Application Architecture

HS2023

Marco Agostini
Jan Untersander
Joshua Beny Hürzeler

Based on the lecture by
Dr. Prof. Olaf Zimmermann

Computer Science
University of Applied Sciences of Eastern Switzerland

Contents

1 Introduction

There is a difference between Information Technology/Software Architecture and Application Architecture. Software Architecture includes the operation and security of the running software, on the other hand Application Architecture does mainly cover the static architecture.

What is Software Architecture?

Bass, Clemens, Kazman, 1998 The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them.

Jansen, Bosch, 2005 A software systems architecture is a set of principle design decision made about the system.

ISO/IEC/IEEE 42010, 2011 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 designs and evolution.

What is Architectural significant?

- Number of Users
- Availability
- Lifetime
- Extendability
- Regulations
- Laws
- Safety and Security

1.1 High-Level Example

Architecture is everywhere in software engineering. An example can be the process automation and supervision in the production industry. All layers in the automation pyramid, from sensors and actuators to Distributed Control Systems (DCS) are part of it. An example shows Figure 1 of ABB.

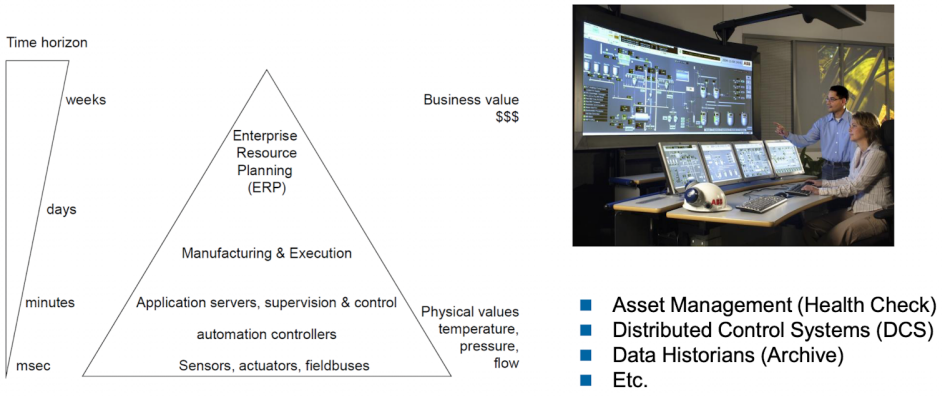


Figure 1: Automation Pyramid of ABB

1.2 Architectural Significance Requirements (ASRs)

Architecturally significant requirements are those requirements that have a measurable effect on a computer system's architecture. ASRs help the developer/architect to prioritize technical issues or requirements quickly so that architecturally significant issues are addressed at their most responsible moment and we do not have to revise decisions and designs unnecessarily later.

The time budget per issue is one to two minutes at most, as it is not unusual to be confronted with 10 to 100 issues per day — thanks to email and auto-notifications in tools such as Kanban boards, issue trackers, and source code management systems.

Architecturally Significant Requirements
The following points enable a developer to check if a requirement is architecturally significant, therefore make an ASR test. The points six and seven are very subjective and often not considered as must have for a story to be architectural significant.

1. The requirement is directly associated with high **business value** or **business risk**.
2. The requirement is a **concern** of a particularly **important stakeholder** (for instance, the project sponsor or an external compliance auditor).
3. The requirement has runtime **Quality-of-Service (QoS)** characteristics (e.g., performance needs) that deviate from those already satisfied by the evolving architecture substantially.
4. The requirement causes new or deals with one or more existing **external dependencies** that have unpredictable, unreliable and/or uncontrollable behaviour.
5. The requirement has a **cross-cutting nature** and therefore affects multiple parts of the system and their interactions; it may even have **system-wide impact**.
6. The requirement has a **first-of-a-kind** character: e.g., the team has never built a component that satisfies this particular requirement.
7. The requirement has been **troublesome** and caused critical situations, budget overruns or client dissatisfaction **in a previous project** in a similar context.

Example ASR Test
The following paragraph gives an example analysis of requirements regrading their architectural significance. Please make sure to always justify the answer in order to make them understandable.

Requirement	Score	Mapping	Explanation
Autoscaling for Spinnaker microservices running on Kubernetes	High (H)	RC-2, RC-3, RC-4 Possibly RC-7	Auto scaling may have an impact on the billing, so external stakeholder affected (cloud provider); scalability is a key quality attribute; dependency on 3rd party software; autonomous system behavior has been reported to be hard to test and maintain due to number of external stimuli and time dependency

Table 1: Sample Requirement Analysis for Architectural Significance

1.3 SMART Non-Functional Requirements

The SMART criteria are frequently used in project and people management, but they can also be applied to Non-Functional Requirement (NFR) engineering. Most of the time only *S* and *M* are used in NFR engineering.

Specific Which feature or part of the system should satisfy the requirement.

Measurable How can testers and other stakeholders fond out whether the requirement is met (or not)? Is the requirement quantified.

Aagreed Upon The goal must be realistic and achievable and a consent in the team.

Relevant/Realistic One must be aware of why one wants to achieve it.

Time-Bound With a clearly defined timeline, including a starting date and a target date.

Architectural Significance of Design Elements
In addition to the ASRs there is also the need to to specify is a structural design element (e.g. component, connector, class, OOP method) is considered architecturally significance.

1. The element is associated with some **critical functionality** of the system / e.g. money transfer

2. The element is associated with a **critical quality** on the solution / e.g. performance of distributed communication
3. The element is associated with a **critical constraint** of the solution / e.g. access to an external system
4. The element incurs a particular **technical risk** / e.g. access to a never-before-tried capability
5. The element presents a **particular architectural change** / e.g. high transaction volume

A table is the obvious format for the resulting seven-criteria ASR test as displayed in the Figure 2.

Issue	Value/Risk	Key Concern	New QoS	Ext. Dep.	X-Cutting	FOAK	Past Pb	Score
issue 1	☐	☐	☐	☐	☐	☐	☐	___
issue 2	☐	☐	☐	☐	☐	☐	☐	___
...	☐	☐	☐	☐	☐	☐	☐	___

Figure 2: A Scoring/Assessment Tempalte Table

1.4 FURPS+

The FURPS+ acronym, devised by Robert Grady of HP, provides a way to define requirements by non-functional stories, and also provides a good way to categorize such needs. The breakdown here suggests some representative questions around potential needs.

F unctionality: represents the main product features that are familiar within the business domain of the solution being developed. The functional requirements can also be very technically oriented.

U sability: includes looking at, capturing, and stating requirements based around user interface issues — things such as accessibility, interface aesthetics, and consistency within the user interface.

R eliability: includes aspects such as availability, accuracy, and recoverability — for example, computations, or recoverability of the system from shut-down failure.

P erformance: involves things such as throughput of information through the system, system response time (which also relates to usability), recovery time, and start-up time.

S upportability: we tend to include a section called supportability, where we specify a number of other requirements such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, localizability, and so on.

+ allows us to specify constraints, including design, implementation, interface, and physical constraints.

Example FURPS+ Requirements

F As a meeting organiser, i would like to propose some time slots so that invitees can indicate their availability, which eases my decisions making and further planning.

U All information should be displayed on one page/screen; adding a time slot should be a single step that can be completed in a few seconds.

R The calendar service should be highly available during office hours (8am to 5pm); it should not crash when invalid data is entered

P Time slot additions should be confirmed and displayed correctly within two seconds; the overview page should load within one second.

S Severity 1 bugs are fixed within 48 business hours, severity 2 issues within one calendar week.

2 Quality Attributes Scenarios (QAS)

System quality attributes have been of interest to the software community at least since the 1970s. There are a variety of published taxonomies and definitions, and many of them have their own research and practitioner communities. From an architect's perspective, there are three problems with previous discussions of system quality attributes:

- The definitions provided for an attribute are not operational. It is meaningless to say that a system will be modifiable. Every system is modifiable with respect to one set of changes and not modifiable with respect to another. The other attributes are similar.
- A focus of discussion is often on which quality a particular aspect belongs to. Is a system failure an aspect of availability, an aspect of security, or an aspect of usability? All three attribute communities would claim ownership of a system failure.
- Each attribute community has developed its own vocabulary. The performance community has "events" arriving at a system, the security community has "attacks" arriving at a system, the availability community has "failures" of a system, and the usability community has "user input." All of these may actually refer to the same occurrence, but are described using different terms.

A solution to the first two of these problems (nonoperational definitions and overlapping attribute concerns) is to use Quality Attribute Scenarios as a means of characterizing quality attributes. A solution to the third problem is to provide a brief discussion of each attribute-concentrating on its underlying concerns-to illustrate the concepts that are fundamental to that attribute community.

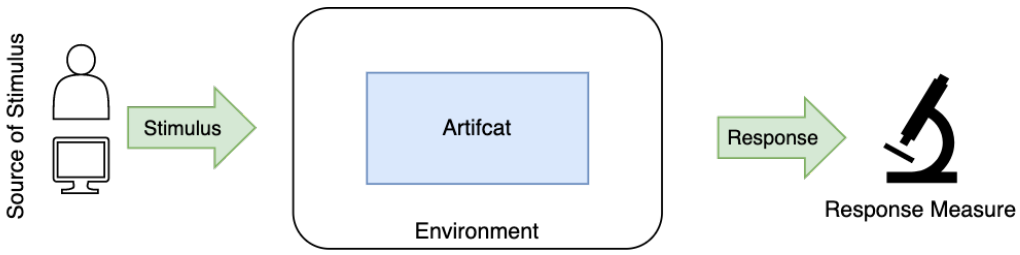


Figure 3: Quality Attribute Scenario Visualization

Scenario for Requirement XYZ		
Scenario(s)		*[Name]*
Business Goals		*[Reference project vision statement and functional requirements]*
Relevant Quality Attributes		*[List of NFRs/QAs in keywords, from a taxonomy]*
Scenario Components	**Stimulus**	*What does the source do to start the scenario (event, trigger)?*
	Stimulus Source	*Who initiates the scenario (internal, external stakeholder)? Which feature of other (functional) requirement is it attached to?*
	Environment	*When can the specified response behavior be observed? Which state of the system does it pertain to (e.g. normal operations, peaks, error mode)?*
	Artifact	*Where can the specified response behavior be observed? Which part of the system or the engineering and evolution process is affected?*
	Response	*How does the artifact in the environment react to the stimulus? What should be measured?*
	Response Measure	*How can the response behavior be quantified to make it observable? What is the measurement goal (target)?*
Questions		*Which design or planning discussions and decisions are needed?*
Issues		*Which technical/organizational problems can be identified in the scenario?*

Figure 4: Quality Attribute Scenario (QAS) Template

2.1 Landing Zone

At first glance a landing zone seems nothing more than a glorified table. Each row in the landing zone represents a measurable requirement. Each requirement has a range of acceptable values labeled Minimum, Target, and

Outstanding. The goal is to have each requirement within this range at the end of development. This can be used to defined a better Response Measure in the QAS.

The Landing Zone allows for some flexibility in the development of an application and tolerance in the acceptance values.

Response Time (per Business Activity)	Minimal Goal (Less Than)	Target (Within)	Outstanding (Within)
Order fully processed	2 weeks	24 hours	3 hours
Relocation processed	3 weeks	2 weeks	1 week
Technician appointment scheduled	2 days	1.5 days	1 day
Address validated	10 seconds	3 seconds	1 second
Billing system configured	1 week	3 days	1 day

Figure 5: Landing Zone

2.2 Quality Utility Trees

The Quality Utility Tree is a structure to visualize QAS, which are the leaves of the tree which are priorized py value and risk (V, R) which is shown in Figure 6.

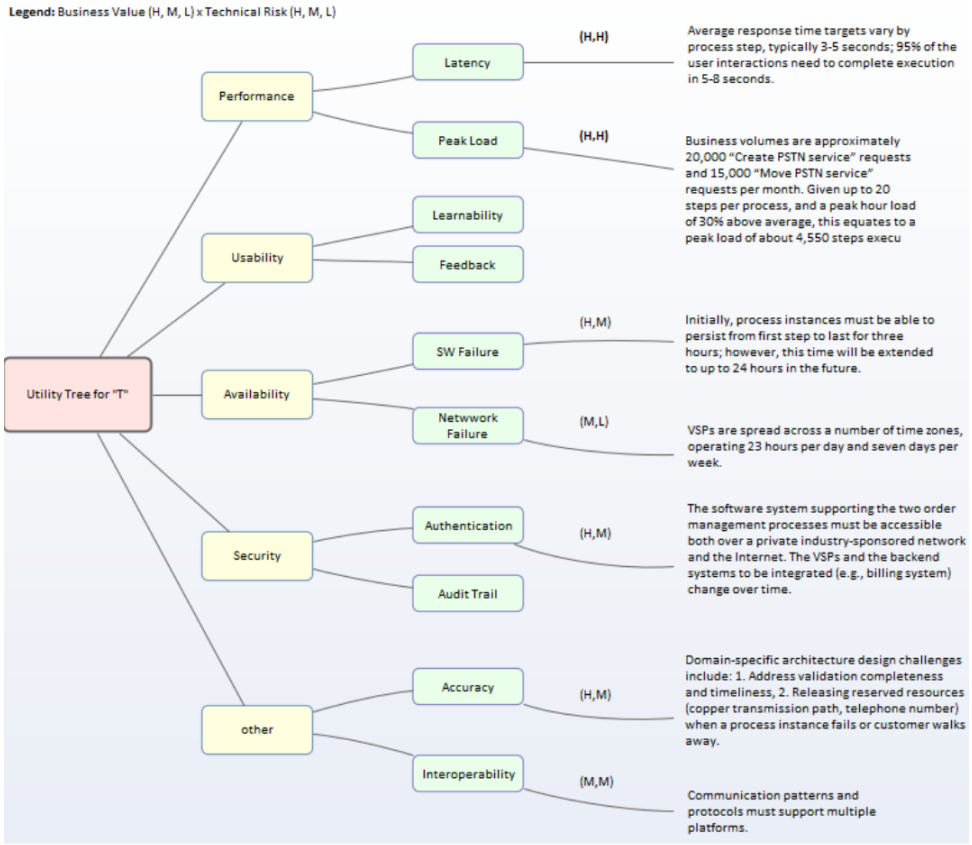


Figure 6: Quality Utility Tree

2.3 Twin Peaks Model

The importance of interleaving the tasks of eliciting and specifying requirements with that of designing a software solution has long been recognized. The traditional waterfall process produces artificially frozen requirements that

often lead to suboptimal architectural solutions, which themselves are often inflexible to future change. Incremental development processes partially address this problem by allowing developers to evaluate repeatedly changing project risks in order to embrace new requirements and to address changing project constraints. The twin peaks model provides an even finer-grained iteration across requirements and architecture that acknowledges the need to develop software architectures that are stable, yet adaptable, in the presence of changing requirements.

As shown in Figure 7, the twin peaks model focuses on the co-development of requirements and architecture. Through a series of iterations, the model captures the progression from general to detailed understanding and expression of both requirements and design. Although the schematic of the twin peaks model shows the process initiated on the requirements peak, projects involving modifications to existing systems could be initiated equally well at the architecture peak.

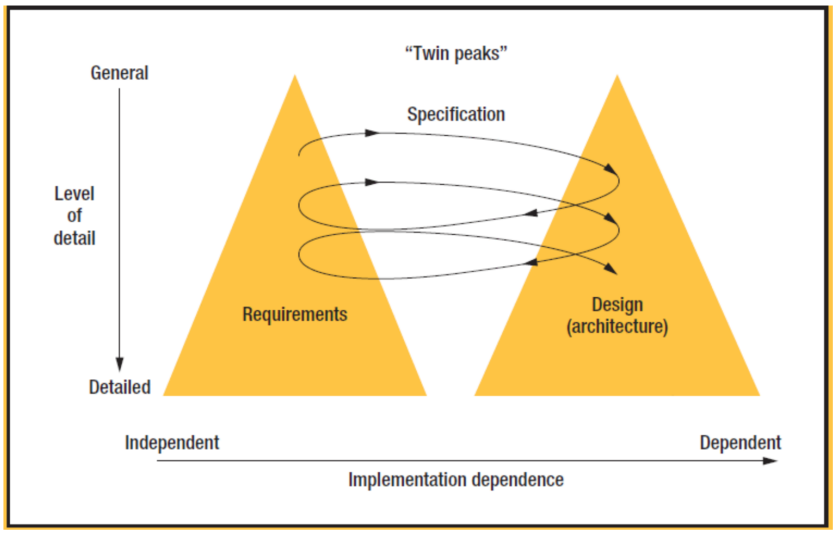


Figure 7: IEEE Twin Peak Model

3 C4 Model for Architecture Visualization

The C4 model considers the static structures of a software system in terms of containers (applications, data stores, microservices, etc.), components, and code. It also considers the people who use the software systems that we build. The C4 model has predefined modelling entities defined in Figure 8. The following section does display multiple sample visualizations and there are still further diagrams such as: System Landscape, Dynamic, Deployment diagram. Additionaly the follwing talk by the creator of the C4 Model, Simon Brown, gives a good overview of it.

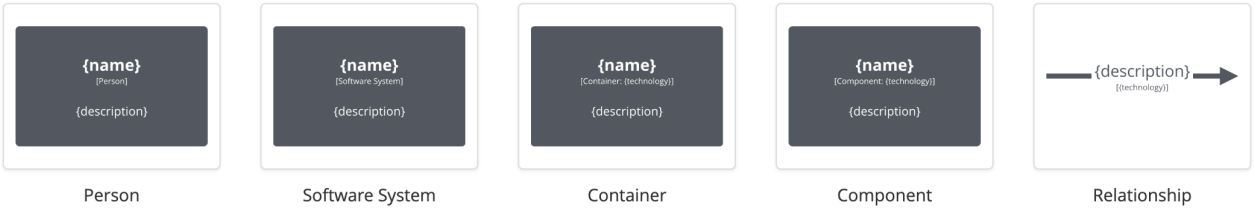


Figure 8: C4 Model Notation

3.1 Level 1: System Context Diagram

A system context diagram, shows the software system you are building and how it fits into the world in terms of the people who use it and the other software systems it interacts with.

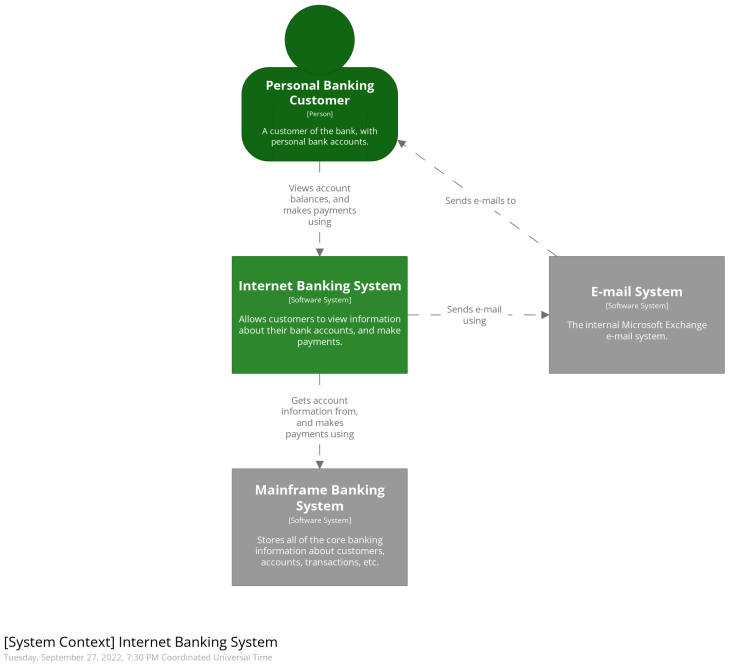


Figure 9: C4 Context Diagram

3.2 Level 2: Container Diagram

A container diagram, zooms into the software system, and shows the containers (applications, data stores, microservices, etc.) that make up that software system. Technology decisions are also a key part of this diagram.

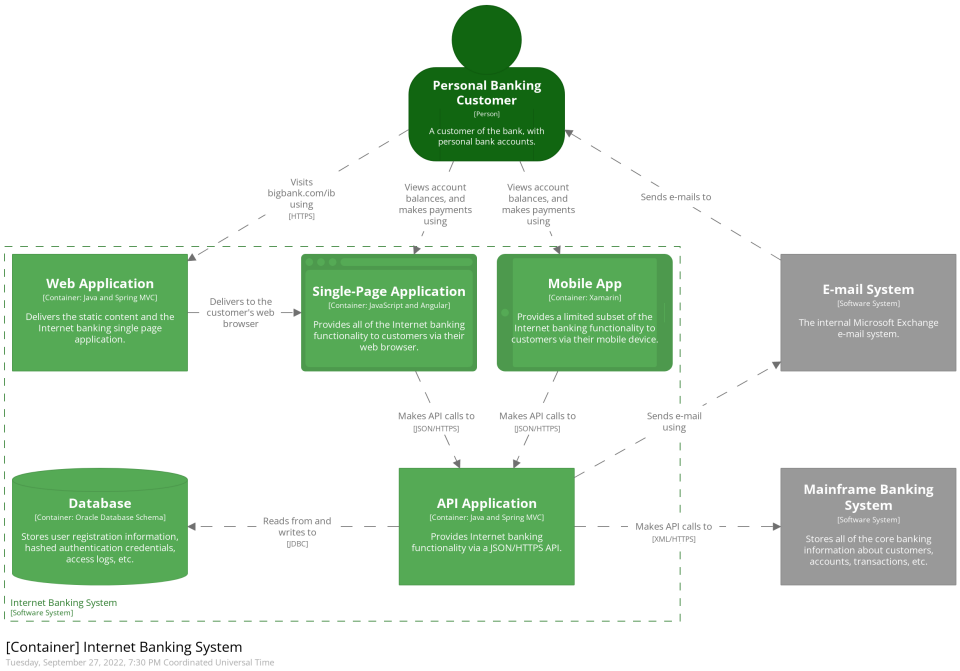


Figure 10: C4 Container Diagram

3.3 Level 3: Component Diagram

A component diagram, zooms into an individual container to show the components inside it. These components should map to real abstractions (e.g., a grouping of code) in your codebase.

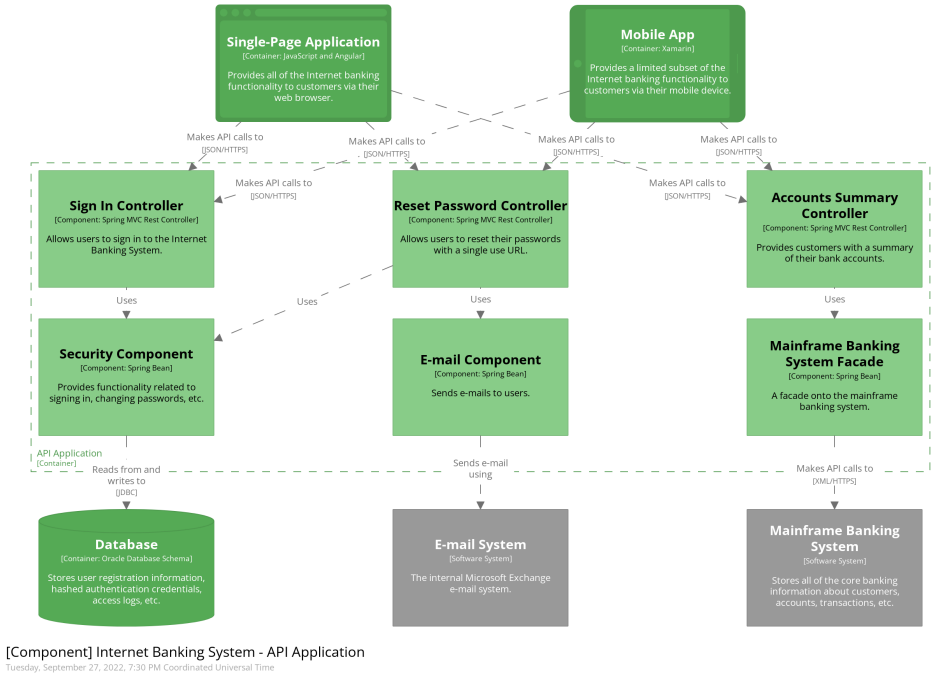


Figure 11: C4 Component Diagram

3.4 Level 4: Code

Finally, if you really want or need to, you can zoom into an individual component to show how that component is implemented. There nearly is never the need to draw a Code diagram, since it gives a very low level idea on how the code is structured.

4 Architectural Decisions, Layers and Tiers

4.1 Solution Strategy

It includes the fundamental decisions and solutions strategies that shape the system's architecture. These decisions form the cornerstones for your architecture. They are the basis for many other detailed decisions or implementation rules.

These include:

- Technology decisions
- Decisions about the top-level decomposition of the system, e.g. usage of an architectural pattern or design pattern
- Decisions on how to achieve key quality goals
- Relevant organizational decisions, e.g. selecting a development process or delegating certain tasks to third parties.

4.2 Architectural Decisions (AD)

Architectural decisions capture key design issues and the rationale behind chosen solutions. They are conscious design decisions concerning a software-intensive system as a whole or one or more of its core components and connectors in any given view. The outcome of architectural decisions influences the system's nonfunctional characteristics including its software quality attributes.

An **Architectural Decision Record (ADR)** captures a single AD, such as often done when writing personal notes or meeting minutes; the collection of ADRs created and maintained in a project constitute its decision log.

Y-Template
Writing formal ADD can be hard and time consuming. Also it is a good idea to make them accessible to the developer team, to make sure they can be reread again and also made sure they are not forgotten. The Y-Model in figure 12 from ABB provides a template to write ADDs.

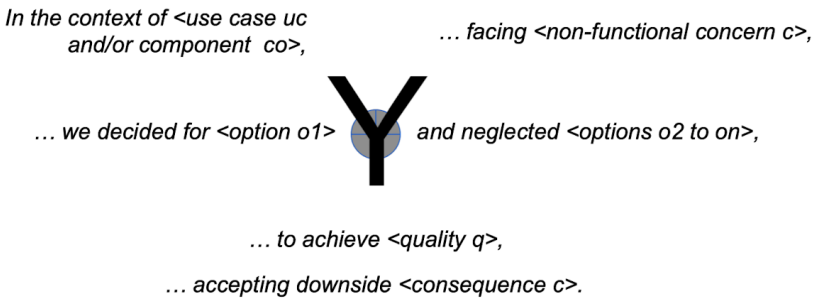


Figure 12: Architectural Design Decision

Example ADR with the Y-Model: In the context of the frontend facing the NFR fast response time, we decided for Deck, a non-distributed presentation layer and neglected ExpressJS to achieve fast response times and accepting the heavier load on the user machines.

4.3 Logical Layering

Most of the time we have a system needs to be segmented due to its size and specialisation. The problem is, that we need to have a system design, that supports a mixture of tasks on different levels. The solution we have in software engineering is, we organise the system into cohesive layers that are stacked on top of each other. This approach is called the Layers Pattern and are typically applied twice: **logical and physical** view.

Two of the big decisions in Solution Strategy is on choosing layers and tiers. It is important to understand the distinction between layers and tiers.

Layers describe the logical groupings of the functionality and components in an application

Tiers describe the physical distribution of the functionality and components on separate servers, computers, networks, or remote locations

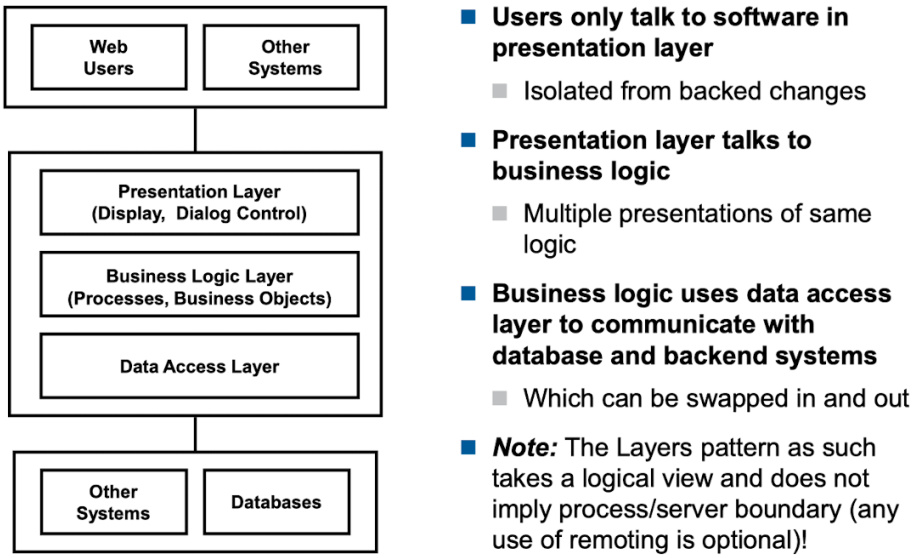


Figure 13: Layer Pattern

4.4 Distribution Patterns

The Problem is now, how do I partition an application into a number of client and server components so that my users' functional and non-functional requirements are met?

To distribute an information system by assigning client and server roles to the components of the layered architecture we have the choice of several distribution styles. Figure 14 shows the styles which build the pattern language which are called **Client/Server Cuts**.

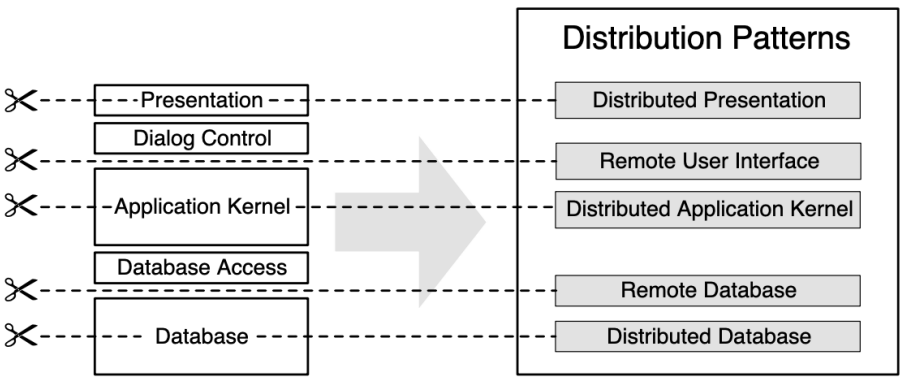


Figure 14: Overview of the patterns resulting from different client/server cuts

To take a glance at the pattern language we give an abstract for each pattern:¨

Distributed Presentation This pattern partitions the system within the presentation component. One part of the presentation component is packaged as a distribution unit and is processed separately from the other part of the presentation which can be packaged together with the other application layers. This pattern allows of an easy implementation and very thin clients.

Remote User Interface Instead of distributing presentation functionality the whole user interface becomes a unit of distribution and acts as a client of the application kernel on the server side.

Distributed Application Kernel The pattern splits the application kernel into two parts which are processed separately. This pattern becomes very challenging if transactions span process boundaries (distributed transaction processing).

Remote Database The database is a major component of a business information system with special requirements on the execution environment. Sometimes, several applications work on the same database. This pattern locates the database component on a separate node within the system's network.

Distributed Database The database is decomposed into separate database components, which interact by means of interprocess communication facilities. With a distributed data- base an application can integrate data from different database systems or data can be stored more closely to the location where it is processed.

There are pros and cons for using different distribution patterns. Take a look at figure 15, where they are explained.

Pattern	Advantage (Pro)	Disadvantage (Con)
Remote User Interface	Connects distributed target audiences, isolates frequent technology changes from application core, user experience	Software distribution required, security threat(s) introduced, portability of client device-specific code questionable
Distributed (Split) Presentation	Display (only) can be optimized for devices/channel (e.g. mobile, Web, PC OS) to achieve best user experience	Management effort, input must be validated twice, may double technology dependencies, interoperability/portability?
Distributed (Split) Application Kernel	Worker components can specialize on tasks, difficult ones can be "outsourced" to specialized nodes (hardware, tools)	May violate DRY principle, introduces interoperability issues and complexity
Remote Database	Can evolve and be managed independently (by specialists), can host multiple applications (information hub)	Requires extra hardware and (sub-) network that need to be managed and secured
Distributed (Split) Database	Scales well and can improve reliability (when designed right)	Consistency endangered, data retention policies more complex, has to be secured

Figure 15: Pros and Cons of Distribution Patterns

Each pattern presents a partial distribution view of the application architecture and maps this to a physical system structure. As a solution addresses only a part of the application components the whole distribution model may result from applying more than one distribution pat- tern. Instances of application components can „live" on different nodes of the technical architecture, because of components being part of more than one distribution unit or distribution units mapped to different nodes.

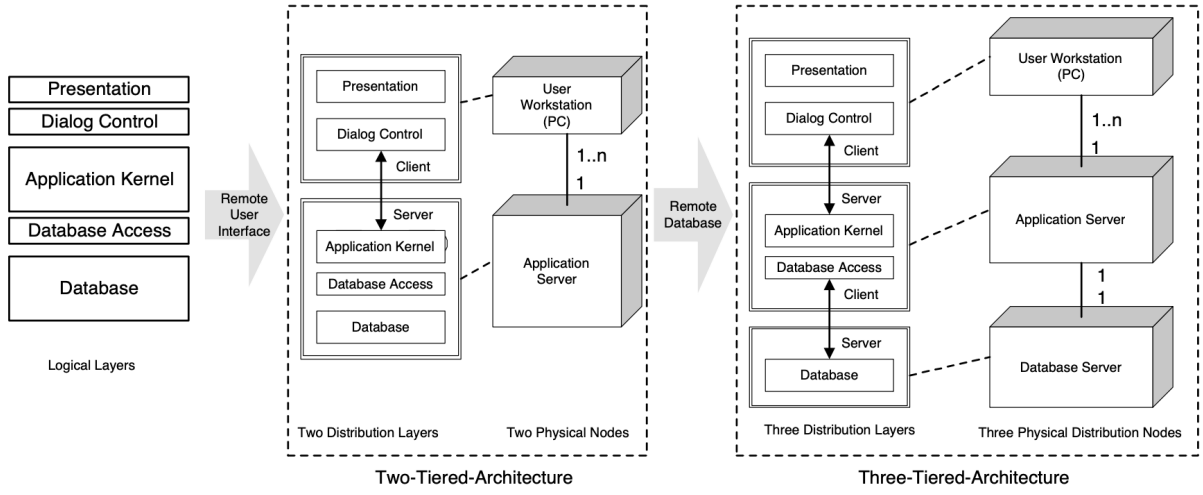


Figure 16: Building a Three-Tiered-Architecture applying the Remote User Interface pattern and the Remote Database pattern

4.5 arc42

arc42 is an open source documentation template and is based on practical experience of many systems in various domains, from information and web systems, real-time and embedded to business intelligence and data warehouses. arc42 answers the following two questions in a pragmatic way and can be tailored to your specific needs: *What should you document/communicate about your architecture?, How should you document/communicate?*.

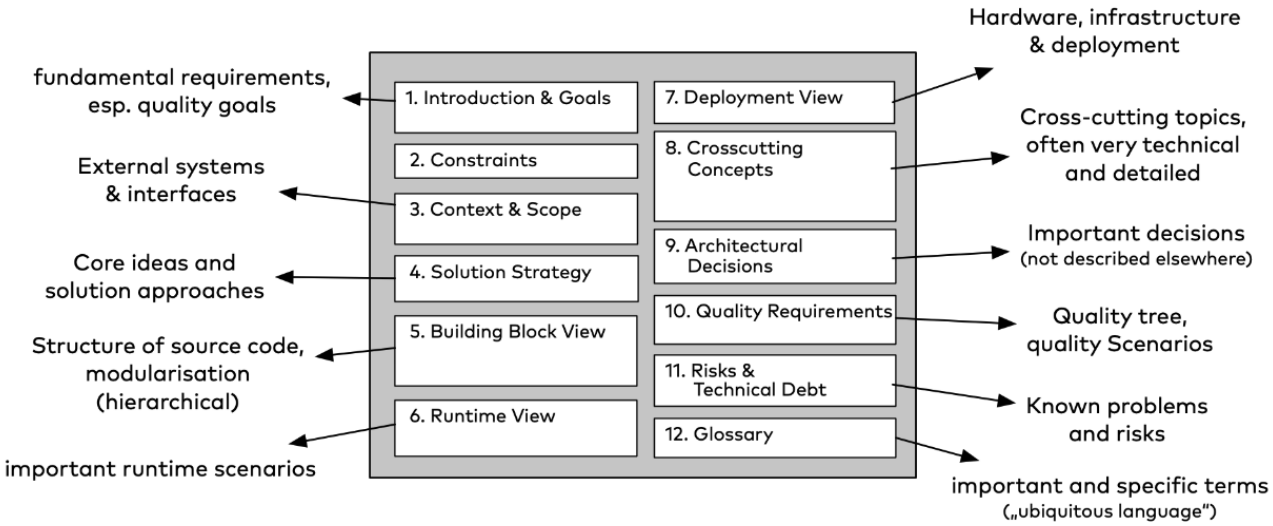


Figure 17: arc42 Overview

5 Spring Framework

The Spring Framework is an open source application framework and inversion of control container for the Java platform. Applications can choose which modules they need. At the heart are the modules of the core container, including a configuration model and a dependency injection mechanism. Beyond that, the Spring Framework provides foundational support for different application architectures, including messaging, transactional data and persistence, and web.

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the Figure 18.

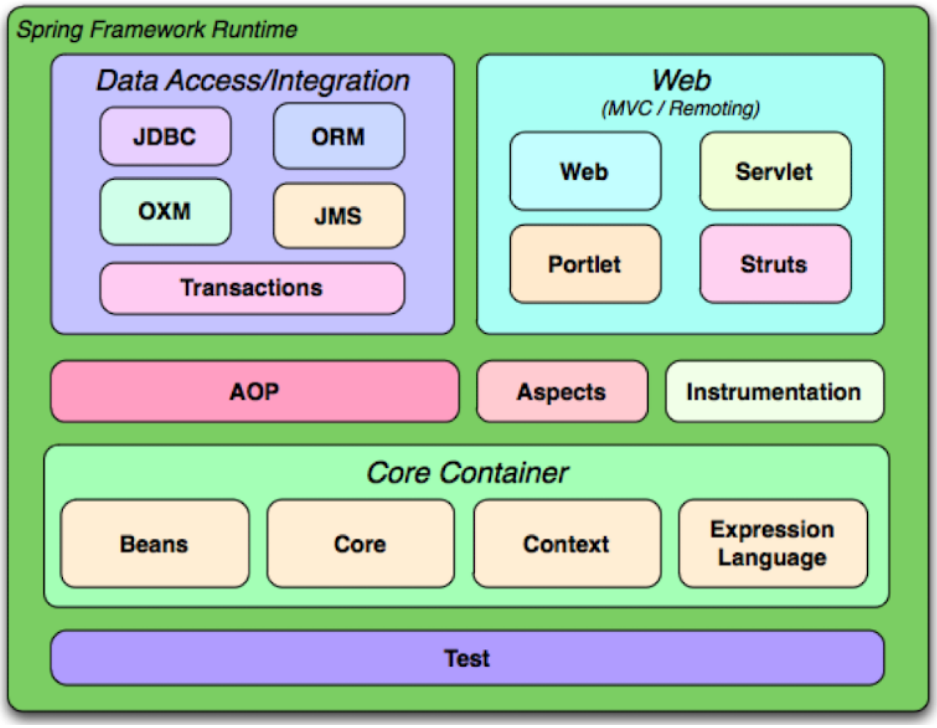


Figure 18: Spring Framework Overview

5.1 Justification

Why should we even use a Framework for the development of our software?

- To reduce tedious and error prone programming work (create better code)
- To raise level of abstraction and convenience by viewpoint/by layer (create less code)

Where can and should framework development be considered?

- Routine work that has to be done in many place (not domain-specific)
- Risky activities such as embedded system development (facing real-time requirements such as guaranteed response time)
- Labor-intensive system parts, e.g. user interfaces (server side, client side) and the Web, configuration management and (cloud) deployment

5.2 Spring Annotations

Annotation	Meaning
@SpringBootApplication	Convenience annotation, bundles @Configuration, @EnableAutoConfiguration and @ComponentScan
@Component	The generalized form considered as a candidate for auto-detection when using annotation-based configuration and classpath scanning. It extended to more specific forms such as @Controller, @Repository, and @Service.
@Controller	To indicate C in MVC pattern (presentation layer, dialog control)
@RequestMapping	URI pattern for of RESTful HTTP resources (JEE: JAX-RS)
@Service, @Repository	Specializations of @Component in Spring for DDD (lesson 6)
@Transactional	System transaction (ACID), container is transaction manager
@EnableCaching	Responsible for registering the necessary Spring components that power annotation-driven cache management
@Table, @Entity	Spring Data JPA, steer O/R mapping
@Query	Spring Data JPA, associates query with method
@Profile, @Primary, @Qualifier	Conditional bean registration (OS level, development stage, etc.)
@Autowired	Spring Framework; no need for DI getters/setters (JEE: @Inject)
@JMSListener	Defines the name of the Destination that this method should listen to. (JEE: @MessageDriven)
@JMSTemplate	JmsTemplate class handles the creation and releasing of resources when sending or synchronously receiving messages. (JEE: @Resource)

Table 2: Spring Annotations

5.3 Spring Concepts

Bean : A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Beans are created with the configuration metadata that you supply to the container. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

Bean Factory : The actual container and is responsible for instantiating, configuring, and assembling the beans. The BeanFactory provides an advanced configuration mechanism capable of managing any type of object.

Application Context : Is built on top of the BeanFactory and adds more enterprise-specific functionality. It includes all functionality of the BeanFactory and much more. This includes: *a* sophisticated event mechanism for use in a multicaster pattern, *a* resource loading abstraction (for example, to load files from the classpath or the filesystem), *a* hierarchical bean factory capability for managing namespaces, *a* message source for i18n (internationalization), and *a* application-layer specific context such as the WebApplicationContext for use in web applications.

Spring Boot : makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

Service Activator : A component that receives a message and then delegates to a service to process it.

Message Channel : A channel is a conduit that messages can be sent through. Channels are connected to each other, or to message endpoints. Channels can also be connected to message gateways.

Message Endpoint : A component that receives messages from a channel.

Message Gateway : A component that sends messages to a channel.

Message Router : A component that routes a message to one or more channels.

Data Enricher : A component that adds data to a message before it is sent to a channel.

5.4 Inversion of Control

Inversion of Control is a software design principle, one form of it is Dependency Injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

The Spring Framework uses the Dependency Injection (DI) pattern to manage the components that make up an application. The Spring Framework provides a number of ways to configure a Spring container, including XML, Java @Annotations(...).

5.5 Component Interaction Diagrams (CIDs)

Component Interaction diagrams describe the dynamic interactions between objects in the system, i.e. pattern of message-passing. This type of diagram can be used additionally in the C4 Model architecture, where it is called Dynamic Diagram, to give a better understanding.

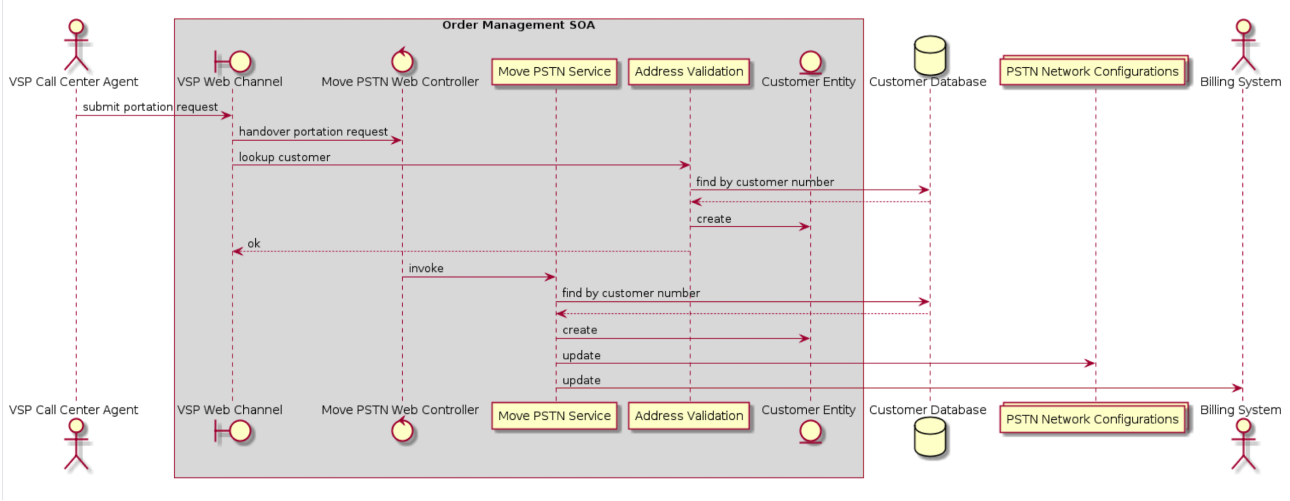


Figure 19: Component Interaction Diagram Example

6 Story Splitting

In the following section the concept of story splitting is introduced which can be used for component identification.

The acronym INVEST helps to remember a widely accepted set of criteria, or checklist, to assess the quality of a user story. If the story fails to meet one of these criteria, the team may want to reword it, or even consider a rewrite (which often translates into physically tearing up the old story card and writing a new one).

I ndependent (of all others)

N egotiable (not a specific contract for features)

V aluable (or vertical)

E stimable (to a good approximation)

S mall (so as to fit within an iteration)

T estable (in principle, even if there isn't a test for it yet)

The following Figure 20 displays nine splitting patterns for userstories.

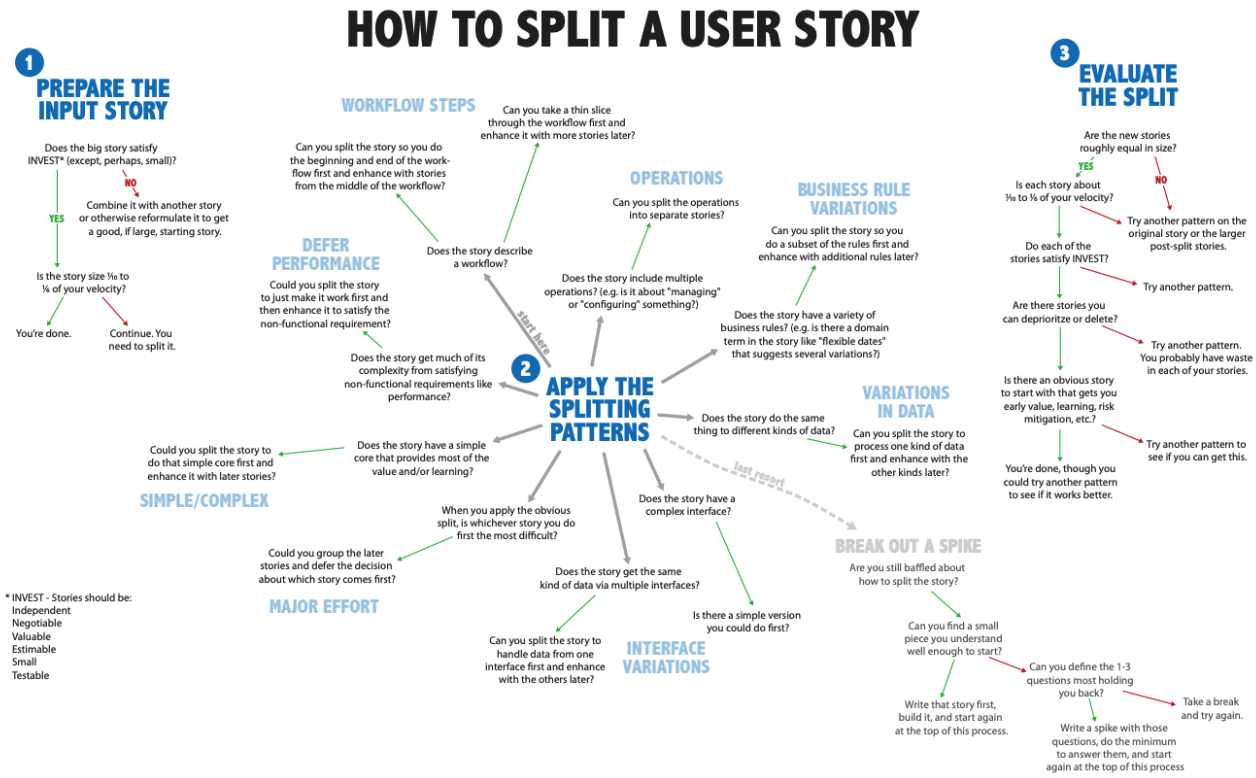


Figure 20: Split a User Story

Example Story Splitting
User Story: As a veterinarian working for the Pet Clinic, I want to schedule visits of pets and their owners so that I can heal as many animals as possible and send invoices to their owners.

Pattern	Story Element	Impact on Architecture
Pattern 1: Workflow Steps	A visit must be requested, scheduled, executed, documented, billed, invoices must be created, sent, monitored	Request visit etc.; at least one new input view per step/story; business logic to assign vets to slots and patents (owners with pets)

7 Component Modeling

A candidate component is an architectural element in the logical viewpoint grouping related responsibilities that jointly satisfy one or more (non-)functional requirements so that design and implementation work can be planned and component realization decisions can be made.

The C4 component diagram takes a logical view point. It is initialized during solution strategy, populated in project iterations and well suited to show layers and candidate components.

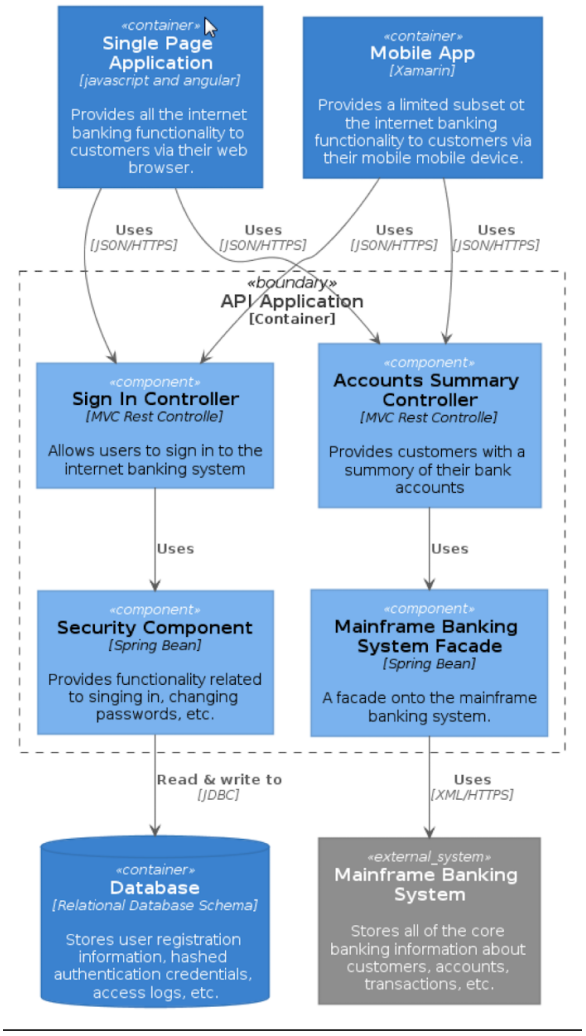


Figure 21: Sample Component Diagram

7.1 C4 Notation

Diagrams

- Every diagram should have a title describing the diagram type and scope (e.g. "System Context diagram for My Software System").
- Every diagram should have a key/legend explaining the notation being used (e.g. shapes, colours, border styles, line types, arrow heads, etc).
- Acronyms and abbreviations (business/domain or technology) should be understandable by all audiences, or explained in the diagram key/legend.

Elements

- The type of every element should be explicitly specified (e.g. Person, Software System, Container or Component).

- Every element should have a short description, to provide an "at a glance" view of key responsibilities.
- Every container and component should have a technology explicitly specified.

Relationship

- Every line should represent a unidirectional relationship.
- Every line should be labelled, the label being consistent with the direction and intent of the relationship (e.g. dependency or data flow). Try to be as specific as possible with the label, ideally avoiding single words like, "Uses".
- Relationships between containers (typically these represent inter-process communication) should have a technology/protocol explicitly labelled.

7.2 ZIOs Component Identification Algorithm

Output: Refined requirements and domain model, C4 diagrams, CRC cards

1. First iteration: Find Candidate Components (**CanCons**)
   (a) One channel component per primary actors (user/system) in context
   (b) One component per layer per feature (story) and domain model partition
   (c) 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
   - List potential realiziation technologies
   - Run a sanity/completeness check

8 Patterns of Enterprise Application Architecture (PoEAA)

Martin Fowler created a good overview of PoEAA. The book is from 2003, logical layers and most base patterns are still valid until today.

8.1 Presentation Layer Pattern

The template view pattern is used in distributed presentation and the Template View Pattern is one example of a Presentation layer Pattern. The known uses are in: JavaScript (Handlebars), ASP.NET, JSPs.

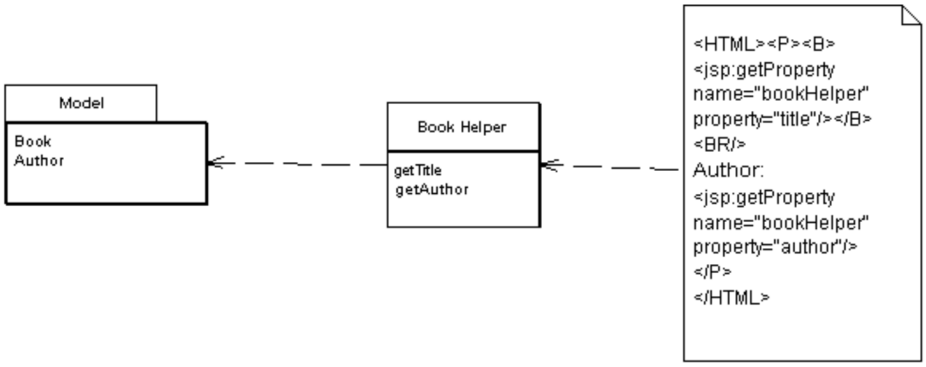


Figure 22: Template View Pattern

8.2 Domain Model Pattern

The domain model is an object of the domain that incorporates both behaviour and data. At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behaviour, and it's this complexity that objects were designed to work with. A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.

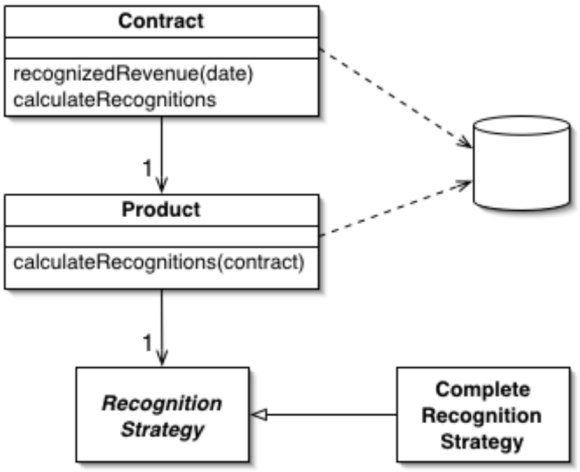


Figure 23: Domain Model Pattern

9 Domain Driven Design

DDD is a major software design concept introduced by the programmer Eric Evans in 2004, focusing on modelling software to match a domain according to input from that domain's expert. Properly applied it can lead to software abstractions called domain models. These models encapsulate complex business logic, closing the gap between business reality and code. It does this by analysing the problem first in a top-down approach.

The main advantages of using Domain-Driven Design is: it improves our craft, it provides flexibility, it prefers domains over interfaces, it reduces communication gaps between teams through ubiquitous language. On the other hand the disadvantages are: it requires a professional with strong domain expertise, it encourages the team to follow iterative practices.

Figure 24: Pattern Language Overview

Tactical and Strategic Design

Domain-driven design knows two types of design tools: Strategic design tools and Tactical design tools. The developers usually use tactical design tools and should have a good understanding of strategic design tools in order to architect good software.

9.1 Strategic Design

Strategic Design helps the developer to solve all problems regarding software modelling. It has similarities to Object-oriented design. In strategic design the architect is also forced to think in terms of a context.

Domain a sphere of knowledge or activity and refers to the business.

Model a system of abstractions representing selected aspects of a domain.

Ubiquitous Language Is a common language used by all team members to describe activities of the team around the domain model. It is used to to talk with domain experts and team members and describe classes, methods etc.

Bounded Context Refers to a boundary condition for a context. It descends a boundary and acts as a threshold within which, a particular domain model is defined and applicable. *Heuristic:* Implement one (micro-)service per Bounded Context

9.1.1 Bounded Context vs. Subdomain

Bounded Contexts implement parts of one or multiple subdomains.

Subdomain is in the problem space and the result of Object-oriented analysis (OOA)

Bounded Context is in the solution space and the result of Object-oriented design (OOD)

9.1.2 Bounded Context Map

The Context Map as in Figure 25 defines how Bounded Contexts do integrate. Additionally the information flow between the context is modelled. To create Such Context Maps the Context Mapper can be used.

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. This is a symmetric relationship.

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 may expose a Published Language.

Published Language (PL) The interacting bounded contexts agree on a common a language (for example a bunch of XML or JSON schemas over an enterprise service bus) by which they can interact with each other.

Conformist (CF) One BC uses the services of another but is not a stakeholder to that other BC. As such it uses "as-is" (conforms to) the protocols or APIs provided by that bounded context (which may be an OHS).

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 – an anti-corruption layer.

Customer/Supplier (CS) 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. The supplier may expose a PL.

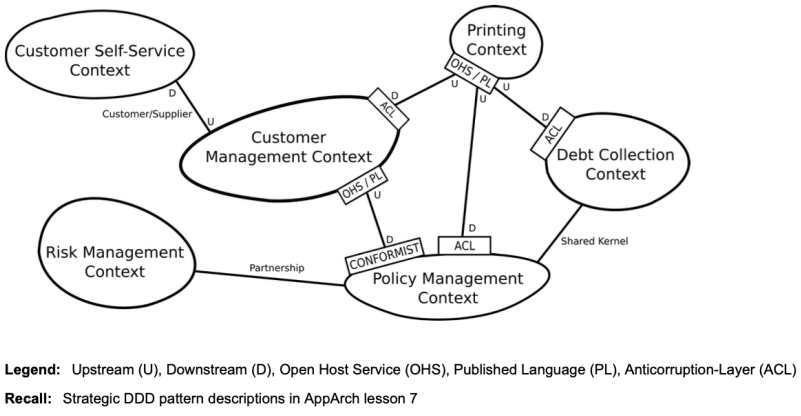


Figure 25: Bounded Context Example

9.2 Tactical Design

In contrast to the strategic design, tactical design talks about implementation details. It takes care of component inside a bounded context. Known artefact's such as services, entities, and factories are defined in this step. This process occurs during the product development phase.

Parts of the domain model created in the Tactical design phase:

Entity An Entity is a class which has some properties. Each instance of the class has a global identity and keeps the identity throughout the lifespan.

Value Object Are *immutable* and light-weight objects without an identity.

Service A service is a stateless class that fits somewhere else other than an entity or value object. A service is a functionality that exists somewhere between entities and values objects but it is neither related to an entity nor values objects.

Aggregate An Aggregate is a collection of entities and values which come under a single transaction boundary.

Repository A Repository helps in persisting aggregates.

Factory A Factory does create aggregates in order for them to be used in the application.

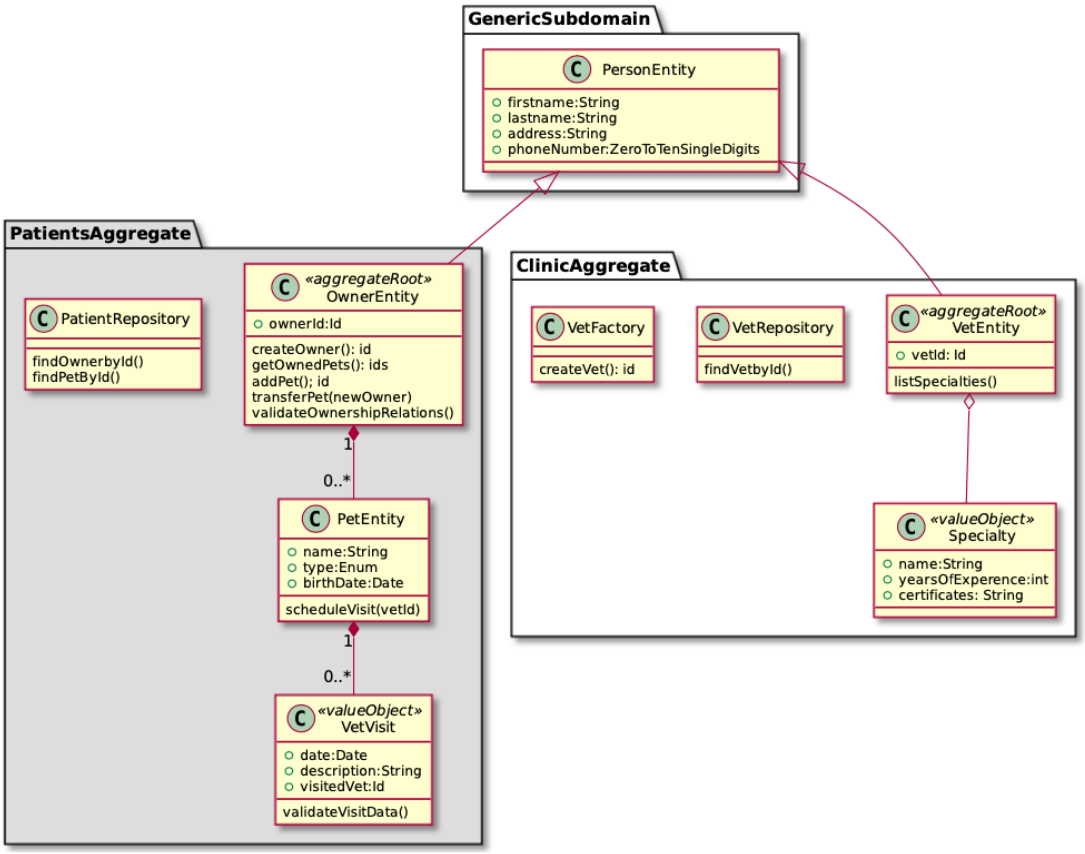


Figure 26: Tactic DDD Model

10 Component Responsibility Collaborators Cards (CRCs)

CRC cards are used to propose architectural elements, describe their responsibilities and show how they are combined together to from an architecture. This breaks down the functionality of an application to the basic entities (class), what the have to do (responsibly) and with which other entities hey communicate (collaborator).

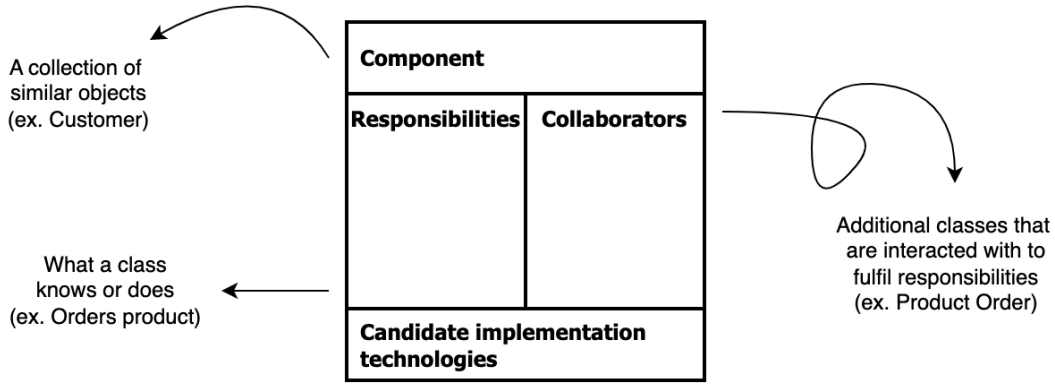


Figure 27: CRC Card Tempalte

The following CRC card in Figure 28 shows a filled out example from the Spring PetClinic.

***Component: :* Patients Aggregate (PA)**	
Responsibilities: • Groups pets and their owners into single transactional storage unit • Provides access to owner information (root entity) and to pet information (via pet owners) • Provides interface to update owner and pet information • Makes sure that each pet actually has an owner (in the system) • Controls access to sensitive information such as addresses, payment data, health records	*Collaborators (Interfaces to/from):* • (downstream) Patient Repository • (upstream) Web presentation layer (e.g., REST controllers in Spring)
Candidate implementations technologies (and known uses): • Do-it-yourself with Java, Spring Boot (@Component), DDD library from IFS • Sculptor (DSL for tactic DDD, briefly mentioned in lecture) • (to be evaluated) Commercially-Off-The-Shelf (COTS) Software package for patient/clinic management?	

Figure 28: CRC Example for PetClinic

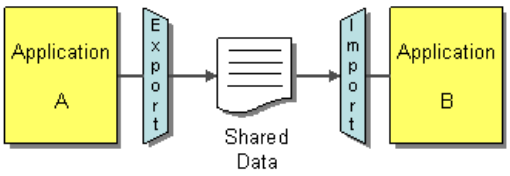
11 Enterprise Integration Patterns (EIP)

Integration patterns are a set of design patterns that can be used to solve common integration problems when connecting different systems and applications. These patterns provide a common language and best practices for designing and implementing integration solutions.

11.1 Integration Styles

11.1.1 File Transfer

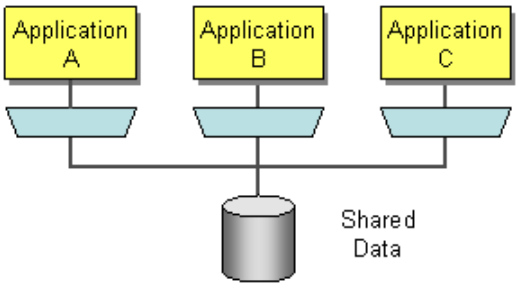
Figure 29: Shared Database



Have each application produce files containing information that other applications need to consume. Integrators take the responsibility of transforming files into different formats. Produce the files at regular intervals according to the nature of the business.

Figure 30: File Transfer

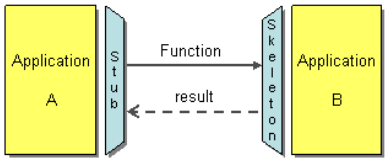
11.1.2 Shared Database



Integrate applications by having them store their data in a single Shared Database.

Figure 31: Shared Database

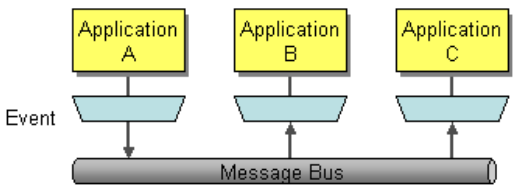
11.1.3 Remote Procedure Call



Develop each application as a large-scale object or component with encapsulated data. Provide an interface to allow other applications to interact with the running application.

Figure 32: Remote Procedure Call

11.1.4 Messaging



Use Messaging to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats.

Figure 33: Messaging

11.2 Asynchronous Messaging

Messaging capabilities are typically provided by a separate software system called a messaging system or message-oriented middleware (MOM), which is comparable to database systems that manage data persistence.

- Database administrator populates database with schema for application data; MOM administrator configures the messaging system with channels that define the paths of communication between the applications. Messaging system manages sending and receiving of messages.
- Database makes sure each data record is safely persisted, and likewise main task of messaging system is to move messages from the sender's computer to the receiver's computer in a reliable fashion.

The reason a messaging system is needed to move messages from one computer to another is that computers and the networks that connect them are inherently unreliable. Just because one application is ready to send a communication does not mean that the other application is ready to receive it. Even if both applications are ready, the network may not be working, or may fail to transmit the data properly. A messaging system overcomes these limitations by repeatedly trying to transmit the message until it succeeds. Under ideal circumstances, the message is transmitted successfully on the first try, but circumstances are often not ideal.

Loose coupling dimensions:

Reference Autonomy Producers and Consumers don't know each other

Platform Autonomy Producers and consumers may be located in different technical environment (e.g. containers), can be written in different languages.

Time Autonomy Producers and Consumers access channel at their own pace: Communication is asynchronous, Data exchanged is persistent

Format Autonomy Producers and consumers may use different formats of exchanged Data

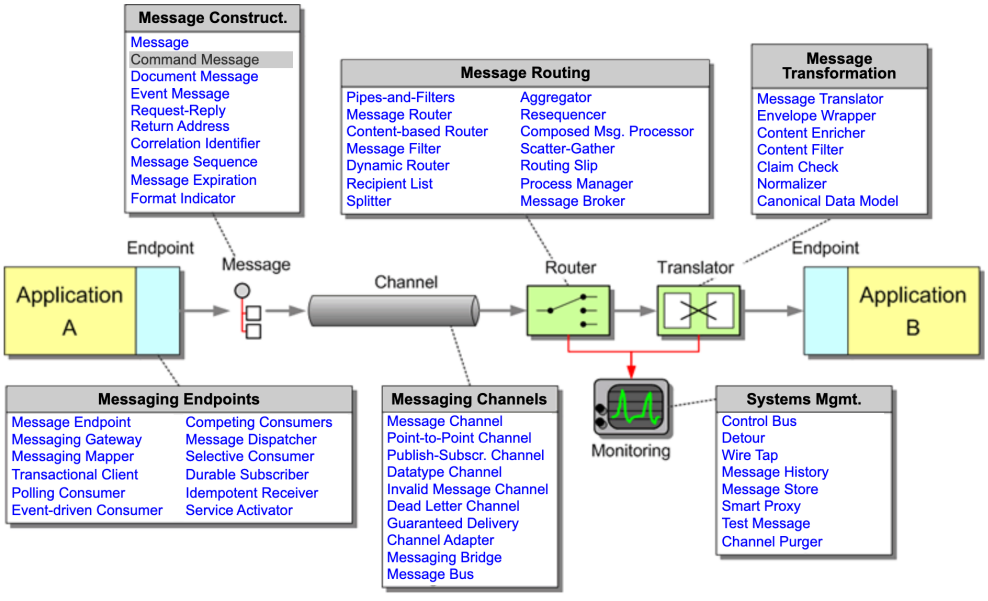


Figure 34: EIP Pattern Categories

11.3 Message Construction Patterns

Describes the intent, form and content of the messages that travel across the messaging system.

11.3.1 Document Message Pattern

Use a Document Message to reliably transfer data structure between applications. Whereas a Command Message tells the receiver to invoke certain behavior, a Document Message just passes data and lets the receiver decide what, if anything, to do with the data. The data is a single unit of data, a single object or data structure which may decompose into smaller units.

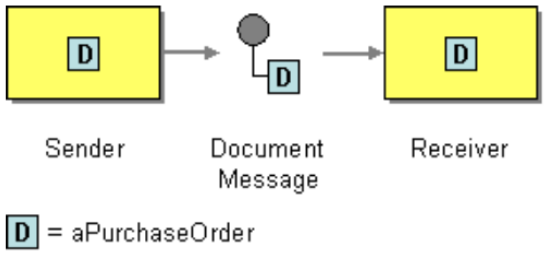


Figure 35: Document Message

11.3.2 Command Message Pattern

Use a Command Message to reliably invoke a procedure in another application. There is no specific message type for commands; a Command Message is simply a regular message that happens to contain a command. In JMS, the command message could be any type of message; examples include an ObjectMessage containing a Serializable command object, a TextMessage containing the command in XML form, etc.

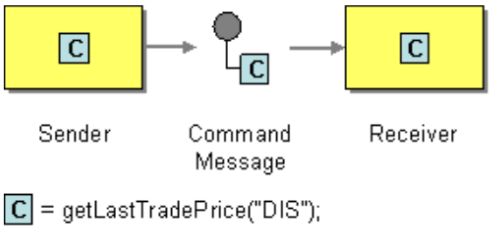


Figure 36: Command Message Pattern

11.3.3 Message Expiration

The Message Expiration can be used to specify a time limit, how long the message is viable. Once the time for which a message is viable passes, and the message still has not been consumed, then the message will expire. The messaging system's consumers will ignore an expired message; they treat the message as if it where never sent in the first place. Most messaging system implementations reroute expired messages to the Dead Letter Channel, while others simply discard expired messages; this may be configurable.

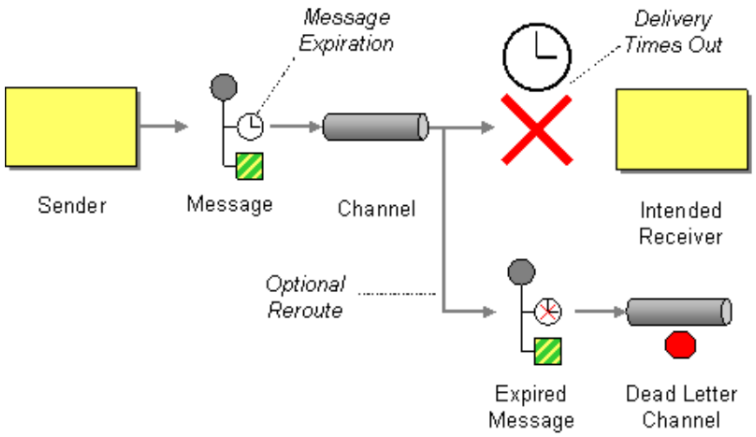


Figure 37: Message Expiration

11.3.4 Request-Reply Pattern

When an application sends a message, how can it get a response from the receiver? Send a pair of Request-Reply messages, each on its own channel. The request message should contain a Return Address that indicates where to send the reply message.

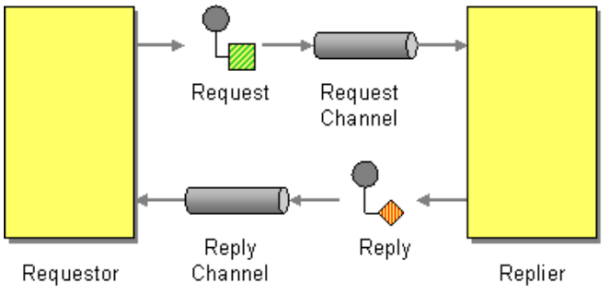


Figure 38: Request-Reply Pattern

11.4 Messaging Channels

11.4.1 Point-to-Point Channel

Send the message on a Point-to-Point Channel, which ensures that only one receiver will receive a particular message. Ensures that only one receiver consumes any given message. If the channel has multiple receivers, only one of them can successfully consume a particular message. If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other. The channel can still have multiple receivers to consume multiple messages concurrently, but only a single receiver consumes any one message.

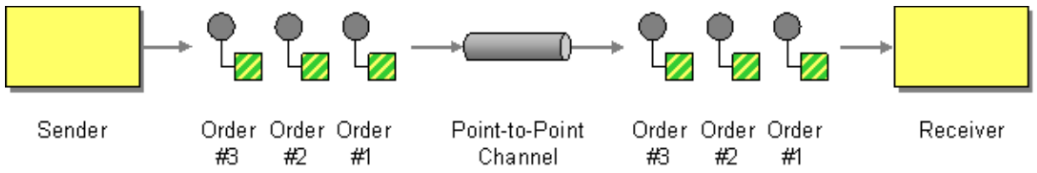


Figure 39: Point-to-Point Messaging

11.4.2 Publish-Subscribe Channel

Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each receiver. Has one input channel that splits into multiple output channels, one for each subscriber. When an event is published into the channel, the Publish-Subscribe Channel delivers a copy of the message to each of the output channels. Each output channel has only one subscriber, which is only allowed to consume a message once. Each subscriber only gets the message once and consumed copies disappear from their channels.

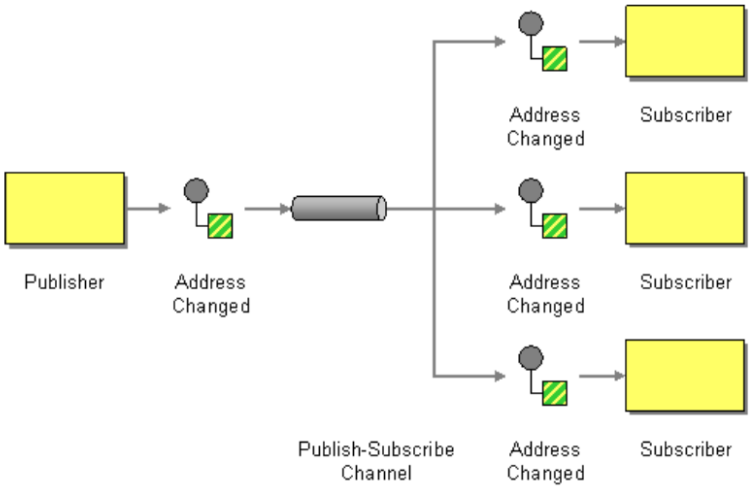


Figure 40: Publish-Subscribe Channel

11.4.3 Guaranteed Delivery Pattern

Use Guaranteed Delivery to make messages persistent so that they are not lost even if the messaging system crashes. With Guaranteed Delivery, the messaging system uses a built-in data store to persist messages. Each computer the messaging system is installed on has its own data store so that the messages can be stored locally. When the sender sends a message, the send operation does not complete successfully until the message is safely stored in the sender's data store. Subsequently, the message is not deleted from one data store until it is successfully forwarded to and stored in the next data store.

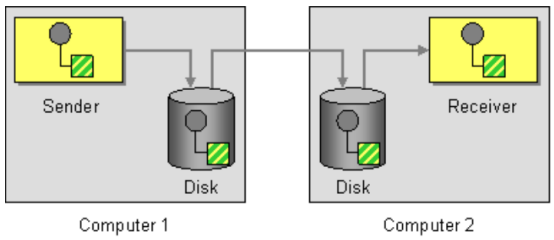


Figure 41: Guaranteed Delivery Pattern

11.4.4 Dead Letter Channel Pattern

When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a Dead Letter Channel. The specific way a Dead Letter Channel works depends on the specific messaging system's implementation, if it provides one at all. The channel may be called a "dead message queue" or "dead letter queue."

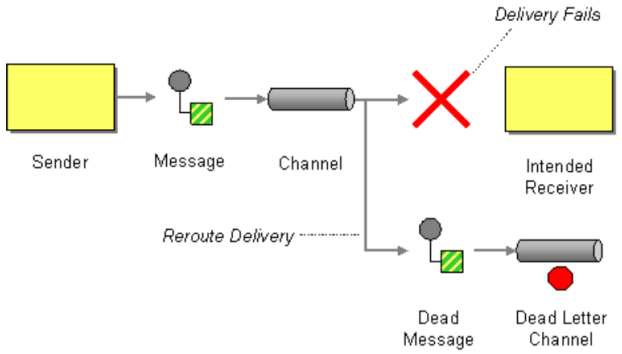


Figure 42: Dead Letter Channel Pattern

11.5 Message Endpoint Pattern

Connect an application to a messaging channel using a Message Endpoint, a client of the messaging system that the application can then use to send or receive messages.

Key considerations:

- Number of consumer endpoints? Push vs. pull (be event-driven)?
- Behavior of consumers: competing? browse message or remove it from queue?
- Idempotency (i.e., resending identical request messages does not change state of receiving application)?
- transactional semantics (ACID properties, local/distributed 2PC)?

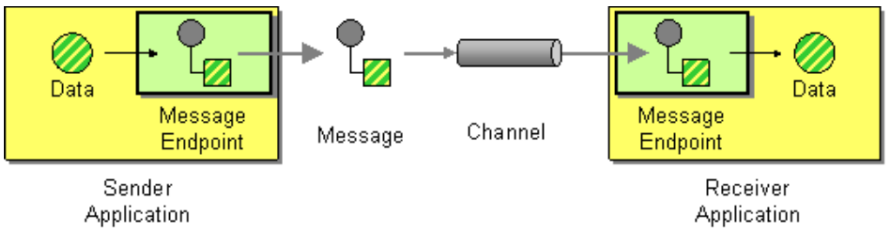


Figure 43: Message Endpoint Pattern

11.5.1 Message Consumption: Competing Consumers

Create multiple Competing Consumers on a single channel so that the consumers can process multiple messages concurrently. Competing Consumers are multiple consumers that are all created to receive messages from a single

Point-to-Point Channel. When the channel delivers a message, any of the consumers could potentially receive it. The messaging system's implementation determines which consumer actually receives the message, but in effect the consumers compete with each other to be the receiver. Once a consumer receives a message, it can delegate to the rest of its application.

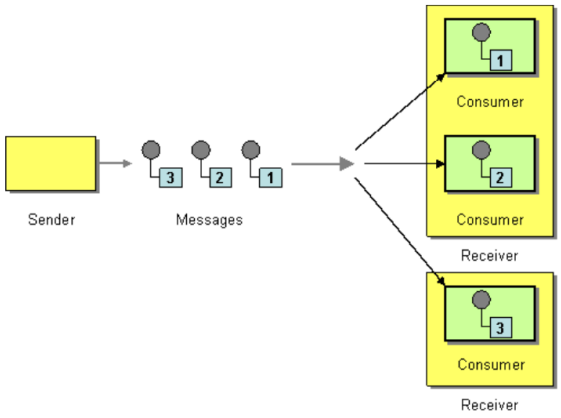


Figure 44: Competing Consumers

11.5.2 Message Consumption: Selective Consumers

Make the consumer a Selective Consumer, one that filters the messages delivered by its channel so that it only receives the ones that match its criteria.

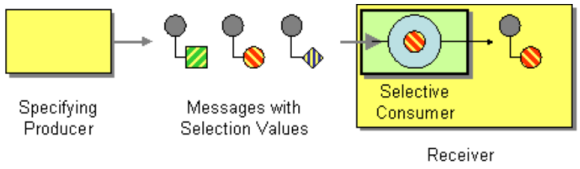


Figure 45: Selective Consumer

11.5.3 Message Consumption: Polling Consumer

The application can use a Polling Consumer, one that explicitly makes a call when it wants to receive a message. Also known as a synchronous receiver, because the receiver thread blocks until a message is received. We call it a Polling Consumer because the receiver polls for a message, processes it, then polls for another. As a convenience, messaging API's usually provide a receive method that blocks until a message is delivered, in addition to methods like receiveNoWait() and Receive(0) that return immediately if no message is available. This difference is only apparent when the receiver is polling faster than messages are arriving.

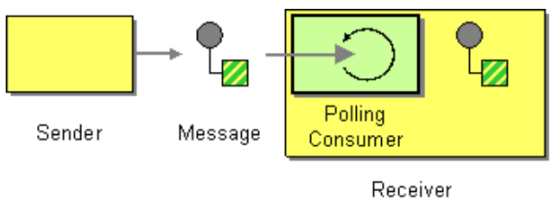


Figure 46: Polling Consumer

11.5.4 Event-Driven Consumer

he application should use an Event-Driven Consumer, one that is automatically handed messages as they're delivered on the channel. This is also known as an asynchronous receiver, because the receiver does not have a running thread

until a callback thread delivers a message. We call it an Event-Driven Consumer because the receiver acts like the message delivery is an event that triggers the receiver into action.

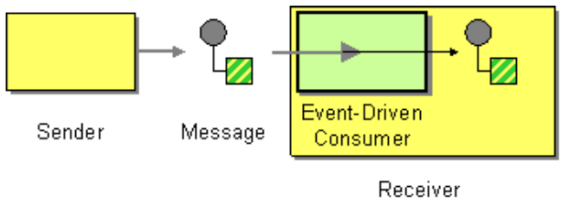


Figure 47: Event-Driven Consumer

11.5.5 Transactional Client Pattern

Use a Transactional Client – make the client's session with the messaging system transactional so that the client can specify transaction boundaries. Both a sender and a receiver can be transactional. With a sender, the message isn't "really" added to the channel until the sender commits the transaction. With a receiver, the message isn't "really" removed from the channel until the receiver commits the transaction.

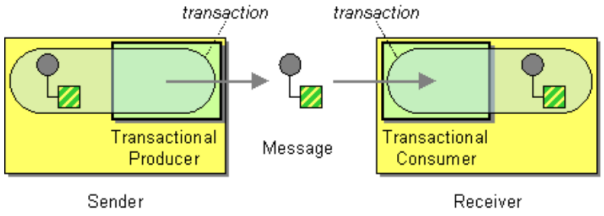


Figure 48: Transactional Client Pattern

11.6 Message Routing Patterns

11.6.1 Message Router Pattern

Insert a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel depending on a set of conditions. The Message Router differs from the most basic notion of Pipes and Filters in that it connects to multiple output channels. Thanks to the Pipes and Filters architecture the components surrounding the Message Router are completely unaware of the existence of a Message Router. A key property of the Message Router is that it does not modify the message contents. It only concerns itself with the destination of the message.

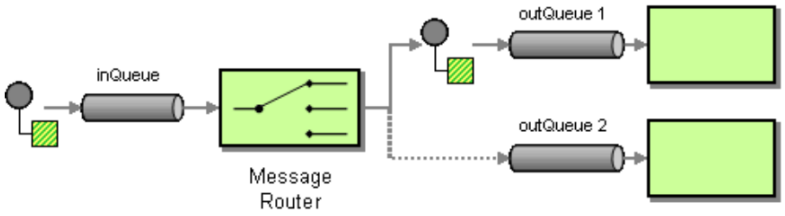


Figure 49: Message Router Pattern

11.6.2 Content-Based Router Pattern

Use a Content-Based Router to route each message to the correct recipient based on message content. The Content-Based Router examines the message content and routes the message onto a different channel based on data contained in the message. When implementing a Content-Based Router, special caution should be taken to make the routing function easy to maintain as the router can become a point of frequent maintenance.

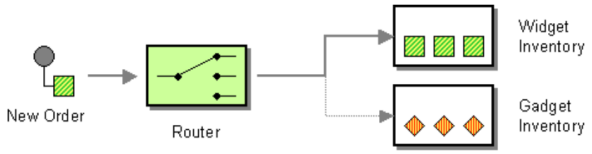


Figure 50: Content Based Router Pattern

11.6.3 Recipient List Pattern

Define a channel for each recipient. Then use a Recipient List to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list. The logic embedded in a Recipient List can be pictured as two separate parts even though the implementation is often coupled together. The first part computes a list of recipients. The second part simply traverses the list and sends a copy of the received message to each recipient.

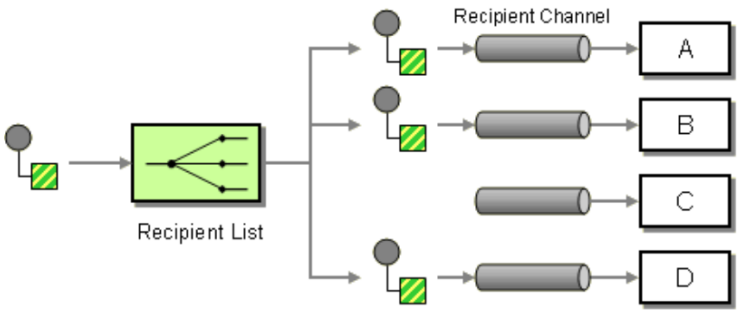


Figure 51: Recipient List Pattern

11.6.4 Splittern Pattern

Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item. Use a Splitter that consumes one message containing a list of repeating elements, each of which can be processed individually.

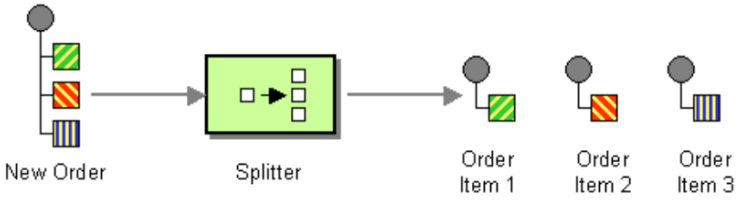


Figure 52: Splitter Pattern

11.6.5 Aggregator Pattern

Use a stateful filter, an Aggregator, to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages. The Aggregator is a special Filter that receives a stream of messages and identifies messages that are correlated. Once a complete set of messages has been received the Aggregator collects information from each correlated message and publishes a single, aggregated message to the output channel for further processing.

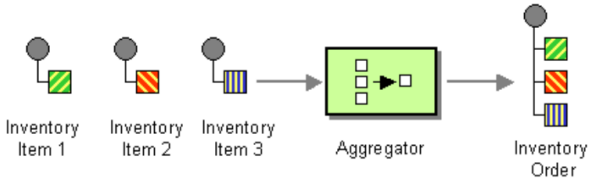


Figure 53: Aggregator Pattern

11.7 Message Transformation Patterns

The EIP book describes six universal message transformation patterns. These patterns are highly applicable to Enterprise Application Integration (EAI).

11.7.1 Message Translation Pattern

Use a special filter, a Message Translator, between other filters or applications to translate one data format into another. The Message Translator is the messaging equivalent of the general Adapter design pattern. An adapter converts the interface of a component into a another interface so it can be used in a different context.

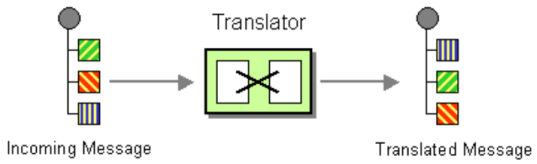


Figure 54: Message Translator Pattern

11.7.2 Content Filter Pattern

Use a Content Filter to remove unimportant data items from a message leaving only important items. The Content Filter does not necessarily just remove data elements. A Content Filter is also useful to simplify the structure of the message. Often times, messages are represented as tree structures. Many messages originating from external systems or packaged applications contain many levels of nested, repeating groups because they are modeled after generic, normalized database structures.

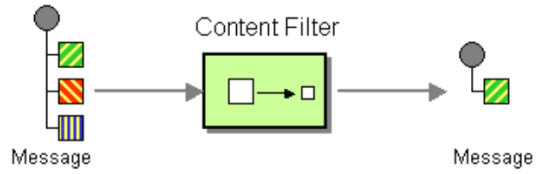


Figure 55: Content Filter

11.7.3 Content Enricher Pattern

Use a specialized transformer, a Content Enricher (a.k.a. Data Enricher), to access an external data source in order to augment a message with missing information. The Content Enricher uses information inside the incoming message (e.g. key fields) to retrieve data from an external source. After the Content Enricher has retrieved the required data from the resource, it appends the data to the message.

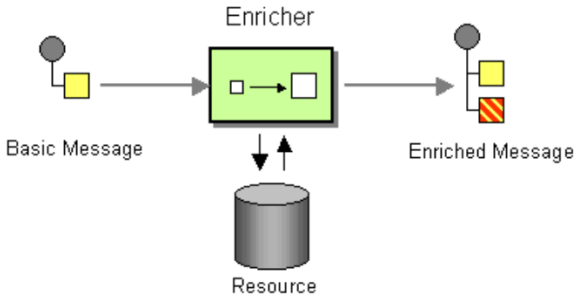


Figure 56: Content Enricher Pattern

12 Patterns of Enterprise Application Architecture for Integration (PEAA)

Patterns of Enterprise Application Architecture for Integration (PEAAI) are a set of design patterns for integrating enterprise applications, written by Martin Fowler. These patterns provide solutions to common integration problems faced in enterprise software development, such as connecting different systems, data migration, and communication between components.

12.1 Service Layer Pattern

Define an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

Other responsibilities of Service Layer:

- Role-Based Access Control (RBAC)
- Transaction Control, business-level undo (compensation)
- Activity logging (e.g. metering, monitoring)
- Exception handling

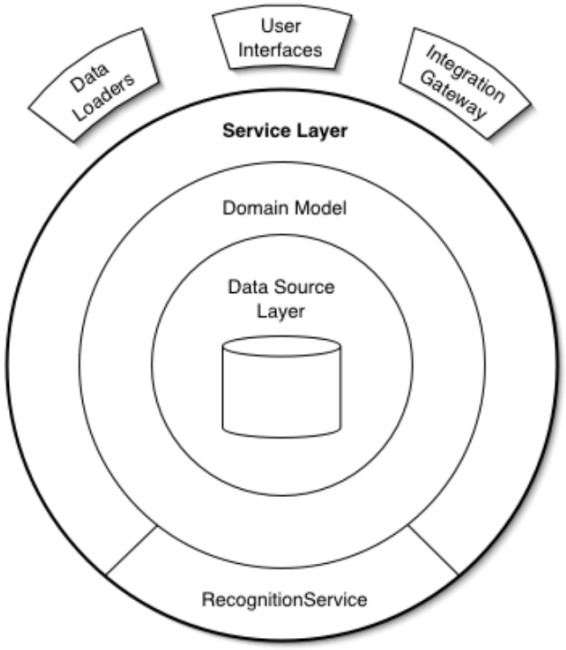


Figure 57: Service Layer Pattern

12.2 Remote Facade Pattern

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

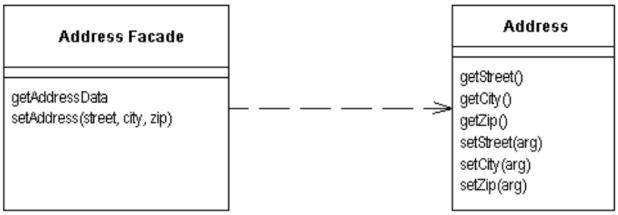


Figure 58: Remote Facade

12.3 Data Transfer Object (DTO)

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

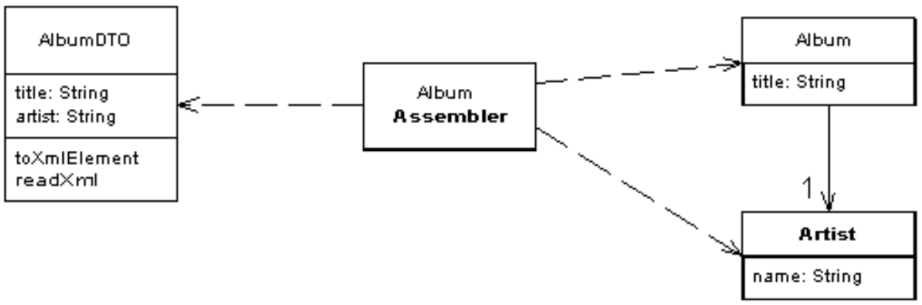


Figure 59: Data Transfer Object

13 Service-Oriented Architecture (SOA)

Mostly consists of separating your application into multiple services (most commonly HTTP services) which can be classified as different types like subsystem or tiers. There is not a single definition for SOA, since it means different things to people. A microservices architecture is a further evolution of SOA.

- A set of services and operations that a business wants to expose to their customers and partners, or other portions of the organization.
- An architectural style which requires a service provider, a service requestor (consumer) and a service contract (a.k.a. client/server).
- A set of architectural patterns such as service layer (with remote facades, data transfer objects), enterprise service bus, service composition (choreography/orchestration), and service registry, promoting principles such as modularity, layering, and loose coupling to achieve design goals such as reuse, and flexibility.
- A programming and deployment model realized by standards, tools and technologies such as Web services (WSDL/SOAP), RESTful HTTP, or asynchronous message queuing (AMQP etc.)

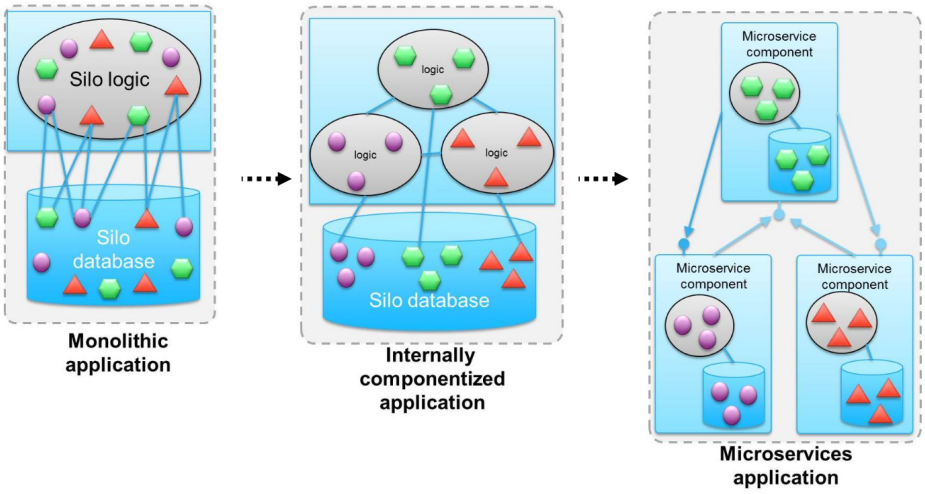


Figure 60: From Monolith and Components to SOA and MicroServices

13.1 Microservices

Microservices architectures evolved from previous incarnations of Service-Oriented Architectures (SOAs) to promote agility and elasticity. Microservices are independently deployable, scalable and changeable services each having a single responsibility. These are often deployed in lightweight containers, encapsulate their own state, and communicate via message-based remote APIs (HTTP, queueing), ideally in a loosely coupled fashion.

13.1.1 Benefits of Microservices

- They support agile development practices with continuous delivery and deployment. Ex. each microservice is owned by a single team, which allows each team to independently develop, deploy and operate their respective service.
- They are well suited for implementing 'IDEAL' cloud-native applications (i.e., **isolated** state, **distributed**, **elasticity**, **automated** management, **loose** coupling). When deployed independently, horizontal on-demand scalability can be achieved through container virtualization and elastic load balancing.
- They allow an incremental migration of monolithic applications, which reduces the risk of failure of software modernization efforts.

13.1.2 Challenges of Microservices

- The communication overhead within a distributed architecture combined with poor API design choices can impact the performance of microservice architectures.

- Scaling the architecture to include a large number of microservices requires a disciplined approach to their life cycle management, monitoring, and debugging.
- Single points of failure and cascading failure proliferation effects need to be avoided, for example through redundancy or circuit breakers. This reduces the risk that failing downstream microservice instances bring down upstream ones (and, eventually, the entire system).
- Data consistency and state management challenges are introduced, for example when decoupling monolithic, stateful applications into independent, autonomous microservices (Furda et al. (2018)). Autonomy and consistency for the whole microservice architecture cannot be both guaranteed when employing a traditional backup and disaster recovery strategy

14 Representational State Transfer (REST)ful HTTP

REST is an architectural style (for integration), defined via constraints. There it is not an API technology or protocol. RESTful HTTP is one prominent incarnation of this style, when it is done right.

- Client-Server
- Stateless
- Cacheable
- Uniform Interface (URI, HTTP)
- Layered System
- Code on Demand (Optional)

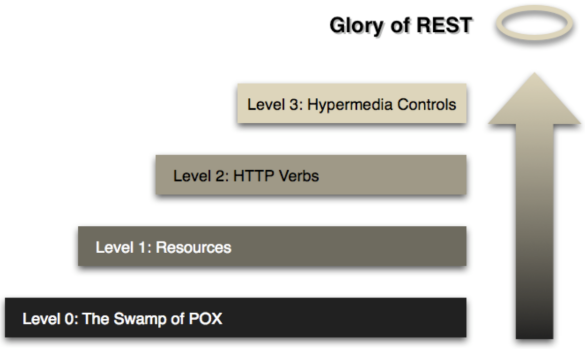


Figure 61: The Glory of REST

14.1 Hypermedia as the Engine of Application State (HATEOAS)

Before we can grasp the definition of HATEOAS we need to have some parts defined. HATEOAS is a part of RESTful API architecture and design. HATEOAS make the APIs self-descriptive. Therefore the consumer does not need to know everything about the API.

Hypermedia are links to different parts of the API. These links are documentation and teaches the developer on how to use the API.

The following listing shows a JSON http answer with the hypermedia links which provide further actions.

```
HTTP/1.1 200 OK
Content-Type: application/+json
Content-Length: ...
{
   "payroll": {
       "employee_number": "employee_123",
       "salary" : 1000,
       "links": {
           "increment": "/payroll/employee_123/increment",
           "decrement": "/payroll/employee_123/decrement",
           "close": "/payroll/employee_123/close"
       }
   }
}
```

14.2 HTTP and REST

HTTP protocol primitives: sent by client, understood by server, client e.g. web browser.

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

Figure 62: REST Verbs

15 Service Contract Notations

A Service contract refers to the definition of a service provided by an application component, detailing its inputs, outputs, and behavior. A service contract is a fundamental building block of service-oriented architecture (SOA), and serves as a shared understanding between the provider and consumer of the service. Examples of Service Contract notations are: OpenAPI, gRPC, Web Service Description Framework, Representational State Transfer.

15.1 OpenAPI

The OpenAPI specification (OAS), formerly known as the Swagger specification, is a format for describing RESTful APIs. It uses a simple YAML or JSON file to describe the structure of an API, including its endpoints, input and output parameters, authentication methods, and other information. The specification can be used to generate documentation, client libraries, and other tools that make it easier to consume and work with an API. It is widely supported and has a large ecosystem of tools and libraries built around it.

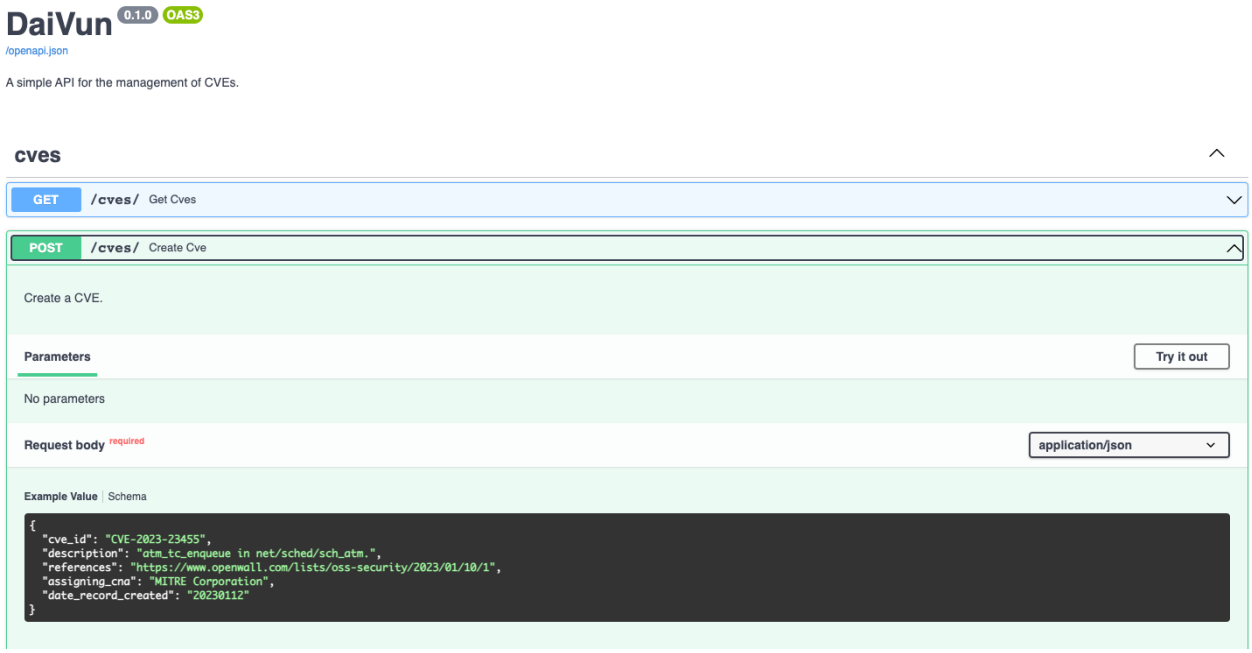


Figure 63: OpenAPI Example Documentation

15.2 Web Service Description Language (WSDL)

The Web Service Description Language (WSDL) is an XML-based language used to describe the functionality offered by a web service. It defines the operations (methods) that can be called on the service, the inputs and outputs for those operations, and the location of the service (the endpoint).

15.3 Microservice Domain-Sepcific Language (MSDL)

Microservice Domain-Specific Language (MDSL) is a language specifically designed to describe the structure and behavior of microservices. It is used to define the interface, dependencies, and other aspects of a microservice, and can be used to generate code, documentation, and other artifacts.

16 Microservice API Patterns (MAPI)

Microservice API Patterns (MAPI) refer to a set of best practices and design patterns for building and exposing APIs in a microservice architecture. MAPI helps ensure that APIs are consistent, scalable, secure, and easily consumable by clients.

Figure 64: Mircoservice API Patterns

17 The Hard Parts of Software Architecture

The following section contains the content of a guest lecture about the book: Software Architecture: The Hard Parts.

17.1 Service Granularity

Granularity Disintegrators Provide guidance and justification for when to break a service into smaller pieces.

- Service scope and function: Is the service doing too many unrelated things?
- Code volatility: Are changes isolated to only one part of the service?
- Scalability and throughput: Do parts of the service need to scale differently?
- Fault tolerance: Are there errors that cause critical functions to fail within the service?
- Security: Do some parts of the service need higher security levels than others?
- Extensibility: Is the service always expanding to add new contexts?

Granularity Integrators Provide guidance and justification for putting services back together (or not breaking apart a service in the first place).

- Database transactions: Is an ACID transaction required between separate services?
- Workflow and choreography: Do services need to talk to one another?
- Shared code: Do services need to share code among one another?
- Database relationships: Although a service can be broken apart, can the data it uses be broken apart as well?

Tradeoff Analysis

1. Find what parts are coupled together
2. Analyze how they are coupled to one another
3. Assess the tradeoffs by determine the impact of change to interdependent system

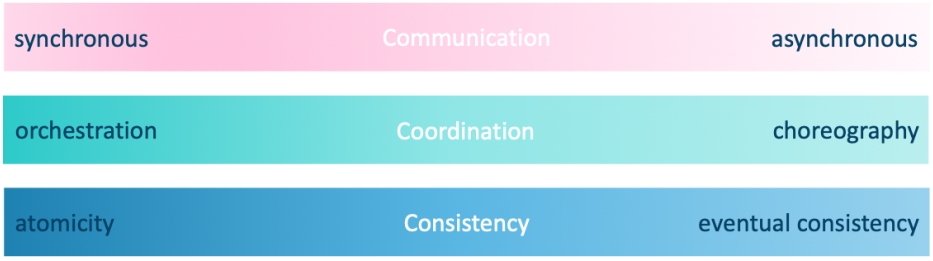


Figure 65: Tradeoff Analysis

17.1.1 Communication

Synchronous Tradeoffs

- - Perfomance impact on highly interactive systems
- - Create dynamic entanglements
- - Creates limitations in distributed architectures
- + Easy to model transactional behaviour
- + Mimics non-distributed method calls
- + Easier to implement

Asynchronous Tradeoffs

- - Complex to build and debug
- - Presents difficulties for transactional behaviours
- - Error handling
- + Allows highly decoupled systems
- + Common performance tuning technique
- + High performance and scale

17.1.2 Orchestration

Orchestration Tradeoffs

- - Responsiveness
- - Fault tolerance
- - Scalability
- - Service coupling
- + Centralised workflow
- + Error handling
- + Recoverability
- + State management

Choreography Tradeoffs

- - Distributed workflow
- - State management
- - Error handling
- - Recoverability
- + Responsiveness
- + Fault Tolerance
- + Scalability
- + Service decoupling

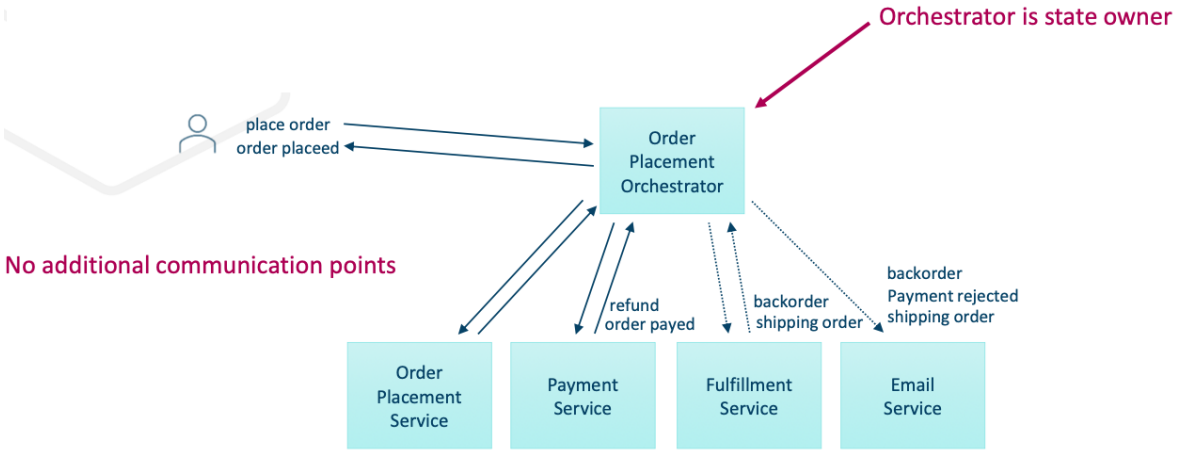


Figure 66: Ochestration Workflow

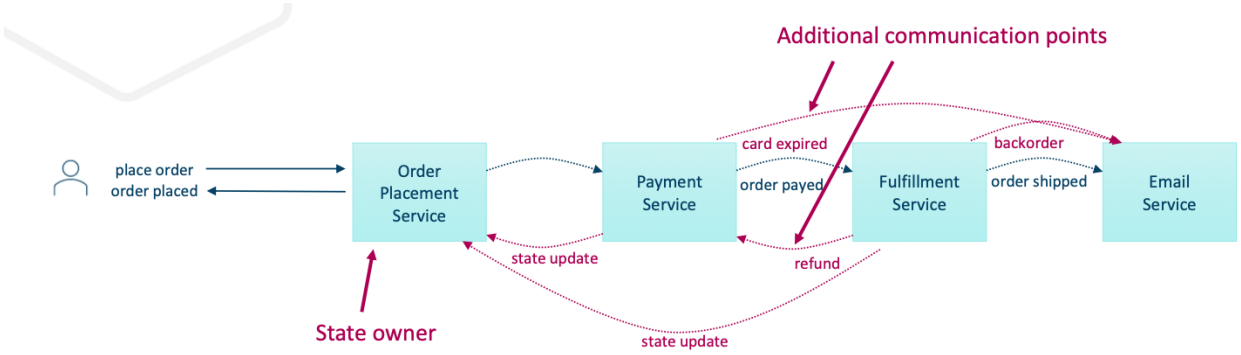


Figure 67: Choreography Workflow