Curriculum for

CPSA Certified Professional for Software Architecture®

– Advanced Level –

Module: FLEX

Flexible Architecture Models - Microservices and Self-Contained Systems

Version 1.1 (November 2015)

Table of contents

0 Introduction: General information about the iSAQB Advanced Level

0.1 Which information imparts the advanced level module?

- The iSAQB Advanced Level provides a modular training in three different domains with flexible, customisable training routes. It takes individual tendencies and priorities into account.
- The certification is done in homework. The evaluation and the oral exam is done by experts named by the iSAQB. Details can be found in the web under http://isaqb.org.

0.2 Which skills do graduates of the Advanced Level (CPSA-A) have?

CPSA-A graduates are able to:

- Autonomously and methodically design medium to large IT systems in a methodically well-founded manner.
- Assume technical and contentual responsibility for IT systems with medium to high criticality.
- Identify, design and document actions to reach non-functional requirements and assist development teams with their implementation.
- Manage and perform the communication of system's architecture in medium to large development teams.

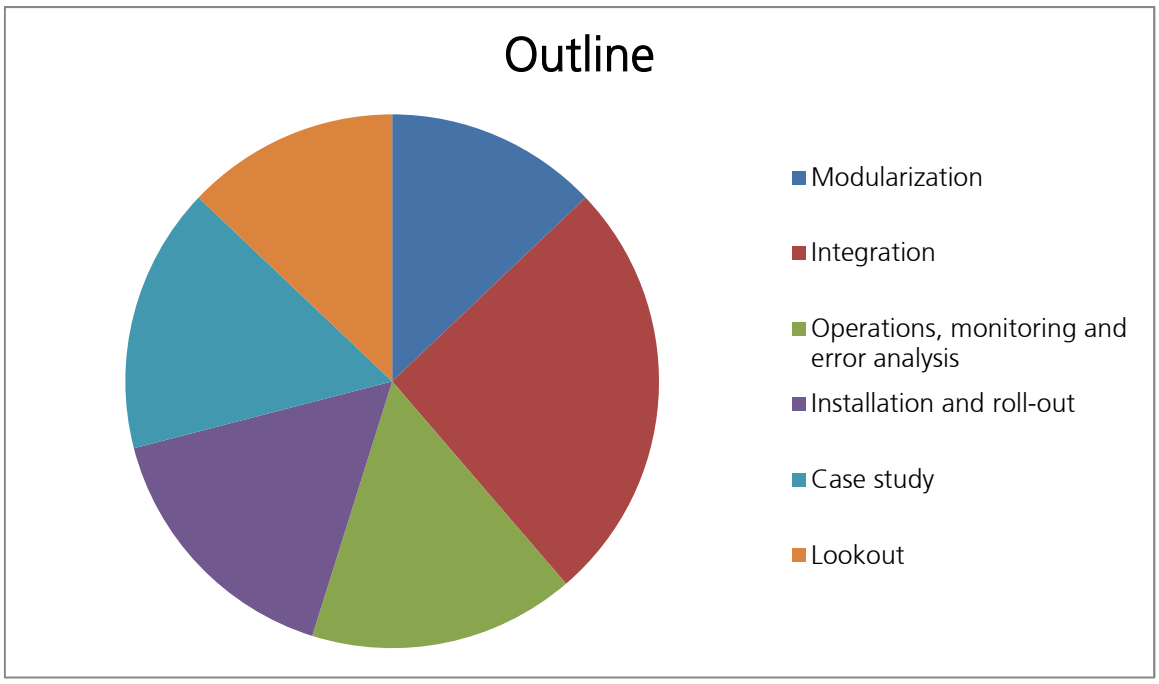
0.3 Prerequisites for the CPSA-A certification

- A successful training and certification as CPSA-F Certified Professional for Software Architecture Foundation Level®
- A least three years of full-time professional experience in the IT industry working on the design and development of at least two different IT systems
    - Exceptions allowed on request (for example: participation in OpenSource projects)
- Training and further education within the iSAQB Advanced Level training courses with a minimum of 70 credit points from all three different areas of competence (detailed in section 1.2)
    - Existing certifications may be charged to these credit points. The list of current certificates for which credit points are charged can be found on the iSAQB homepage.
- Successful completion of the CPSA-A certification test.

CPSA-A®
Certified Professional for Software Architecture
Advanced Level

1 Basic information about the FLEX Module

1.1 Outline of the curriculum for FLEX and recommended temporal breakdown



- Motivation 02:00
- Modularisation 04:00
- Integration 02:30
- Operations, monitoring and error analysis 02:30
- Installation and roll-out 02:30
- Case study 02:30
- Perspective 02:00

1.2 Duration, didactic and additional details

The times below are recommendations. The duration of the FLEX training should be at least 3 days, but may be longer. Vendors can distinguish themselves by duration, didactics, type and structure of the exercises as well as the detailed course organisation. In particular, the type of examples and exercises leaves the curriculum completely open.

Licensed trainings for FLEX contribute the following points (credit points) to the permission for the final Advanced-Level certification exam:
Methodical competence: 10 points
Technical competence: 20 points
Communicative competence: 00 points

1.3 Prerequisites for the FLEX module

Participants should have the following knowledge and / or experience:

- Fundamentals of the description of architectures using various views, comprehensive concepts, design decisions, boundary conditions etc., as taught in the CPSA-F (Foundation Level).
- Experience with implementation and architecture in agile projects.
- Experiences from the development and architecture of classical systems with the typical challenges.

Useful for understanding some concepts are also:

- Distributed systems
  - Problems and challenges in the implementation of distributed systems
  - Typical distributed algorithms
  - Internet protocols
- Knowledge about modularisations
  - Functional modularisation
  - Technical implementations like packages or libraries
- Classical operation and deployment processes

1.4 Outline of the curriculum for FLEX

The individual sections of the curriculum are described as follows:

- Terms/Concepts: Main core concepts of this topic.
- Teaching/Practice Time: Specifies the teaching and practice time that at least has to be spent on this topic or its exercise in the accredited training.
- Learning goals: Describes the content to be communicated, including its core terms and concepts.

This section also outlines the knowledge to be acquired in appropriate training courses. The learning goals are differentiated into the following categories or sub-sections:

- What should the participants be capable of? The participants should be able to apply these contents independently after the training. Within the course, these contents are covered by exercises and are part of the FLEX module examination and / or the final examination of the iSAQB Advanced Level.
- What should the participants understand? These contents can be verified in the FLEX module examination.
- What should the participants know? These contents (terms, concepts, methods, practices, etc.) can help to understand or motivate the topic. These contents are not part of the examinations. They are discussed in trainings, but not necessarily very detailed.

1.5 Additional information, terms and translations

As far as required for the understanding of the curriculum, we have included and defined functional terms in the iSAQB glossary, and also added translations of the original literature if necessary.

2 Motivation

Duration: 120 min	Exercise time: none

2.1 Terms and concepts

Availability, reliability, time-to-market, flexibility, predictability, reproducibility, homogenisation of the stages, internet / web-scale, distributed systems, parallelism of feature development, evolution of the architecture (build for replacement), heterogeneity, automation.

2.2 Learning goals

2.2.1 What shall participants be capable of?

- Architectures can be optimised for different quality goals. In this module, participants learn how to create flexible architectures that allow rapid deployment and thus rapid feedback from the application of the system.
- They have understood the drivers for the architecture types taught in this curriculum module, the implications of these drivers for the architectures, and the interaction of the architectures with the organisation, processes, and technologies.
- They have understood the trade-offs of the architecture types presented (at least Microservices, Self-Contained Systems and Deployment Monoliths) and are able to communicate them as well as apply them in the context of concrete projects / system developments in order to make appropriate architectural decisions.

2.2.2 What should participants understand?

- The architecture has crucial influence on the ability to quickly bring new features into production.
- Dependencies between components of different development teams influence the duration until software can be put into production because they increase the communication costs and delays are propagated.
- Automation of activities (such as test and deployment processes) increases the reproducibility, predictability, and quality of results of these processes. This can lead to an improvement of the overall development process.
- Unification of the different environments (e. g., development, test, QA, production) reduces the risk of lately detected and (in other environments) non-reproducible errors due to different configurations.
- Unification and automation are essential aspects of continuous delivery.
- Continuous integration is a prerequisite for continuous delivery.
- A suitable architecture is the prerequisite for the parallelisation of the development as well as the independent commissioning of independent modules. This can, but do not have to be "services".
- Some change scenarios are easier to implement in monolithic architectures. Other change scenarios are easier to implement in distributed service architectures. Both approaches can be combined.

- There are different types of isolation with different advantages. A failure, for example, can be limited to a single component or changes can be limited to a single component.
- Certain types of isolation are much easier to implement between processes with remote communication.
- Remote communication, however, has disadvantages - for example many new sources of errors.

2.2.3 What should participants know?

- Conway's law
- Partitionability as a quality feature
- Round trip times with the IT supply chain as a competitive factor
- Building a continuous delivery pipeline
- The different test phases of a continuous delivery pipeline

2.3 References

- Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
- Eberhard Wolff: A Practical Guide to Continuous Delivery, Addison-Wesley, 2017, ISBN 978-0-13-469147-3
- Jez Humble, Barry O'Reilly, Joanne Molesky: Lean Enterprise: Adopting Continuous Delivery, DevOps, and Lean Startup at Scale, O'Reilly 2014, ISBN 978-1-44936-842-5

3 Modularisation

Duration: 120 min	Exercise time: 30 min

3.1 Terms and concepts

- Motivation for decompositioning into smaller systems
- Different kinds of modularisation, coupling
- System limits as a way of isolation
- Hierarchical structure
- Application, Self-Contained System, Microservice
- Domain-Driven Design Concepts and "Strategic Design", Bounded Contexts

3.2 Learning goals

3.2.1 What shall participants be capable of?

- Participants can design a decomposition into individual blocks for a given problem.
- The participants should consider the organisation's communication structure when setting the module boundaries (Conway's law).
- The participants should be able to evaluate and select technical modularisation concepts in a project-specific manner.
- The participants should be able to illustrate and analyse the relationships between modules as well as between modules and subdomains (context mapping).
- Participants can evaluate the consequences of different modularisation strategies and compare the efforts of the modularisation with the expected benefits.
- Participants can assess the impact of the modularisation strategy on the autonomy of building blocks at development time and at run time.
- The participants can draw up a plan to divide a deployment monolith into small services.
- Participants can develop a concept to build a system of services.
- The participants can choose a suitable modularisation as well as a suitable granularity of the modularisation - depending on the organisation and the quality goals.

3.2.2 What should participants understand?

- Participants understand that each type of building blocks requires a handy label, as well as a description,
    - what makes up this kind of building block
    - how such a building block is integrated at runtime
    - how such a building block is built (in the sense of the build system)
    - how such a building block is deployed
    - how such a building block is tested
    - how such a building block is scaled
- Participants understand that an integration strategy decides whether a dependency

    - emerges at runtime
    - emerges during development time, or
    - emerges at the deployment.
- Modularisation helps to achieve goals such as parallelisation of development, independent deployment / interchangeability at runtime, rebuild / reuse of modules and easier understanding of the overall system.
- Therefore, techniques like continuous delivery and the automation of test and deployment are important influences on the modularisation.
- Modularisation means the decomposition of a system into smaller parts. Re-integrating these parts after the decomposition causes organisational and technical efforts. These efforts have to be exceeded by the advantages achieved by the modularisation.
- Participants understand that in order to achieve higher autonomy of the development teams, it is better to divide a component along functional boundaries rather than along technical boundaries.
- Depending on the chosen modularisation technology, there is coupling on different levels:
    - Sourcecode (modularisation with files, classes, packages, namespaces etc.)
    - Built target (modularisation with JARs, libraries, DLLs, etc.)
    - Runtime environment (operating system, virtual machine or container)
    - Network protocol (distribution to different processes)
- A coupling at the source code level requires very close cooperation as well as common SCM. A coupling at the level of the built target means that the building blocks must usually be deployed together. Only a distribution to different applications / processes is feasible with regard to independent deployment.
- Participants understand that a complete isolation between building blocks can only be ensured by a separation in all phases (development, deployment and runtime). If this is not the case, undesirable dependencies cannot be excluded. At the same time, the participants also understand that it can be useful to forego complete isolation for reasons such as efficient resource usage or complexity reduction.
- Participants understand that, when distributing to different processes, some dependencies no longer exist in the implementation, but rather arise at runtime. This increases the requirements for monitoring these interfaces.
- Microservices are independent deployment units and therefore independent processes that expose their functions through lightweight protocols, but may also have a UI. Different technology decisions can be made for the implementation of each individual microservice.
- A Self-Contained System (SCS) is a functionally independent system. It usually includes UI and persistence. It may consist of several Microservices. Usually a SCS covers a functional bounded context.
- The module division can be done along functional or technical boundaries. In most cases, a functional division is recommended, because in this case functional requirements can be assigned more clearly to a concrete module and therefore it is not necessary to adapt several modules for the implementation of a single functional

requirement. Thereby, each module can have its own domain model in the sense of a bounded context and thus different views on a business object with its own data.
- Transactional consistency through process boundaries can only be achieved via additional mechanisms. So, if a system is divided into several processes, the module boundary often also represents the limit for transactional consistency. Therefore, a DDD aggregate must be managed in one module.
- Participants understand which modularisation concepts can be used not only for transactional but also for batch- and data-flow-oriented systems.
- Different levels of specifications can be useful for the development of a module. Some specifications should better be superior valid for the integration with other building blocks of this type, in general. The overarching decisions that affect all systems can form a macro architecture, including, for example, communications protocols or operating standards. Micro architecture can be the architecture of a single system. It is largely independent of other systems. Excessive limitations at the macro architecture level will lead to an overall architecture that can be applied to fewer problems

3.2.3 What should participants know?

- The participants should know various technical modularisation options: e. g., source code files, libraries, frameworks, plugins, applications, processes, Microservices, SCS.
- The participants should know the following terms from the domain-driven design: Aggregate Root, Context Mapping, Bounded Contexts and relationships between them (e. g., Anti-Corruption Layer).
- The participants should know "The Twelve-Factor App".
- The participants should know Conway's law.

3.3 References

- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003
- http://12factor.net/
- Sam Newmann: Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2015
- Eberhard Wolff: Microservices – Flexible Software Architecture, Addison-Wesley, 2016, ISBN 978-0134602417
- http://martinfowler.com/articles/microservices.html

4 Integration

Duration: 90 min	Exercise time: 30 min

4.1 Terms and concepts

Frontend integration, legacy systems, authentication, authorisation, (loose) coupling, scalability, messaging patterns, domain events, decentralised data management.

4.2 Learning goals

4.2.1 What shall participants be capable of?

- The participants should choose an integration strategy that best suits the particular problem. This can be, for example, front-end integration, integration via RPC mechanisms, message-oriented middleware, REST, or replication of data.
- Participants should be able to design a suitable approach for implementing security (authorisation / authentication) in a distributed system.
- Based on these approaches, participants should be able to design a macro architecture that covers at least communication and security.
- For the integration of legacy systems, it has to be defined how to handle old data models. For this purpose, the approach of strategic design can be used with essential patterns such as anti-corruption layers.
- The participants can propose a suitable integration depending on the quality goals and the knowledge of the team.

4.2.2 What should participants understand?

- The participants should know the benefits and drawbacks of different integration mechanisms. These include front-end integration with Mash Ups, Middle Tier integration, and integration through databases or database replication.
- Participants should understand the implications and limitations that arise from the integration of systems across different technologies and integration patterns, relating to, for example, security, response time, or latency.
- Participants should have a basic understanding of the implementation of integrations using Strategic Design from Domain Driven Design and know fundamental patterns.
- RPC means mechanisms for synchronously calling functionality in another process over computer boundaries. This results in coupling in many respects (time, data format, API). This coupling has a negative effect on the availability and response times of the system. REST makes guidelines that can reduce this coupling (Hypermedia, standardised API). However basically, the temporal coupling remains.
- With integrating through messaging, systems communicate through the asynchronous exchange of messages. The systems are thus decoupled in time. Technically, this is achieved by means of indirection via a middleware. Messages can optionally be persisted, filtered, transformed, etc. There are different messaging patterns like Request / Reply, Publish / Subscribe or Broadcast.

- An integration via data enables high autonomy, nevertheless it is bought by the necessity for redundant data storage and the necessary synchronisation. It must not be assumed that other systems use the same schemes, because this prevents an independent development of the schemata. Therefore, an adequate transformation has to be provided for the integration.
- In an event-driven architecture (EDA), RPC is avoided or reduced by publishing domain events. Domain events describe state changes. Interested systems can process these messages (Publish / Subscribe). This procedure affects how the state is stored. While, in an RPC-based integration the server has to store the data, with EDA this is the responsibility of the subscriber. Thus, replicas arise (decentralised data storage). Thereby, the subscribers act as a cache for the publisher. Additional subscribers can be added without affecting the publisher (except by polling). Monitoring of the event flows is important.
- Domain events can be published via messaging. The publisher pushes the messages into a messaging system. Alternatively, the messages can be polled from the publisher (e. g., Atom / RSS). When using a messaging system, subscribers can receive the messages by push or pull. This has implications for dealing with backpressure.

4.2.3 What should participants know?

- Typical distributed security mechanisms such as OAuth or Kerberos
- Approaches for front-end integration
- Technologies for the integration of services: REST, RPC, message-oriented middleware
- Challenges of the usage of shared data
- Database replication mechanisms using ETL tools or other approaches
- Messaging Patterns (Request / Reply, Publish / Subscribe, etc.)
- Messaging systems (RabbitMQ, Kafka etc.), protocols (AMQP, MQTT, STOMP etc.) and APIs (JMS)

4.3 References

- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 978-0-32112-521-7
- http://oauth.net/
- Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 978-0-32120-068-6

5 Installation and Roll Out

Duration: 90 min	Exercise time: 30 min

5.1 Terms and concepts

Modern operations, DevOps, Infrastructure as Code, Configuration Management.

5.2 Learning goals

5.2.1 What shall participants be capable of?

- The participants should be able to roughly sketch a concept and understand how to automatically deploy a system as simple as possible. They should be able to weigh between the different technological approaches.
- Participants should be able to design and evaluate deployment automation. For example, they have to be able to assess the quality and testing of these approaches. They have to be able to select an appropriate approach for a project scenario.
- The participants should be able to sketch a team structure based on the DevOps organisation model.

5.2.2 What should participants understand?

- The foundation for automating deployment is virtualisation or the cloud with Infrastructure as a Service (IaaS). A lightweight alternative are Linux containers as implemented by Docker.
- Deployment of a large number of servers and services is virtually impossible without a deployment automation.
- Modern deployment tools allow to automatically install software on computers. In addition to the application itself, the complete infrastructure can also be created automatically. In this case installations are idempotent, which means that they always lead to the same result regardless of the initial state of the system.
- Immutable servers are never changed. If a new version of the software has to be taken into operation, the server is rebuilt from scratch. This can be simpler and more reliable than to rely on idempotent tools.
- PaaS (Platform as a Service) provides a complete platform where applications can be deployed in. Since in this case the infrastructure is not built up by oneself, the approach is simpler, but also less flexible.
- Concepts such as tests or code reviews are also indispensable for deployment automation. Infrastructure becomes code that must meet the same requirements as productive code (Infrastructure as Code).
- To support the deployment, the result of a build process can be packages for an operating system or even images for virtual machines.
- The environments of a developer should ideally match the production environments. With modern tools, it is possible to create and maintain such an environment at the push of a button.

- The complexity of the deployment becomes a further quality feature of the system and affects architecture tools.
- DevOps leads to a different team structure. Beside development, also more attention has to be paid to operations. In addition to provisioning, this also affects continuous delivery (see chapter "Continuous Delivery").

5.2.3 What should participants know?

- Basic concept of modern infrastructure such as IaaS, PaaS and virtualisation
- Concepts of deployment tools like Chef, Puppet, Ansible or Salt
- Organisation forms for DevOps
- Concepts of deployments with package managers or Linux containers
- Various PaaS platforms and their concepts

5.3 References

- Gottfried Vossen, Till Haselmann, Thomas Hoeren: Cloud-Computing für Unternehmen: Technische, wirtschaftliche, rechtliche und organisatorische Aspekte, dpunkt, 2012, ISBN 978-3-89864-808-0
- Eberhard Wolff, Stephan Müller, Bernhard Löwenstein: PaaS - Die wichtigsten Java Clouds auf einen Blick, entwickler.press, 2013
- Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
- Eberhard Wolff: A Practical Guide to Continuous Delivery, Addison-Wesley, 2017, ISBN 978-0-13-469147-3

6 Operations, Monitoring and Error Analysis

Duration: 90 min	Exercise time: 30 min

6.1 Terms and concepts

Monitoring, Operations, Logging, Tracing, Metrics.

6.2 Learning goals

6.2.1 What shall participants be capable of?

- The participants should be able to roughly sketch and understand a concept that allows to monitor a system, that means assess the status, avoid errors and deviations from the regular operation as far as possible or at least detect and handle them as early as possible.
- In the concept, they may focus on logging, monitoring and the data that is required for that purpose, depending on the specific project scenario
- Participants should be able to meet architecture requirements in a way that supports the usage of appropriate tools, while reasonably dealing with system resources.

6.2.2 What should participants understand?

- Logging and monitoring can include both functional as well as technical data.
- The appropriate selection of data is essential for reliable and meaningful monitoring and logging.
- In order to achieve systems that are ready to operate, in particular those composed of many individual subsystems, the support of operations has to be an integral part of the architecture concepts.
- In order to achieve the highest possible transparency, a great deal of data has to be collected, but also be pre-aggregated and made evaluable.
- The participants should understand which information can be obtained from log data and which may (better) be obtained by instrumentation of the code with metric probes.
- The participants should understand how a typical centralised log data administration is built and what impact it has on the architecture.
- The participants should understand how a typical centralised metrics pipeline is built (capture, collect & sample, persist, query, visualise) and the impact it has on the architecture (performance overhead, memory consumption, ...).
- The participants should understand the various options of logging, monitoring and an Operations DB (see M. Nygard, Release IT!), which to use wherefore and how to meaningfully combine these tools.

6.2.3 What should participants know?

- Tools for centralised log data management
- Tools for centralised metrics processing
- Difference between business, application and system metrics
- The meaning of important, system-independent application and system metrics

6.3 References

- Eberhard Wolff: A Practical Guide to Continuous Delivery, Addison-Wesley, 2017, ISBN 978-0-13-469147-3
- Michael Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8

7 Case Study

Duration: 90 min	Exercise time: 60 min

Within the scope of a curriculum-oriented training, a case study has to explain the concepts in practice.

7.1 Terms and concepts

The case study does not introduce new terms and concepts.

7.2 Learning goals

The case study is not intended to impart new learning goals, but to deepen the topics through hands-on exercises and to illustrate the practice.

7.3 References

None. The training providers are responsible for the selection and description of examples.

8 Perspective

Duration: 120 min	Exercise time: 0 min

The lookout presents advanced topics that participants may dive into. So, they gain a deeper understanding of the challenges of implementing flexible systems. In addition, they learn other factors influencing the choice of technologies.

8.1 Terms and concepts

- Consistency models: ACID, BASE, partitioning, CAP
- Resilience: Resilient Software Design, Stability, Availability, Graceful Degradation, Circuit Breaker, Bulkhead

8.2 Learning goals

8.2.1 What shall participants be capable of?

- The participants should know various consistency models. They should basically know the trade-offs of the different consistency models.
- Depending on the requirements and general conditions, they should be able to decide whether traditional stability approaches are adequate or resilient software design is required.

8.2.2 What should participants understand?

- Consistency models
  - The need for ACID transactions is much lower than is often assumed.
  - Different scaling, distribution, and availability requirements require different consistency models.
  - The CAP theorem describes a spectrum in which, depending on the given requirements, a suitable consistency model can be selected very fine-grained.
  - BASE transactions guarantee consistency, however, they may not be atomic and isolated as ACID transactions, which may lead to temporary inconsistencies becoming visible.
- Resilience
  - Traditional stability approaches (error avoidance strategies) on infrastructure levels are generally no longer sufficient for today's distributed, highly networked system landscapes.
  - There is no Silver Bullet for Resilient Software Design, i. e., the relevant measures and applied patterns and principles depend on the requirements, the general conditions and the persons involved.

8.2.3 What should participants know?

- Consistency models
  - Characteristics of and differences between ACID and BASE transactions

    - Some product examples from different categories (e. g., NoSQL, configuration tools, service discovery)
    - CAP for describing and explaining consistency models
- Resilience
    - The formula for availability and the different approaches to maximise availability (maximising MTTF, minimising MTTR)
    - Isolation and latency monitoring as useful starting principles of Resilient Software Design
    - Basic Resilience patterns such as bulkhead, circuit breaker, redundancy, failover

8.3 References

- Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006
- Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169
- Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000
- Mikito Takada, Distributed Systems for Fun and Profit, http://book.mixu.net/distsys/ (Guter Einstieg und Überblick)
- Michael T. Nygard, Release It!, Pragmatic Bookshelf, 2007
- Robert S. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007
- James Hamilton, On Designing and Deploying Internet-Scale Services, 21st LISA Conference 2007

9 Sources and references to information systems for agile environments

This section contains sources, which are referenced in whole or in part in the curriculum.

B
Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000

E
Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003

F
Martin Fowler: Microservices, http://martinfowler.com/articles/microservices.html

H
James Hamilton, On Designing and Deploying Internet-Scale Services, 21st LISA Conference 2007
Robert S. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007
Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 978-0-32120-068-6
Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
Jez Humble, Barry O'Reilly, Joanne Molesky: Lean Enterprise: Adopting Continuous Delivery, DevOps, and Lean Startup at Scale, O'Reilly 2014, ISBN 978-1-44936-842-5

L
Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

N
Sam Newmann: Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2015
Michael T. Nygard, Release It!, Pragmatic Bookshelf, 2007

O
OAuth: http://oauth.net/

T
Mikito Takada, Distributed Systems for Fun and Profit, http://book.mixu.net/distsys/ (Guter Einstieg und Überblick)
Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006

V
Gottfried Vossen, Till Haselmann, Thomas Hoeren: Cloud-Computing für Unternehmen: Technische, wirtschaftliche, rechtliche und organisatorische Aspekte, dpunkt, 2012, ISBN 978-3-89864-808-0

W
Eberhard Wolff: Microservices – Flexible Software Architecture, Addison-Wesley, 2016, ISBN 978-0134602417
Eberhard Wolff: A Practical Guide to Continuous Delivery, Addison-Wesley, 2017, ISBN 978-0-13-469147-3
Eberhard Wolff, Stephan Müller, Bernhard Löwenstein: PaaS - Die wichtigsten Java Clouds auf einen Blick, entwickler.press, 2013