In this course unit [CM-W-WEB] a structured software development process of web applications is illustrated. The application development makes use of current software development concepts, such as Behavior-Driven Development (BDD), Domain-Driven Design (DDD), microservice architectures including a systematic design of the web Application Programming Interfaces (API) of the microservices.

1) An overview of the relevant development concepts is given and the application development process using these concepts is introduced. The core concepts and technologies are illustrated with the example of a web application, by name TodoListManagement (TLM), is introduced.

2) The development of advanced web application requires a complex tool environment. The organization of software development projects is supported by a specific set of project management and version control tools which are described in more detail in this chapter.

3) This chapter deals with the analysis, the first phase of the development process, in which the requirements are specified. In the approach described in this course unit, so-called Gherkin features are used. Those features are the central artifact of the BDD concept.

4) The domain model is the central artifacts of the subsequent design phase. In the domain model established according to DDD principles, the (structural and functional) knowledge needed to implement the software system is formally captured.

5) In the implementation phase the functionality as is was specified in the analysis and design phase is coded based on a microservice architecture. The domain model is systematically transformed into code and the code is extended by annotations by which the microservice API is specified as a part of the code.

6) The implemented web application is deployed in a container-virtualized infrastructure. Leading technologies used to build and run this infrastructure are Docker and Kubernetes.

API Application Programming Interface
BDD Behavior-Driven Development
C&M Cooperation & Management
DDD Domain-Driven Design
EA Enterprise Architect
KIT Karlsruhe Institute of Technology
TLM TodoListManagement

[CM-W-WEB] Cooperation & Management: WEB APPLICATION DEVELOPMENT. WASA Course Unit. https://team.kit.edu/sites/cm-tm/Mitglieder/2-2.WASA_Lecture
The motivation behind Behavior-Driven Design (BDD) is to build and deliver better software [Sm15:3].

(1) According to a number of studies, nearly half of all software projects fail to deliver in some significant way.

(1.1) In a CHAOS Report published by the Standish Group, in 42% of the software projects one or more of these problems occurred.

(1.2) This results in billions of dollars in wasted effort.

(2) The two goals center around the questions of WHAT to develop and HOW to develop.

(2.1) This goal describing the WHAT is shown by the vertical (y) axis in the figure.

(2.2) This goal describing the HOW is shown by the horizontal (x) axis in the figure.

(3) BDD encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand.

BDD Behavior-Driven Development

Throughout this course unit, the TodoListManagement (TLM) web application by which a user can manage his/her todos in lists, is used to demonstrate the technologies and methods of web application development. This application is also in the focus of the initial phase of the practical and seminar course offered in parallel to the lecture. A detailed technical description can be found in [CM-W-TLM].

(1) To be able to develop complex web applications it is not only necessary to (theoretically) understand the quite complex concepts, but also to (practically) apply them. With the TLM example the practical use of the concepts is demonstrated.

(<<client>> TLM Frontend, HTTP. <<server>> TLM Backend) Since TLM is a web application it consists of a web client and a web server which communicate via the Internet application protocol HyperText Transfer Protocol (HTTP).

The diagram describes the physical architecture of the TLM application and uses modeling elements specified by the Unified Modeling Language (UML). It is a so-called deployment diagram by which the physical systems (in this case, TLM Frontend and TLM Backend) are modeled as specific UML symbol called nodes.

(Web Technologies) Technologies play an important role in web application development. According to the physical architecture of a web application, frontend and backend technologies as well as technologies concerning the interface (Application Programming Interface, API) can be distinguished.

(A, TypeScript, JS) The “A” logo stands for Angular which is a leading frontend technology. Angular uses the script language TypeScript which is based on JavaScript (see JS logo).

(TLM GUI) This is a cutout of the TLM Graphical User Interface (GUI) which is implemented in Angular.

(Spring, Java) Spring is one of the leading Java-based backend technologies.

(OpenAPI Specification) The OpenAPI Specification is a manufacturer-independent defacto standard for the definition of REST APIs.

(REST API) (TLM REST API) REST stands for Representational State Transfer which is the most widely used concept for the specification of web APIs.

(POST /todo-lists/ [id] todos, ... GET /todo-lists/... PUT ...) This is an excerpt of the REST API which is provided by the todo-list microservice. Each REST operation is a specific HTTP method, such as GET or POST. For example, the REST operation "POST /todo-lists/ [id] todos" has a parameter which describes the todo which is added to the todo list identified by "id".

API Application Programming Interface
HTTP HyperText Transfer Protocol
JS JavaScript
REST Representational State Transfer
TLM TodoListManagement
UML Unified Modeling Language

The first phase of the development process is called the analysis phase. Therefore, the language sketch is an analysis artifact. C&M's development process is introduced on page ++Development of a Microservice-based Application++.

(2) The User can manage the Todo Lists and the Todos they contain. Every Todo List can contain multiple Todos. A Todo has a Description and its Content. The status of a Todo can be marked as done or as outstanding. All central terms appearing in the language sketch are part of a so-called ubiquitous language. These terms are written in upper case and they should be uniquely used by all stakeholders, i.e. the business experts and the development team.

(3) GitLab is a modern and powerful development tool which supports the whole development lifecycle. It used Git as version control system and organizes all files generated during the software development in so-called repositories. Besides the different types of repositories used for the code, a specific documentation repository exists for each software developed by C&M. (CMTeam > TodoListManagement > 0Doc_TodoListManagement > Repository) The figure shows the documentation repository of the TLM application [CM-G-0Doc-Todo]. Besides the language sketch, other analysis artifacts (e.g. Vision, Business Goals and Capabilities or Features) and design artifacts (e.g. Context Map or API Specification) can be found in this repository.

The TodoListManagement (TLM) web application serves as an example to demonstrate the concepts of web application development introduced in this course. All artifacts and the code of the web application is available in the GitLab [CM-G-TLM].

1. A concept that is used to specify and test the functionality is the Behavior-Driven Development (BDD). The main artifact of BDD are the so-called features which are described in the language Gherkin.

2. The sample project is a simple todo administration software.
   2.1. It should be possible to create a new todo list and add it to the existing todo lists. Lists can have a title and an unlimited number of todos.
   2.2. Existing todo lists can be edited.
   2.2.1. The title can be changed or a new todo can be added to the list.
   2.2.2. It is also possible to mark a todo as done.
   2.3. Another requirement is to delete existing lists.

1. Feature: Creating a todo list Most of the feature description is written in a natural language. In order to allow interpretation with the computer, Gherkin prescribes some keywords (in the example: Feature, Background, Scenario, Given, When, Then, And) which must be considered. Gherkin is used by the Cucumber framework [Cuc-Ger].
2. As a ... (3. I want ...) (4. So that ...) Each feature starts with a story which informally introduces the role and what this role wants to do why.
6. (11.) The feature consists of two scenarios. Each scenario describes a specific part of the behavior of the application which is specified by preconditions (Given), actions or events (When) and the testable outcomes (Then).

(edu.kit.cm.springBootDemo.cucumber.RunCucumberTests) In all later phases of the software development process the definition can be used to test the software. In this way, it is possible to test whether a software meets the requirements and, in case of changes to the code, to check whether all requirements are still being met. The tests are started as Junit tests and can be integrated into all existing tools [Jun-Ju4].

BDD Behavior-Driven Development
TLM TodoListManagement

The figure introduces the different phases that have to be carried out to develop software systems, esp. web applications. Besides the programming language, natural language and several description languages are needed to develop a web application in a sound and systematic way.

(1) Iterative in this context means that the phases are not completely passed in a sequential order but jumps to a the preceding phase are the usual case.

(Feature, Domain Model) Features are the main artifacts from Behavior-Driven Development whereas the domain model is the main artifact introduced by the Domain-Driven Design (DDD).

(Unified Modeling Language, eXtensible Markup Language, Hypertext Markup Language) are widely used description languages in the development process of web applications. Gherkin is a language which is used in the BDD approach.

(OpenAPI) This is a standardized language by which an API (Application Programming Interface) of a microservice based on REST (REpresentational State Transfer) can be specified.

<table>
<thead>
<tr>
<th>API</th>
<th>Application Programming Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDD</td>
<td>Behavior-Driven Development</td>
</tr>
<tr>
<td>DDD</td>
<td>Domain-Driven Design</td>
</tr>
<tr>
<td>REST</td>
<td>REpresentational State Transfer</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
This figure illustrates the various architectural concepts and their relationships. It should be noted that no standardized definitions exist. For example, [CMU-SA] lists more than 30 definitions for software architecture.

(Software Architecture) The software architecture takes a logical view on the system by dividing it into logical components.
(System Architecture) The system architecture takes a physical view on a system and describes its structure which consists of network and hardware components [VA+08]. Furthermore, their properties and relationships to each other as well as to their environment and other systems are shown. A possible modeling type is the deployment diagram.
(Implementation Architecture) The implementation architecture describes the structure of the system from a technical point of view [RM+06]. This includes packages, libraries and frameworks.
(Macro Architecture, Micro Architecture) The macro architecture is used to describe the principles of the subsystems, e.g. microservices or modules, on a black box level. The micro architecture deals with the internal structure of a single subsystem [Ot13].

(Architectural Style) Architectural styles [RH08] are application-independent solution principles that are used throughout the project. These can be assigned to the following categories: communication (e.g. message bus oriented), deployment (e.g. N-tier architecture) and structure (e.g. layer architecture).
(Design Pattern) Design patterns [RH08] provide program code-specific patterns (e.g. visitor pattern) for recurring problems.
(Architectural Pattern) Architectural patterns [RH08] are solutions to recurring problems which, unlike design patterns, affect several architectural elements.

(Component / Deployment / Class Diagram) Deployment View) The Unified Modeling Language (UML) provides different types of diagrams to specify the different types of architecture as models.

(1) ++Challenges in Software Development++
- Goal 1: To develop the right software (WHAT should be developed?)
- Goal 2: To develop the software right (HOW should the software be developed?)

(2) ++Example: Web Application Demo TodoListManagement (TLM)++
- Physical systems: Web client (frontend), web server (backend)
- HTTP (HyperText Transfer Protocol) which is an Internet application protocol.

(3) ++Example: Web Application Demo TodoListManagement (TLM)++
- Angular: A leading frontend technology
- TypeScript: Used by Angular and based on Java Script
- Spring, Java: Spring is one of the leading Java-based backend technologies.
- OpenAPI Specification: A manufacturer-independent defacto standard for the definition of REST APIs.

(4) ++Example: TLM Language Sketch++
- Language Sketch
- The terms are part of the so-called ubiquitous language

(5) ++General Software Development Process++
- In the analysis phase
- Remark: Features are described by using the language Gherkin. They are the central artifact of the Behavior-Driven Development (BDD).

(6) ++Types of Architectures and Relationships++
- Software architecture: logical view, UML component diagram
- System architecture: physical view, UML deployment diagram
(1) The fundamental principles of each plane (e.g. layering) are described in the following.

(Network Plane) Web applications run on several independent systems which heavily use Internet technologies for communication purposes. The network plane consists of communication layers. In each such layer the communication is organized by standardized protocols (e.g. IP, HTTP).

(System Plane) Internet protocols (such as the web protocol HTTP) are based on a client-server principle. The system plane is characterized by different operating systems leading to heterogeneous IT infrastructures.

(Application Plane) Web-based applications are located on the application plane. The term "web-based" especially means that the presentation layer is implemented by a standard web browser.

(DATA, BUSINESS LOGIC, PRESENTATION) The logical separation concerns the different (logical, conceptual, architectural) aspects a distributed application has to cover. There are three aspects – data, function (or business) and presentation – which can be found in every application.

(2) The Unified Modeling Language (UML) is an adequate language to graphically describe the architecture of a software application. Modeling is necessary in order to develop the application in a structured and systematic way.

(Stereotype describing a component) A component in UML is modeled as a rectangle with a specific icon in the right upper corner. This icon is a graphical representation of a stereotype which is further defined in the UML meta model. An equivalent textual representation of this stereotype defining a component is <<component>>.

(Ball-Socket symbol describing an interface) The ball represents the provided part and the socket represents the required part of the component interface.

DB Data Base
GUI Graphical User Interface
HTTP HyperText Transfer Protocol
IP Internet Protocol
UML Unified Modeling Language
There are so-called communication reference models that define the functionality of the layers that build the network plane. Two important models are shown on this page [KR05, Ta06].

(HTTP, Request Content, Provide Content) The HyperText Transfer Protocol (HTTP) is an Internet application protocol which allows a HTTP client to request and also change content residing at an HTTP server by offering different HTTP operation, such as GET or POST.

(1) (2) The merging of the two lower layers in the Internet architecture has "organizational" reasons whereas the missing layering in the application system is conceptually motivated.

HTTP HyperText Transfer Protocol
IEEE Institute of Electrical and Electronics Engineers
IP Internet Protocol
ISO International Standards Organization
OSI Open Systems Interconnection
TCP Transmission Control Protocol

On the system plane of a web application the involved systems take the roles of clients and servers.

(1) Processing is usually organized according to a client-server principle: The client asks the server to do some processing.

(2) Central tasks the server systems listed in the slide have to provide, are:
- File server: Upload and download of files
- Database server: SQL-based search queries
- Print server: offers functionality specific to the print service, such as queuing or prioritizing of print orders
- Application server: contains executable code specific to one or more applications

SQL Standard Query Language
A microservice architecture is located on the application plane as it is introduced in ++Planes and Three-Layer Application Architecture++. It consists of three parts (backend, backend-for-frontend BFF, frontend) and two types of application programming interfaces (BFF) via which these parts are interacting (backend microservices API, BFF microservices API).

(Presentation Layer) This layer renders the UI elements in the browser. Technologies that support the implementation are Angular and Bootstrap.

(Presentation Logic Layer) The presentation includes a logic which controls the interaction with the BFF microservice API.

(Application Logic Layer) This layer realizes the orchestration of the backend microservices in order to provide the BFF microservices required by the frontend. A technology that supports the implementation of this functionality is Spring.

(Domain Logic Layer, Infrastructure Layer) These layers implement the backend services which are CRUD operations on the domain objects.

In contrast to a traditional three-layer application architecture the business logic layer in a microservice architecture is split into two layers, the domain logic layer and the application logic layer. The reason for that is to promote the reuse of business logic functionality by distinguishing between application-agnostic (= domain logic) and application-specific (= application logic) functionality.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BFF</td>
<td>Backend For Frontend</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Read, Update, Delete</td>
</tr>
</tbody>
</table>
The software development process applied by C&M combines the concepts of Behavior-Driven Development (BDD) and Domain-Driven Design (DDD). Both concepts provide complementary contributions to the layered microservice architecture as the figure illustrates.

1. An implemented feature can be seen as a deployable increment of the software system. (Feature 1, Feature 2, ...) The ordering of the features in the figure implies that the first feature should cover the core functionality of the software system.

2. The domain model makes sure that the static and dynamic domain knowledge is consistently used by each feature. This ensures that the features build a consistent whole although each feature is developed and deployed independently from other features.

BDD  Behavior-Driven Development
DDD  Domain-Driven Design
The activity diagram gives an overview of the whole engineering process which in fact is divided into two sub-processes: According to our understanding of DDD, a domain model is the foundation of all software applications belonging to this domain. We regard the domain model as the knowledge of a domain expert he or she makes available to the software developer by the model. This knowledge is the application-agnostic part of the application meaning that each application of the domain implements this knowledge in its domain logic layer. We understand this domain knowledge as the "physics" that each application must adhere to. Evans calls this part of functionality the heart of the software system [Ev03].

(Starting Points) (Strategic Modeling of the Initial Context Map) The starting point of the domain modeling process is set earlier as the starting point of the application development process. The reason is that it makes sense to have an initial context map of the domain before the development of the first application starts. In the following we describe the main characteristics of the application development process.

(Context Map) The main artifact of the domain modeling process is the context map in which all micro-service-based applications belonging to the domain must fit into.

(Strategic Modeling of the Bounded Contexts as Part of the Domain's Context Map) In the application development process, the integration of the application under development into the context map of the domain is done in the second step after the first step in which the BDD-based requirements analysis is carried out.

(Tactical Modeling of a Bounded Context) A bounded context is part of a context map. It represents a candidate for a microservice which can be developed by an independent team [Ne15]. DDD considers the modeling of the content (i.e. the functionality) of a bounded context as tactical modeling.

(Relation View) The most relevant artifact of the tactical modeling is the relation view which is built in the third step. By the relation view central DDD concepts (such as entity and value objects) including their relationships and operations are represented.

(Implementation of each Bounded Context as a Microservice Accessible Via its Web API) The functionality modeled by the relation view is implemented in the domain logic layer of the microservice architecture. A systematic approach to implement the bounded context as a microservice (fourth step) is a core characteristic of the proposed development process. One of the key demands of DDD is to keep the model and its implementation in sync in order to assure that the model and the implementation is the same.

The two remaining steps concern development activities around the core implementation of the microservice, namely the implementation and test and the deployment of the application.

The figure shows that concepts specific to the microservice field do not replace, but complement, existing software engineering concepts.

(Analysis Techniques) In the first phase of the engineering process, the analysis phase, different techniques to elicit and specify the requirements exist.

(Ubiquitous Language (BDD, DDD)) In the process we apply for the development of microservice-based software, the requirements are specified by Gherkin features which are part of Behavior-Driven Development (BDD). There is a match between BDD and microservices since BDD provides a ubiquitous language which is needed to design the APIs of a microservice.

(Domain Modeling (DDD)) BDD fits well to the the Domain-Driven Design (DDD) which we estimate to be one of the most relevant concept for the design of microservices.

(Design Patterns) DDD is a set of design patterns which are a well-known engineering concept to capture the knowledge of the structure (architecture, code) of a software system.

(Microservice Architecture) One of these DDD patterns, called Layered Architecture, provides the foundation of the microservice architecture. By the Layered Architecture pattern, the functional layer from the well-known three-layer application architecture is divided into a domain logic layer and an application logic layer. The functionality captured in the domain logic layer has to be application agnostic. This is exactly the functionality a microservice API should provide. By using the DDD patterns, a domain model of the domain, the software system belongs to, is built.

(API Design) The domain model supports both the implementation of the microservice and the design of the web API.

(Frontend/Backend Frameworks, Microservice-Supporting Frameworks) For the implementation of a micro-service-based web application, powerful frameworks for the implementation of the frontend and the backend are available. Certain frontends provide a specific support for the development of microservices, especially related to the web API via which the functionality of the microservice can be accessed.

(CI/CD) One of the driving forces of microservices is the continuous integration (CI) and continuous deployment (CD) of implemented software systems. In contrast to monolithic architectures, microservices are loosely coupled. Changes in the software system relate to single micro-services which can be tested and deployed independently from the other part of the software system. In practice, each microservice is deployed in a so-called container which provides the computing infrastructure to run the microservice.
(1) ++Planes and Three-Layer Architecture++
   - Application plane
   - System plane, network plane

(2) ++Planes and Three-Layer Architecture++, ++Microservice Architecture++
   - Separation of the business logic layer (middle layer) into two layers, the application layer and the domain layer
   - Explicit introduction of two APIs, the Backend Microservice API and the Backend for Frontend Microservice API

(3) ++Features and the Domain Model in a Microservice Architecture++
   - BDD: Each feature is a vertical cut through all layers of the microservice architecture
   - DDD: The domain model is implemented in the domain logic layer and provides the semantical foundation for all features

(4) ++BDD/DDD-Based Software Development++
   - BDD: Influences the analysis by providing the specification based on features. In addition, a feature defines the software increment which is implemented in an agile way.
   - DDD: The domain model is the central design artifact of the development process.
(1) On the one hand, the tools support the software engineers and make software development easier and more efficient. On the other hand, the broad spectrum of different tools makes software development more complex since the correct use of the tools is a time-consuming challenge for each developer.

(1.1) The phases are: analysis, design, implementation, testing, deployment. Management tools cannot be assigned to one specific phase are which are used by the teams for communication purposes and for the organization of the software project.

(2) This type of development tools is indispensable for the development of software.

(2.1) Construction tools are used in the implementation phase.

(3) An example of a software requirements tool is Cucumber which supports the approach of behavior-driven development (BDD). Examples of software design tools are UML modeling tools or tools to specify the APIs (application programming interface) of a software architecture.

(4) Software testing tools include test generators, test execution frameworks, test evaluation, test management and performance analysis. Tools which support the deployment are called software configuration management tools [Ca01].

(5) The main support provided by this type of development tools is project planning and tracking of tasks.

Software engineering tools cover two major aspects which must be handled in every software project: the management aspect dealing with organizational and project-related problems and the development aspect dealing with technical and (software) system-related problems.

(Communication and Collaboration Platform) The tools for solving either organizational or technical problems of software engineering are not decoupled. If a technical problem occurs this has influence on the overall planning. In order to manage these relationships a common communication and collaboration platform for management and development is useful.

(Software Repository) A software repository enables the storage of different versions of the software in a way that each software engineer can communicate the changes he has made in a systematic way to the other team members.

(Project Management and Organization) The relevance of this group of software engineering tools rises with the size of the projects and the software engineers involved.

(Project Planning and Tracking) This type of tool is used by a project manager (in Scrum this role is called project owner) to plan the resources (esp. the developers) that are available in the project. In [Ca01] the project planning and tracking is part of the so-called software engineering management tools.

(Issue and Problem Tracking) A team of software engineers uses this type of tool to organize technical, i.e. software-related tasks.

(Software Development) In this group of tools the development process starting with the analysis and ending with the deployment of the software is supported. The list of tool types mentioned in the figure is not complete. Tool types which are missing include the software quality tools or software maintenance tools.
The main dimension according which the development tools can be ordered are the development phases (from analysis to deployment). The tools supporting the analysis and design and the implementation and testing can be grouped according to the software architecture (which in our case is a microservice architecture).

(Project Management and Version Control) These tools support the overall organization of the software project and the communication between the project members. The most important tool used by C&M for this purpose is GitLab which is based on Git.

(Analysis and Design) Analysis requirements at C&M is done by taking the approach of behavior-driven development (BDD) based on the tool Cucumber. For the design the two most relevant tools are Enterprise Architect for the domain model and Swagger for the API specification. In addition to these tools the Microsoft Office tools (Word, PowerPoint) and the GitLab repository are applied for documentation purposes.

(Implementation and Testing) In this phase the frontend and backend of the web applications are constructed. The integrated development environment (IDE) used for frontend development is Visual Studio Code and frontend frameworks are Angular and Bootstrap. As IDE for backend development Eclipse is used and Apache Spring (esp. Spring Boot for the microservice implementation) is used as backend framework.

(Deployment and Operation) The deployment of the microservices is carried out via a build pipeline by which the concept of continuous integration and continuous deployment is provided. The result of the build pipeline is a Docker image (= application container image) since Docker is used as the container environment at C&M. Kubernetes is used to manage the Docker containers in order to reach a high scalability and robustness of the service landscape in the operation phase of the microservices.

BDD  Behavior-Driven Development
IDE  Integrated Development Environment
Analysis and design comprises the development work that is needed to implement the right software in the right way.

(1) Analysis is concerned with the question which software the user requires [CM-W-BEH].
(1.1) Gherkin is a language which introduces a minimal structure into the largely informal description of the requirement description.
(1.2) Cucumber allows to test an implemented software based on the specified features. The test are called user acceptance tests.

(2) Design is concerned with the static and dynamic structures of the software system that is to be implemented [CM-W-DOM].
(2.1) The concept of domain-driven design (DDD, [Ev04]) does not include a way how to specify the parts of a domain model. The Unified Modeling Language (UML) is a widely accepted modeling language which provides the flexibility to express the needed DDD modeling elements. Enterprise Architect is one of the leading professional UML tools to implement the DDD-to-UML approach.
(2.2) Based on a formalized UML-based domain model, the web application programming interfaces (API) of the microservice architecture can be systematically derived. The specification of the APIs is based on the OpenAPI defacto standard. Swagger is a well-known tool to specify and test the web APIs.

[CM-W-BEH] Cooperation & Management: BEHAVIOR-DRIVEN DEVELOPMENT, WASA Course Unit. https://team.kit.edu/sites/cm-tm/Mitglieder/2-1.WASA
[CM-W-DOM] Cooperation & Management: DOMAIN MODELING, WASA Course Unit. https://team.kit.edu/sites/cm-tm/Mitglieder/2-1.WASA
(1) **Spectrum and Types of Tools**
- Software construction tools: Program editors, compilers / interpreters, debuggers

(2) **Classification of Software Engineering Tools**
- Project Management and Organization: The relevance of this group of software engineering tools rises with the size of the projects and the software engineers involved.
- Software Development: In this group of tools the development process starting with the analysis and ending with the deployment of the software is supported.

(3) **Example: Tool Environment**
(3.1) SharePoint, Confluence, Jira, Bitbucket, Trello
(3.2) Git, Bitbucket
(3.3) Cucumber
(3.4) Enterprise Architect
(3.5) Angular and Bootstrap
(3.6) Spring
(3.7) Docker, Nexus, Portainer
In this chapter the software engineering tools are introduced that are used to manage and organize the project and the people involved in the whole software engineering process.

(1) Software systems tend to become complex which means that a whole team of software engineers is needed to develop the system. That is why software engineering means teamwork.

(1.1) The project teams are built in the beginning of each semester.

(1.2) The coach takes care that all team members contribute to the teamwork.

(2) In this context, project management comprises all tasks that should be carried out to organize the development tasks in a systematic and clear way.

(2.1) Features specify the requirements the software system must fulfill. They are described in a language called Gherkin. Since the agile development process is organized along the features with highest priority this artifact plays an important role for the project planning.

(2.2) The division into frontend-related and backend-related tasks make sense since they demand different competences.

(2.3) At C&M, the Bitbucket wiki is used to store the artifact descriptions which are edited collaboratively by the members of the development team.

(3) The development of Git started in 2005. It is currently the most widely used version control system.
A version control system (VCS) is one of the most relevant software tools to develop complex software in a team [Atl-Wha].

(1.1) A local VCS is a database that keeps all the changes to files under revision control. The differences from one file version to the next file version usually is stored in a certain format. An example of a local VCS is RCS (Revision Control System). The central and distributed VCS are explained in what follows [CS14].

(2) A disadvantage of local VCS is that a collaboration of developers working on distributed systems is not supported which has led to the central VCS. In a central VCS, clients running on different systems check out files from the central place.

(2.1) Subversion is the successor of CVS which stands for Concurrent Versions System.

(3) In a central VCS, clients only check out single files and not the repository (i.e. the whole file system).

(3.1) The full mirror includes the full history of all the files in the repository. This leads to a high stability.

(3.2) BitKeeper today is an open-source distributed VCS which was used to develop the Linux kernel. Some conflicts caused by the fact that BitKeeper at that time was proprietary (owned by a company called BitMover) has led to the development of Git. Mercurial is a free software which was announced and developed at the same time as Git in 2005.

(Computer A <---> Computer B) Note: There is only an "indirect" relationship between the participating computers A and B (leading via the server computer).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
</tr>
<tr>
<td>RCS</td>
<td>Revision Control System</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
</tbody>
</table>


Git was developed by Linus Torvalds, the initiator of the Linux kernel. Development started in 2005 [HP16]. The reason for the development of Git was the need of a version control system for the Linux kernel development.

1.1 Git allows to record and manage the changes of files.

1.2 Git is the appropriate tool both for single persons and for large projects in which hundreds of developers cooperate (such as the Linux kernel development).

1.3 Examples of other VCS include CVS (Concurrent Versions System) or Subversion (successor of CVS) which store data as a list of file-based changes (since the changes of files are stored as deltas, they are named delta-based version control).

2. A snapshot in Git is a picture of all the files of the project. For efficiency reasons for all files that did not change from version to the other only a link to the previous identical file is stored.

(Version 3, A1, B, C2) The snapshot Git takes for Version 3 are links for A1 and B since they did not change (illustrated by the dotted lined ellipses). Only C2 is stored since it is different from C1 (which is file C in Version 2).


This page gives an overview of the most relevant commands provided by the distributed version control system Git [Atlassian: Getting Started].

(1) A Git repository (also called Git repo) is a virtual storage of the versions of code from a project.

(1.1) `git init` creates a new .git subdirectory in the current working directory. It will also create a new master branch.

`git clone <repo url>` creates a copy of the indicated remote repository. The URL format depends on the network protocol that is used.

`git config` sets Git configuration values (e.g. email or username) on a global or local project level. It also allows to create shortcuts for frequently used Git operations in order to work with Git more efficiently.

(2) In Git, each change of the local repository is done via a so-called staging area which can be seen as a buffer between the working directory and the project history.

(2.1) `git add` and `git commit` compose the fundamental Git workflow. They are the means to record versions of a project into the repository's history. The stage is used by Git to group related changes into highly focused snapshots before actually committing it to the project history (when `git commit` is executed).

`git stash` (germ. bunkern) temporarily saves uncommitted changes and reverts them when a developer later wants to continue his work.

.gitignore is a file which contains all names of files that should be ignored by Git (e.g. build artifacts or machine generated files, such as compiled code).

(3.1) `git status` displays the state of the working directory and the staging area.

`git log` provides information regarding the committed project history.

(4) These commands allow to explore old commits and undo the changes in different ways.

(5) By using these commands the Git history can be rewritten or altered.

(6) These commands are used by a developer to share a series of commits with other developers.

GitLab is an open source software solution which uses the functionality of the version control system Git. Examples of solutions comparable to GitLab are: GitHub (owned by Microsoft), Bitbucket (owned by Atlassian).

(1) Several large tech companies (e.g. NASA, CERN, Boeing, IBM, LRZ, KIT) use GitLab.
(1.1) The management includes the documentation by a wiki, workflow support by an issue tracker and automatic build function by a CI/CD pipeline.
(1.2) Go (often referred to as Golang) is a statically typed, compiled programming language syntactically similar to C and designed at Google. Ruby on Rails, or Rails, is a server-side web application framework written in Ruby. Vue.js (commonly referred to as Vue; pronounced like view) is an open-source JavaScript framework for building user interfaces and single-page applications and it is inspired by Angular.

(2) All KIT members (students, employees) who have a KIT Active Directory (AD) account can use KIT GitLab [KIT-GitLab].
(2.1) The Steinbuch Centre of Computing (SCC) is the IT service provider at the KIT.
(2.2) Each KIT AD account has the structure "xy1234".
(2.3) The profile can be added by clicking on the entry "Settings" from the menu which can be opened by clicking on the arrow on the right hand side of the public avatar.

(3) C&M uses GitLab for the documentation of all artifacts and the versioning of all the code that are developed during the structured development process (see also page ++BDD/DDD-Based Software Development Process++).
(3.1) The origin of and main concepts behind the repository stem from [Zi19].

AD  Active Directory
SCC  Steinbuch Centre for Computing

[KIT-GitLab] Karlsruhe Institute of Technology: GitLab am KIT SCC. https://git.scc.kit.edu
The goal is to make all relevant information on the software, developed by C&M, available in GitLab. The documentation especially includes the analysis and design artifacts specified during the development.

(1) It should be taken into account that the documentation is carried out in a "normal" Git repository (and not in the Wiki as one might expect). The reason for this is the missing branch concept in a Wiki which is needed by C&M to implement the reviewing as shown in the process on the right-hand side.

((content change)) (Commit Changes to New Branch) As shown by the documentation process, each major change which concerns the content (i.e. the semantics) leads to the generation of a new branch in which the changes are committed.

(Merge Request) (Accept / Merge Changes) By this GitLab operation, the changes are provided to other persons involved in the documentation. Usually, the main responsible person or owner of this part of the documentation is merging (i.e. adopting) the changes.

(2) GitLab uses an extension, called GitLab Flavored Markdown (GFM) [GL-Mark] of the standard Markdown, which is a lightweight markup language with plain text formatting syntax. The use of the Markdown editor Visual Studio Code is recommended. Alternatives are Typora, which is a WYSIWYG editor or GitLab's Web IDE (but not GitLab's single-file online code editor).

(1 # C&M – Documentation Guidelines ...) The figure on the lower right-hand side is a screen dump taken from Visual Studio Code. It shows the README Markdown file from the CM-Documentation repository [CM-G-CMD] in the left-hand window and its Markdown preview in the right-hand window.

Note: In all C&M documents screen dumps with black background color should be avoided. Therefore, the default setting of Visual Studio Code should be changed (File | Preferences | ColorTheme | Light+) when screen dumps are taken.

GFM GitLab Flavored Markdown

[CM-G-CMD] Cooperation and Management: CMDocumentation, Git Repository. https://git.scc.kit.edu/cm-tm/cm-documentation

Example: Repositories of the TLM Application

1. Repositories available for each C&M project
   1. Documentation (0Doc)
   2. Frontend (1UI)
   3. BFF (2BFF)
   4. Backend (3MS)
   5. Test (4Test)

2. The analysis and design artifacts can be found in the "0Doc_TodoListManagement" repository

3. The project details view shows the files of the repository, the last commit, the status of the last pipeline run and the README file

The structure and the content of a GitLab repository is shown with the repositories of the TLM web application.

1. The TLM demo provided by C&M is accessible via GitLab [CM-G-TLM].
   (CMTeam) This is the main C&M Group in GitLab. It contains guidelines for the correct usage of GitLab and subgroups for the different projects of C&M.
   (TodoListManagement) This is a subgroup of the "CMTeam" group and it contains the repositories of the TLM project. Every project of C&M has its own subgroup in the "CMTeam" group.
   1.1) 0Doc_TodoListManagement For each project a repository "0Doc_<project name>" should exist in which the artifacts that are valid for the whole project (i.e. frontend and backend) are stored.
   1.2) 1UI_TodoListManagement This is the repository in which the frontend code (UI stands for user interface) of the TLM application is stored.
   1.3) 2BFF_TodoListManagement In this repository the backend for frontend (BFF) code of the TLM application is stored.
   1.4) 3MS_TodoListManagement This is the repository in which the backend microservices (MS stands for micro service) of the TLM application is stored.
   1.5) 4Test_TodoListManagement Here the end-to-end tests (Cucumber tests) are stored.

2. Every C&M project subgroup has a "0Doc_TodoListManagement" repository that documents the projects and contains the analysis and design artifacts. The structure of this repository is defined in the C&M documentation guidelines [CM-G-CMD]. The guidelines also define the process that is used to make contributions to the documentation repository.

3. All the information stored in the source folder is loaded into the development environment by using the appropriate Git command (which is "git clone https://git.scc.kit.edu/cm-tm/todolistmanagement/3ms_todolistmanagement.git" in the case that the backend source code is loaded).
   (green check mark) Pipeline status for the last commit. A green check mark means the pipeline was finished successfully, a red cross means the pipeline failed.
   (master) This field is used to select the branch that is currently shown.
   (src) The source code files are part of the folder "src". An example of a source code file which describes the todo domain object is: "src/main/java/edu/kit/tm/cm/springdemo/domain/model/Todo.java".
   (.gitignore, Dockerfile, .gitlab-ci.yml, pom.xml) These are the control files needed for building and deploying the software.
   (README.md) In this file the most relevant information can be found which describes what the software provides, how it can be used, and what the current status and open tasks are.

[CM-G-TLM] Cooperation & Management: GitLab TLM group. https://git.scc.kit.edu/cm-tm/todolistmanagement
[CM-G-CMD] Cooperation and Management: CMDocumentation, Git Repository. https://git.scc.kit.edu/cm-tm/cm-documentation
(1) **Overview**
- Chronological planning
- Distribution of tasks
- Documentation

(2) **Version Control System**
- Local: VCS Client and VCS server run on one system. Disadvantage: collaboration of developers working on distributed systems is not supported.
- Central: Clients running on different systems check out files from the central place.
- Distributed: Client fully mirrors the repository. Each system holds a full backup of all data.

(3) **Git**
- Git stores snapshots in each client. A snapshot is a picture of all the files of the project. For efficiency reasons for all files that did not change from version to the other only a link to the previous identical file is stored.
- Subversion (successor of the Concurrent Versions System CVS) stores data centrally on the server as a list of file-based changes (since the changes of files are stored as deltas, they are named delta-based version control).

(4) **C&M Documentation in GitLab**
- Documentation takes place in the documentation repository (0Doc_...)
- Major change is not directly carried out in the master branch, but first committed in a new branch
- Merge request carried out by the person responsible for this part of the documentation leads to the changed documentation
The right software can only be built when the requirements the software must fulfill are precisely elicited. Only in this case the problem the software should solve is well understood.

(1) At project start the level of ignorance is at its maximum. The learning path is neither linear nor predictable.

(2) It is hard to predict what the development team will learn as the project progresses. It is not advisable trying to force reality to fit into your plan according to a Swiss Army proverb: "When the terrain disagrees with the map, trust the terrain."

(1) Behavior-Driven Development (BDD) is an approach to specify and test the behavior of a software system.

1. Software requirements are defined in a semi-formal way
2. Automated testing of the requirements is one goal
3. Ubiquitous language from the concept of Domain-Driven Design (DDD) is used

(2) BDD provides a conceptually new perspective on Test-Driven Development (TDD)

1. Formulation of tests as sentences
2. Use of a simple sentence template for the name of a test method
3. Replacement of the word "test" by "behavior"
4. Business value to prioritize not yet implemented features
5. Provision of a ubiquitous language for analysis

(1) Behavior-Driven Development (BDD) can be seen as an agile software development technique. Since the specified behavior is the basis of BDD it is also called Specification-Driven Development.

(1.1) The requirements comprise the tasks, goals and expected results of the software.

(1.2) The founder of BDD, Dan North, started from the concept of Test-Driven Development (TDD) which pursues the same goal of an automated and consistent testing of the software.

(1.3) The domain model constructed according to the concept of Domain-Driven Design (DDD, [CM-W-DOM]) provides the syntax and semantics (behavior) of the terms that are used to describe the tests according to the BDD concept [Wik-Beh].

(2) Dan North developed the ideas of BDD from deficiencies he saw in the TDD concept. In [No06], he wrote: "I decided it must be possible to present TDD in a way that gets straight to the good stuff and avoids all the pitfalls."

(2.1) The idea of formulating Junit test classes as sentences stems from a utility called agiledox from Chris Stevenson. Developers were motivated to use sentences for documentation purposes.

(2.2) The sentence template used by Dan North was: "The class should do something"

Example: testShouldFailForMissingSurname

(2.3) By this replacement it became clear what test means and how to proceed. The sentence describes the next behavior one is interested in.

(2.4) The question behind the determination of the business value is: What is the next most important thing the system doesn't do?

(2.5) The behavior-driven thinking was applied to defining requirements based on a consistent vocabulary for analysts, testers, developers, and the business. This idea of a ubiquitous language Dan North (together with the business analyst Chris Matts) had at about the same time when the DDD book from Eric Evans was published in 2004.

BDD     Behavior-Driven Development
DDD     Domain-Driven Design
TDD     Test-Driven Development

[CM-W-DOM] Cooperation & Management: DOMAIN MODELING, WASA Course Unit. https://team.kit.edu/sites/cm-tm/Mitglieder/2-1.WASA


(1) This is the definition of a ubiquitous language in the DDD context [Ev04].

(1.1) In the DDD approach the domain model becomes the core of a common language for a software project.

(1.2) The language is expressed by the domain model which should not only be understood by the software developer but also by the domain expert. The language connects all the activities of the team with the software.

(Figure) The ubiquitous language is defined by the overlap of the technical language and the business language.

(Domain Model Terms) The domain model terms are the terms that are understood by both the domain experts and the developer. Besides these terms there are technical terms and business terms which are used solely by developers and domain experts.

(Bounded Contexts) This is a delimited applicability of a particular model Bounded contexts give team members a clear and shared understanding of what has to be consistent and what can be developed independently.

(Large-Scale Structure) A set of high-level concepts, rules, or both that establishes a pattern of design for an entire system. A language that allows the system to be discussed and understood on a broad and high level.

A main characteristic of the process behind the Behavior-Driven Development (BDD) is the collaboration of three central roles (business analyst, developer, tester) in the development process. A fourth role is the business owner using the software.

(1) The business owner ... business needs.) The first step is similar to the traditional development process.

(2) The business analyst ... requirements together.) Two aspects in BDD are new and different compared to the traditional process:
(i) The collaboration of the business analyst, developer and tester.
(ii) The structured specification all three roles agree on.

(3) The scenarios guide the developer ... For the developer the specification defines what should be implemented.

(4) The tester ... for the tests) For the tester the specification defines what should be tested.

(5) The automated tests ... document the application) For the business analyst (and the two other roles) the specification and the outcome of the tests document the software and its current state of implementation. The scenarios can be seen as a low-level technical documentation and provide up-to-date examples of how the system works.

(6) The structured collaboration of the roles taking part in the development process leads to a software which does the right thing and which does the thing right.

(1) The advantage of BDD compared to the traditional development process is the common specification which prevents misunderstanding and miscommunication, especially between the business analyst and the developer. This specification which is structured according to a defined grammar and keywords is the core of BDD. It consists of a so-called story and scenarios described in natural language.

(Extended, Adapted) The text "6 The tested ..." was extended. The role of the business analyst (i.e., a green instead of a grey stickman) was adapted in step 5.
BDD provides a language Gherkin consisting of specific keywords used to define the features (and the scenarios to test these features) which the software system must fulfill. The proposed template is taken from two sources: The upper feature description part is taken from [DNA-Story] and the lower scenario description part is taken from [WH12].

(1) In the description the business value of the feature should be recognizable.
Although Gherkin does not prescribe any structure and use of specific keyword in this part, it is recommended to use the structure "As a – I want – So that" proposed by Dan North.

(2) Behind BDD's Given-When-Then concept a finite state machine can be seen.
The structure and the keywords "Given – When – Then" are prescribed by Cucumber.

Developers use lots of different words to describe what they want to build. The following terminology taken from [Sm15:88] is used to provide a clear understanding in this confusing part of software development.

1. The project vision provides a high-level guiding direction for the project. The problem must be clarified to understand which software system is needed by which users to solve the problem.

2. The software system must have a measureable, positive impact on the business of the customer the software is built for.

3. Capabilities give users or stakeholders the ability to realize some business goal or perform some useful task. A capability represents the ability to do something; it does not depend on a particular implementation. For example, "the ability to book a flight" is a capability.

4. A feature is a piece of functionality that is delivered to the end users or to other stakeholders to support a capability that they need in order to achieve their business goals [:98].

A feature is something that users can test and use in isolation. A feature can deliver business value in itself; once a feature is completed, it could theoretically be deploy into production immediately, without having to wait for any other features to be finished first. The description of the features is generally more effective than describing what the application does.

5. The typical format of a story is "As a – I want – So that".

6. Scenarios written in the format "Given – When – Then" make up the core of an executable specification.
Example: TLM Vision, Business Goals and Capabilities

(1) Vision
   (1) The customer should be given the opportunity to organize his/her todos. He/She should be able to do this in the simplest and quickest way possible.

(2) Business goals
   (1) Increase of the number of customers
   (2) Increase of customer loyalty
   (3) Generation of revenue from advertising

(3) Capabilities
   (1) Create todos in todo lists
   (2) Maintain several lists
   (3) Mark todos as completed via a status

By defining the vision, business goals and capabilities, the software system to be developed can be described on a high abstraction level. Since this description is part of the TLM documentation, it is stored in the documentation repository [CM-G-CMD].

(1) The vision should make clear for which purpose the software system is used. In this case the minimal core functionality of the todo list management is to organize the todos in the easiest and quickest way possible. By this vision a Minimum Viable Product (MVP) is described.

(2) More customers are attracted by a modern application which is easy to use and which provides intelligent functionality.
(2.1) More customers are attracted by a modern application which is easy to use and which provides intelligent functionality.
(2.2) Offering such an application, customer loyalty will increase and additional products can be sold to the customer.
(2.3) Revenue can also be generated from the advertising shown in the application.
Remark: The business goal pursued by C&M is to practically demonstrate the students the BDD/DDD-based software development with an easy example.

(3) The formulation of the capabilities in this example is close to the features that the (minimal) todo list management system should provide.

MVP Minimum Viable Product

[CM-G-CMD] Cooperation and Management: CMDocumentation, Git Repository. https://git.scc.kit.edu/cm-tm/cm-documentation
The example is taken from the TodoListManagement (TLM) which allows a user to organize todos in different lists.

(1) As the example shows (user acceptance) tests are written in a way that they can be understood by everyone, especially also by non-technical experts who know the domain and users who should use the system.

1. Feature: Creating a Todo List
   2. As a User
   3. I want to create a Todo List
   4. So that I can track my Todos

5. Scenario: Create a Todo List
6. Given I have the possibility to create a Todo List
7. When I create a Todo List with the title "WASA course"
8. Then the Todo List with the title "WASA course" is created

9. Scenario: ...

(1) Features can be read and written both by developers and stakeholders
   (1) Terms of the ubiquitous language are written in upper case

(2) Scenarios are (user acceptance) tests specified as examples

(3) This kind of acceptance tests are executable specifications

---

(2) Each scenario describes one example case.

6. Given I have the possibility to create a Todo List) The steps should be formulated in a way that no assumptions are made with respect to the GUI (Graphical User Interface). Visualization can be further elaborated by additional UI prototypes illustrating the interaction of the user with the software.

7. When I create a Todo List with the title "WASA course") The use of a concrete title (here: WASA course) makes clear the characteristic of a scenario. It describes one specific example or instance of the functionality. The use of instances also supports the specification of the test cases.

(3) This is the reason for Dan North to use the term Behavior-Driven Design instead of Test-Driven Design since the specification of the behavior is the goal and the test is only a means to reach this goal.

GUI Graphical User Interface
Dealing With Gherkin Features

(1) Gherkin features should be documented in a way that the stakeholder can be involved in the documentation process.

(1) C&M supports external reviews of the features stored in the GitLab documentation repository.

(2) Best Practices

   (1) Start with features that cover the core functionality of the software system.
   (2) Do not think too small since each feature should be self-contained and it should provide a real business value.
   (3) Think in (coarse-grained) business terms and values, and not in (fine-grained) technical terms and functions.

The artifact which describes the Gherkin features according to the BDD is the main outcome of the requirements analysis. Since all further development steps, esp. the modeling of the domain, depend on this artifact it should be documented in an adequate and defined way.

(1) The Gherkin features provide a requirements specification which is an integral artifact of the software development process. Therefore, developers must have access to the features in their IDE (Integrated Development Environment) since they run through the features in order to carry out user acceptance tests of their software implementation. Nevertheless, features are not only relevant for the developers, but also for the stakeholders for which the software is developed. Therefore, an adequate way of documentation of the feature collection must be made available not only for the developers, but also for the stakeholders.

(1.1) At C&M, the feature are stored in the GitLab documentation repository. Stakeholders who have no access to the GitLab can be involved by generating PDF documents which are sent to these stakeholders so that they can make comments. This process is further described in the C&M Documentation Guidelines [CM-G-CMD], Section "Reviews (external)".

(2) The specification of features is a creative process which makes it impossible to state complete rules how to go through this process. Nevertheless, best practices can be formulated which give some valuable hints for the process.

(2.1) The first (one to three) features should specify the functionality of the MVP.

(2.2) A business value will usually not be provided by one single functionality but by a set of coherent functions by which a user can fulfill a specific task.

(2.3) The terms of the ubiquitous language should be used. These terms should be understood by the software developer and the business owner/analyst.

Another best practice is to use a declarative and not an imperative way to express the scenario steps.

[CM-G-CMD] Cooperation and Management: CMDocumentation, Git Respository. https://git.scc.kit.edu/cm-tm/cm-documentation
(1) ++How to Build the Right Software++
The proverb concerns the analysis of the requirements the software should fulfill. It means that the plan must always be adapted to reality – and not the other way round.

(2) ++Behavior-Driven Development (BDD) Process, Roles and Main Artifact++
- Three central roles: business analyst, developer, tester
- Scenario description: This specification which is structured according to a defined grammar and keywords is the core of BDD.
  -- Hint: Feature description including stories and scenarios would be the correct term.

(3) ++Systematic Analysis of Requirements++
- Vision: states what the project wants to achieve
- Business goal: describes what the business will get out of it
- Capability: expresses what users and stakeholders should be able to do to deliver these goals
- Feature: represents software functionality to support capabilities
- Story: makes clear which role wants to carry out what for which purpose
- Scenario: expresses a concrete example in a defined format by which a feature is illustrated

(4) ++Gherkin Feature and Including Scenarios++
A story consists of two parts
- Narrative description of a requirement structured according to "As a – I want – So that"
- A set of acceptance criteria presented as scenarios structured according to "Given – When – Then"

(5) ++Example: TLM Feature++
- Scenario: Create a Todo List
  Given I have the possibility to create a Todo List
  When I create a Todo List with the title "WASA course"
  Then the Todo List with the title "WASA course" is created

(6) ++Dealing With Gherkin Features++
- The developer and the stakeholder.
  This leads to the requirement that features should not only be documented in the IDE of the developer but also in external format (e.g. Word or Bitbucket Wiki).
Cucumber was created by Aslak Hellesoy [WH12]. The company behind the software is called Cucumber Ltd [Cuc-Web]. Cucumber is available as an open-source version and as a professional version (Cucumber pro).

(1) Cucumber allows to formulate requirements on a software as a textual specification and to automatically check the implementation if the specification is fulfilled [Wik-Cucumber].

(1.1) Originally, Cucumber was developed in Ruby to apply BDD to Ruby applications.

(2) Gherkin is based on a natural language and defines specific keywords which have specific meanings [Git-Ghe].

(2.1) It brings together business people and developers and serves as a ubiquitous language.

(2.2) But there are no details how the behavior is implemented.

(2.3) Examples of keywords are: Feature, Scenario, Given, When, Then, And Like Python and YAML (YAML Ain’t Markup Language, Yet Another Markup Language), Gherkin uses indentation to define structure. Line endings terminate statements.

(3) The template provided by Cucumber does only prescribe the structure of the scenarios.

(3.1) Usually, the description consists of only one short sentence.

(3.2) A user story is structured according to "As a [role] I want [feature] So that [benefit]".

(3.3) The structure "Given – When – Then" is also used in Cucumber. The concept of a step is additionally introduced. A step carries out the action defined after the "Then" keyword.

<table>
<thead>
<tr>
<th>BDD</th>
<th>Behavior-Driven Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>YAML</td>
<td>YAML Ain’t Markup Language, Yet Another Markup Language</td>
</tr>
</tbody>
</table>


BDD introduces specific concepts and terms to describe the behavior of a software system in a systematic and uniform way.

(1) Each feature is described in a separate file.
   (1.1) A feature describes a functionality of the software system that can be tested.
   Example: A user can log in to the system.
   (1.2) Features (and also scenarios and steps) cannot be interpreted by a system but serve the developer and customer to group the scenarios.

(2) A feature can contain several scenarios which are part of the feature file.
   (2.1) Both success and failure usage procedures are described.
   (2.2) "Given" lists the preconditions, "When" names the actions leading to the steps and "Then" specifies the outcomes that must be testable.

(3) Scenarios consist of steps. In Cucumber, steps are not part of the feature file.
The main building blocks of Cucumber are introduced and their interaction is described on a high level and by using an illustration.

(Gherkin Features with Scenarios and Steps) The features written in the Gherkin language are read in by Cucumber which is a command-line tool. The text in such ".feature" files must follow some basic rules prescribed by Gherkin (such as use of the specific keywords).

1) The step is the part in a Gherkin feature which can be executed. To do so, to each step a step definition must be added by which the (operational) semantics of this step is expressed.

1.1) (Step Definitions) The step definition bridges the gap between the informal, text-based description of how the software system should behave and the testing of this behavior.

1.2) There exist Cucumber versions which provide other programming languages (e.g. Java, JavaScript) to implement step definitions. In a mature test suite, the step definition itself will probably just be one or two lines of Ruby.

1.3) The step definition code delegates to a support code, specific to the domain of the application under test, that knows how to carry out common tasks on the system. For each step the developer responsible for the BDD tests must provide the support code.

1.4) An example is a web driver (e.g. Selenium, Protractor) which allows to automate the input via a web browser in order to test a web application.

2) If an error occurs in one step Cucumber marks the scenario as failed and moves on to the next scenario. If it gets to the end of the scenario without any of the steps raising an error, it marks the scenario as having passed.

3) The Gherkin features and step definitions to run the user acceptance tests are stored in the GitLab test repository “4Test_TodoListManagement” of the TodoListManagement (TLM) application [CM-G-4Test-Todo].

TLM TodoListManagement


A step is part of the Gherkin scenario expressed as plain text whereas a step definition is a piece of code that says to Cucumber, "If you see a step that looks like this..., then here is what I want you to do...." [RW+15].

(1) In the example, the feature including the step "When I create a Todo List with the title "WASA course" (written in Gherkin) might be stored in a file "todolist.feature" and the step definition "@When("I create .....");" in a Java source code called "Steps.java".

(2) In the example, the annotation and method stub to the "@When" is shown, which contains a regular expression followed by a Java method.

(3) The regular expression in the example is "^I create a todo list with the title \"([^\"]*)\"$". By the expression following the title starting with "\" is defined that any substring can appear as title (such as "WASA course").

(4) The reason for this flexibility is given by the wildcards that are contained in regular expressions. The flexibility can be utilized to make the Gherkin steps readable while the Java step definitions can be kept free of duplication.

(TodoListGenericRequest request = new TodoListGenericRequest(); request.setTitle(title);)

In the step definition the class "TodoListGenericRequest" and its method "setTitle()" provided by the TLM application are needed to create an new todo list.

This chapter illustrates how scenarios can be written in a way that they express what the software should provide to its stakeholders [WH12].

(1) Features are a specific kind of specification document and not computer programs. The reader should not need any technical background but only background from the domain to be able to understand the content.

(1.2) The real key to expressive scenarios is having a healthy vocabulary of the ubiquitous language to use to express the requirements. According to a guideline followed by C&M, the subjects and objects of the ubiquitous language are written in upper case letters so that the reader of the Gherkin feature realizes the relevant terms. (User, Todo, Todo List, Description, Content) These are terms from the ubiquitous language which is indicated by the upper case letters.

(2) Using only the basic set of Gherkin keywords can often make your features repetitive, making them cluttered and awkward to read. Therefore, some more keywords and related concepts are provided. The following list is only a subset of additional concepts provided by Gherkin.

(2.1) A background section allows to specify a set of steps that are common to every scenario in the file. (2.2) Data tables give you a way to extend a Gherkin step beyond a single line to include a larger piece of data.

Example:

<table>
<thead>
<tr>
<th>Given the Todo List exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Todo List</td>
</tr>
<tr>
<td>WASA course</td>
</tr>
<tr>
<td>travel preparation</td>
</tr>
<tr>
<td>shopping</td>
</tr>
</tbody>
</table>

(2.3) Subfolders allow to organize features according to some criteria, e.g. roles or high-level tasks a user might try to do.

Tags are a further dimension of organizing features by putting a word prefixed with the wildcard (@) character on the line before the "Scenario" keyword.
Avoidance of Incidental Scenarios

1. Details that are not necessary should be avoided.
2. In the example:
   1. Password information is irrelevant.
   2. Only the user sending the mail must be introduced.

It is important to formulate a scenario as focused as possible and to avoid all incidental (unnecessary) information which are of secondary concern.

1. Such detailed information leads to confusion (so-called noise) which makes the scenario hard to read. To be able to avoid noise it is important to know what the focus of the scenario is. Scenarios which try cover several aspects should be avoided.

   1. (2.1) (...) In the scenario on the upper side the password aspect covers a specific aspect of the sign-in functionality which is not in the focus of this scenario.

2. In the case of the scenario the noise was produced by the password information and the second user.

   2. (2.2) The user "Dave" is not needed since this role is taken by "I" which avoids to introduce incidental information into the scenario.
Duplication and the Principles DRY and DAMP

(1) Gherkin provides means to reduce duplication on a syntactical level
   (1) Duplication often is a sign of a low and too technical abstraction level
   (2) Only those duplications can remain that nontechnical team members accept

(2) DRY and DAMP principles
   (1) Don't Repeat Yourself (DRY) is an important, but not the most relevant principle with respect to the scenarios
   (2) Scenarios should contain Descriptive And Meaningful Phrases (DAMP)
   (3) DAMP trumps DRY

(3) The order of features and scenarios should support their readability
   (1) The first feature should describe what happens when the application is started for the first time

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAMP</td>
<td>Descriptive And Meaningful Phrases</td>
</tr>
<tr>
<td>DRY</td>
<td>Don't Repeat Yourself</td>
</tr>
</tbody>
</table>

Duplication is a dangerous property, both on the code and scenario level. Therefore this problem should be handled with care [WH12].

(1) The syntactical means (and related keywords) that Gherkin provides are the concepts of background sections and scenario outlines.
   (1.1) In this case, a pure syntactical treatment does not provide the solution.
   (1.2) Not minimal redundancy, but maximal expressiveness on the level of the stakeholder must be the must be the major goal.

(2) Both principles, DRY and DAMP, are concerned with redundancy and duplication.
   (2.1) When Don't Repeat Yourself (DRY) leads to scenarios that are hard to understand controlled redundancy should be accepted.
   (2.2) A sequence of scenarios should read like a book containing a logical and understandable story and not like a dictionary which has no story. This principle is called Descriptive And Meaningful Phrases (DAMP)
   (2.3) For the case of writing scenarios, readability is more important than duplication

(3) The logic of the developed software system should be expressed by the sequence in which features (functionality) and scenarios (test cases) are ordered.
   (3.1) This will provide the fundament on which all following features can be built.

DAMP Descriptive And Meaningful Phrases
DRY Don't Repeat Yourself
Best Practices Related to a Feature

1. Concentrate on features that cover the core functionality of the software system
2. Do not think too small since each feature should be self-contained and it should provide a real business value
3. Think in (coarse-grained) business terms and values, and not in (fine-grained) technical terms and functions
4. Use a declarative and not an imperative way to express the scenario steps

(1) The first (one to three) features should specify the functionality of a so-called Minimum Viable Product (MVP).

(2) A business value will usually not be provided by one single functionality but by a set of coherent functions by which a user can fulfill a specific task.

(3) The terms of the ubiquitous language should be used. These terms should be understood by the software developer and the business owner/analyst.

(4) Example:
Imperative: When I fill in "Username" with "dave" And I fill in "Password" with "secret"
Declarative: When I authenticate with valid credentials
(1) **BDD Tool Support**
- A domain-specific language (DSL)
- Describes the software's behavior
- Line-oriented language usually starting with a keyword

(2) **Feature, Scenario, Step**
- A scenario is part of a feature
- A scenario describes the acceptance criteria of the user story which is another part of the feature
- A scenario consists of steps.

(3) **Working of Cucumber**
- Implemented in a programming language (e.g. Ruby)
- Map the business-readable language into executable code
- Coupling with the application via a library of support code
- Use of an automation library

(4) **Avoidance of Incidental Scenarios**
   **Duplication and the Principles DRY and DAMP**
   **Best Practices Related to a Feature**
- As focused as possible and avoidance of all incidental information which are of secondary concern
- Maximal readability (DAMP trumps DRY)
- Declarative style instead of an imperative style

(5) **Duplication and the Principles DRY and DAMP**
- Descriptive And Meaningful Phrases (DAMP)
- Don't Repeat Yourself (DRY) is an important, but not the most relevant principle with respect to the scenarios (DAMP trumps DRY)
The motivation of Domain-Driven Design (DDD) is taken from Martin Fowler's foreword of Eric Evans's book "Domain-Driven Design – Tackling Complexity in the Heart of Software" [Ev04].

1) A general definition of a domain is: A sphere of knowledge, influence, or activity.

1.1) A model itself can be defined as a system of abstractions.

1.2) By this language domain experts and technologists are tight together.

2) There are many things that make software development complex. A deep understanding of the problem domain is in the heart of this complexity.

2.1) An effective domain modeler can both use the whiteboard during his talk with the customer and work together with a programmer on a Java program.

2.2) Only few people can do it well – and the competence to build good domain models is not easy to teach.

2.3) Domain models are not first modelled and then implemented. Instead, an experienced modeler gets splendid ideas from earlier releases of the domain model.

2.4) DDD teaches a lot of people how to apply modeling concepts to add structure and cohesion to the software development.

3) Based on this vocabulary the activity and the hard-to-learn skill of domain modelling can be explained.

DDD Domain-Driven Design

The figure shows a navigation map of the building blocks of Domain-Driven Design (DDD) [Ev04]. Each building block can be seen as a design pattern. Besides the pattern group of "Building Blocks" there exist further pattern groups related to "Supple Design", "Context", "Distillation", and "Large-Scale Structure".

1. (1.1) Most of the DDD patterns are not new, but a summary of existing modeling concepts such as object-oriented design (as e.g. described by Larman); responsibility-driven design (by Wirfs-Brock); contract by design (described by Meyer).

2. The navigation map shows the most relevant relations between the patterns starting from overall DDD pattern. The following explanations are mainly taken from the glossary.

(DOMAIN-DRIVEN DESIGN) A design in which some subset of software elements correspond closely to elements of a model. Also, a process of co-developing a model and an implementation that stay aligned with each other.

(LAYERED ARCHITECTURE) A technique for separating the concerns of a software system, isolating a domain layer, among other things.

(ENTITY) An object fundamentally defined not by its attributes, but by a thread of continuity and identity.

(SMART UI) A systematic approach for connecting the user interface (UI) to the application and domain layers (the most common pattern is the "Model View Controller", MVC). The Smart UI Pattern is an alternative, mutually exclusive fork in the road, incompatible with the approach of domain-driven design (since in a Smart UI the presentation layer (UI) and the business layer are explicitly not separated).

(SERVICE) An operation offered as an interface that stands alone in the model, with no encapsulated state.

(VALUE OBJECT) An object that describes some characteristics or attributes but carries no concept of identity.

(REPOSITORY) A mechanism for encapsulating storage, retrieval, and search behavior which emulates a set of objects.

(AGGREGATE) A cluster of associated objects that are treated as a unit for the purpose of data changes.

(FACTORY) A mechanism for encapsulating complex creation logic and abstracting the type of a created object for the sake of a client.

(Core) The following two patterns can be seen as the core of DDD:
"Layered Architecture" illustrates where in the software architecture the domain model is located.
"Entities" and "Value Objects" can be seen as the fundamental modeling elements of a domain model.

MVC Model View Controller
UI User Interface
To understand how domain modeling can be practically applied it is important to understand where it can be located in the overall software development process.

(1) The domain model is located in the design phase of a software development process. Since the domain model is used to clarify specific aspects of the domain with a domain expert (not with the user) it also contributes to the analysis phase.

(1.1) The domain model concentrates on the domain knowledge that is needed to provide the functionality of the software system. A precise and semi-formal description as a domain model is the main challenge of the design of the software system.

(1.2) Only one layer of the overall software architecture, the so-called domain logic layer is covered by the domain model. Since this layer can be seen as the heart of a software system it provides the kernel of the system. A major goal of domain modeling is to keep the model (of the domain) and implementation (of the domain logic layer) close together.
According to the overall BDD/DDD-based development process in the subsequent step the domain logic contained in features should be added to the domain model.

Note: In what follows the term feature is synonymous to a Gherkin feature.

(1) A feature specifies a specific part of the business logic from the viewpoint of a user who applies the functionality in a certain role ("As a ..." in the story) and in certain situations ("Given ..." in the scenario).

(1.1) The other part is the application logic. It is important to notice that in the DDD approach the business logic is divided into the domain logic and the application logic. The domain logic is application agnostic which means that this logic is relevant for other applications of this domain. Therefore, the domain logic is the re-usable part of the business logic appearing in a feature.

(2) Like the process of feature specification the derivation of domain knowledge from the features is a highly creative process. Nevertheless, some best practices do exist.

(2.1) By answering this question those terms of the feature should be identified that are relevant for the understanding of the user requirement.

(2.2) Only those aspects identified by Question 1 should be added to the domain model which are not specific to the regarded application.

(2.3) Three types of relationships can be distinguished: general associations, containment (composition, aggregation), inheritance
The derivation of domain knowledge is shown with a part of the TLM feature that was already introduced in the chapter ANALYSIS.

(1) The approach is to go through the lines of the feature and identify those terms which are presumably relevant for the understanding of the domain and the functionality related to the domain, i.e. the domain logic.

(1.1) This analysis artifact was introduced in the beginning of the course unit.

(2) The objects and subjects are written in upper case in order to draw the attention of the reader to the relevance for the understanding of the domain.

(3) Opposed to the objects and subjects, the activities are written in lower case.

(4) The relation view is specified as a UML class diagram as illustrated in what follows.
The domain model emerges from two types of modeling, the tactical modeling and the strategic modeling.

(1) The tactical modeling corresponds to the well-known proceeding of modeling knowledge as conceptual class diagrams added by dynamic diagrams to model certain time-related aspects. The result from the tactical modeling can be seen as the core content of the domain model.

(1.1) Four types of views are distinguished. Only the process view allows to model dynamic aspects, whereas the relation view, domain object view, and interaction view focus on the static aspects.

(1.2) The relation view corresponds to conceptual class diagrams as will be shown with an example.

(2) The strategic modeling is the real new part of domain modeling and the result, a so-called context map, provides the link to the microservice architecture.

(2.1) The structuring of the domain knowledge (as it is described by tactical modeling based on the four view types) is carried out in two steps: The whole domain is structured into subdomains and the subdomains are structured into so-called bounded contexts. To find an adequate structure coupling that domain knowledge which is coherently connected "from its nature" is one of the main challenges in domain modeling. The term "bounded context" is derived from the approach of a Domain-Driven Design (DDD, [Ev04]). A bounded context form the architectural perspective can be seen as a microservice candidate. This is the reason why a domain model serves as a blueprint for the microservice architecture which implements the model in the domain logic layer of the layered architecture. Domain objects can be shared by bounded contexts. These objects are called shared entities.

(2.2) DDD does not prescribe a concrete representation of the model. At C&M, the modeling elements from the Unified Modeling Language (UML) are used to model the results from the strategic (and also from the tactical) modeling.

(<<package>> SubDomain1) The subdomain is modeled as a standard UML package. "SubDomain1" stands for the name of the subdomain formulated in UpperCamelCase.

(<<bounded context MicroserviceCandidate1>> MicroserviceCandidate1, MicroserviceCandidate2) To be able to model a bounded context in UML a new stereotype <<bounded context>> is introduced. The UML modeling symbol of a package is used as graphical representation. "MicroserviceCandidate1" stands for the name of the bounded context formulated in UpperCamelCase.

(<<shared>> Entity, uses) This is the way how domain models shared between bounded contexts. In the example, the bounded context MicroserviceCandidate1 <<uses>> the domain object "SharedEntity" with a bounded context MicroserviceCandidate2 which is part of SubDomain2 and which <<uses>> the SharedEntity:

<table>
<thead>
<tr>
<th>Tactical modeling provides the objects and their static and dynamic relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeled by using different types of views</td>
</tr>
<tr>
<td>Relation view is the most relevant view type</td>
</tr>
<tr>
<td>Strategic modeling provides the overall structure of the domain model</td>
</tr>
<tr>
<td>Modeled by introducing subdomains and bounded contexts</td>
</tr>
<tr>
<td>The diagram is called a context map</td>
</tr>
</tbody>
</table>

DDD Domain-Driven Design
UML Unified Modeling Language

The context map is a diagram which can be seen as the starting point of domain modeling since it provides the domain structure in which the tactical domain models are placed in a systematic and consistent way. A context map does not focus the domain content (i.e. domain objects and its relationships), but a coherent structure of the business functionality.

1. The modeling elements that can be used in a context map are introduced in what follows.

1.1) By introducing subdomains the overview of a larger domain can be kept. Each subdomain should have a size which can be easily overlooked.

<<subdomain>> TodoListManagement This is the central subdomain of the TodoListManagement domain. A stereotype <<subdomain>> is introduced which uses the UML modeling element of a package as graphical representation.

1.2) A bounded context packages coherent functionality in a component. In the software architecture derived from the context map a bounded context is a candidate for a microservice.

<<bounded context>> TodoManagement This bounded context is part of the subdomain TodoListManagement. The stereotype <<bounded context>> is represented as a UML PackagingComponent. This allows to use UML compliant associations to model different types of relationships between bounded contexts as described on the next page.

1.3) The modeling elements of a foreign bounded contexts and an anti-corruption layer allow to model functionality from foreign domains.

<<foreign bounded context>> UserDirectory <<anti-corruption layer>> UserDirectoryACL The user directory is part of the Identity and Access Management (IAM) subdomain. Therefore, this functionality is modeled as a foreign bounded context which is integrated into the core functionality by using an anti-corruption layer (ACL) called UserDirectoryACL.

1.4) If there is an overlap of a part of the domain model in two or more bounded contexts it might make sense to model this overlap in a shared kernel.

<<shared kernel>> TodoInformation Since a shared kernel can be seen as a kind of directory and not as a running software component, it is modeled a a package. In the example, the shared kernel TodoInformation describes the domain logic with respect to the todos which are shared by two bounded contexts, TodoManagement and Notification.

ACL Anti-Corruption Layer
IAM Identity and Access Management
The context map is a design artifact of the TLM web application. It is stored in the GitLab documentation repository [CM-G-0Doc-Todo].

1. Not only the modeling elements (e.g. subdomain or bounded context) but also the relationships between these modeling elements are described by a context map.

1.1. As the modeling elements itself the relationships between them are specific to a context map. In the following each of the four specific relationships are described with the exemplary context map.

<<conformist>> This type of relationship is a directed association between two bounded contexts. The bounded context building the source of the directed association is called upstream (in this case, UserDirectory) and the destination (UserDirectoryACL) is called downstream. In a conformist relationship the downstream must conform to (i.e. strictly obey) any changes made by the upstream even if the model and the implementation from the upstream does not fit to the downstream.

<<customer/supplier>> A directed association between two bounded contexts in which the downstream is the supplier (in the example, TodoManagement) and the upstream is the customer (Calendar). The association means that the upstream team which develops the Calendar must coordinate planned changes with the downstream team which develops the TodoManagement.

<<partnership>> An undirected association between two bounded contexts. A partnership association means that the two teams responsible for the bounded contexts depend on each other and must, therefore, cooperate closely. In the example, this is the case for the bounded contexts TodoManagement and Notification although the shared kernel TodoInformation might lead to a looser coupling.

<<import>> This directed association is used between a bounded context and a shared kernel which needs this part of the domain model described by the shared kernel. In the example, the shared kernel is imported – and in this way, shared – by the two bounded contexts TodoManagement and Notification.

(Membership) This type of association allows to model the relationship of a membership. In the example, User Directory ACL is a member of (or is included in) TodoManagement.

(1) The context map is the main artifact of strategic modeling. It introduces the bounded contexts and their relationships. The tactical modeling starts from one bounded context by showing its content and inner structure.

(2) In DDD, entities and value objects are the fundamental domain objects (see also the page ++Domain-Driven Design++).

(2.1) The graphical notation is a rectangle which expresses the stereotype <<class>>.

(2.2) This notation is also used for all other modeling elements, such as the structuring element of a package that has been introduced before.

(User, TodoList, Todo) These are examples of domain objects from the TLM domain.

(3) The following types of relationships are standardized by UML which can be used in a class diagram.

(3.1) The association is the most general relationship between two domain objects DO1 and DO2. The association can be named (e.g. defines, is responsible for, creates). An arrow indicates the direction of the association.

(3.2) The aggregation is a stronger version of association and it is used to indicate that a domain object DO1 contains another domain object DO2 [MH06:86]. The lifetimes of the containing domain object (DO1) and the contained domain object (DO2) do not depend on each other. An aggregation is modeled by using an empty diamond arrowhead.

(3.3) The composition is as the aggregation a containment relationship, i.e. domain object DO1 contains another domain object DO2 [MH06:86]. In contrast to an aggregation, the lifetime of the containing domain object (DO1) depends on the lifetimes of the contained domain object (DO2). This means that the contained domain object (DO2) cannot exist without the containing domain object (DO1). A composition is modeled by using a filled diamond arrowhead.

(3.4) Generalization which is otherwise known as inheritance is used to describe that a domain object DO1 is a generalization of a domain object DO2. Therefore, the sub domain object DO2 inherits all properties from the super domain object DO1. The generalization relationship can be denoted as "is a" or "is a type of". It is modeled by an empty generalization arrow.

DO  Domain Object

The tactical modeling of a bounded context is shown with the example of the "bounded context" TodoManagement.

1. Domain objects are extended by stereotypes derived from DDD (Domain-Driven Design) concepts. Each domain object belongs to one or more such stereotype.

2. Entities are domain objects that can be clearly identified. They can contain attributes and methods.
   (1) If two Todo domain objects have the same status and contain the same information, they are the same Todo domain object.
   (2) Example of an attribute and a method of the Todo entity are content and changeContent().
   (3) If objects are shared between bounded contexts, the stereotype is annotated with "shared".

3. Value objects represent a descriptive aspect of the domain for which is only important what they are, but not who or which they are.
   (1) This lack of constraint gives design freedom and simplifies to optimize performance (e.g. by copying and not sharing a value object).

Remark: Further stereotypes like aggregates or domain functions do exist but they are not explained here.

IAM Identity and Access Management
EXERCISES: DESIGN (I)

1. What means DDD and what does it define?
2. What is the relationship between the domain model and (a) software architecture? (b) features?
3. How is a composition and an aggregation modeled and what is the difference of these two types of containment associations?
4. What does strategic modeling mean and how is it related to microservice architectures?
5. What is the difference between a conformist, a consumer-supplier, and a partnership relationship of two bounded contexts?
6. What is a relation view and how does it relate to a bounded context?

(1) ++Domain-Driven Design++
- Domain-Driven Design
- Patterns, such as Layered Architecture, Entity, Shared Object

(2) (a) ++Relationship between the Domain Model and other Development Artifacts++
The domain model provides the functionality of the domain logic layer.
(2) (b) ++From Features to the Domain Model++
- Each feature contain domain knowledge which should be formalized in a domain model according to DDD
- Each feature contains application-agnostic domain logic which should be separated from the application logic.

(3) ++Modeling the Domain Content++
- Composition: filled diamond arrowhead
- Aggregation: white diamond arrowhead
- Difference: In contrast to an aggregation, the lifetime of the containing domain object (DO1) depends on the lifetimes of the contained domain object (DO2)
- Example: TodoList is a composition of Todos -> Deletion of the TodoList leads to the deletion of all contained todos
  TodoList is an aggregation of Todos -> Deletion of the TodoList does NOT delete the contained Todos

(4) ++Tactical and Strategic Modeling++
- Strategic modeling provides the overall structure of the domain model by defining subdomains and bounded contexts.
- Relationship: Bounded contexts are the candidates for microservices.

(5) ++Example: TLM Context Map++
- <<conformist>>: This type of relationship is a directed association between two bounded contexts.. In a conformist relationship the downstream must conform to (i.e. strictly obey) any changes made by the upstream even if the model and the implementation from the upstream does not fit to the downstream.
- <<customer/supplier>> The association means that the supplier must coordinate planned changes with the custmer.
- <<partnership>> A partnership association means that the two teams responsible for the bounded contexts depend on each other and must, therefore, cooperate closely.

(6) ++Tactical Modeling of a Bounded Context++
- An extended class diagram containing entities, value objects and their relationships
- A relation view represents the tactical model of a bounded context.
Domain Modeling Guideline

(1) A domain modeling guideline helps a software development team to apply DDD in the design phase of the development process

(2) Specifics of the applied domain modeling guideline

(1) A subset of the DDD patterns are applied in the guideline

(2) UML is used to describe a domain model based on DDD

(3) The guideline includes

(1) Overall organization of the modeling artifacts

(1) Syntax and semantics of the used modeling elements

(2) A method describing the process of the modeling steps

(1) Structuring of the domain model into packages and bounded contexts (strategic modeling)

(2) Modeling of the bounded contexts by relation views (tactical modeling)

In this chapter a part of the guideline is introduced which shows how the DDD concept can be applied in software development. Starting from the software development process defined by the research group Cooperation & Management (C&M), concrete modeling elements and their usage in specific software projects are described.

(1) All members of the team should have a common understanding which DDD concepts should be applied which way. The result should be a domain model that supports the implementation of a high-quality software system.

(2) The domain modeling approach taken by C&M is only one possible way how to cope with the DDD concept.

(2.1) These patterns include the context map and the central building block patterns Entity and Value Object (introduced on the page ++Domain-Driven Design++).

(2.2) The Unified Modeling Language (UML) provides adequate means to adapt to the DDD patterns so that a formally precise domain model can be built.

(3) A main focus of C&M’s guideline is to prescribe the activities of domain modeling in a way that a team member produces modeling results that s/he can exchange with other members of the team.

(3.1) This is a more or less technical part of the guideline which describes the repository where the artifacts are stored and which tools are needed to edit the artifacts. At C&M, domain models are built by using PowerPoint and the resulting files are stored together with all other artifacts in the main folder of the respective C&M software product on the team server as described in detail in [CM-G-TOO].

(3.2.1) In order to find a common description of the DDD model concepts that are part of the approach, modeling elements defined by UML are introduced. This provides a common understanding of the modeling syntax and the semantics (which is additionally described by an informal text).

(3.2) The guideline provides a logical sequence of modeling steps that team members should carry out to come to build the domain model. For sure, this process can only be a recommendation since (domain) modeling is a highly creative process.

(3.2.1) In a first step the overall logical structure of the domain model of the software must be determined. Packages are used to model the different domains appearing in the domain model. According to the DDD, a domain is further structured into so-called bounded contexts.

(3.2.2) In a second step the bounded contexts are modeled using the building blocks introduced by DDD.

C&M Cooperation & Management

DDD Domain-Driven Design

UML Unified Modeling Language

(1) One problem with the concept of domain-driven design (DDD) is that it does not include modeling elements by which the domain model can be represented in a formally defined way.

(1.1) The Unified Modeling Language (UML) standardized by the Object Management Group can be seen as the leading and widely accepted modeling language used to model software systems. Enterprise Architect (EA) is a professional modeling tool that supports the complete spectrum of UML features.

(2) One relevant feature of UML that is used in the context of domain modeling language is its extensibility by defining so-called profiles. The definition of profiles is also supported by EA as shown in the figure which represents the profile of the context map.

The diagram shows the UML profile that was constructed to model a context map as part of the strategical modeling of domains.

(<<metaclass>> Package, PackagingComponent, Association) These metaclasses represent the UML base classes.

(direction, compositionKind) There exist two types of Association metaclasses, directed and undirected associations. This expressed by the attribute "direction". customer/supplier and conformist are directed associations whereas partnership is undirected. All of these are "normal" associations which means that they are neither composition nor aggregation relationships.

DDD Domian-Driven Design
EA Enterprise Architect
UML Unified Modeling Language

https://www.omg.org/spec/UML/2.5.1/
A central design artifact in the C&M development process is the domain model for which the Enterprise Architect is used as modeling tool.

(1) Enterprise Architect is one of the most popular UML (Unified Modeling Language) modeling tools with more than 200,000 licensed users in over 100 countries.

(2.1) Always the latest UML specifications (www.omg.org) are supported by EA.

(2.2) A wide range of software systems can be designed and constructed. For business purposes the most popular notation for business process modeling, the Business Process Model and Notation (BPMN), is supported by EA.

(2.3) EA provides model repositories that can be used by large teams to collaboratively work on models that are built in the design phase of software development.

(2.4) This property of EA is used to adapt the tool to the domain modeling concepts followed by C&M.

Note: The term domain has different semantics in the context of domain-specific modeling (or domain-specific language DSL) and domain modeling according to the approach described by Evans [Ev04].


BPMN  Business Process Model and Notation
DSL  Domain-Specific Language
EA  Enterprise Architect
UML  Unified Modeling Language

On this page the necessary steps are summarized in order to be able to use the UML tool Enterprise Architect (EA) for the strategic and tactical modeling of domain models. A detailed description of how to carry out these steps is given in [CM-G-TOO].

(1) (1.1) C&M has an academic license exit to use the EA software which is provided on a software distribution server.
(1.2) The installation can be easily carried out by executing an installation script. A step-by-step description can be found in [CM-G-TOO].
(1.3) All model information is stored in a repository for which an MySQL ODBC driver for access is required.

(2) The profiles are one part of the so-called Model-Driven Generation (MDG) technologies provided by the EA. Further examples of such technologies are metamodels and code transformation.

(3) The structure of the repository which is used by C&M is shown in the right-hand figure which is a screen dump taken from the EA user interface.

(1) UML_Profiles) In this folder the DDD profiles are stored.
(2) Research_Project_Work Each of the subfolders of this folder contains models that belong to an individual person or project. All further folders contain one specific domain model.
(CarSharing) The first of the alphabetically ordered domain models.

(TodoList) The domain model underlying the demo TodoList application.
(Domain Model, TodoList) Each domain model folder (i.e. root node) contains a package named "Domain Model". The first diagram appearing in this package is the context map which has the same name as the domain model, in this case "TodoList". All elements that follow are part of the context map diagram. According to the profile, subdomains are modeled as Packages and bounded contexts are modeled as PackagingComponents.

MDG Model-Driven Generation
ODBC Open Database Connectivity
SQL Structured Query Language

The process modeled on this page as a UML activity diagram describes the way changes are made in the domain models which are related to the context map. It is taken from [CM-G-TOO] where further information can be found on how to use the EA operations in order to carry out the process steps.

(Create or Choose the Project Folder) In most new projects, software applications are developed which use one of the existing domain models and add a new part to it. The initial model is developed in the “2. Research Project Work” and not directly in the domain model of the domain. The reason is that the models in the specific domains should be first approved before they are used by projects. New models that are not yet approved may confuse other project members because the domain knowledge could be wrong and may change rapidly. Therefore the models are only added to the appropriate domain model after their approval.

(Copy the Context Map of the Domain) To add a new bounded context to the own folder, it is necessary to copy the context map of the existing domain which should be extended or changed to an individual (personal or project) project folder. The bounded contexts that are relevant for the project are added to the copied context map.

(Add New Bounded Context to the Copied Context Map) The new bounded contexts (and subdomains, if needed) are added to the copied version of the context map. Therefore, the original domain model is not affected by the experimental changes.

(Integrate the New Bounded Context) The bounded contexts are integrated into the (copied) context map meaning the relationships between the bounded contexts are modeled. These relationships can be easily added to the original context map later on.

(Add Diagrams to the Bounded Context) It is important to use the C&M DDD profile for modelling.

(Create Merge Request(s), Discuss the New Diagram(s)) The newly modeled diagrams are discussed and reviewed. Since EA does not provide adequate means for discussion and review of the diagrams they are moved to GitLab where the well-known review process is initiated.

(Move the Bounded Context to the Concerning Domain) The created bounded contexts are moved (via drag and drop) from the working directory to the original domain.

(Add the Bounded Context to the Original Context Map) In a last step the context map needs to be updated. The new bounded contexts are added to the context map. This is simply done in the EA by dragging the bounded context from the project browser to the context map diagram.

(Consider to create a new baseline) A baseline is a snapshot of the current status of a model. If the model gets corrupted the model can be reverted to the baseline.

PNG Portable Network Graphics
The process described on this page determines the way changes are made in the domain models. It is taken from [CM-G-TOO] where further information on how to use the EA operations in order to carry out the steps appearing in the process.

Choose an Individual Project Folder) In most new projects, software applications are developed which use one of the existing domain models and add a new part to it. The initial model is developed in the "2.Research_Project_Work" and not directly in the domain model of the domain. This because the models in the specific domains are approved and are used by projects. New models that are not yet approved may confuse other project members because the domain knowledge could be wrong and may change rapidly. Therefore the models are not added until they are approved.

(Copy the Context Map of the Domain) To add a new bounded context to the own folder, it is necessary to copy the context map of the existing domain which should be enhanced to an individual (personal or project) project folder. The bounded contexts that are relevant for the project are added to the copied context map.

(Add new bounded context to the copied context map) The new bounded contexts (and subdomains, if needed) are added to the copied version of the context map. Therefore, the original domain model is not affected by the experimental changes.

(Integrate the new bounded context) The bounded contexts are integrated into the (copied) context map meaning the relationships between the bounded contexts are modeled. These relationships can be easily added to the original context map later on.

(Add Diagrams to the Bounded Context) It is important that the C&M DDD profile is used for modelling. There are separate diagram types, such as the relation view. The diagram type has to be chosen accordingly to what should be modeled.

(Discuss the New Diagram(s)) The newly modeled diagrams are discussed and reviewed. If the diagrams get approved, they are added to the original domain and its model.

(Move the Bounded Context to the Concerning Domain) The created bounded contexts are moved (via drag and drop) from the working directory to the original domain.

(Add the Bounded Context to the Original Context Map) At last, the context map needs to be updated. The new bounded contexts are added to the context map. This is simply done by dragging the bounded context from the project browser to the context map diagram.

(Consider to create a new baseline) A baseline is a snapshot of the current status of a model. If the model gets corrupted or changed in a way that is not supported, the model can be reverted to the baseline.
EXERCISES: DESIGN (II)

(1) What is the goal and content of a domain modeling guideline?
(2) Which UML concept is used to provide the model elements for domain modeling?
(3) An examples of a domain modeling element and its relation to the corresponding standard UML element should be given
(4) How is C&M’s domain modeling repository structured?
(5) Which steps must be carried out in order to update a domain model?

(1) ++Overview++
- Goal: To help a software development team to practically domain modeling in the design phase of the development process
- Content:
  (i) Organization of the modeling elements (including tools, repository, used elements and their syntax/semantics)
  (ii) Method describing the modeling steps
    -- Strategic modeling: Structuring of the domain model into packages and bounded contexts
    -- Tactical Modeling: Modeling of the domain parts by domain views

(2) ++Tool-Supported Domain Modeling++
- The UML concept is called profiles by which UML can be extended by the needed domain modeling elements.

(3) ++Tool-Supported Domain Modeling++
- domain modeling element: subdomain – UML element: package
- bounded context – packaging element
- customer/supplier – association

(4) ++Use of Enterprise Architect for Domain Modeling++
- Folders containing the profiles, individual domain model content and the common domain models
- One folder for each domain (such as Carsharing, CampusManagement, IdentityAndAccessManagement)

(5) ++EA- and GitLab-based Modeling Process Related to the Context Map++
- Copy the part of the domain model that should be changed or extended to the individual folder
- Make changes in this folder
- Discuss and agree upon these changes with the other team members
- Transfer the accepted changes to the common domain model
In the subsequent implementation phase the functionality as is was analyzed and designed is coded based on a microservice architecture and frameworks supporting the frontend and backend implementation. All implementation work is carried out in a test-driven manner based on unit tests and user acceptance tests which are defined by the scenarios being part of the Gherkin features. In the deployment phase the tested application is deployed in a Docker-based execution environment. As the microservice approach is strictly followed, the concept of Continuous Integration/Deployment/Delivery (CI/CD) can be supported.

(1) The BDD/DDD-based development process (Behavior-Driven Development / Domain-Driven Design) is an agile approach which means that implementation of the software should start as soon as possible.

(2) In the C&M development process the API specification is derived from the implementation of a bounded context. This part of the implementation concentrates on the domain logic being part of the domain logic layer of a microservice architecture.
(2.1) Unit tests check if the implementation of the domain model fulfills all domain-specific assertions included in the domain model.

(3.1) The API of a backend microservice is derived from the implementation carried out in the first implementation step. The BFF microservice composes the backend microservices and provides an API to the frontend.
(3.2) User acceptance tests are carried out by testing the scenarios occurring in the feature. In order to be able to test a feature the step definitions occurring in the features must be implemented.

API  Application Programming Interface
BDD  Behavior-Driven Development
BFF  Backend For Frontend
CD   Continuous Deployment
CI   Continuous Integration
DDD  Domain-Driven Design
In the first step of the C&M development process related to the implementation phase, each bounded context is implemented as a microservice. From this implementation the API specification is derived.

(1) DDD brings together the code and the model. Therefore, the concepts of the model equal the code definitions and vice versa. If two business functions are defined, each methods is explicitly modeled. In each implementation step, not the whole domain model, but only the relation view of one bounded context is implemented. This specific part of the domain model is simply called model.

(2) Entities and value objects are DDD patterns introduced on a previous page ++Domain-Driven Design++. While transferring the domain objects to code, naming conventions have to be considered.

(2.1) Therefore, the naming of attributes, methods and classes are always the same. When changes to code or model are applied, this principle needs to be handled with care in order to guarantee that both are in sync.

(3) A relationship is implemented according to its type. Depending on the relationship type, the implementation is carried out. For example, a composition is implemented by defining an object of the related entity (or value object).

(public class TodoList ...) The relation between "TodoList" to "Todo" is a composition.

(private List<Todo> todos;) A composition is simply implemented as a Java list. Therefore, when the Todo list is removed, each containing todo is deleted as well.
(1) The structure is achieved by step-wise implementation and separating the different concerns in corresponding folders. The TLM application and the implementation structure of the microservice stored in the GitLab repository "3MS_TodoListManagement" [CM-G-3MS] serves as the example to illustrate the structure.

(2) The folder structure refers to the hexagonal architecture. The reference frame is a microservice and therefore, the structure does not display the layered architecture. In addition, a BFF microservice also consists of these three parts. Therefore, the term ”domain” is misleading in the context of an BFF microservice. Each microservice consist of the following structure.

(2.1) The hexagonal architecture [Fi13] is a powerful architectural style to support the Domain-Driven Design, esp. its concept of bounded contexts [Ve13]. The term was coined by Alistair Cockburn [Co09]. The "hexagonal" symbol in this context means that an application has many sides which represent ports into and out of the application. The architectural style is also called "Ports and Adapters" [Br14].

Together with the port a kind of protocol is defined which takes the form of an application programming interface [Co09].

(2.2) The domain contains the domain logic which can be seen as the "heart" of the software. The application contains the controllers which define the API endpoints. The infrastructure folder contains the interfaces to the microservice calls and DTOs. The infrastructure manages the relational data mapping.

(3) The domain model is located in the model folder ("/tlm/todomanagement/domain/model"). In addition, the domain folder contains exceptions concerning the domain logic.

DTO Data Transfer Object
TLM TodoListManagement

[CM-G-3MS] Cooperation & Management: 3MS_TodoListManagement, Git Repository. https://git.scc.kit.edu/cm-tm/todolistmanagement/3ms_todolistmanagement
1 As with modeling, the focus is first on the core of the domain in order to prevent concerns from the application and the interface from merging into the domain. This domain kernel is implemented and tested with unit tests. The implementation of the domain requires several steps.

(Core) In order to implement interfaces, services, and tests, the domain logic needs to be implemented first. The domain logic is modeled by the relation views of each bounded context being part of the context map and of each bounded.

(EntityBase) The class EntityBase is a base class from a microservice library, which provides an ID and corresponding equals- and hashCode methods. The aim of this library is to achieve the boilerplate code of microservices on the one hand and a uniform configuration of HTTP responses, logging behavior and other aspects on the other.

(Infrastructural Annotation Enrichment) This step adds database annotations to the domain and is necessary to enable persistence mapping. To a database. This mapping can be done by Java Persistence API (JPA) or an ORM framework.

(@OneToMany)(@ManyToOne) These annotations are processed and define database cardinalities. @OneToMany for example defines, that each object related to one object is deleted when the one object is deleted.

(Domain Logic) In this step, the domain methods are implemented. Getter and setter methods are only implemented when they are part of the business logic. In most cases, getter and setters are not defined. Using Lombok only requires to add the annotations @Getter or @Setter.

(Domain Tests) Since the domain logic is implemented, it needs to be tested. Unit tests concentrate on the formal correctness of the domain at the technical level. In concrete terms, this means that the tests test the methods of the domain objects with valid and invalid parameters to ensure correct behavior.
The example taken from [Ir18] shows how the TodoList which is a central TLM domain object is systematically transferred into its implementation. Several JavaX annotations (e.g. @Entity, @Column, @OneToMany [Ora-JEE]) are used to solve the persistency issues of the implementation.

(<<shared entity>> TodoList) The TodoList is a <<shared entity>> which means that it is owned by one bounded context and used by at least one other bounded context. This domain object is defined by three (alphabetically ordered) attributes, color, nextTodoNumber, and title, and a number of methods (e.g. createTodo).

(1 @Entity 2 public class TodoList ...) This is the source code of the implemented domain object TodoList. Only the packet imports and the method bodies are left out.

(1 @Entity) This JavaX annotation is used for persistency purposes. The class is marked as an entity when it is written into the database.

(4 @Column ...) Specifies the mapped column for this property.

(11 @OneToMany ...) The class TodoList can be mapped to many Todos. The annotation @ManyToOne works the other way round.

(14 @OrderBy ...) The mapped objects are ordered by ascending positions.

(20 public TodoList (...) The constructor has no parameters. New created TodoLists are empty.

(26 createTodo(String content) ...) This is the first method provided by the class TodoList. In contrast to the model of the domain object the methods are not alphabetically, but logically ordered.

(1) A core part of the software development according to the Domain-Driven Design (DDD) is to make sure that the model and the implementation are the "same". A change of one of the two artifacts must lead to an equalizing change of the other artifact.

(1.1) The domain logic is only found in the domain logic layer. The other layers of the microservice architecture are separated from the domain logic as to promote reusability. The application logic ensures the access to the domain layer.

<<shared entity>> TodoList …) TodoList is a central domain object of the TLM domain model as described on previous pages.

(43 public Todo createTodo ...) The method createTodo takes the parameter content that is required to create a new Todo.

(46 throwExceptionIfDuplicate ...) Two identical Todos in a TodoList are not allowed. Hence the new Todo is checked against duplicates.

(47 Todo todo = new Todo...) A new Todo is created and added to the todos list. Then the newly created Todo is returned.

(1) ++Implementation Steps++
- Implementation of a bounded context as a microservice + Web API
- Implementation of BFF microservice and frontend

(2) ++Implementation of a Bounded Context++
- Model and code should always be the same
- The domain model and the source code which implements this domain model, must be compared after the implementations and possible differences must be adapted by changing one or both of the artifacts.

(3) ++Implementation of a Bounded Context++
- Relationship view of the bounded context which has to be implemented
- Model elements: entity, value object

(4) ++Implementation Structure++
- Hexagonal Architecture
- Domain, Application Infrastructure
- Starting point of the implementation is the domain

(5) ++Example: TLM Domain Model and Implementation++
- @OneToMany: The class TodoList is related to many Todos since a todo list cant contain any number of todos.
- @ManyToOne: A user can have many, i.e. any number of todo lists.
Definition of a Microservice API

1. A microservice API is also called web API
2. REST provides a description format
3. OpenAPI provides a standardized language
4. Two approaches to define the API of a microservice exist
   1. API-First Approach
      1. The API is derived from an intermediate resource model
      2. API is derived during the design phase
   2. Code-First (Domain-Implementation-First) Approach
      1. API is derived from the implementation of the domain model
      2. The elements of the API are directly specified in the code

(1) API is an acronym of Application Programming Interface. This term already exists before the microservice era. An API, in general, describes an interface that a software application provides in a way that it can be integrated into another software program. In the context of microservices the API is accessible via the web. Therefore, the term web API is often used instead of microservice API.

1.1) (1.2) The REpresentational State Transfer (REST) and OpenAPI are two important concepts used for the definition and description of web API that are used in today’s web applications. Both will be introduced on the succeeding pages.

(2.1) By following the API-First approach, the API is defined manually using an API editor (e.g. Swagger) during modeling. This is especially useful to get a grasp on how to map the domain to the API and to establish a contract between the teams, on which they can rely during implementation. Due to the limited insight of the domain before the actual implementation, several changes to the API will be necessary, but can be coordinated through updated API files. After the domain model is created, the API is specified and can be seen as contract between the frontend and backend teams.

(2.1.1) A systematic derivation process is presented in [Gi18].

(2.1.2) In the API-First approach, no implementation is carried out before the API is specified. Therefore, all actions of this approach take place in the design phase.

(2.2) The Code-First approach or Domain-Implementation-First approach requires that first the domain model must be implemented before the API is specified.

(2.2.1) Therefore, the API specification is part of the implementation phase. Nevertheless, the API specification itself is still a design artifact.

(2.2.2) The specification of the API is done via annotations by which the different parts of an API, such as requests, responses and status codes are defined and integrated into the code.

REST  REpresentational State Transfer

The code-first (or domain implementation-first) approach is used by C&M in ist BDD/DDD-based development process. A man reason for this decision is the fact that the code-first approach supports the model-code synchronization. This is not the case for the API-first approach.

(1) The concept of annotations was also used in the first implementation step for different purposes, especially to make the domain objects persistent. The annotation concept is illustrated with the example of a TLM REST operation which finds a todo list.

(Deriving Endpoints) Each endpoint results in a corresponding interface and controller classes for the API and handling the HTTP requests.

(RequestMapping, 1.) This annotation maps HTTP requests to handler methods of the REST controller.

(Extending the Domain Logic by API-defining Annotations) This is the central step of the code-first approach since in this step the OpenAPI specification is defined by extending the code by specific annotations.

(@Api, 2.) This annotation is required to declare an API resource.

(@ApiOperation, 6.). This annotation is required to declare an API operation. It specifies a short description of the endpoint (in the example: "Finds a specific todo list")

(@ApiResponse 7.) A wrapper to allow a list of multiple ApiResponse objects.

(@ApiResponse 8.) Describes a possible response of an operation.

(Implementing the Controller) The OpenAPI specification is implemented using the @Override annotation.

(Handling of the Resources) For the specification, the domain objects will be mapped onto so-called "resources" that act as data transfer objects (DTOs) These DTOs contain (partial) information of the respective domain objects. By using such an approach, an abstraction layer is introduced so that the domain objects can develop independently of each other without a necessary web API change.

(OpenAPI Specification of TLM) The API is located in the GitLab repository [CM-G-3MS] in the backend development folders "tlm/application/controllers/api/".

DTO Data Transfer Object

[CM-G-3MS] Cooperation & Management: 3MS_TodoListManagement, Git Repository. https://git.scc.kit.edu/cm-tm/todolistmanagement/3ms_todolistmanagement
The origin of the architectural principle called the REpresentational State Transfer (REST) is the PhD thesis [Fi00] from Roy Fielding.

(1) The core of REST is to provide a concept how a microservice API can be structured. REST replaces the Web Service Description Language (WSDL) which was used to specify web services. In contrast to WSDL, REST is a lightweight approach and, therefore, much easier to apply for software developers.

(1.1) Resources are the main concept and the fundamental abstraction of information in REST. Each resource is addressed by a Uniform Resource Locator (URL).

(1.2) Only a small set of well-defined method to manipulate the resources is made available.

(1.3) Stateless means that the server does not hold any session information. Therefore the client must transfer all information in the request that the server needs to process this request which leads to a loss of performance. Advantage of stateless communication is an increase of scalability and robustness.

(1.4) The representation principle is strongly related to the resources. Each resource is represented by different formats according to the platform on which the resource is stored. Examples of formats and corresponding platforms are: HTML-Browser; JSON – JavaScript application; XML – Java application. Different formats used in Internet protocols are defined by the Multipurpose Internet Mail Extensions (MIME) standard.

(1.5) The state transitions are driven by the format of the resources. This principle is related to hypermedia and the so-called "Hypermedia as the engine of application state" (HATEOAS). In this context the concept of hypermedia (especially the hypermedia links) is used to control the state transitions of an application.

(2) Although REST is defined independently from HTTP it is usually based on this Internet application protocol.

(2.1) (2.2) HTTP operations (GET, POST, HEAD, PUT, DELETE, ...) and the URL which identifies the resource are the main parts of a REST API.

(POST /todo-lists ...) This is an example of a REST API of the backend microservice being part of the TLM application.

HATEOAS Hypermedia As The Engine Of Application State
HTTP HyperText Transfer Protocol
JSON JavaScript Object Notation
MIME Multipurpose Internet Mail Extensions
URL Uniform Resource Locator
WSDL Web Service Description Language

OpenAPI is one of the most broadly accepted API description formats.

1. The OpenAPI Specification is a manufacturer-independent defacto standard. Examples of other description formats are API Blueprint or RESTful API Modeling Language (RAML).

1.1. The OpenAPI Initiative was founded in 2015 as an open source project under the Linux Foundation [OAI-FAQ]. A major driver was SmartBear which donated the Swagger Specification to the OpenAPI Initiative.

1.2. Examples of such companies are Google, Microsoft, IBM, Apigee.

2. JSON and YAML are both markup language which are based on the same concept (lists and scalars). Each JSON document is a valid YAML document. The first line of the document (i.e. swagger: '2.0') defines that this is an OpenAPI 2.0 specification.

2.1. In the "info" JSON/YAML object, metadata of the OpenAPI specification can be found, such as the "title" or the "contact" person of the service. All information is described by a cascaded list.

2.2. In "paths" the (relative) URLs of the endpoints and the HTTP methods are indicated. In "responses" the results including the HTTP status codes are described.

2.3. Reusable parts are objects that consist of a set of properties. To be able to reuse the object a name is defined which is use as a reference.

<table>
<thead>
<tr>
<th>JSON</th>
<th>JavaScript Object Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAML</td>
<td>RESTful API Modeling Language</td>
</tr>
<tr>
<td>YAML</td>
<td>YAML Ain’t Markup Language</td>
</tr>
</tbody>
</table>

The YAML document of the TLM OpenAPI specification is available on the Bitbucket repository [CM-B-swa].

(2. info) The “info” part contains general information (such as title, description, contact) that describe the TLM API.

(1. (5. host ...)) This is a central part of the API path [SWA-API] which is built as follows:
<operation> (e.g. GET) <scheme> (HTTP) <host> (todo-management.cm.tm.kit.edu) <path> (todo-lists)

(2. (7. paths:) An endpoint corresponds to a resource that the API exposes. An examples of endpoint or resource of the TLM API is "/todo-lists" (line 8.).

(3. (9. get:) (22. post:) As can be seen in the example, a single path can support multiple operations. In this case, GET /todo-lists and POST /todo-lists.

(4. (10. tags:) (23. tags:) For example. SwaggerUI (see also page ++Example: TLM API++) uses "tags" to group the displayed operations. Often the endpoint names are used as tags which is also done in this example.

(5. (13. operationId: getTodoListUsingGET) (26. operationId: createtodoListUsingPOST) These are the names of the corresponding methods in the backend code. Usually, the (optional) operationIds are automatically generated, i.e. when the OpenAPI specification is generated from the code).

(6. (27 parameters:) If an operation (such as POST or PUT) has parameters in its request these are described as body parameters ("- in: body", line 28). In the example, the parameter "newTodoList" is specified as body parameter of the operation POST /todo-lists by which a new todo list is created.

(32. schema:) (33. Sref: '#/definitions/TodoList-GenericRequest') The schema describes the ... The "Sref" indicates that for modularity purposes this description is part of the "definitions" section of the OpenAPI specification.

(7. (14. responses) (34. responses) In this part of the "paths" object the possible results of the get operation are described. In the example a successful (200, line 15) is defined.

(Sref ...) This is a reference to the "definitions" part of the OpenAPI specification where the different data structure related to the domain objects (e.g. Todo or TodoList) are defined.

[Swa-API] Swagger: API Host and Base URL. https://swagger.io/docs/specification/2-0/api-host-and-base-path/
For the specification of the REST-based TLM API the tool Swagger was used [CM-S-TLM]. The page shows screen dumps taken from the tool SwaggerUI.

(GET /todo-lists) This HTTP GET operation of the TLM API returns all todo lists and all todos for each todo list.

The complete request URL is: 

https://todo-management.cm.tm.kit.edu/todo-lists

When clicking on a list element representing an operation, details of this operation is presented. The screen dump on the right hand side shows the details of the TLM API operation "GET /todo-lists".

(Parameters) Since is TLM API operation is based on a HTTP GET it has no input parameters.

(Responses) The response of a successful request (i.e. "Code 200" meaning "OK") is listed in the text field underneath "Example Value".

("id", "title", "color", "todos") These are the elements which define a todo list.

("number", "position", "content", "done", "description") A todo is defined by these five elements.

(POST /todo-lists/ ...) (GET /todo-lists/{id} ...) In what follows all REST-based operations of the TLM API are listed. For each operation a short text describes what is performed by the operation. For example, by sending a request "POST/todo-lists" a new todo list is created and a request "GET /todo-lists/{id}" finds the todo list with the specific "id" (or provides an error response if a todo list with the "id" does not exist).

This page gives an overview of the modular architecture the Angular framework is based on.

1. An Angular module is a class decorated with NgModule. Decorators (such as @NgModule) are functions that modify JavaScript classes.

1.1 Each Angular App has at least one such Angular module class, the root module (conventionally called AppModule). The Angular module system is complementary to the JavaScript module system in which each file is a module and all objects defined in the file belong to that JavaScript module. The two types of module systems must be well separated which is not that easy since both use the same vocabulary of "imports" and "exports".

(Library Module) A library module is a collection of JavaScript modules. Angular ships with several such library modules, called Angular libraries each beginning with the @angular prefix. Examples are: @angular/core, @angular/platform-browser.

(Metadata) Describe how a class needs to be processed. They are inserted into classes with decorators.

2. A component controls parts of the screen called view and each defined by a template. The component supports the view by providing presentation logic which is implemented as a class.

3. A template is a form of HTML that tells Angular how to render the component.

3.1 Directives give instructions on how templates should be rendered. There are three types of directives in Angular. First, there are components that are directives with templates. Second, there are structural directives, such as ngIf and ngFor, that manipulate the DOM layout by adding and removing DOM elements. There are also attribute directives, such as ngStyle, that change the appearance or behavior of elements, components or directives.

(Property Binding, Event Binding) The mechanism for coordinating parts of a template and parts of a component is supported by the data binding concept of Angular. Data binding is conducted by adding markup bindings (specific forms) to the HTML template to tell Angular how to connect the component with the DOM.

4. A service can be nearly everything, such as a logging service or data service. Or it can be a tax calculator or an application configuration.

There is nothing specifically Angular about services. Angular has no definition of a service. There is no service base class, and no place to register a service.

4.1 (Injector) Angular uses dependency injection to provide new components with the services they need. When Angular creates a component, it first asks an injector for the services that the component requires.

The main structure of the Angular architecture is demonstrated with the example of the part of the TLM frontend which presents the data and functions related to a todo list.

(TodoList) This component handles everything a todo list offers.

(HTML 5 & Bootstrap) The HTML defines which elements are shown in the browser. It includes Angular syntax like "*Ngfor" which is used for addressing each list element. The todolist.component.html uses the Bootstrap element "card".

(CSS) Elements can use custom style sheets. These are defines in the "*.css" classes.

(TodoListComponent) The file todolist.component.ts handles the actions from the HTML elements. For example, if a button on the web page is pressed, it is forwarded to this file. If an action requires a request, this is forwarded to the service.

(TodoService) The TodoService is the interface to the BFF. It forwards every request.
(1) ++Definition of a Microservice API++
- API-First approach and Domain-Implementation-First (or Code-First) approach
- Main differences:
  -- Sequence of API specification and implementation (API-First: specification before implementation; Code-First: specification after implementation)
  -- Assignment to a development phase (API-First: design phase; Code-First: implementation phase)

(2) ++API Description Format OpenAPI++
- OpenAPI is a specification language for APIs defined based on the REST approach

(3) ++Example: TLM OpenAPI Specification++
- REST property resources: /paths/todo-list
- REST property service interface: get: responses: ; post parameters: responses:

(4) ++Architecture of an Angular Application++
- Component: controls parts of the screen called view which are defined by a template. The component supports the view by providing application/presentation logic which is implemented as a class.
- Template: is a form of HTML that tells Angular how to render the component.
- Relation: The mechanism for coordinating parts of a template and parts of a component is supported by the data binding concept of Angular. Data binding is conducted by adding markup bindings (specific forms) to the HTML template to tell Angular how to connect the component with the DOM (https://angular.io/guide/architecture#data-binding).
The microservice system architecture consists of a frontend, backend-for-frontend (BFF) and a backend. Angular is used as implementation technology for the frontend and Spring as implementation technology for the BFF and the backend microservices.

(1) A microservice architecture divides the software system into three distributed subsystem (or system parts) frontend, BFF, and backend. These systems communicate via resource-oriented web APIs.

(Diagram) The Unified Modeling Language (UML) provides the deployment diagram to model this physical view on the software application [MH06] as shown on the right hand side of the slide. The three-dimensional boxes represent the UML modeling element of a so-called node (which describes a computer system). On a node contain so-called artifacts are located which describe software programs running on that node.

(2) Angular is a popular open-source framework to implement the frontend based on HTML (HyperText Markup Language) and Typescript which is an extension of JavaScript.

(.angular-cli.json) A JSON file (JavaScript Object Notation) which contains configuration information of the whole project setup.

(app.module.ts) A Transcript file which defines the modules that are bootstrapped by the framework at its start.

(3) Spring is one of the most popular open-source development framework for Java-based application servers. Many available open JavaEE technologies, such as Java Persistence API (JPA) are used and extended by Spring.

(4) The three system parts of the microservice architecture (usually) run on distributed client and server systems which communicate via the Web. The way how specification of the web APIs are specified is described in a previous chapter.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>JPA</td>
<td>Java Persistence API</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>

Virtualization allows to divide a physical system into separate virtual parts (machines, containers) each of which can run different things. This concept is especially interesting for microservice architectures where a microservice is one of these things.

(1) The virtualization based on virtual machines (VM) is also called "standard" virtualization.
(1.1) The use of VM-based virtualization technology among Unix, Linux and Unix-like operating systems started around 2005.
(1.2) If the hypervisor directly runs on the hardware, it is called type 1 virtualization. In the case it runs on top of another operation system, it is a type 2 virtualization. Examples of type 2 virtualization technologies are VMWare, AWS, and Xen. Main resources managed by the hypervisor are CPU and memory.
(1.3) Each VM can be considered almost hermetically sealed machines, kept isolated from the underlying physical host and the other virtual machines by the hypervisor.

(2) The concept of a container-based virtualization stems from Linux, more concrete from LXC (LinuX Container) which is available in any modern Linux kernel.
(2.1) A container is a subtree of the overall system process tree.
(2.2) The allocation of resource to a container is done by the kernel.
(2.3) The missing hypervisor and the lightweight design lead to a much faster provisioning of a container (a few seconds) compared with a virtual machine (many minutes). Resources can be assigned to containers in a more targeted way.
(2.4) The kernel is where the process tree lives and must therefore be shared.

AWS Amazon Web Service
LXC LinuX Container
VM Virtual Machine

Docker is a software product developed in Go by Docker Inc. which published the first version in 2013.

(1) Docker extends the existing container technology from Linux in a way that it provides a complete and easy-to-use solution for the creation and the operation of containers.

(2) Container virtualization offer applications the isolated usage of resources (CPU, RAM, network) without use of heavyweight virtual machines. Each container runs as an isolated process on the host operating system. This offers important advantages compared to virtual machine virtualizations, such as lightweight resource consumption (esp. CPU, storage), shorter start time, easier distribution.

(3) A container is a runnable instance of an image while an image is a read-only template with instructions for creating a Docker container. The standardized description leads to a portability of containers. In addition, deployment becomes easier and more stable.

(4) The instructions are formulated in a simple syntax. An example of a command that can be used in a Dockerfile is "docker run".

"$ docker run -i -t ubuntu /bin/bash" (i) creates a new container containing an Ubuntu image, (ii) allocates a read-write filesystem as its final layer, (iii) creates a network interface including the assignment of an IP address to the container, (iv) starts the container and executes the /bin/bash command.

(4.1) One important property of layers is that they are shared between all installed images. In the example, the (operating system) layer "Ubuntu" is shared by all three applications A, B and C, while the (programming language) layer Java is shared by the applications A and B.

This page illustrates an example of a microservice environment as it is established at C&M.

(Humane Service Registry) This registry is used during the development to guarantee a consistent design of the architecture. The humane service registry visualizes the service landscape by offering a user-defined list which is stored in the C&M Teamserver.

(Version Control) The source code of the services is organized with the version control system Git and stored at GitLab.

(CI/CD Pipeline) The CI/CD pipeline is a customized pipeline executed whenever code is pushed to the repository. Hereby a step initiates the next one, when its run was successful. Each step runs in its own Linux Docker container as defined by the gitlab.ci.yml which is stored in the version control system.

(Runner Server, PaaS Server) The pipeline includes two servers: (i) the runner server executing the build, and (ii) the PaaS (Platform as a Service) server which is responsible for storing and deploying the images. A GitLab runner executes the tests and builds a runnable artifact. For example, in the case of a Java application, a JAR is the resulting artifact.

(Tested Code) The test results are produced by running Maven Verify [Mav-Int]. It includes to validate whether all information, like dependencies, are available, compile the source code and to test the code. The project information and dependencies which are needed are given by the pom.xml.

(Runnable Artifact) Furthermore Maven Verify packages the compiled code to an executable. In our case it is an lightweight jar file. This jar is then stored as a pipeline artifact to be used in the next step.

(Docker Image, Running Container) Next, the runnable artifact is copied into a bare image, the result is the so called Docker Image. This Docker image is pushed to the registry on the PaaS server. The registered Docker image is deployed in a running container which can be monitored with Portainer.

CD Continuous Deployment (or Delivery)
CI Continuous Integration
PaaS Platform as a Service

After the software has been implemented it must be built and deployed which requires another specific set of tools.

(1) DevOps contains the two terms development and operations. Since security is an important issue, DevOps is extended to DevSecOps. It summarizes all practices that bring together software development and software operations in a way that the development cycles are shortened and the deployment frequency is increased. Microservice architectures support the DevOps concept since microservices have the property that they can be tested and deployed independently from the whole software system. This is not the case for monolithic systems.

(1.1) Continuous integration (CI) is a DevOps practice by which the code from the involved developers are built and tested in short time intervals (e.g. several times a day) in order to discover and solve integration problems as soon as possible. Continuous delivery (CD) extends the CI practice by releasing the software after it was built and tested (i.e. continuously integrated).

Continuous deployment is also abbreviated by CD which is often confused with continuous delivery, means that every change is automatically deployed to production. Since the software must be released before it can be deployed, continuous deployment is an extension of the DevOps practice of continuous delivery.

(Maven) A Java tool by which a project’s build, reporting and documentation can be managed.
(Gradle) A Java-based build management tool (comparable to Apache Maven) which uses a domain-specific language
(Cucumber) A library by which user acceptance tests based on Gherkin features can be carried out.
(JUnit5) A framework to test Java programs based on unit tests.
(Nexus) A repository for build artifacts produced by Docker, Maven, npm, and others.
(Docker) A software system which supports container virtualization in order to deploy applications in isolated containers.
(Portainer) An open-source management user interface to manage a Docker host or swarm cluster.
(Kubernetes) An open source solution for the container management which is describes in further detail on succeeding pages.

DevOps Development and Operations
DevSecOps Development and Security and Operations

Continuous Deployment and Build Pipelines

1. Continuous Deployment (CD) treats each and every check-in as a release candidate
   1. The software has to go through multiple stages from check-in to production
   2. The multiple stages inside a build are modeled as build pipeline

2. A CD tool provides specific functionality to support the build pipeline
   1. Define and visualize the pipeline
   2. Support manual actions when stage transitions are not fully automated

3. One pipeline per microservice should be the goal
   1. An exception might be acceptable in the initial project phase

---

1 Continuous Deployment (CD) gives constant feedback on the production readiness of each and every check-in [Ne15].
1.1 The reason for the stages is that there is a sequence of the needed activities. An example is to conduct fast tests before slow tests (more general: to conduct different types of tests in a specific sequence).
1.2 The figure on the right side shows the different stages of a build pipeline on an abstract level.

2 The complexity of such a CD tool is often underestimated. It does not make sense to try to hack and extend Continuous Integration (CI) tools to make them do CD.
2.1 The tool supports the developer to keep a good overview of the process and the current status.
2.2 In the case of a manually executed "User Acceptance Test" (UAT) the tool supports this process by showing the next available build ready to be deployed in the UAT environment, deploy it and show the test result in the visualized build pipeline.

3 The service is an artifact (coming with different, platform-specific shapes) that is created and moved through the pipeline.
3.1 In the initial phase, the service boundaries are often not clear and changes across service boundaries are more likely. As soon as a service API stabilizes it should be moved into an own build.

UAT User Acceptance Test

GitLab CI/CD Pipeline

(1) CI/CD pipeline
   (1) Configured by .gitlab-ci.yml in each repository
   (2) Multiple jobs per pipeline
   (3) Divided into stages

(2) GitLab Runner
   (1) Runs the jobs
      (1) Multiple executors: SSH, Shell, Docker, ...
      (2) Is assigned to a group or a repository

(1) GitLab provides a CI/CD pipeline for every repository.
(1.1) Every repository that wants to use the CI/CD pipeline needs to configure the pipeline with the .gitlab-ci.yml file. This file is stored in the root directory of the repository and it contains the definitions of the jobs and the order of their execution. A complete configuration reference can be found in the GitLab documentation [GL-CIC].
(1.2) (1.3) Each pipeline can contain multiple jobs which can be divided into several stages. Jobs of the same stage can be executed in parallel. Jobs of the next stage can only be executed when all jobs of the previous stage have been finished successfully.

(2) The code of the GitLab Runner is open source and is available in the GitLab Runner repository on Gitlab.com [GL-git]. It is written in Go and is designed to run on GNU/Linux, macOS and Windows operating systems.
(2.1) To execute the pipeline and its jobs, a GitLab Runner is needed. The GitLab Runner can be run on a server or a desktop computer.
(2.1.1) The GitLab Runner supports multiple ways to execute the jobs in so-called executors:
(SSH) The SSH executor can be used to connect to a remote machine and execute the commands of the pipeline job on this machine over SSH.
(Shell) The Shell executor is used to execute the commands of pipeline jobs locally. Depending on the operating system that is used, different shells are used.
(Docker) GitLab Runner can execute jobs on Docker images which are specified in the .gitlab-ci.yml by using the Docker executor. To run Docker, commands inside the Docker container needs to be in privileged mode. Further information can be found in the GitLab documentation [GL-Use]
(... More executors for VirtualBox, Kubernetes and more are also available.
(2.2) A GitLab Runner can be assigned to a group or a single repository. If a GitLab Runner is assigned to a group, all child groups and repositories of that group can use it.

(Pipeline) The image shows the currently used C&M CI/CD pipeline with three stages: Build, Package, Deploy. Each stage contains one job and the execution of the jobs was finished successfully, as shown by the green checkmarks.

The screen dump on the right shows an excerpt of the .gitlab-ci.yml that is used in the microservice repository of the ElectricCarCharger project.

(1) The .gitlab-ci.yml file contains the configuration of the GitLab CI/CD pipeline for a GitLab repository. The main elements of the pipeline configuration are the jobs. A job has no predefined name or prefix, it can have any name possible. The only limitations are already existing keywords like image, services, stages, and a couple more. The full list can be found in the GitLab documentation [GL-CIC].

(1.1) (25. script) Every job requires the script keyword which defines the commands that are executed in this job. These commands are executed by the GitLab Runner that runs the script in a shell.

(1.2) (21. cache) The files that are used by a job can be cached. This way the files do not need to be downloaded or created for every run. As an example, the build-app job (15.) caches the Maven repository (24.) so it does not have to be created in every run.

(1.3) (18. artifacts) With the artifacts keyword the artifacts that were created by running the job can be copied and attached to the job on GitLab after the job finished successfully. The artifacts of a job can also be used by jobs which are part of a following stage.

(2) Jobs can be assigned to stages. Jobs of the same stage can be executed in parallel. Jobs of the next stage can only be executed if all jobs of the previous stage have been finished successfully.

(10. stages) The stages are defined outside the jobs by using the stages keyword, followed by the stages that should be used. The stages are ordered by their order in the pipeline config.

(3) (3. variables) With the variables keyword it is possible to define environment variables that are available in all jobs of the pipeline.

(4) (1. image) The GitLab CI/CD pipeline allows the use of Docker images. The default Docker image for the pipeline can be set with the image keyword on the same layer in the pipeline config as the stages and jobs. Inside each job the image keyword can be used to assign another Docker image to a job.

(1) ++Microservice System Architecture++
- Nodes: Frontend, BFF, Backend system
- Interfaces: Web APIs, e.g. specified by using OpenAPI

(2) ++Virtualization of an IT Infrastructure++
- Differences
  (i) Granularity: Container are much more fine-grained and lightweight compared to VMs
  (ii) Performance: A container can be started and stopped much quicker than a VM
  (iii) Architecture: Container share a common kernel and need no hypervisor
  (iv) Security: VMs run in a more close environment and their protection is easier
- Microservices
  -- Microservices usually run on virtual nodes
  -- A container is the favorite virtualization unit for a microservice (especially due to the horizontal scaling aspect)

(3) ++Docker++
- A Docker container runs as an isolated process on the host operating system and provides a virtualized environment to run applications.
- A Docker container is a runnable instance of an image.
- Created by instructions in a Docker file

(4) ++DevOps, DevSecOps and CI/CD++
- DevOps uses the CI/CD approach to couple development (Dev) and operation (Ops) in a continuous way

(5) ++DevOps, DevSecOps and CI/CD++
- Continuous Deployment extends Continuous Delivery by automatically deploy a delivered release onto a (usually virtualized) infrastructure

(6) ++GitLab CI/CD Pipeline++
- The artifact is a YAML file named gitlab-ci.yml
- The component is called GitLab Runner which supports multiple ways to execute the jobs (e.g by SSH, Shell, Docker)
A container technology, such as Docker, provides the technology to establish a containerized infrastructure. But it does not provide the functionality that is needed to manage such an infrastructure.

1. The automated deployment of a microservice is a central aspect of the DevOps (Development and Operations) concept which makes sure that a change of the software system is continuously integrated and deployed to the infrastructure. In the case of a microservice architecture deployed on a container-virtualized infrastructure this concerns the container on which the microservice is running.

2. Kubernetes supports the Container Runtime Interface (CRI) which is not only provided by the container technology Docker, but also by other technologies such as XEN and Rocket.

CNCF Cloud Native Computing Foundation
CRI Container Runtime Interface
DevOps Development and Operations

[1] The operation of a containerized infrastructure used to run microservices needs complex management functionality

1. Automated deployment of a microservice
2. Scaling of the microservices
3. Operation of microservice containers across machines and clusters

2. Kubernetes is a leading container management technology

1. Open source software originally developed by Google
2. Client-server architecture running on Linux machines
3. Based on a container technology, esp. Docker

2. Alternative solutions for the management of container-virtualized infrastructures include Rancher, Apache Mesos, or Docker Swarm. Kubernetes [Lin-Kub] today is one of the most important and powerful technologies in this field.

2.1) Kubernetes is hosted by the Cloud Native Computing Foundation (CNCF) which serves as the vendor-neutral home for many of the fastest-growing open source projects.

2.2) In the case of Kubernetes, the client is a so-called node and the server is a so-called master as further illustrated on a succeeding page **Kubernetes Architecture**. Each node runs on a separate Linux machine.

2.3) Kubernetes supports the Container Runtime Interface (CRI) which is not only provided by the container technology Docker, but also by other technologies such as XEN and Rocket.

Kubernetes (short name K8s, Greek for governor) was first announced by Google in 2014 and its design is heavily influenced by Google's Borg system. While Borg is entirely written in C++, the rewritten Kubernetes is implemented in Go. The main goal of Kubernetes is to automate the deployment, scaling, and management of applications that run in a container-virtualized infrastructure. One of the most relevant supported container tools is Docker. Since many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) it is a key technology in the field of cloud computing.

Google describes Kubernetes as a platform for the orchestration of containers. Orchestration in this context means that all resources are made available (in a highly elastic and efficient way) in order to provide the services.

(kubectl) Administration interface (kubernetes control) that serves for the communication between client and master.

(master) The central management unit of a Kubernetes cluster consisting of the four components (i) API server, (ii) etcd, (iii) controller manager, and (iv) scheduler.

(kube-apiserver) Provides the whole Kubernetes API.

(etcd) Stores all metadata and configuration data of Kubernetes in a key-value store (i.e. configuration data "/etc" distributed).

(Scheduler) Schedules the pods by placing them in a queue from where they are assigned to a node. Opposed to schedulers in operating systems, it is not responsible for executing the pods.

(node, Docker) A Kubernetes node, also called minion, is a single server for containers. The container technology used by Kubernetes, such as Docker, must be installed on each node.

(kubelet) Fulfills the task of executing a pod. Since kubelets run on a node, an overload of the master is prevented.

(kube-proxy) Distributes the requests to the container based on a load balancing algorithm.

(pod) This is a running process on the Kubernetes cluster. Each pod contains one or more containers. The one-container-per-pod concept is used in most situations. Containers are able to intercommunicate between pods which have their own internal IP address.

etcd /etc distributed
IaaS Infrastructure as a Service
K8s Kubernetes
PaaS Platform as a Service
Elasticity and Scaling

1. The goal is to guarantee the elasticity of an infrastructure
   - Extends scalability by the dynamic aspect of adaption
   - Highly relevant in container-virtualized infrastructures

2. Two types of scaling
   1. Vertical Scaling
      - One pod gets more (scale up) or less resources (scale down)
   2. Horizontal Scaling
      - The number of pods is increased (scale in) or reduced (scale out)

Elasticity = scalability + automation + optimization

(1) Elasticity can be expressed by the "formula" scalability + automation + optimization [AP+18] which means that elasticity is an automation of the concept of scalability (a static property of a system) aiming the optimization of the needed resources.

(1.1) Scalability is the time-independent property of a system to be technically able to cope with increasing load by using additional resources. Elasticity adds a dynamic aspect of adaption which leads to the concept of autoscaling.

(1.2) The automated scaling, also called autoscaling, is one of the key features of Kubernetes.

(2) The difference of the two types of scaling is illustrated by the figure.

(2.1) Vertical scaling generally means that a computing node (computer, virtual machine, container, pod) receives more or less resources.

(2.1.1) In the context of Kubernetes, horizontal scaling means changing the resource requests and limits of pods for CPU or memory.

(2.2) Horizontal scaling in general, means that the number of computing node (computer, virtual machine, container, pod).

(2.2.1) In the context of Kubernetes the number of pods changes. This means that more or less pods are running.

The figure illustrates relevant aspects which must be taken into account when introducing and running a Kubernetes-based infrastructure.

(Developers, Application, Code Repository, Continuous Delivery System, Docker Registry) This part of the figure concerns the influence of Kubernetes on the software development process. The only requirement of Kubernetes is that the application must be containerized. The outcome of the CI/CD pipeline is the containerized application in a Docker registry. This means, the pipeline is not responsible to distribute the application in the Kubernetes cluster.

In the C&M environment, GitLab is used as continuous delivery system. In order to integrate a Kubernetes cluster into GitLab, the address of the cluster (uniform resource locator of the API and the port) and and authorization information is needed.

(Deployment Strategies, Kubernetes Manifest, Configuration) In order to distribute the application in the Kubernetes cluster, the developer or operator must manually apply the Kubernetes manifest files.

(Operating System) Well established operating systems, such as Ubuntu, are not appropriate for the use with Kubernetes. The reason is, that these operating systems contain redundant elements (e.g. package manager) which make their use in the context of Kubernetes insecure. Instead, specific operating systems tailored to the needs of Kubernetes do exist (e.g. RancherOS, k3OS, Talos).

(Installer) The installation of Kubernetes requires the installation and integration of several components, such as the master, the key-value database etcd, and the start of a node using kubelet. Many different installation tools (e.g. Minikube, MicroK8s, k3s, Kubeadm, Kubespray) do exist which all have their specific advantages and disadvantages.

(Monitoring) Kubernetes allows to use any monitoring tool that fulfills the requirements of the involved roles (operator, developer, end user). In general, it must be distinguished between monitoring of the Kubernetes cluster and monitoring of the pods.

A relevant monitoring tool is called Prometheus that offers the possibility to record metrics in Kubernetes. By user-defined queries and the usage of an HTTP interface, pods and controllers are able to access the measurement data.
The Git repository is responsible for the version management of the source code. Any commit to the versioning system also triggers the CI/CD pipeline.

The continuous integration and continuous deployment pipeline links development and operations together. It consists of six steps which will be explained in the following.

1. **Code Style Check** The first step of the pipeline performs a static code analysis.
2. **Unit Tests** After that, manually written unit tests confirm if the application still behaves as expected.
3. **Review** To review the application, it is deployed in a Kubernetes environment.
4. **Penetration Tests** The penetration tests are automatically carried out by the OWASP Zed Attack Proxy Project (ZAP) and the report is checked by the Neural Evaluator.
5. **Load Tests** At this stage the pipeline evaluates the behavior of the application under load.
6. **Release** In the last step the tested application will be pushed to a container registry to be available for deployment and also deployed in the production cluster.

The container registry is necessary to manage different container images of an application. This allows different application versions to be reused. It is used by the CI/CD pipeline to upload new images and by the deployment cluster to download the required images.

The development cluster is a Kubernetes-based environment which is used to deploy the applications. This cluster is part of the pipeline and ensures both application and deployment functionality.

The production cluster is the final step of the CI/CD pipeline and serves to release a tested and functioning application.

As attackers always adapt and improve their approaches, limiting security testing to the development process is insufficient. There are several approaches to implement a self-learning security system which evolves alongside the attackers.

**OWASP** Open Web Application Security Project
**ZAD** Zed Attack Proxy Project
The scaler scales the cluster horizontally and vertically according to the current load. To make a decision about whether scaling makes sense, it uses Prometheus to obtain metric data.

All microservices in the Kubernetes cluster are protected by a gateway. Its tasks are to regulate the consumption of the services regarding authentication and billing. Further tasks of the gateway could include enforcing security policies, possibly determined through machine learning. Apigee by Google contains a gateway, which can work together with the PingIntelligence API Security Enforcer (ASE) to provide such security feature.

The Neuronal Decision Point learns from incoming requests and derives policies to determine if a new request represents an attack or not. One possible implementation of this concept is the API Behavioral Security component of the PingIntelligence solution provided by Ping.

The Function-as-a-Service concept can be integrated into Kubernetes using a serverless framework. The serverless framework allows to deploy individual functions within the Kubernetes cluster. Two suitable use cases are extensions and asynchronous jobs.

Functions to extend existing software environments with additional functionality. If the software product to be extended is closed, which means that it is not possible to make your own changes to the source code and the development of your own service for the extension is too much effort, then this functionality is suitable to be developed as a function. As a result, the development time and thus the development costs are reduced.

Async Jobs) Faas is suitable for complex calculations that can be executed asynchronously, i.e. for which it is not relevant that they deliver a result as quickly as possible. If no calculations are pending, the associated function can be scaled to zero, so that no resources are consumed during this time. When calculations are due, the function is scaled accordingly. Since the result does not have to be available immediately, there are no disadvantages from a cold start.

<table>
<thead>
<tr>
<th>ASE</th>
<th>API Security Enforcer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>Decision Point</td>
</tr>
<tr>
<td>FaaS</td>
<td>Function-as-a-Service</td>
</tr>
</tbody>
</table>
1. What are functions provided by a container management?
2. What are relationships between Docker and Kubernetes?
3. Which Kubernetes components enable the communication between the master and the node?
4. What is the effect of scale up, scale down, scale in, scale out?
5. What is to be done to distribute an application in the Kubernetes cluster?
6. Which step in the introduced DevSecOps concept relates to the Sec aspect of the concept?

(1) **++Container Management++**
- Automated deployment of a microservice
- Scaling of the microservices
- Operation of microservice containers across machines and clusters

(2) **++Container Management++ ++Kubernetes Architecture++**
- Kubernetes can be based on Docker as container technology.
- One or more Docker containers are contained in a Kubernetes pod.

(3) **++Kubernetes Architecture++**
- kube-apiserver: Master component which provides the whole Kubernetes API.
- kubelet: Node component which fulfills the task of executing a pod.

(4) **++Elasticity and Scaling++**
scale up, scale down: Resources are added (up) to or withdrawn (down) from a node -> vertical scaling
scale in, scale out: An additional node is created (in) or an existing node is deleted (out) -> horizontal scaling

(5) **++Relevant Systems and Related Processes Around Kubernetes++**
- Kubernetes manifests must be applied by the operator or the developer
- From the viewpoint of the CI/CD pipeline no further demands related to Kubernetes must be fulfilled.

(6) **++Kubernetes-based DevSecOps Concept++**
- Penetration Tests: These tests are automatically carried out by the OWASP Zed Attack Proxy Project (ZAP) and the report is checked by the Neural Evaluator.