" case study • BLL in PQG case study • domain.model package in DDD Sample Application • HSR enterprise applications such as unterricht.ch, AVT • Many more	

10.3 Component Dynamics

Both normal operations and error situations should be investigated when designing component dynamics. Dynamics are important for design review based on FURPS and other criteria:

- Performance
- Reliability / Error handling (e.g. retries, fallback solution)
- Isolation and caching (e.g. for database access)
- Timing (e.g. timeouts due to slow network)
- Lifecycle management (e.g. instance pool per application, session or request)
- Security



Dynamics can be visualized as UML sequence diagram or written down as list of interaction steps.

10.4 Tips and Tricks

- Write CRC cards in a **SMART** way as well

- **Names** should communicate what application/architecture are about (e.g. Domain Concept + Architectural Role / Pattern)

- Each **outgoing collaboration** relationship of a CRC card must **correspond to an incoming** one elsewhere in system or its context (e.g. Service Consumer and Service Provider)

- Model on **same level of detail** on all CRCs cards and find a good middle way ("If in doubt leave it out")

- **Sunny days** and **rainy days** to be taken into account

11 Synthesis: Domain-Driven Design (DDD)

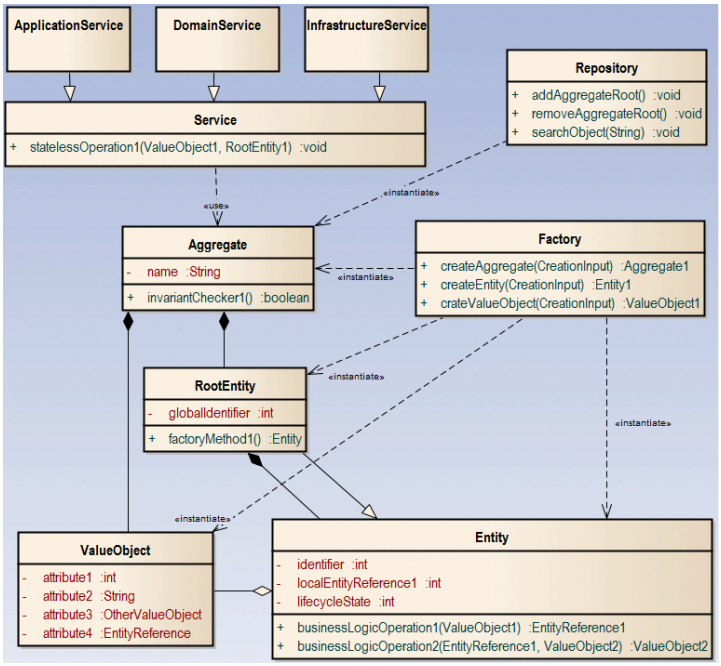
11.1 Tactic DDD

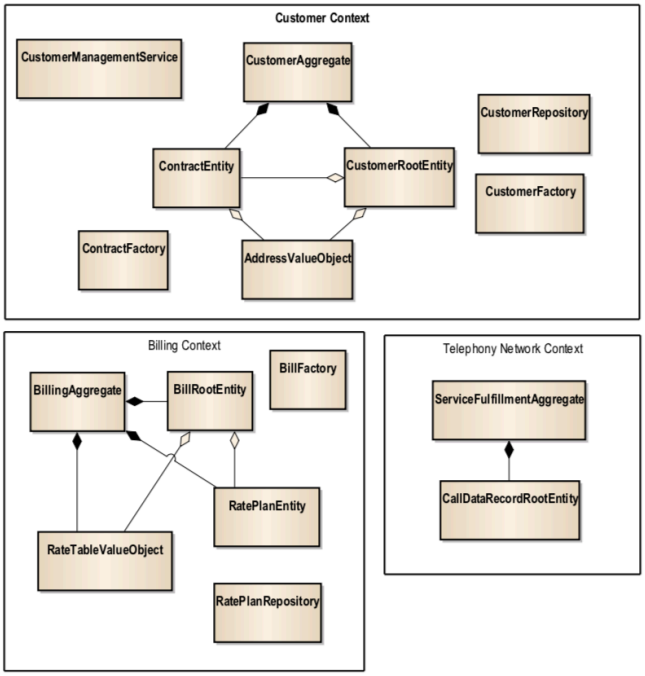
DDD Core Patterns:

- Ubiquitous Language ("language structured around domain model and used by all team members")
- Entity (see section 15.4)
- Value Object (see section 15.4)
- Service (see section 15.4)
- Aggregate (see section 15.5)
- Repository (see section 15.6)
- Factory (see section 15.6)

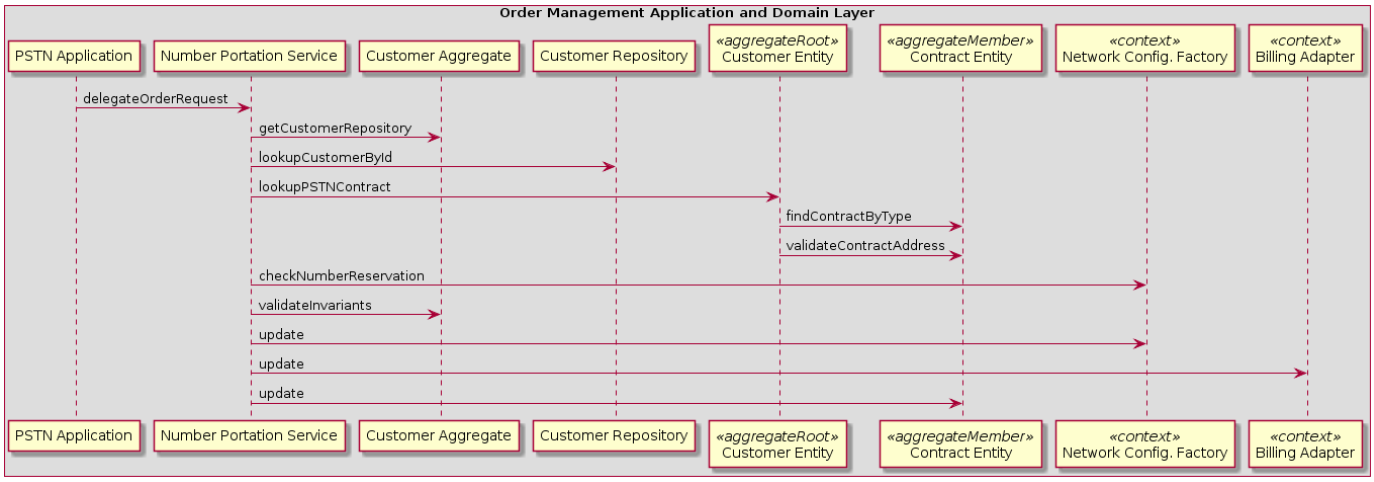
DDD Layers:

- Interface Layer
- Application Layer (BLL refinement)
- Domain Layer (BLL refinement)
- Infrastructure Layer

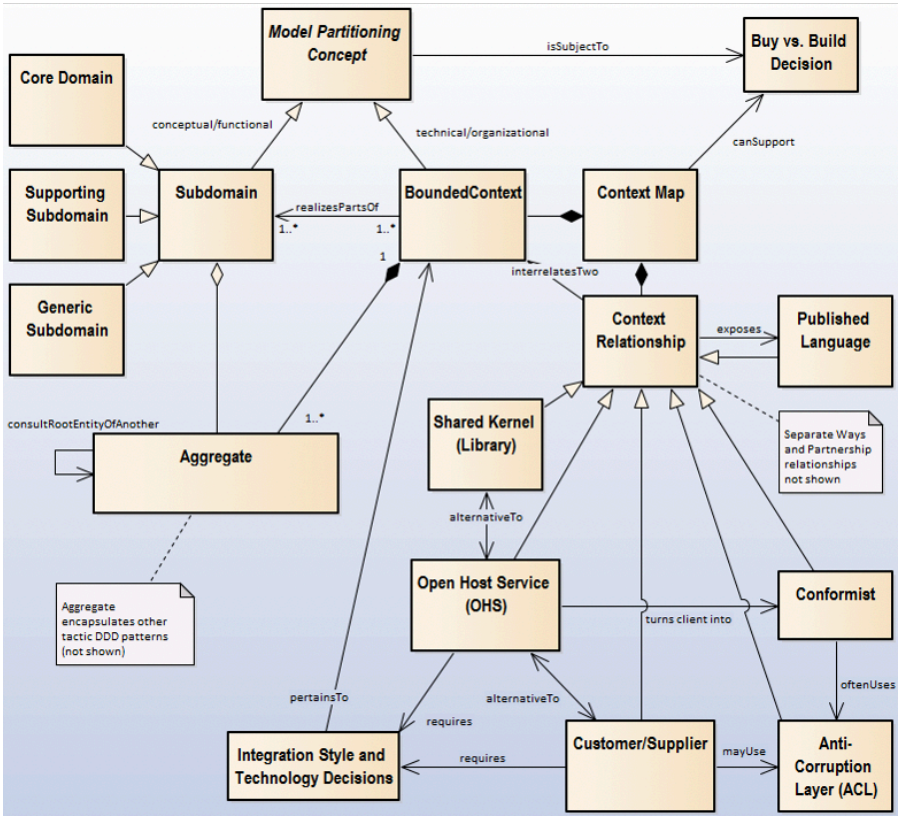


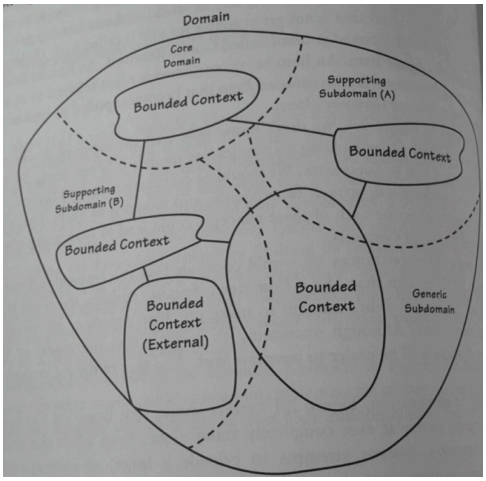


Example for interaction:



11.2 Strategic DDD





11.2.1 Subdomain

There are 3 subsets inside a domain:

- Core Domain (most important)
- Supporting Domain
- Generic Domain

See section 15.7 for additional information as well as the illustration above.

11.2.2 Bounded Context (BC)

- A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable
- 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
- A boundary around source code

Relationship Types (do not exclude, but complement each other):

- **Published Language (PL)**: 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.

- **Open Host Service (OHS)**: 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.

- **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 bounded context uses the services of another but is not a stakeholder to that other bounded context. As such it uses ”as-is” (conforms to) the protocols or APIs provided by that bounded context.

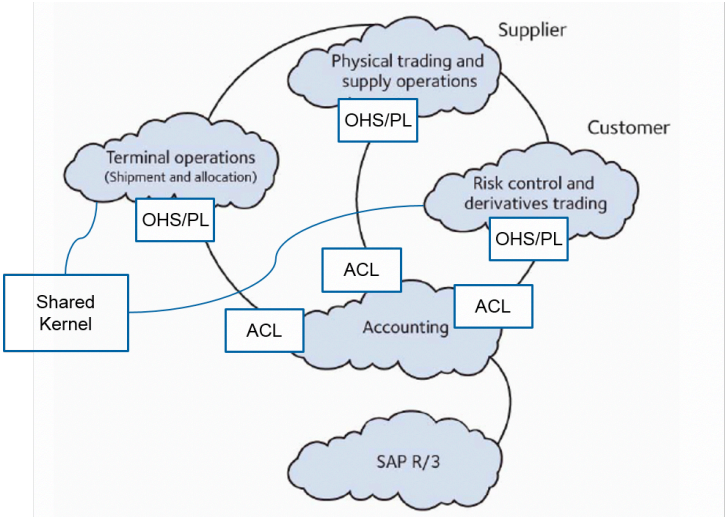
- **Anti-Corruption Layer (ACL)**: One bounded context uses the services of another and is not a stakeholder, but aims to minimize impact from changes in the bounded context it depends on by introducing a set of adapters.

See section 15.7 for additional information.

11.2.3 Context Map

Questions to be answered while drawing the map:

- **Topology**: Local vs. Remote?
- **Visibility** (of collaboration/communication partners to each other)?
- **(A)symmetry** of relationship?
- Amount of **control** and **influence** for client/constumer?



11.3 Strategy vs. Tactic

A **strategy** is a larger, overall plan that can comprise several **tactics**, which are smaller, focused, less impactful plans that are part of the overall plan. **Tactic DDD** deals with implementing domain layer components. **Strategic DDD** deals with integrating these components and managing complexity in end-to-end application landscapes for the long run.

11.4 Subdomain vs. Bounded Context

	Subdomain	Bounded Context (BC)
Purpose	Partition problem space, focus and prioritize work	Partition solution space, identify communication and integration needs
Stakeholder concern	Complexity and size (of requirements); special skills needed for certain design tasks	Complexity and size (of realization); deal with terminology conflicts (homonyms, polysemes)
Viewpoint	Scenario, Logical	Implementation, Process
Types/Variants	Core Domain Supporting Subdomain Generic Subdomain	Team BC, subsystem BC
Units of decomposition	Groups of features (e.g. epic in agile, use case model/category in UML), classes in OOA domain model	Subdomains from OOA, one single OOD domain model
Relationship to Tactic DDD	Identifies need for Entities, Value Objects, Factories, Repositories; suggests initial set of Aggregates	Identifies need for implementation of Aggregates, groups them (and their content), leads to refactoring of Aggregate landscape
Mutual relationship	1:1 (in ideal world), n:m (in practice)	

Subdomain = Top-down approach for partitioning
Bounded Context = Bottom-up approach for partitioning

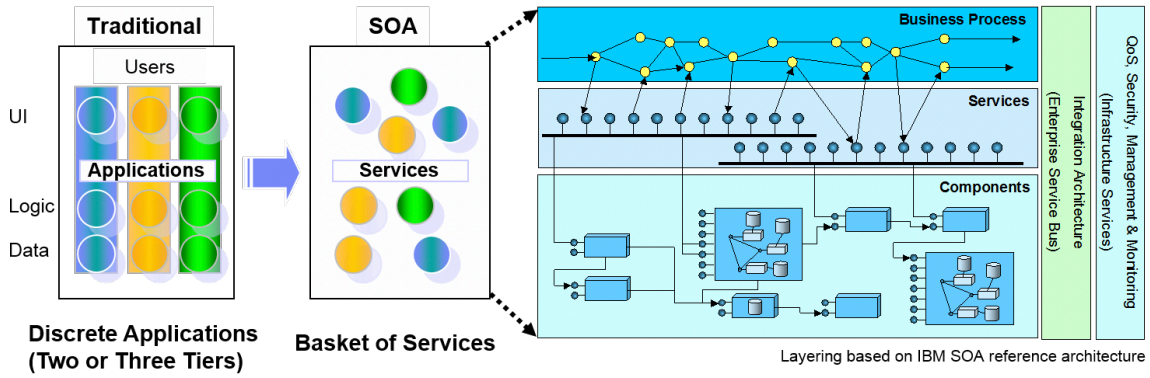
11.5 DDD vs. OOAD vs. PoEAA

The DDD patterns **refine** the Domain Model patterns from PoEAA (the DDD pattern "Service" can also be seen as a DDD version of "Transaction Script" in PoEAA). A design- and implementation-level domain model that populates the Business Logic Layer (BLL) of a layered application architecture is one of the major outputs and artifacts of OOADD (which is a family of methods, or analysis and design techniques).

11.6 Tips and Tricks

- Design **small aggregates**
- Use **asynchronous communication** between Aggregates
- **Reference** other aggregates **by identity**
- Use **eventual consistency** outside the boundary
- Give enforcement responsibilities (for **invariants**) to **root entity**
- Keep one Aggregate on one server, allow different Aggregates to be distributed among (hardware) nodes
- Use the **same boundaries** for **transactions** and **distribution**
- Model true invariants in **consistency boundaries**

12 Synthesis: Service-Oriented Architecture (SOA)



12.1 Definition

SOA mean different things to different stakeholders:

- **Business People**: A set of services that a business wants to expose to their customers and partners, or other portions of the organization.
- **Software Architects**: An architectural style which requires a service provider, a service requestor (consumer) and a service contract: “A service is a component with a remote interface.”
- **Developers / Operators**: Requires and defines one or more programming and deployment models realized by standards, tools and technologies such as Web services (WSDL/SOAP based or RESTful HTTP).

Architectural principles of SOA:

- Modularity
- Layering
- Loose Coupling

Architectural patterns of SOA:

- Service Routing, Transformation, Adaptation (e.g. with an Enterprise Service Bus)
- Service Composition (services "stitched" together to implement user needs)
- Service Registry (services must be discoverable)

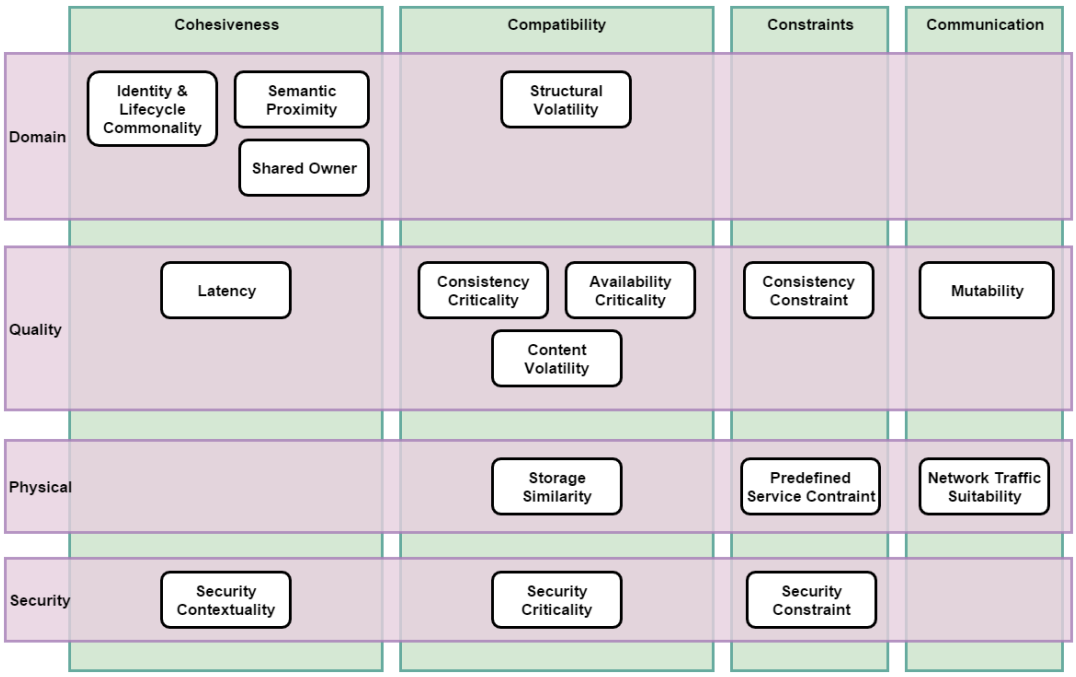
We want to achieve separation of concerns, reuse and flexibility!

12.2 Service Cutting

Primary Question: How do I split my system into services?

- **Step 1**: Extract coupling information based on coupling criteria (see illustration below)
- **Step 2a**: Create graph with scored edges that represent coupling between data fields, operations and artifacts*
- **Step 2b**: Calculate candidate service cuts based on graph clustering algorithm(s)
- **Step 3**: Visualize candidate service cuts (for visual evaluation)

*Artifacts = Entity-relationship model, Use Cases, System characteristics, Aggregates (DDD)



12.3 Loose Coupling

Involves 4 types of autonomy

- **Reference autonomy** (or location transparency): Producers and consumers don't know each other.
- **Platform autonomy**: Producers and consumers may be in different environments and written in different programming languages.
- **Time autonomy**: Producers and consumers access channel at their pace; communication is asynchronous, data exchanged is persistent.
- **Format autonomy**: Producers and consumers may use different formats of data exchanged; this requires transformation "on the wire" (mediation).

12.4 Integration Styles

Services have to exchange data, this can be achieved in multiple ways

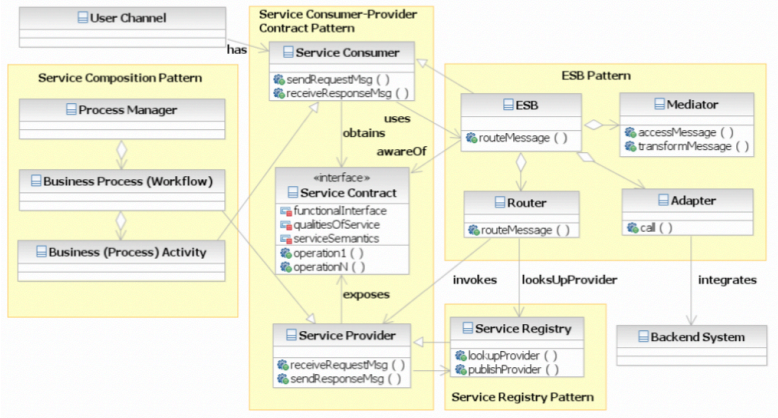
- File Transfer
- Shared Database
- Remote Procedure Invocation
- Messaging
- Web (REST)

Integration Style	Time Autonomy	Reference Autonomy	Platform Autonomy	Format Autonomy
File Transfer[12]	+	+	(+)	-
Shared Database[13]	+	+ or (+)	(+)	-
Remote Procedure Invocation or Call (RPC)[14]	-	(+)	(+)	-

Integration Style	Time Autonomy	Reference Autonomy	Platform Autonomy	Format Autonomy
Messaging[15]	+	+	+	*(+) with EIPs and protocol headers*
The Web (REST)	+	+	+	*(+) with CMTs and HTTP content negotiation (but actual format still need to be agreed upon)*

12.5 Core Patterns

SOA **combines** and **extends PoEAA, EIP and DDD** patterns (and others) in an enterprise computing context.



12.5.1 Service Contract

Motivation:

- Modularity
- Platform Transparency

A service contract defines a **service invocation interface**. The contract has a functional and a behavioral part. The **functional part** is machine- readable and specifies one or more **operations** which comprise request and, optionally, response messages. The **behavioral part** defines the **non-functional characteristics** of the message exchange (e.g. Security) and the operation **invocation semantics** (e.g. Pre- and Postconditions).

Service Contract: **[Name]**	
Business domain (scenario viewpoint, functional area): • […]	*User stories and quality attributes (design forces):* • […]
Service quick reference (synopsis of what this service provides to consumers): • […]	
Invocation syntax (functional contract): IDL specification, security policy; request/response data samples; endpoint addresses (test deployment, production instances); sample service consumer program (source code); error handling information (error codes, exceptions) • […]	
Invocation semantics (behavioral contract): informal description of preconditions, postconditions, invariants, parameter meanings; FSM; service composition example; integration test cases • […]	
Service Level Agreement (SLA) with Service Level Objectives (SLO); Quality-of-Service policies • […]	
Accounting information (service pricing); external dependencies/resource needs • […]	
Lifecycle information: current and previous version(s); limitations, future roadmap; service owner with contact information, link to support and bug tracking system • […]	

12.5.2 Enterprise Service Bus (ESB)

Motivation:

- Loose coupling
- Protocol transparency
- Format transparency
- Systems Management (Logging, Load-Balancing etc.)

Responsibilities:

- **Routing** of messages (to services)
- **Translation** of messages (into diff. formats)
- **Adaptation** of services (consumers, providers)

ESB is the SOA refinement of the general **broker** pattern and enables many to many connectivity for technically diverse and physically distributed communication parties. It provides a **remote communication infrastructure** that allows service consumers and service providers to **exchange request and response messages** using **one or more** message **exchange patterns**, communication **protocols**, and message **exchange formats**.

Exchange Patterns:

- Synchr. Request-Reply
- Asynchr. One Way Messaging
- Publish-Subscribe
- …

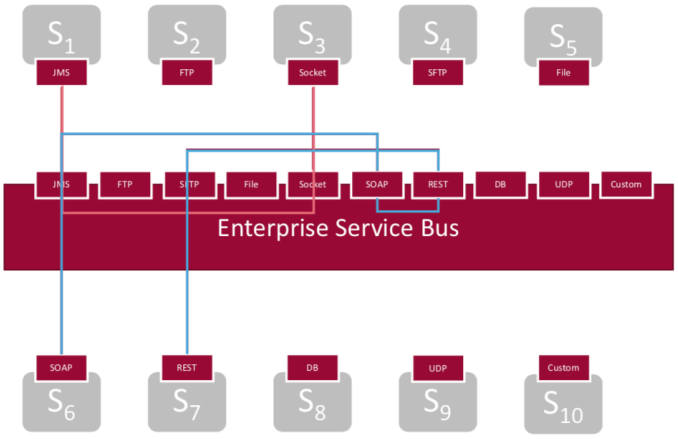
Exchange Formats:

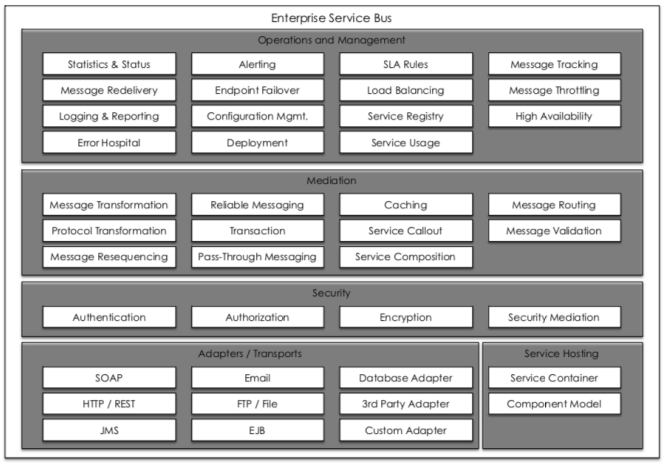
- JSON
- XML
- …

Communication (Protocols):

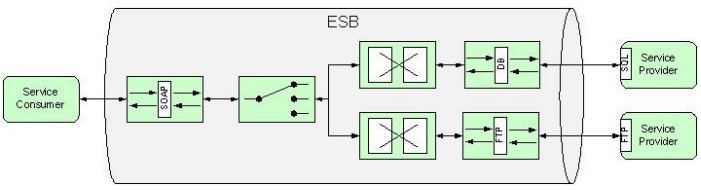
- Message Queuing (e.g. AMQP)
- RPC (e.g. SOAP, JSON-RPC via HTTP)
- Propriertary (with adapters)
- …

Note that ESBs can typically perform content-based routing as well as fixed routing (e.g. based on rules).

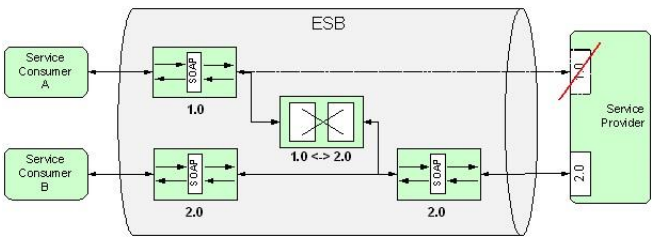




ESB for Content-Based Routing and Translation:



ESB for Service Versioning:



12.5.3 Service Composition

Motivation:

- Loose Coupling
- Modularity
- Maintainability
- Portability

If **two or more service providers are assembled** into another service provider we speak of **service composition**. The assembled service provider invokes the composed service providers via their service invocation interfaces.

The characteristics of service providers differ. To avoid a tight **coupling between service consumers and providers** with **different responsibilities and quality attributes** as well as **undesired dependencies** between the services, the permitted invocations must be defined. Therefore the SOA should be organized into **three or more logical layers**.

- **Presentation Layer**: Contains all rich or thin client logic displaying user interfaces to human users. In an SOA, many service consumers reside in the presentation layer.

- **Domain Layer**: Contains business logic such as control flow, but also calculations and modifications of enterprise resources.

- **Data Layer**: Lets enterprise resources and other data persist. It also provides interfaces allowing the domain layer to access the data when executing its logic.

The service composition pattern refines the above logical layering scheme. The **domain layer** is divided into two sub-layers:

- **Service Composition Layer**: Business activities that are assigned to end users are placed in this layer. A business process manager (workflow management system) is the central middleware component in the service composition layer. It is aware of the business activities that have to be performed and the appropriate order, which defines an executable business process control flow (workflow).

- **Atomic Service Layer**: Business activities invoke services in the atomic service layer. The atomic service layer contains calculations and manipulations of enterprise resources, which are not permitted to invoke services in the service composition layer.

Service providers either reside in the atomic service layer or, as composed services, in the service composition layer.

12.5.4 Service Registry

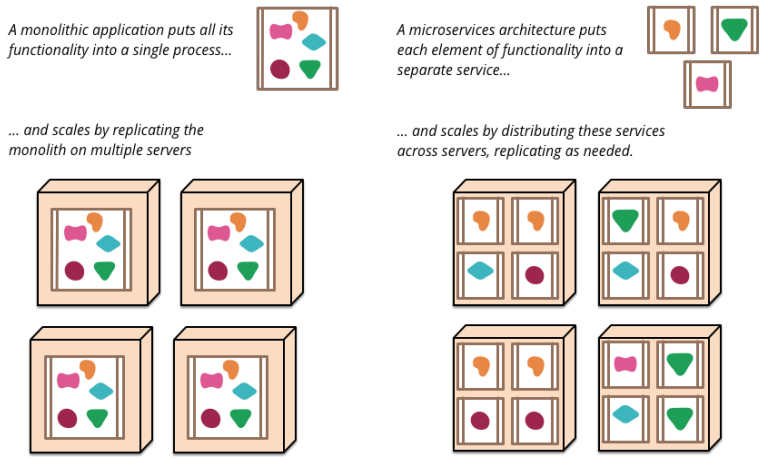
Motivation:

- Location transparency

To ensure flexibility during deployment and service invocation, service consumers **should not use fixed service provider addresses**; ideally, they should even be unaware of the actual service provider and **let the ESB decide** where to **route** a service request to (e.g. for load balancing purposes).

A 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. Organizational information such as service ownership, service level agreements, and billing information can optionally be stored in the service registry as well.

12.6 Microservices



Microservices evolved as an **implementation approach and physical refinement to SOA** that leverages recent advances in agile practies, cloud computing and DevOps.

- 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**.
- **Business-driven development practices** and pattern languages such as Domain-Driven Design (DDD) are employed to identify and conceptualize services.
- Cloud-native application design principles are followed, e.g. as summarized in **Isolated State, Distribution, Elasticity, Automated Management and Loose Coupling** (IDEAL).
- Multiple storage paradigms are leveraged (SQL and NoSQL) in a **polyglot persistence strategy**. Each service implementation has **its own data store**.
- **Lightweight containers** are used to **deploy** and **scale** services.
- **Decentralized continuous delivery** is practiced during service **development**.
- Lean, but holistic and largely **automated approaches** to **configuration** and **fault management** are employed within an overarching DevOps approach.

12.6.1 Sizing / Granularity

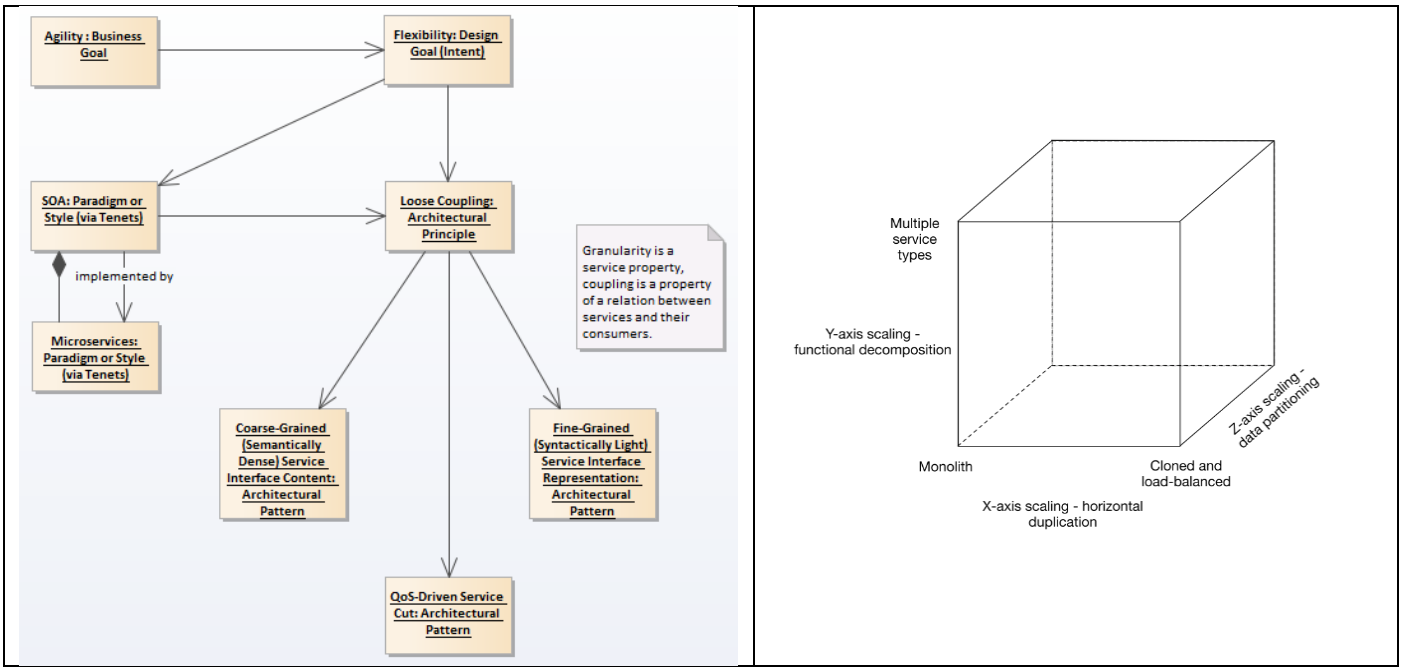
The "micro" part can be interpreted as follows:

- **Developed and operated by a single team**. New or changed business requirements should ideally lead to changes in just a single microservice (including the user interface).
- Fully **understood by each developer** on the team.
- **Replaceable** by a new implementation if necessary.
- Should be large enough to **contain the data it needs** to operate, and **loosely coupled** with others.

Further considerations:

- Communication and deployment **overhead**
- **Transactions** spanning multiple microservices are hard to manage
- Data **consistency** (consistency boundaries) is hard to manage

One has to be careful not to end up with a (distributed) monolith again!

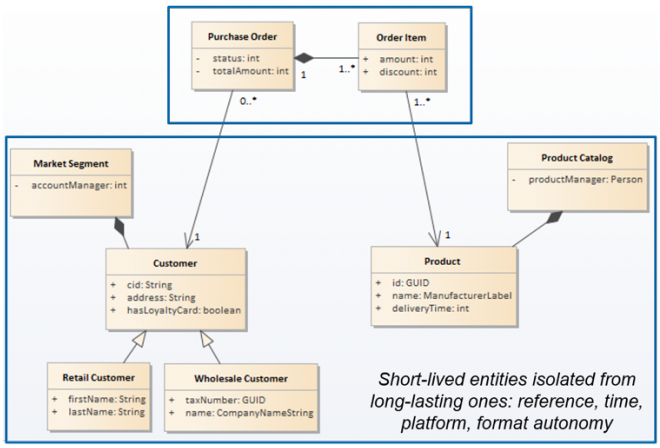


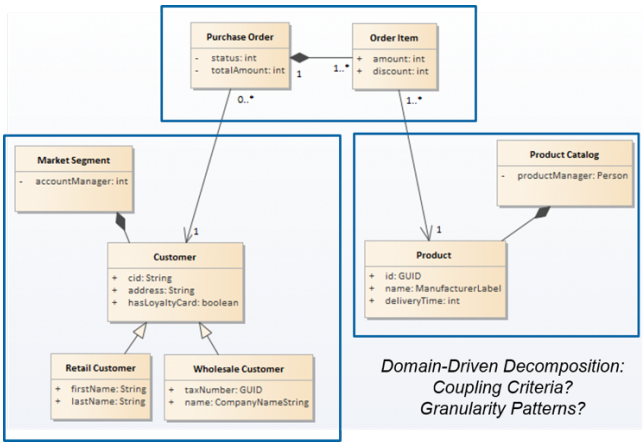
Business granularity (a.k.a. semantic density) has a major impact on agility and flexibility, as well as maintainability

- Position of service operation in Business Architecture, e.g., expressed in a Component Business Model (CBM) or DDD Domain Model (analysis level)
- Amount of business process functionality covered: Entire process? Subprocess? Activity?
- Number and type of analysis-level domain model entities touched

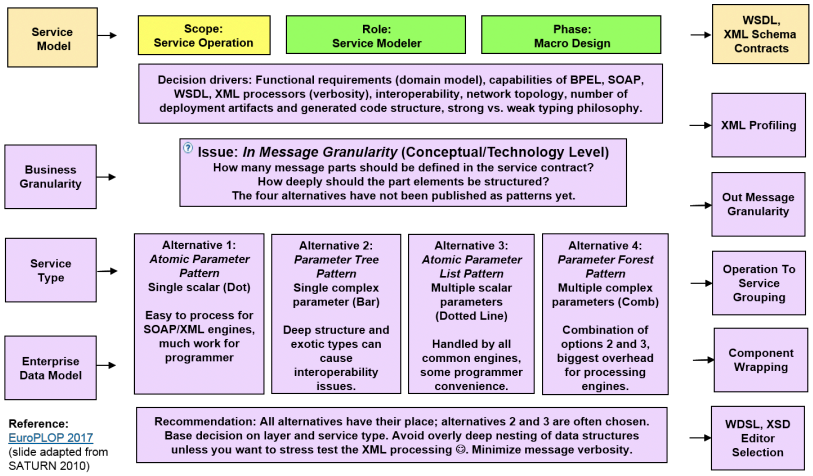
Technical granularity (a.k.a. syntactic weight) determines runtime characteristics such as performance and scalability, interoperability – but also maintainability and flexibility

- Number of operations in service contract, number of resources exchanged
- Structure of payload data in request and response messages
- QoS entropy adds to the maintenance effort of the service component
- Backend system interface dependencies and their properties (e.g. consistency)
- Security, reliability, consistency requirements (coupling criteria)





Web API Design and Evolution (WADE) patterns offer four options for structuring API call request and response messages:

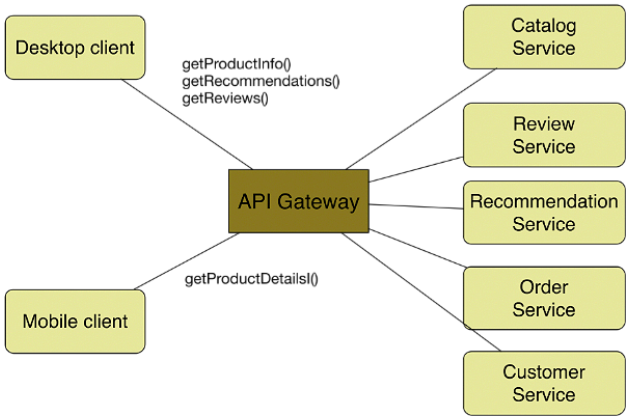


Another approach to solve this problem is "Service Cutting" described in section 12.2.

There is no single definite answer to the “what is the right granularity?” question, which has several context-specific dimensions and criteria!

12.6.2 API Gateway (Pattern)

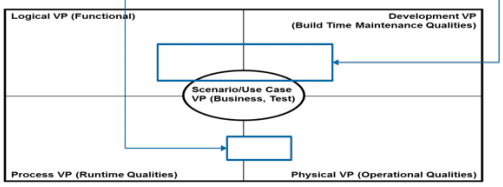
The API Gateway pattern is sort of an equivalent to the Enterprise Service Bus (ESB) known from SOA. Although it seems to be more focused on **connecting clients to services** than connecting services to services. But the **functionality (routing, translation, adaptation)** is basically the **same**.



12.6.3 Microservices vs. SOA

Characteristic	Viewpoint/Qualities/Benefit	SOA Pendant
Componentization via services	Logical Viewpoint (VP): separation of concerns improves modifiability	Service provider, consumer, contract (same concept)
Organized around business capabilities	Scenario VP: OOAD domain model and DDD ubiquitous language make code understandable and easy to maintain	Part of SOA definition in books and articles since 200x (e.g. Lublinsky/Rosen)
Products not projects	n/a (not technical but process-related)	(enterprise SOA programs)
Smart endpoints and dumb pipes	Logical Viewpoint (VP): information hiding improves scalability and modifiability	Same best practice design rule exists for SOA/ESB (see e.g. here)
Decentralized governance	n/a (not technical but process-related)	SOA governance (might be more centralized, but does not have to; "it depends")
Infrastructure automation	Development/Deployment VP: speed	No direct pendant (not style-specific, recent advances)
Design for failure	Logical/Development/Deployment VP: robustness	Key concern for distributed systems, SOA or other
Evolutionary design	n/a (not technical but process-related): improves replaceability, upgradeability	Service design methods, Backward compatible contracts

Application Component Property (Gartner/TMF)	Mapping to 4+1 Viewpoint Model (Kruchten 1995)	Mapping to ZIO Tenet	Novel or "Same Old Architecture"?
tightly scoped	Scenario/Use Case, Logical	1, 2	Good practice in SOA
strongly encapsulated	Logical, Development	1	SOA definition
loosely coupled	Development, Process (Integr.)	1, 3	SOA definition
independently deployable	Process, Physical	1	(somewhat) novel
independently scalable	Process, Physical	1	(somewhat) ovel

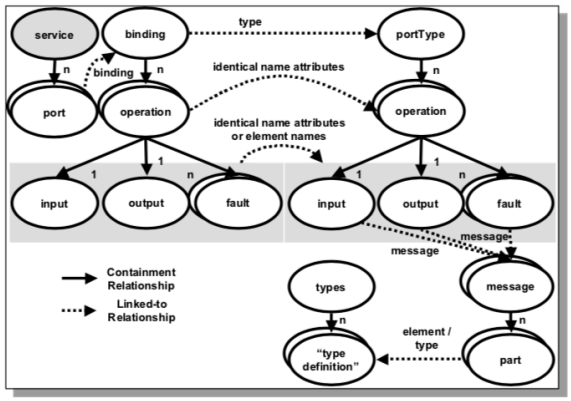


View model adapted from: P. Kruchten, 4+1 views on SWA, IEEE Software.

12.6.4 DevOps

DevOps (a clipped compound of "development" and "operations") is a **software engineering culture and practice** that aims at **unifying software development (Dev) and software operation (Ops)**. The main characteristic of the DevOps movement is to strongly **advocate automation and monitoring** at all steps of **software construction**, from **integration**, **testing**, **releasing** to **deployment** and **infrastructure management**. DevOps aims at **shorter** development **cycles**, **increased** deployment **frequency**, more **dependable releases**, in close alignment with business objectives.

12.7 WSDL Basics



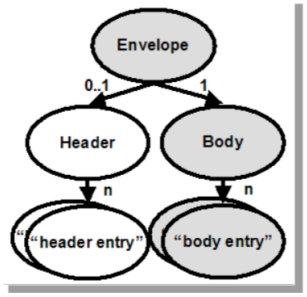
Logical relationships between WSDL elements

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://ws.handling
    <xs:element name="submitReport" type="tns:submitReport"/>
    <xs:element name="submitReportResponse" type="tns:submitReportResponse"/>
    <xs:complexType name="submitReport">
        <xs:sequence>
            <xs:element minOccurs="0" name="arg0" type="tns:handlingReport"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="handlingReport">
        <xs:sequence>
            <xs:element name="completionTime" type="xs:dateTime"/>
            <xs:element maxOccurs="unbounded" name="trackingIds" type="xs:string"/>
            <xs:element name="type" type="xs:string"/>
            <xs:element name="unLocode" type="xs:string"/>
            <xs:element minOccurs="0" name="voyageNumber" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="submitReportResponse">
        <xs:sequence/>
    </xs:complexType>
    <xs:element name="HandlingReportErrors" type="tns:HandlingReportErrors"/>
    <xs:complexType name="HandlingReportErrors">
        <xs:sequence/>
    </xs:complexType>
</xs:schema>
```

- **XML elements for operation parameters**
  - a.k.a. message parts
- **XML complex types for nontrivial DTOs**
- **XML basic types for scalar DTOs**

```
<wsdl:portType name="HandlingReportService">
  <wsdl:operation name="submitReport">
    <wsdl:input name="submitReport" message="tns:submitReport">
    </wsdl:input>
    <wsdl:output name="submitReportResponse" message="tns:submitReportResponse">
    </wsdl:output>
    <wsdl:fault name="HandlingReportErrors" message="tns:HandlingReportErrors">
    </wsdl:fault>
  </wsdl:operation>
```

12.8 SOAP Basics



SOAP message containment structure

```
<SOAP-ENV:Envelope
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:xsd="http://www.w3.org/2001/Schema"
            xmlns:xsi="http://www.w3.org/2001/Schema-instance">
    <SOAP-ENV:Header>
        <pQP:priority
Header entry  xmlns:pQP="http://premierquotes.com/ns/priority"
        SOAP-ENV:mustUnderstand="1">
        high
        </pQP:priority>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <pQ:getPhoneNumber
Body entry  xmlns:pQ="http://premierquotes.com/ns/employees"
        SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
Method parameters  <first-name xsi:type="xsd:string">Ed U.</first-name>
        <last-name xsi:type="xsd:string">Cate</last-name>
        </pQ:getPhoneNumber>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Header must be processed

Method invocation

Use SOAP encoding

Method parameter types

Method parameter values

SOAP message example

13 Synthesis: State Management

Statelessness of interactions between service consumer/requestor (or client) and service provider (or server) helps to **achieve loose coupling**. It makes it easier to **scale** out and to **deploy** rapidly and flexibly.

State Management is an ASR in most applications!

13.1 Session State vs. Resource State

Session / Resource State are layer-specific refinements of the general concept of Application State:

- Presentation Layer holds the session state.
- Business Logic Layer and its subordinate layers (data access, persistence) deal with resource state and (business) conversation state.

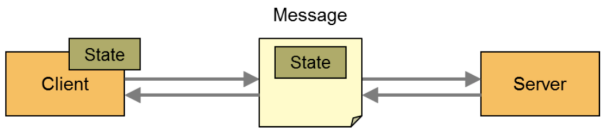
13.2 Client Session State

Pros:

- Scales well

Cons:

- Security concerns
- Performance concerns



Example: Session in HTML/HTTP Cookie, Hidden Field, URL rewrite

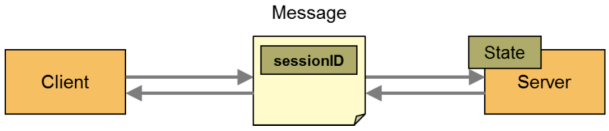
13.3 Server Session State

Pros:

- Development effort small

Cons:

- Reliability concerns
- Scalability concerns ("Session stickiness")



Example: Session in memory or proprietary data store of application server (e.g. JEE servlet container)

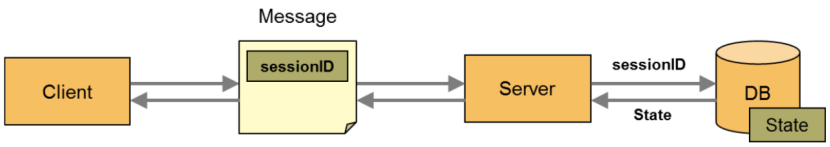
13.4 Database Session State

Pros:

- Highly scalable
- Persistence required anyway (probably)

Cons:

- Additional resources required
- Communication overhead



Example: Sessions stored in key-value storage such as Redis.

13.5 State Transfer vs. State Transition

- Server-internal state management vs. client-server interactions
- External vs. internal state
- Component vs. connector state

13.6 Event Sourcing

This can be seen as an alternative approach to "classical" state management.

- A **stream of events** is the only means **to transfer/initiate state changes**. All changes come in as events and are handled via event processing.

- To get **current values**, the application cannot simple read the property of an entity, but **has to start from empty application state** and use the change log to **recalculate the current state** (unless snapshots are taken)

- All **events are immutable** (they do not have any setter methods, and do not implement any business logic that might have side effects). There are **two types** of such immutable events, those describing an **absolute/full new value** vs. those **reporting differences between old and new values**; the latter type is more reversible than the first one.

Event sourcing brings a set of capabilities that might be very handy depending on the NFRs of the application to be built (e.g. offline availability, traceability):

- Complete rebuilds (of state)
- Temporal queries
- Event replay or undo
- Full audit logs

As a result**, loose coupling** is achieved in **almost all coupling dimensions**:

- Time autonomy
- Reference autonomy
- Platform autonomy

See section 15.10 for pattern definition.

Event Sourcing is good for occasional connectivity scenarios and when availability is more important than consistency!

13.6.1 Event Properties

Example of an event definition in JSON that allows CUD of entities with flexible granularity:

<event>

```
{
    "hash"      : <string (change set hash)>
    "timestamp" : <integer (unix-timestamp)>,
    "fieldspec" : <fieldspec>,
    "old"       : <value>,
    "new"       : <value>
}
```

<fieldspec>

```
{
    "entity" : <string (entity name)>,
    "attr"   : <string (attribute name)>,
    "pk"     : <integer (primary key)>
}
```

"attr" null means "update all"

<value> (scalar or JSON object)

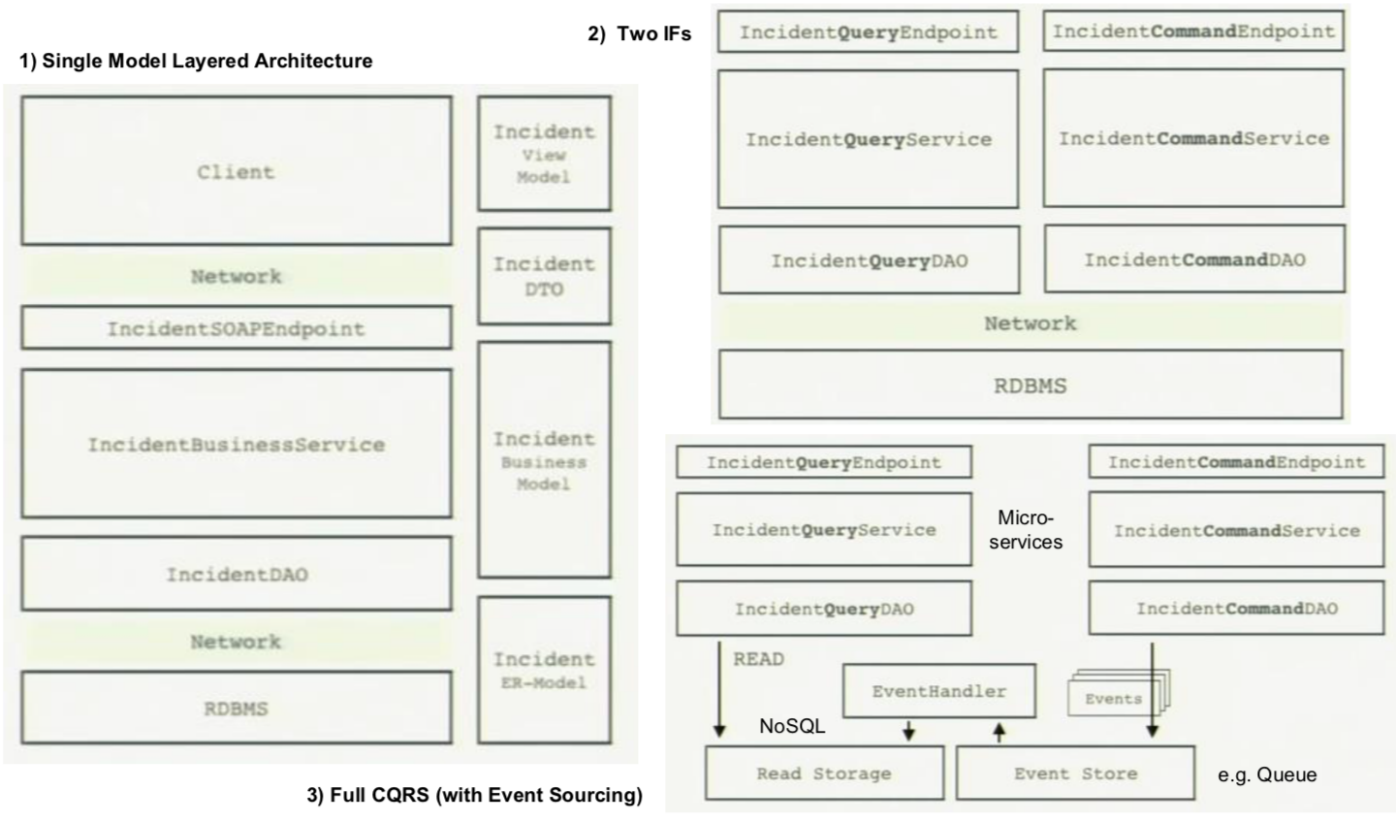
	C	R	U	D
$old = \varnothing \wedge new \neq \varnothing$	×			
$old \neq \varnothing \wedge new \neq \varnothing$			×	
$old \neq \varnothing \wedge new = \varnothing$				×

Event Naming Format: Domain Model Element (noun) + Action Verb (in -ed form)
"CustomerVerifiedEvent", "ShipmentArrivedEvent", "AddressUpdatedEvent"

13.7 Command Query Responsibility Segregation (CQRS)

Different (domain) models are used for **read** and **write** access. This has to be handled with care, it is powerful but risky (increased development and deployment effort).

The pattern can be **combined with Event Sourcing** as shown in the following example architecture:



See section 15.11 for pattern definition.

CQRS allows independent optimization of read and write models to perform/scale well but you have to deal with consistency issues!

14 Synthesis: Representational State Transfer (REST)

REST is an architectural style and defined via constraints:

- Client / Server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand (optional)

RESTful HTTP is the dominating **implementation** of the REST style.

14.1 Key Concepts

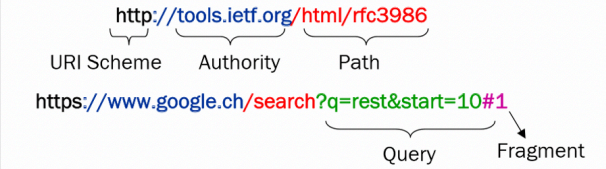
- Linked resources identified by URIs
- Resource representations (external views)
- Unified method set (e.g. HTTP verbs)
- Self-descriptive messages

14.2 RESTful HTTP

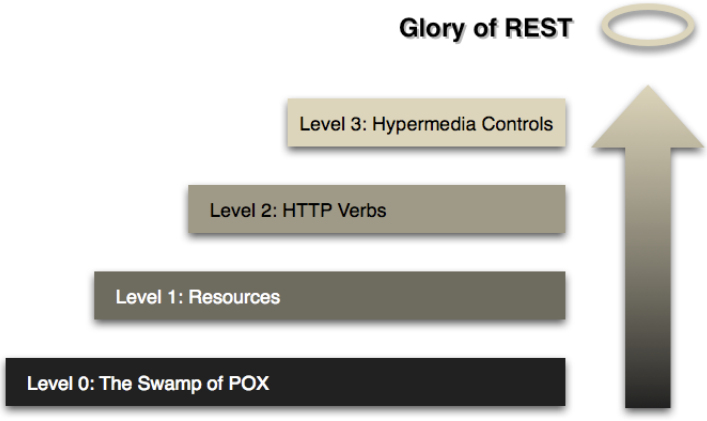
- **HTTP protocol primitives**
  - Sent by client
  - Understood by server
  - Client e.g. Web browser (or application client in integration scenario)

CRUD	REST	
CREATE	POST/PUT	Initialize the state of a new resource at the given URI
READ	GET	Retrieve the current state of the resource
UPDATE	PUT	Modify the state of a resource
DELETE	DELETE	Clear a resource, after the URI is no longer valid

- **URI-reference = [absoluteURI | relativeURI] ["#" fragment]**



14.3 Maturity Levels



- **Level 3**: Hypermedia a.k.a. Hypertext as the Engine of Application State (HATEOAS)
- **Level 2**: Using only HTTP verbs for CRUD operations on resources
- **Level 1**: Using resources uniquely identifiable by URI
- **Level 0**: Using HTTP as tunneling mechanism (e.g. for XML or JSON RPC)

One can only claim to use REST if level 3 is reached, use term "Web API" otherwise!

14.4 Hypermedia as the Engine of Application State (HATEOAS)

Following HATEOAS, the client **discovers possible interactions** with resources in **responses** (sometimes called "Discoverable API"). Only the home URI of a given resource set is fixed and shared upfront. This leads to a **partial specification** of the overall **workflow** and the **need for dynamic contracts**.

- **Application State** represents the state of processing (e.g. activities in business workflow) including preconditions, post conditions and the content/values of backend information systems and databases (e.g. enterprise resources)
- **Media Types** are seen to be the carriers of Application State, serving as typed data transfer objects for HTTP request and response payload (schema language depends on MIME type, e.g. XML Schema for XML, JSON Schema for JSON)
- **Typed links** as identifiers of AS (simultaneous presentation of information and controls that allows selection of next actions and/or obtainment of next options)

14.4.1 Media Type (MIME Type)

Examples of frequently used types specified as "Content-Type" in HTTP pakets:

- application/xml
- application/json
- text/plain
- image/png

These are rather generic ones. **Custom Media Types** (CMT) can be used to implement domain-specifc "type affinity" for exchanged data.

14.4.2 Typed Link

PayPal API uses three components for each HATEOAS link returned in the body of an HTTP response:

- **href**: URL of the related HATEOAS link you can use for subsequent calls.
- **rel**: Link relation that describes how this link relates to the previous call.
- **method**: HTTP method required for related call.

14.5 Service Contract

Ways to define service contract when applying REST architecture style:

- Swagger (OpenAPI Specification)
- RAML (Resource Access Markup Language)
- WSDL 2.0 (Web Services Description Language)
- WADL (Web Application Description Language)

14.6 Tips and Tricks

- Offer both **JSON and XML**
- Provide **resource discoverability** through links
- Support **versioning** via content negotiation

- Do NOT tunnel everything through GET / POST
- Do NOT ignore response codes
- Do NOT misuse cookies

15 Synthesis: Pattern Catalogue

15.1 Logical Layering (Architectural Style)

Context

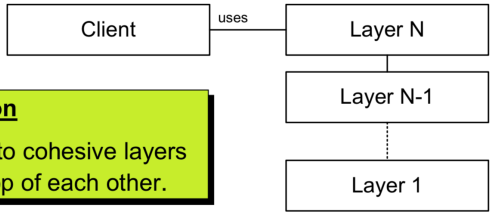
A system that needs to be segmented due to its size and specialization.

Problem

System design that supports a mixture of tasks on different levels.

Solution

Organize the system into cohesive layers that are stacked on top of each other.



15.2 Inversion of Control (Container Pattern)

"Hollywood Principle" -> "Don't call us, we call you!"

Context

Many solutions in same problem domain need to be build; middleware will be used to implement common parts with guaranteed service levels (QoS).

Problem

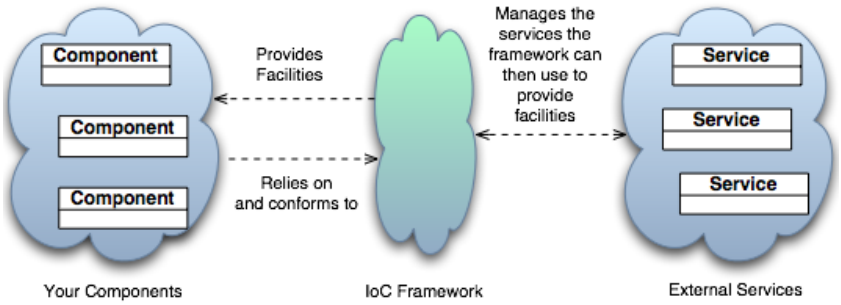
How can the framework control the server-side execution, but still be extensible with application features (without recompilation)?

One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.

-- Ralph Johnson and Brian Foote

Solution

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).



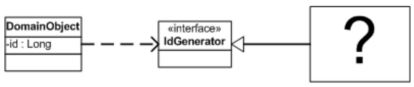
15.3 Dependency Injection (Container Pattern)

Context

A plugin links classes during configuration rather than compilation.

Problem

How do we assemble these plugins into an application?



Solution

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).

15.4 Entity, Value Object, Service (Tactic DDD Pattern)

Context

In the Domain (sub-)layer, the types of business logic and their requirements differ (w.r.t. instance lifetimes, state management, copy semantics). Plain OOAD delivers classes with methods and attributes and relationships.

Problem

Frameworks need to know the properties of their hosted components to address build time and runtime NFRs. How should the components (beans) be classified so that differences w.r.t. identity and continuity become clear?

Solution

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*.

15.5 Aggregate (Tactic DDD Pattern)

Context

Possibly many entities and value objects have been defined and linked. Hence, not all invariants can be expressed locally and assigned to single entities easily.

Problem

How can the dependency and integrity management be organized in such a way that visibility of entity relationships remains limited (local)? How can cross-entity relationships be monitored to conform to pre- and postconditions as well as invariants that are domain- but not entity-specific (e.g., complex business rules)?

Solution

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.

Component: Instance of DDD Root Entity pattern (a.k.a. Aggregate Root)	
Responsibilities: • Can be found via its Id (from outside of aggregate) • Subscribes to domain events (sent by other aggregrates) • Manages its lifecycle, maintains mutable state, including references to non-root entities in same aggregate • Has methods implementing state-changing domain logic (transactional boundary) that might return pointers ot value objects and other entities in the same aggregate • Offers entity modifiers and getter methods that might be subject to access control and audit, logs activities • Enforces business rules (preconditions, invariants, postconditions) for entire aggregate • Sets transaction boundaries and access control policies	*Collaborators (Interfaces to/from):* • Service calls (application layer, domain layer) • Factory object • Repository (single one for this aggregate) • Subordinate entities in same aggregate • Root entities in other aggregates (via their Ids)
Candidate implementations technologies (and known uses): • Spring beans, EJBs in JEE • POJOs, optionally realizing other design patterns such as Strategy, Composite, Specification • Cargo in DDD Sample Application is an exemplary known use	

15.6 Repository & Factory (Tactic DDD Pattern)

Context

Aggregates with root entities, subordinate entities and value objects are defined. Presentation/interface layer (and application sub-layer) want to invoke CRUDS operations on them, but do not know how to instantiate and locate them.

Problem

How can the Create, Read, Update, Delete, Search (CRUD) lifecycle of aggregates and their root entities be managed?
How can root entities and (possibly) other entities be searched for (queried)?

Solution

Define a single *factory* per aggregate and root entity. 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.

15.7 Subdomain & Bounded Context (Strategic DDD Pattern)

Context

A large and complex set of requirements has to be dealt with. Parts of these have already been implemented, possibly multiple times. Both analysis-level domain model and design-level domain models are large and complex as well, and hard to agree upon universally/globally (for instance, organization-widely).

Problem

How can a large set of candidate aggregates (comprising entities and value objects and services) be grouped so that problem and solution space remain manageable and semantic ambiguities are isolated (or at least identified)?

Solution

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.

15.8 Context Map with Dependency Types (Strategic DDD Pattern)

Context

A rich, complex domain model has been decomposed into multiple analysis-level subdomains to be realized/organized in several design-level bounded contexts.

Problem

How can end-to-end functionality be implemented in a controlled manner when dealing with a partitioned domain model that is organized into functional subdomains and bounded contexts that represent (sub-)systems or teams?

Solution

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.

15.9 Context Mapping Best Practices (Strategic DDD Pattern)

Context

The context map of an integrated system of systems (or a single complex system decomposed into services) shows relationships of many types: published language, shared kernel, open host service, customer/supplier, conformist, ACL.

Problem

How can upstream and downstream contexts be protected from each other (to support loose coupling), but still benefit from each other/invoke each other?

Solution

Keep the published language minimal, especially when defining an OHS. Avoid shared kernels between supporting subdomains and in customer-supplier relations. Add an ACL on the conformist side of an OHS-conformist relation. Consider introduction of ACLs also for customer-supplier relations.

Component: **Anti-Corruption Layer (ACL)**	
Responsibilities: • Acts as an intermediary between client and server • Protects downstream client from platform- and model-specifics with model transformations • Makes upstream client independent of change management policy of upstream provider (versioning etc.) • Maps external transfer syntax (JSON, XML) to object notation; maps from model to model when crossing context boundary (including error message mapping) • Cache requests and responses (optional), support conditional requests • Observe rate limits and other quality-related constraints	*Collaborators (Interfaces to/from):* • Downstream client • Upstream data/service provider
Candidate implementations technologies (and known uses): • Many EAI products and open source assets such as Mule ESB, Talend, Pentaho DI • Java Connector Architecture (in previous editions of JEE) • AdCubum SYRIUS; Core Banking SOA (from lesson 1) and most enterprise applications vary this pattern (typically under different names: Gateway, Adapter, Façade, Wrapper)	

15.10 Event Sourcing (State Management Pattern)

Context

A system (application) interacts with other ones. There is shared conversation state and each conversation participant maintains its own application/resource state. The amount and location of application components changes often.

Problem

How to flexibly and asynchronously communicate state changes in loosely coupled systems that consist of multiple components that work together but do not know each other? How to reliably/atomically publish such state changes?

Solution

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.

15.11 Command Query Responsibility Segregation (State Management Pattern)

Context

The read and write access profiles to REST resources and DDD entities (and other stateful components) varies greatly. The application has to perform and scale well. A single OOD domain model handling all CRUD operations turned out not to be able to satisfy the NFRs for both read and write access.

Problem

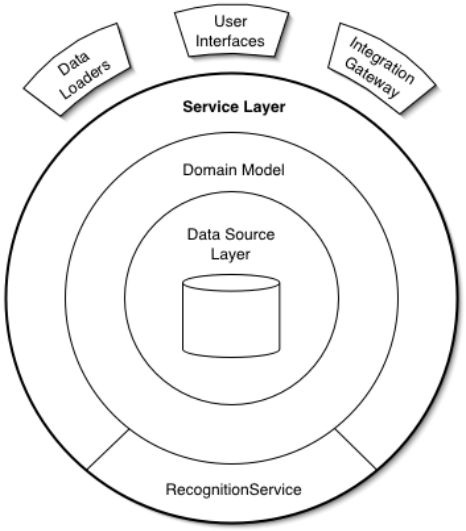
How can state changing traffic and idempotent requests be handled separately so that high traffic scenarios can be supported in complex domains?

Solution

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.

15.12 Service Layer (PoEAA Pattern)

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.

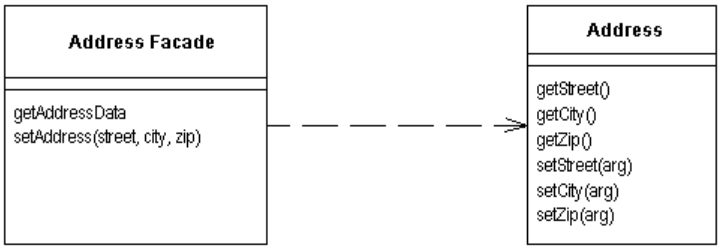


Responsibilities of Service Layer:

- Hide complex domain models
- Role-Based Access Control (RBAC)
- Transaction Control (including "Redo" or "Undo")
- Activity Logging
- Exception Handling

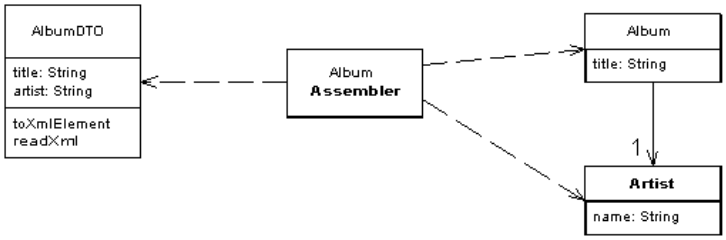
15.13 Remote Facade (PoEAA Pattern)

Provides a **coarse-grained facade** on **fine-grained objects** to improve efficiency over a network.

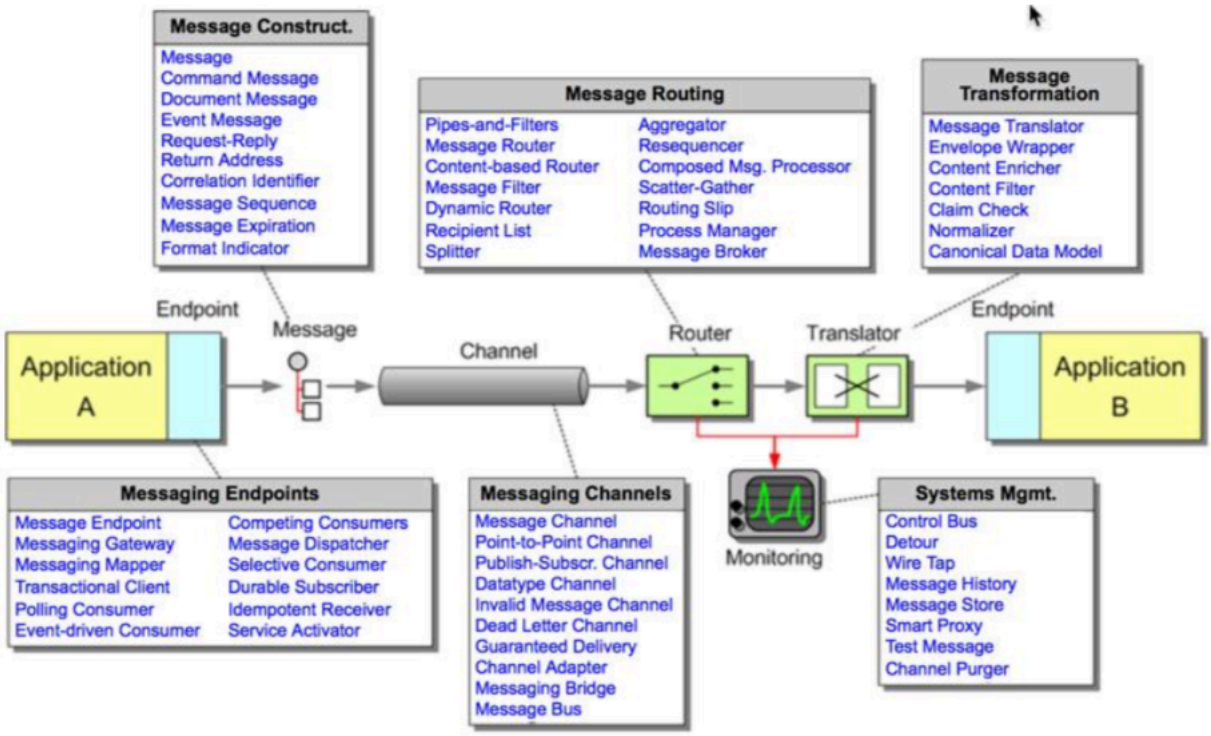


15.14 Data Transfer Object (PoEAA Pattern)

An object that **carries data between processes** in order to reduce the number of method calls.



15.15 Enterprise Integration Patterns (EIP)



16 Evaluation: Architectural Evaluation in a Nutshell

- Have the **right decisions** been made, and have they been **made right**?
- Can the designed architecture **meet the elicited NFRs** (and other requirements, including constraints)?
- Does the **implementation match** the designed **architecture**?

16.1 Questions to ask

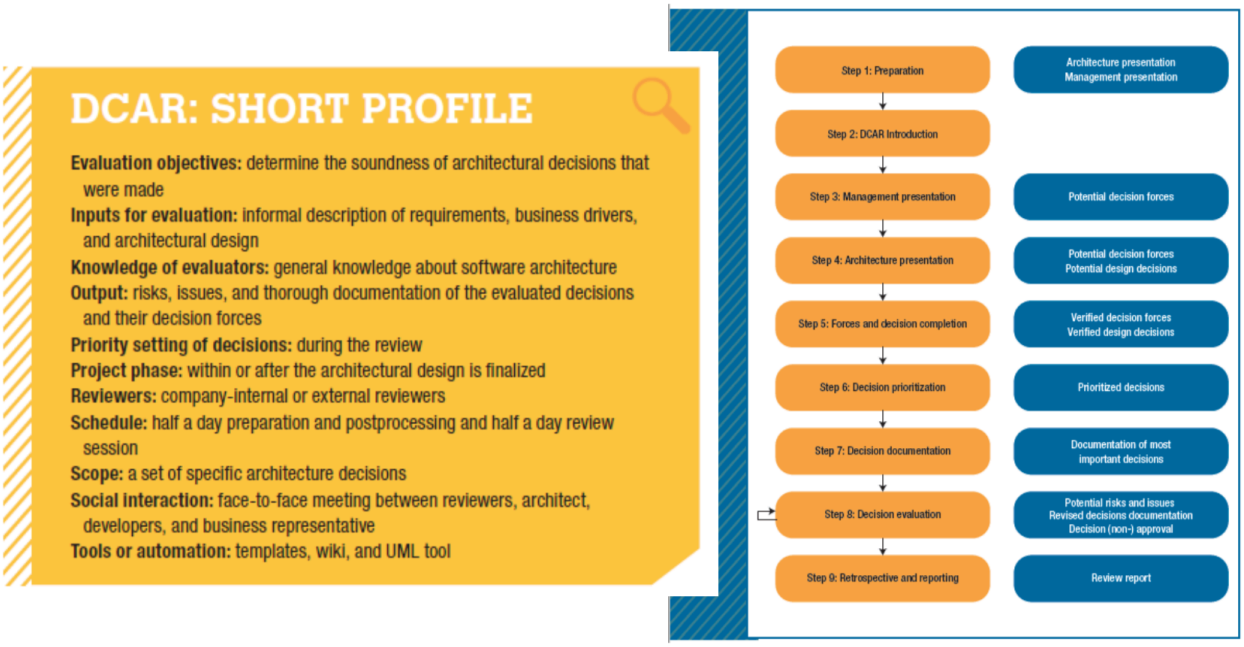
- Which architectural decisions were made to satisfy quality attribute scenario X?
- Which architectural approach supports the satisfaction of scenario X?
- Which compromises (tradeoffs) were made along with this decision?
- Which other quality characteristics or architectural goals does this decision 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?

Use arc42 template as "checklist" during analysis, synthesis and evaluation (see section 17.4)!

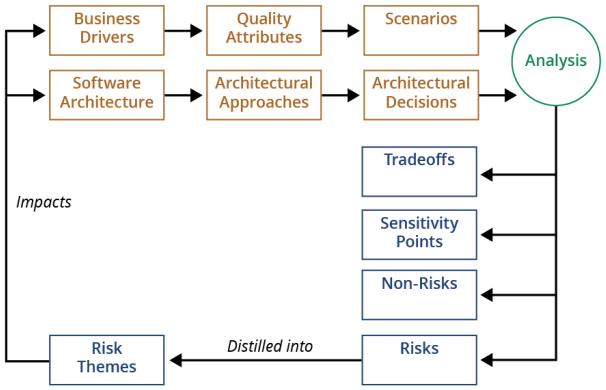
16.2 Typical Tradeoffs

- Performance vs. Security
- Security vs. Usability
- Performance vs. Supportability
- Reliability vs. Simplicity
- Scalability vs. Manageability
- I/O vs. CPU Usage
- Hardware vs. Software Solution

16.3 Decision-Centric Architecture Review (DCAR)



16.4 Architecture Tradeoff Analysis Method (ATAM)



17 Architecture Documentation

17.1 Standard

IEC/IEEE/ISO 42010:2011 (Architecture Description Standard)

17.2 Motivation

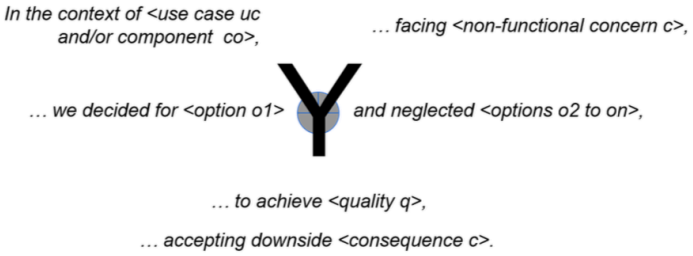
- Provide a **single place to find** important architectural decisions
- Make **explicit rationale and justification** of architectural decisions
- Preserve **design integrity** in the provision of functionality and its allocation to system components
- Ensure that the architecture is **extensible** and can support an evolving system
- Provide **reference** of documented decisions **for new people** who join the project
- **Avoid unnecessary reconsideration** of the same issues

17.3 Architectural Decisions (ADs)

Design decisions that are costly to change. These decisions directly or indirectly determine the non-functional characteristics of a system. A **decision record** can be broken down into

- An architecturally significant design problem
- Several potential solutions (options)
- Rationale for the selection of the chosen solution (e.g. by referencing quality attributes)

17.3.1 (WH)Y Statements




```
AD 1: Apply Messaging pattern and RPC pattern

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.
```

17.3.2 Table Format

Subject Area	Process and service layer design	**Topic**	Integration
Name	Integration Style	**AD ID**	2
Decision Made	We decided for RPC and the Messaging pattern (Enterprise Integration Patterns)		
Issue or Problem	How should process activities and underlying services communicate?		
Assumptions	Process model and NFRs/QA requirements are valid and stable		
Motivation	If logical layers are physically distributed, they must be integrated.		
Alternatives	File transfer, shared database, no physical distribution (local calls)		
Justification	This is an inherently synchronous scenario: VSP users as well as internal Telco staff expect immediate responses to their requests. Messaging system will ensure guaranteed delivery and buffer requests to unreliable data sources.		
Implications	Need to select, install, and configure a message-oriented middleware provider.		
Derived Requirements	Many finer grained patterns are now eligible and have to be decided upon: message construction, channel design, message routing, message transformation, system management (see Enterprise Integration Patterns book).		
Related Decisions	Next, we have to decide on one or more integration technologies implementing the selected two integration styles. Many alternatives exist, e.g., Java Message Service (JMS) providers.		

17.3.3 Minimal

```
ADR-001: We decided for MySQL as our relational database in the backend
because this database has been approved strategically by our EAM group,
has good tool support and sufficient performance under the workload
that we expect to be caused by the user stories to be implemented in the next 10 sprints.
```

17.4 arc42

Effective, lean and pragmatic architecture documentation and communication.

17.4.1 Template (Sections)

1. Introduction and goals	Requirements, stakeholder, (top) quality goals
2. Constraints	Technical and organizational constraints, conventions
3. Context and scope	Business and technical context, external interfaces
4. Solution strategy	Fundamental solution decisions and ideas
5. Building bock view	Abstractions of source code, black-/whiteboxes
6. Runtime view	Runtime scenarios: how do building blocks interact?
7. Deployment view	Hardware and technical infrastructure, deployment
8. Crosscutting concepts	Recurring solution approaches and patterns
9. Architecture decisions	Important decisions
10. Quality	Quality tree and quality scenarios
11. Risks and technical dept	Known problems, risks and technical dept
12. Glossary	Definitions of important business and technical terms

17.4.2 Template (Tips)

Introduction and goals	1. Give a compact summary of requirements and driving forces! 2. Limit yourself to the essential tasks and use cases! 3. Highlight the business goals of the system! 4. Create an overview by grouping or clustering requirements! 5. Make sure you can reference (existing) requirements! 6. Use activity diagrams to describe functional requirements! 7. Use BPMN diagrams to describe functional requirements! 8. Use a numbered list to describe functional requirements! 9. Use (semi) formal text to describe functional requirements! 10. Use 'exemplary business process models' to describe functional requirements! 11. Always work with explicit quality requirements! 12. Explain quality requirements through scenarios! 13. If you do not get quality requirements, make your assumptions *explicit*! 14. Use checklists for quality requirements! 15. Use examples to work out quality goals together with your stakeholders! 16. Describe only the top 3-5 quality goals in the introduction! 17. Combine quality goals with the action points of the 'solutions strategy' section! 18. Defer detailed and complete quality requirements to arc42 section 10! 19. Search broadly for stakeholders! 20. Describe the expectations of stakeholders! 21. Maintain a stakeholder table! 22. Skip the stakeholder table if your management already maintains it! 23. Classify your stakeholders by interest and influence!
Constraints	1. Consider the contraints of other systems within the organization! 2. Clarify the consequences of constraints! 3. Document organizational constraints! 4. Document design and development constraints! 5. Differentiate different categories of constraints!
Context and scope	1. Explicitly demarcate your system from its environment! 2. Show the context as diagram! 3. Combine the context diagram with a table! 4. Explicitly indicate risks in the context! 5. Restrict the context to an overview, avoid too many details! 6. Simplify the context by categorization! 7. If many external systems are involved, aggregate (cluster) them by explicit criteria! 8. Aggregate (cluster) similar neighbour systems with ports! 9. Show all (all!) external interfaces! 10. Differentiate business and technical context! 11. In the business context, show data flows (instead of dependencies)! 12. Show external influences in the context! 13. Show transitive dependencies in the context! 14. Pay attention to quality requirements at external interfaces! 15. Show the technical context (in case hardware is central to your system)! 16. Use the technical context to describe protocols or channels! 17. Combine business context with technical information! 18. Explain the relationship between domain interfaces and their technical realization! 19. Defer technical context to the deployment view!

Solution strategy	1. Explain the solution strategy as compact as possible (e.g. as list of keywords)! 2. Describe the solution approaches as a table! 3. Describe solution approaches in context of quality requirements! 4. In the solution strategy, refer to concepts, views or code! 5. Let the solution strategy grow iteratively / incrementally! 6. Justify the solution strategy!
Building block view	1. Use common structures for sections of the building block view! 2. Organize the building block view hierarchically! 3. Always describe level-1 of the building block view ('Level-1 is your friend')! 4. Ensure consistency of external interfaces to level-1 5. Describe the responsibility or purpose of every (important) blackbox! 6. Hide the inner workings of blackboxes! 7. Use tables to efficiently document/specify blackboxes! 8. Justify every whitebox structure! 9. Use runtime views to explain or specify whiteboxes! 10. Use crosscutting concepts to describe or specify similarities in building blocks! 11. Show multiple levels of the building block view! 12. Refine building-blocks consistently! 13. Explain the mapping of source-code to building blocks! 14. Explain where to find the source code of your building blocks! 15. Align the mapping of source-code to building-blocks along the directory and file structure! 16. Map building blocks according to modularization constructs of your programming language! 17. 'Cohesion' shall be the primary driver when creating architecture building blocks! 18. Ensure **every** piece of source code can be located in the building block view! 19. In exceptional cases include third-party software in the building block view! 20. Clearly indicate third-party elements in the building block view! 21. Describe or specify internal interfaces with minimal effort! 22. Document or specify interfaces with unit-tests! 23. Document or specify interfaces with runtime scenarios! 24. Use building-block level-1 for **other** information! 25. If useful, refine several building blocks at once! 26. Make the origin of lower-level building blocks explicit! 27. Refine only a few building blocks! 28. Explain concepts instead of building blocks!
Runtime view	1. Always map existing building blocks to the activities within runtime scenarios! 2. Document only a few runtime scenarios! 3. Document 'schematic' (instead of detailed) scenarios! 4. Document detailed scenarios (with caution)! 5. Use scenarios primarily to `discover` building blocks, not so much for documentation! 6. Describe excerpts of scenarios (partial scenarios)! 7. Use activity diagrams with swimlanes to describe or specify runtime scenarios! 8. Use activity diagrams with partitions to describe or specify runtime scenarios! 9. Use a textual notation to describe runtime scenarios! 10. Use both small and large building blocks in scenarios! 11. Use sequence diagrams to describe or specify runtime scenarios!

Deployment view	1. Document your technical infrastructure (hardware)! 2. Explain hardware and infrastructure decisions! 3. Document the various environments! 4. Document the deployment view hierarchically! 5. Document the mapping of building-blocks to hardware! 6. Use UML deployment diagrams to document software/hardware mapping! 7. Use tables to document software/hardware mapping!! 8. Explain your nodes! 9. Explain what (else) is relevant for productive use (aka operation) of the system! 10. Leave hardware decisions to hardware-experts!
Crosscutting concepts	1. Explain the Concepts! 2. Concepts are approaches, rules, principles, tactics, strategies etc... 3. Restrict documentation of concepts to the most important topics! 4. In concepts, explain HOW it works! 5. Document business or domain models! 6. Combine business or domain models with the glossary! 7. Document (at least) the (business or domain) data model! 8. Document concepts with source code! 9. Document decisions instead of concepts! 10. Use the collection from arc42 as checklist for concepts!
Architecture decisions	1. Document only architecturally relevant decisions! 2. Document decision criteria! 3. Provide reasons for important decisions! 4. Document decisions as mind-map or as table! 5. Document decisions as `Architecture Decision Record` (ADR)! 6. Document rejected alternatives! 7. Document decisions informally as a blog (RSS-feed)!
Quality	1. Keep the quality goals in arc42-section 1.2 short! 2. Document and explain the specific quality tree! 3. Use a mind-map as quality tree! 4. Use the quality tree as checklist! 5. Consider usage or application (quality) scenarios! 6. Consider change (quality) scenarios!! 7. Consider fault/error/failure (quality) scenarios!! 8. Use (quality) scenarios for architecture analysis or evaluation!
Risks and technical dept	1. Search for problems and risks with different stakeholders! 2. Analyze (external) interfaces for problems and risks! 3. Identify problems or risks by qualitative evaluation! 4. Analyze _processes_ for problems and risks! 5. Analyze data or data structures for problems and risks! 6. Analyze source-code for problems and risks!
Glossary	1. Take the glossary seriously! 2. Document the glossary as a table! 3. Amend the glossary by a (graphical) model! 4. Include translations in the glossary! 5. Keep the glossary compact! Avoid trivia. 6. Make your 'product owner' or 'project manager' responsible for the glossary!

17.5 Tips and Tricks

A few examples of good and bad AD justifications:

Decision driver type	Valid justification	Counter example
Wants and needs of external stakeholders	Alternative A best meets user expectations and functional requirements as documented in user stories, use cases, and business process model.	End users want it, but no evidence for a pressing business need. Technical project team never challenged the need for this feature. Technical design is prescribed in the requirements documents.
Architecturally significant requirements	Nonfunctional requirement XYZ has higher weight than any other requirement and must be addressed; only alternative A meets it.	Do not have any strong requirements that would favor one of the design options, but alternative B is the market trend. Using it will reflect well on the team.
Conflicting decision drivers and alternatives	Performed a trade-off analysis, and alternative A scored best. Prototype showed that it's good enough to solve the given design problem and has acceptable negative consequences.	Only had time to review two design options and did not conduct any hands-on experiments. Alternative B does not seem to perform well, according to information online. Let's try alternative A.

Decision driver type	Valid justification	Counter example
Reuse of an earlier design	Facing the same or very similar NFRs as successfully completed project XYZ. Alternative A worked well there. A reusable asset of high quality is available to the team.	We've always done it like that. Everybody seems to go this way these days; there's a lot of momentum for this technology.
Prefer do-it-yourself over commercial off-the-shelf (build over buy)	Two cornerstones of our IT strategy are to differentiate ourselves in selected application areas, and remain master of our destiny by avoiding vendor lock-in. None of the evaluated software both meets our functional requirements and fits into our application landscape. We analyzed customization and maintenance efforts and concluded that related cost will be in the same range as custom development.	Price of software package seems high, though we did not investigate total cost of ownership (TCO) in detail. Prefer to build our own middleware so we can use our existing application development resources.
Anticipation of future needs	Change case XYZ describes a feature we don't need in the first release but is in plan for next release. Predict that concurrent requests will be x per second shortly after global rollout of the solution, planned for Q1/2009.	Have to be ready for any future change in technology standards and in data models. All quality attributes matter, and quality attribute XYZ is always the most important for any software-intensive system.

Common pitfalls:

- Stick to one template throughout a project. It is less relevant which one you pick - as long as you pick one.
- Focus on the essence of the decision, but make sure the individual ADRs and the entire decision log can serve their purpose (quick orientation, long time reference)
- Don’t spend more time on decision capturing than on decision making (and preparing this activity)
- Assign an identifier and create minimal meta information such as decision owner/makers, status and timestamp.
- “will look good on my CV” does not qualify as a sound decision rationale for a technology selection decision

Keep in mind: we want to avoid knowledge vaporization!

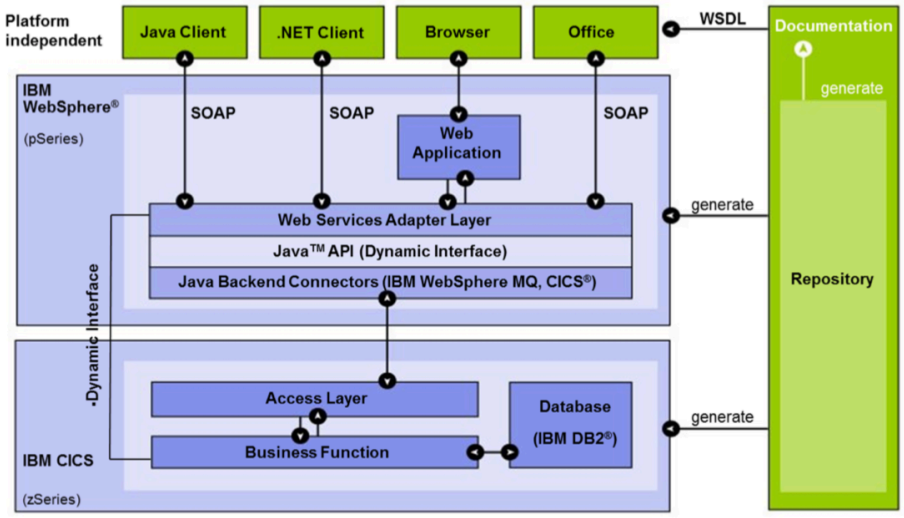
17.6 Architecturally Evident Coding Style (AECS)

- **G. Fairbanks introduces this concept in conference tutorials, based on his "Just Enough Software Architecture" book:**
  - For instance at SATURN 2013 and Agile Roots 2010,
  - Key ideas (organized in 6 slices in talk and 10 intents in book):
    - Components and patterns should be visible in code (e.g., abstract classes be defined for components/patterns)
    - Startup code should be centralized (and explicitly named) so that it can be located easily and tested separately
    - Quality-related properties should be marked as such g, with names or annotations
- **Finding good names is hard**
  - Two hard things: "cache invalidation, naming things, and off-by-1 errors"
- **ZIO's main advice:**
  - 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* (but do not go too far, 2-3 words per name will do)
    - Use concrete concept from domain in name
    - Use pattern name in name to indicate arch. role and resp.
- **Other things you can do:**
  - Make external interfaces very visible, add sample data to comments
  - Indicate memory requirements (state management is key concerns)
  - Practice the Ubiquitous Language everywhere (as the name suggests)
    - Put methods like `checkPrecondition()` or `authorizeRequest()` in same place – and use these names rather than generic `check()` or semi-cryptic `evalRBACPol()`

Names should be intention revealing and related to the problem domain!

18 Example Architectures

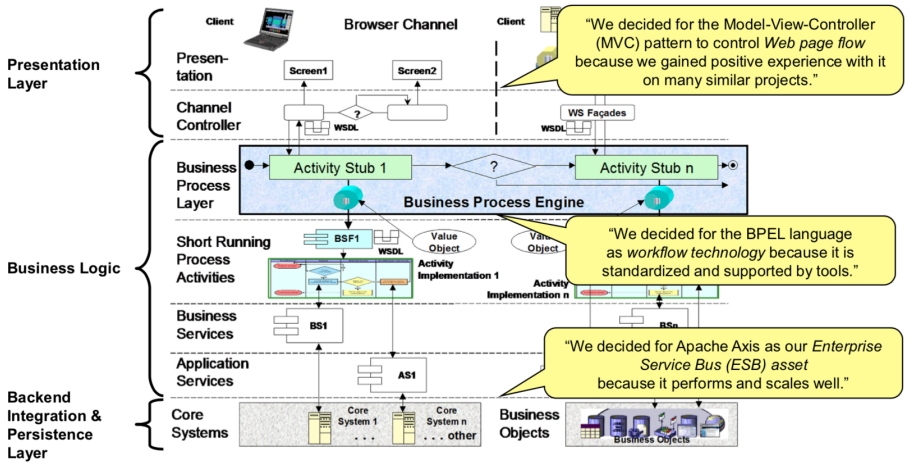
18.1 Core Banking SOA



Architectural Elements:

- Layers pattern
- Web frontend
- Web services
- Data store in backend

18.2 Telecom Order Management SOA

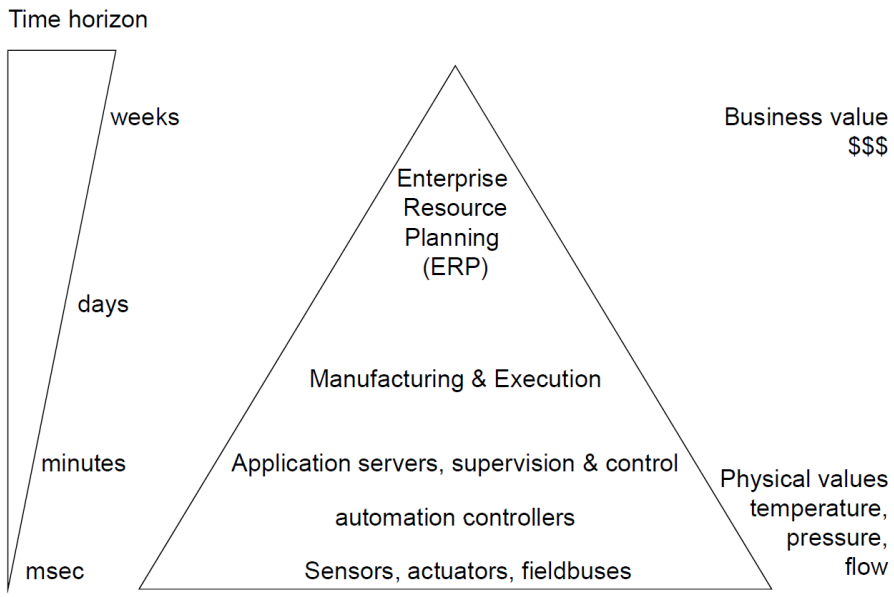


Architectural Elements:

- Layers pattern
- MVC pattern
- Business process engine
- ESB for backend

18.3 Distributed Control System

A DCS is used to control and monitor production facilities (e.g. power plant).



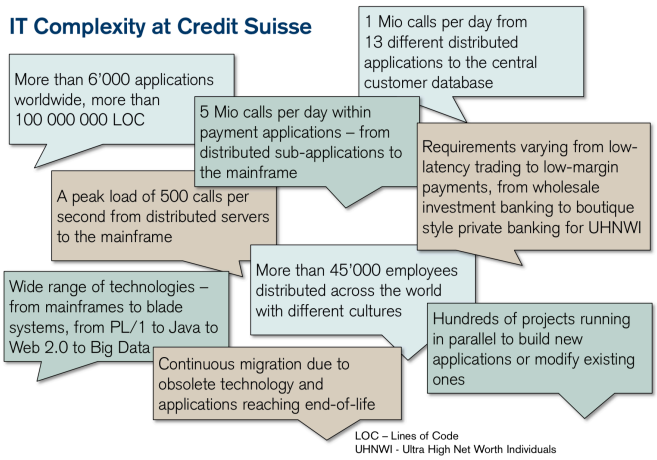
Architectural Elements:

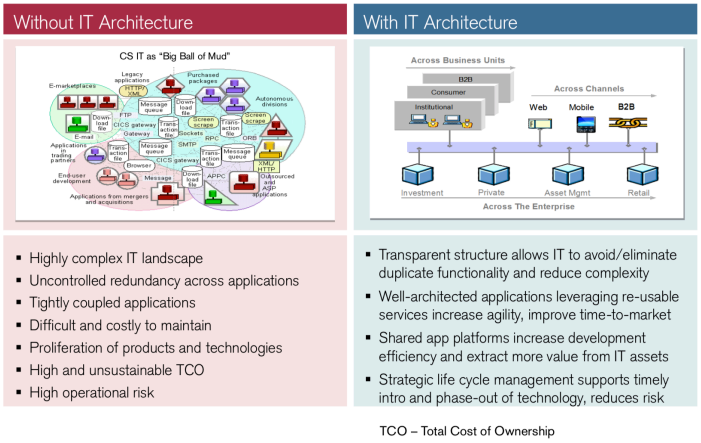
Layers of abstraction (from hardware components to high level production processes)

- Level 4
  (production scheduling)
- Level 3
  (production control, monitoring)
- Level 2
  (supervisory computers, control screens)
- Level 1
  (controllers, firmware)
- Level 0
  (sensors, actuators)

19 Guest Lectures

19.1 Architecting at Credit Suisse

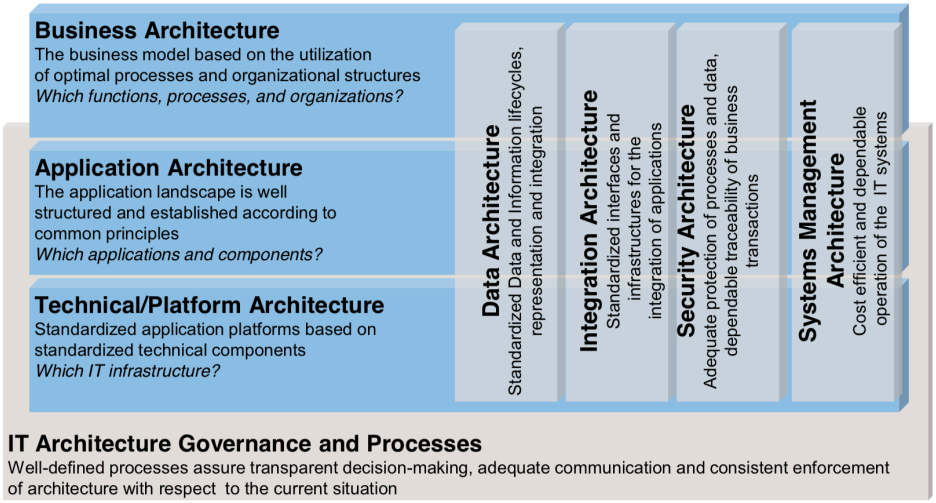




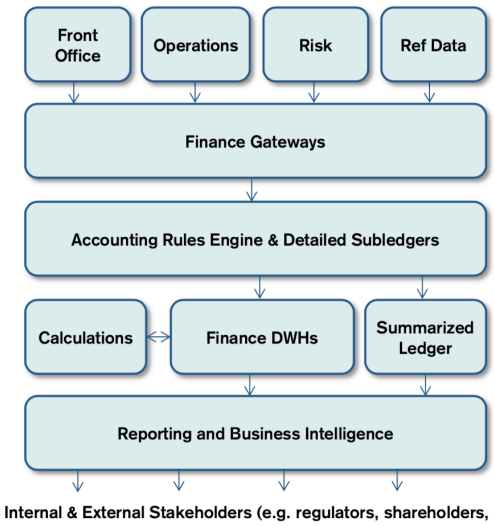
19.1.1 Key Messages

- IT Architecture helps to **address** one of the key IT challenges of large organizations like banks - **complexity**.
- IT Architecture is **multi-dimensional** and spans **many disciplines**, from business and data architecture to security and integration architecture.
- Different requirements, especially **non-functional requirements**, lead to a different choice of **technologies**, **patterns** and architectural **approaches**.
- **Innovative technologies** such as Big Data open up many new opportunities, but are **not necessarily a solution** to all problems.

19.1.2 "Three + Four + One" Disciplines



19.1.3 Logical Architecture



- Data collected for all divisions, regions
- Books and records of the firm
- Daily, monthly, quarterly reporting
- To internal and external stakeholders

- >100 systems globally
- Mainframe, DWH, vendor, …
- PL1, PL SQL, Java, C++, …
- Several Big Data applications