Software Architectures

Lecture Notes for students of Faculty of Mathematics and Physics at Charles University

Martin Nečaský
Department of Software Engineering
Faculty of Mathematics and Physics at Charles University

2020

Recommended reading

- Len Bass, Paul Clements, and Rick Kazman. 2012. Software Architecture in Practice (3rd. ed.). ISBN 978-0321815736. Addison-Wesley Professional.
- Scott Millett, Nick Tune. 2015. Patterns, Principles, and Practices of Domain-Driven Design. ISBN 978-1118714706. Wrox.
- Chris Richardson. 2019. Microservices Patterns. ISBN 978-1617294549 . Manning.
- Martin Fowler. 2002. Patterns of Enterprise Application Architecture 1st Edition. ISBN 978-0321127426

Table of contents

Lecture 1: What is software architecture?

1) Welcome to the first lecture about software architectures.
   This lecture is the introduction to software architectures.
   Our goal today is to understand what software architecture is and why it is important.

2) Let me start with an example.
   I will use it in this and some other lectures later as a practical example to demonstrate things about software architectures.
   The example is a software for cataloguing governmental data available as open data.
   It is called Czech National Open Data Catalogue.
   You can try it on your own, it runs on https://data.gov.cz
   It has been developed by my colleagues at the department of software engineering at our faculty.
   It is operated by our government.

3) Open data is structured data in an open format available on the internet for free reuse.
   Here we can see the so-called 5-star deployment scheme for open data invented by Tim Berners Lee.
   At the first level, we have data which is available under an open license which enables you to do anything you want with the data, unless it is prohibited by law.
   At the second level, we have data which is moreover available in a machine readable format.
   It means any format which allows you to use software to access and process individual data items without any loss of data values.
   At the third level, we have data which is moreover available in an open format.
   It means a format with an open specification which you can implement in your software without fees or any other obstacles.
   The next two levels are not important.
   The third level is considered as the minimal set of properties of any data we want to call open data.

4) Open data is available on the internet in the form of datasets.
   A dataset is a collection of logically related data values about entities of the same kind.
   For example, you can have a dataset comprising data about universities in Germany.
   Or you can have a dataset of data about teachers at Charles University.

5) In our country, governmental datasets must be catalogued in the national portal.
   It is required by law, namely the public sector information act.
   On the slide you can see a catalog record for an existing dataset comprising numbers of paid health insurance benefits by month in districts of Czechia.
   In the catalog record, you can see some metadata about the dataset, such as title, description, publisher, keywords, etc.

6) A dataset is an abstract collection of values.
It is available on the internet in different distributions which are representations of the content of the dataset in different forms.
The difference between two distributions is their format and the access technique which you use to get the content of the dataset.
For example, you can have three distributions of the dataset of all universities in Germany.
One distribution may be a simple CSV file for download.
Another distribution may be the same data but formatted in XML.
And the last distribution may be an API which allows you to access data about individual universities formatted in JSON.

7) Metadata about each distribution of a dataset is also part of its catalog record in the national catalog.
On the slide you can see two distributions of our dataset published by the Czech social security administration.
You can see that the catalog record contains metadata about two distributions of the dataset.
If you use the download link, you will get the content of the dataset in the form of a downloadable file in the selected format.

8) So this will be our running example.
Let me use it in this lecture to demonstrate what a software architecture is.
We will start with functionalities provided by the national catalog.
It is a good practice to summarize functionalities in a form of a UML use-case diagram.
The big box is the system - our national catalog.
The ellipses inside the box are use cases each specifying a required interaction between a user and the system.
To fulfill the required interaction, the system must offer a set of features.
For example, it must enable a user to enter a search query, to actually search for datasets corresponding to the query and display the result in a user interface.

9) Imagine now that you want to explain how the national catalog works to your new colleague.
Or that you are responsible to analyze how the system needs to be changed to fulfil a new requirement.
For example, you may be asked to extend catalog records with a MIME type of a dataset distribution.
Or you may be asked to display the datasets with XML distributions first in the search results.
Or you can imagine that you are responsible to plan the next development iteration, to decide what people you need and how they will expand the system.
The view of the system presented on the slide is not helpful in these situations.

You probably suspect that another view of the system which reveals its internal structure is necessary.

10) Each software system has some internal structure.
In general, it is made of elements which must be designed, developed, tested, delivered and maintained.

11) For example, most software systems are made of source code and data structures.

12) Source code is usually expressed in a high level programming language and defines instructions.
The instructions are executed on machines to provide the desired behavior of the software system.

13) Data structures are usually expressed in a form of database schemas or API schemas and define how information manipulated by the software system is represented as machine interpretable data.

14) Although this structure is inherent to most software systems it does not help us too much in situations I mentioned earlier.
We need something more detailed.
Our example, national open data catalogue is a special kind of a software system called information system.
An information system is a software system whose main purpose is to enable human users to manage and view information persisted as data in one or more data sources.
The internal structure of an information system is usually structured to 3 layers.

15) The presentation layer handles interaction between the user and the software system.
It displays data to users and interprets commands from the users into actions upon the other two layers.
For example, this can be an HTML-based UI in a web browser.

16) In the presentation layer in our running example the user may issue commands such as “show datasets”, “show datasets published by the Ministry of interior” or “show a dataset detail”.
The presentation layer interprets these commands into actions upon the other layers.
The other layers return application data which is displayed by the presentation layer to the user.

17) The presentation layer has its own internal structure.
For example, there can be a part responsible for presenting the list of found datasets and facets for their further filtering.
And there can be a part responsible for presenting dataset details.

18) Another layer common in information systems is the data source layer.
Its name suggests that it contains a database which is responsible for data persistence.
However, this is not so simple.
In general, the data source layer is about communicating with other systems that carry out tasks on behalf of the application.
Communication means data exchange in both directions.
It does not contain these other systems.
It contains the logic which is responsible for the communication.
Yes, if we consider information systems, the communication with a database is the biggest piece of the data source logic.
The data source layer is responsible for sending the data from the other layers to the database where the data is persisted and also for extracting the data from the database using database queries.
However, a data source layer communicating only with a single database is in most cases a story of the past.
Today, it is common that a data source layer is responsible for communicating with more different applications or services.

19) The data source layer in our running example is responsible for communication with three external services.
First, it communicates with a database which persists metadata records about datasets.
The database is a CouchDB database which persists records formatted in JSON.
The data source layer is responsible for extracting metadata records about datasets from the database, for persisting new or updated records and also for deleting them from the database.
It communicates with the database through database queries in a form of CouchDB-specific HTTP requests.
It receives data formatted in CouchDB-specific JSON format, transforms them to the application specific form and passes them to the other layers.

20) Second, the data source layer communicates with a dataset indexing service.
The indexing service is Apache Sorl.
The data source layer is responsible for extracting a list of datasets corresponding to a given set of parameters from the indexing service.
It communicates with the indexing service through search queries in a form of Solr-specific HTTP requests.
And again, it receives data formatted in Solr-specific JSON format, transforms them to the application specific form and passes them to the other layers.

21) Last but not least, the data source layer communicates with the web.
It may seem strange but let me explain.
Let’s go back to the presentation layer for a moment.

22) Here is the screenshot of the dataset detail you have already seen.
Please focus your attention to the central part of the screen where metadata such as dataset theme or spatial coverage are displayed.
These two and frequency are metadata values from external codelists.
Dataset theme characterizes the content of the dataset and it is a value from a codelists of dataset themes published by European Commision.
Frequency specifies how often the dataset content can be updated and it is also a value from a codelist published by European Commision.
Spatial coverage specifies a region on a map for which the dataset content is relevant.
It is a value from the Czech specific codelist of spatial regions published by Czech State Administration of Land Surveying and Cadastre.

23) These codelists and their items are published on the web as Linked Open Data.
Shortly speaking each codelist item has its unique IRI which can be accessed to receive data about the identified entity.
So for example, “health insurance” item of the dataset theme codelist is identified by the URL displayed on the slide.
When you access the URL, you get data about the item.
For example, you have its label in each european language.

24) So, the data source layer from our running example uses the IRIs of the items to get detailed data about the items from the external resources using standard HTTP requests.
It receives data in the RDF format, which is a standard format for Linked Open Data, transforms them to the application specific form and passes them to the other layers.

25) The other layers communicate with the data source layer which returns the required data in the application specific form independent of the original data sources.

26) The remaining layer out of the three is the domain layer.
Sometimes, it is also referred to as a business layer.
It involves all domain (or business) specific logic.
This is probably too generic so let’s be more concrete.
For example, when the user provides some input to the presentation layer it is dangerous to work with the input directly.
The input must be validated.
This validation has nothing to do with the presentation layer neither with communication with the external services.
It depends on the domain how the input is validated, i.e. what is a valid input.
Therefore, user input validation is part of the domain logic.
Another example are various calculations based on inputs from the users and data from the data source layer.
How we handle these data and calculate new application data is a responsibility of the domain logic.

Also decisions about which data source logic should be dispatched based on commands from the presentation are domain logic.

27) Well, you may call for even more concrete examples.
So, here is one in our running example.
On the slide, we have a screenshot of another dataset detail.
Take a look on the red exclamation mark at the title of the dataset.
It informs the user that there is a problem with the dataset record.
When you click on the mark you will get a detailed explanation of the problems with the dataset.
This may include, e.g., that some important metadata is missing or other problems.
The concrete kinds of problems are not important here.
What is important is that our data sources do not store the information that there is a problem, so we have to construct it in the application code.
The construction is the responsibility of the domain logic.
It cannot be done in the data source layer which is responsible only for communication with external services which are not interested in this information.
And it should not be done in the presentation layer which is responsible for presenting application data not their calculation.
The presentation layer is only responsible for the visual style of presenting the information.
In other words, the presentation layer does not decide whether there is a problem.
This decision is up to the domain layer.
The presentation layer gets the information and decides that it will be displayed as the exclamation mark in the red color and that it is displayed at the title of the dataset.

28) The domain layer also has its internal structure.
For example, we may distinguish the domain logic of processing dataset detail from the domain logic of searching datasets.

29) The logic in the domain layer is used by the presentation layer.
The domain layer uses the data source layer to get necessary data from external sources.
On the slide, you can see the resulting structure of our running example.
It also shows that the whole application logic is split between a web client application and a server.
The client provides the presentation logic.
The server provides the domain and data source logic.
Of course, the displayed parts have their own internal structure.
We could decompose them further until we get to the details of the source code and data structures.
However, as you will see later, this is not the intention of software architectures.
Also you may wonder why this structure and not another structure.

Maybe you are familiar with some libraries for building web user interfaces such as React and, therefore, you would expect another internal structure of the presentation layer.
Maybe you would expect that the domain logic or its part is on the client side.
This is one of the reasons why we need software architecture.
If you start with a set of user requirements, you can design different software structures to fulfil them.
Two software systems with different structures will not differ in the functionalities provided to the users.
However, they will differ in other aspects such as performance, security, scalability, user experience or how easy it is for the development team to modify them when user requirements change.

30) Now, we are ready for a definition of software architecture.
So, let's have some theory.
You can find different definitions of software architecture in literature.
I will not read them here all.
Instead, I have picked up one of them which nicely covers what I have shown you on the previous slides.
You can see it on the slide.
The software architecture of a system is a set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
This definition stands in contrast to other definitions that talk about the system’s “early” or “major” design decisions.
They say something like that software architecture is everything which is decided early in the software system lifecycle and which cannot be changed later or which is hard to be changed later.
While it is true that many architectural decisions are made early, not all are - especially in agile or iterative software development projects.
It’s also true that very many decisions are made early that are not architectural.

31) I would like to pick two important points from the definition.
It says that software architecture is a set of structures and that it is an abstraction.
We saw this in our running example already.
Let’s discuss this a little deeper.

32) [ONLY SLIDE]

33) First, we can view a software system as a set of structures which partition a software system into implementation units.
We will refer to such units as modules.

Each module has assigned specific computational responsibilities, shortly speaking it must do something which provides some features either to our users (human as well as other systems) or to other modules of our system.
Modules must be assigned to individual people or teams who develop them, configure them, test them and integrate them with other modules.
Usually, a module means a programming code which must be developed.
Sometimes, a module is something which already exists but must be acquired, configured and integrated.
Therefore, modules are static structures.
We reason about them independently of how they exist at runtime.

34) [ONLY SLIDE]

35) We can also view a software system from the runtime point of view.
We can view it as a set of structures which exist at runtime and interact with each other to carry out the required functions of the system.
We will refer to these structures as components.
A component exists at runtime.
Basically, it may be an instance of a module.
However, it may also be an instance of a set of modules which are logically related.
From the static point of view, it is necessary to distinguish them but we do not distinguish them at the instance level at runtime.
Of course, a module may have more instances.
I will not go into details here.
Let’s keep them to a lecture dedicated to components.

36) The last kind of structure is allocation structure.
Allocation structures are structures which describe mapping from software structures (modules or components) to the system's environments.
These environments may be organizational, developmental, installation or execution.
For example, mapping of modules to developers is an allocation structure.
Mapping of components to physical infrastructure is an allocation structure too.

37) In general, a structure is architectural if it supports reasoning about the systems and the properties of the system.
In the extreme, the set of lines of source code that contain the letter 'z', ordered by increasing length from shortest to longest, is a software structure.
However, it is not an architectural structure.
System properties are attributes of the system that are important to some stakeholders.
For example, this includes functionality achieved by the system, availability of the system in the face of faults, difficulty of making specific changes to the system, responsiveness of the system to user requests or how easily can the system be tested.

You use architectural structures to explain how these attributes are achieved in your system.

38) The definition also tells us that an architecture is an abstraction.
Describing the architectural structures of a software system means describing particular kinds of software elements and relations between them.
Naturally, this is a kind of abstraction of a software system.
Software architecture specifically omits certain information about elements that is not useful for reasoning about the system.
Software architecture always omits private details of elements, because details having to do solely with internal implementation are not architectural.
Moreover, we simply cannot, and do not want to, deal with all of the complexity of the system at once.
So the architecture does not provide the full detail but only the necessary level of abstraction.

39) We can also think about the relationship between the architecture of a software system and the system itself.
We could ask questions.
[SLIDE]
The truth is that every system has an architecture.
However, it does not necessarily follow that the architecture is known to anyone.
People who designed the system are long gone, the documentation has vanished (or was never produced), the source code has been lost (or was never delivered), and all we have is the executing binary code.
But still, it has an architecture.
So we need to distinguish ARCHITECTURE OF THE SYSTEM and REPRESENTATION OF THAT ARCHITECTURE.
An architecture can exist independently of its description or specification.
This raises the importance of ARCHITECTURE DOCUMENTATION.
In other words, the architecture must be documented somehow to be useful.

40) What is also important is that architecture includes behavior.
The behavior of each element is a part of the architecture when the knowledge of that behavior is important to reason about the system.
This behavior must be documented as part of the documentation of the architecture otherwise, architecture descriptions become only box-and-line drawings
Readers may imagine what these drawings mean but it is dangerous since it depends on the imagination of the reader's mind.

41) So let's summarize what we have heard by asking the question.
Why is software architecture important?
First of all, software systems are constructed to satisfy business goals of an organization.

And the architecture is a bridge between the business goals and the final resulting system.

42) But we also have other reasons.
Some of them are listed on this slide.

Lecture 2: Basic Terminology

1) Welcome to the second lecture about software architectures.
In this lecture we will define some basic terminology we will use to talk about software architectures.

2) We will not reinvent a wheel
We will use an existing standard which defines software architectures terminology.
The standard is issued by IEEE under the number 1471.
Its title is Recommended Practice for Architectural Description of Software-Intensive Systems.
IEEE is a global technical professional organization.
Its members are professionals from different engineering fields across the globe.
Naturally, software engineering, including software architectures is also the field of interest of this organization.
Among other things, IEEE issues technical standards which describe a common practice in a given field.
The mentioned standard is important as it introduces a terminology which is understood by software architects independently from where they live and for which company they work.

3) The standard introduces its own definition of the software architecture.
You can see it on the slide.
You can also see that it is not very different from the definition we had in the previous lecture.
Actually, the IEEE definition extends the previous one with the last part about the principles.
The rest is almost the same.

4) The definition considers different kinds of software systems, e.g. application, platform, system of systems, enterprise system, product line, etc.
They differ in their nature and each require a little different approach to designing the software architecture.
For example, a platform is a software system which defines some basic features which can be extended with third-party extensions.
Its architecture has to enable these extensions and show how to integrate them to the platform.

A system of systems combines existing systems to a new one.
Its architecture must show how the systems are integrated together.
Enterprise system is a complex software system or system of systems which supports the business of an enterprise, which means a large organization.
It is strongly related to business processes in the organization.
Its architecture must show how the processes are supported by the software.
A product line is a software which is delivered as an out-of-the box software to its customers which can be customized to their needs.
Its architecture must show what can be customized and what kinds of modifications are necessary, e.g. whether only some parameters are reconfigured or whether something new has to be programmed by developers.
An application today usually means something which runs on many client machines and has some central server part.
Its users are usually more or less known in advance.
It means that we cannot directly ask them about their requirements because we do not know them.

5) The definition considers different contexts in which the software exists or will exist.
The developmental context views the architecture in the context of the software system development process.
The operational context views the architecture in the context of the system operation, administration, and support when it is running in its production environment
The business context views the architecture in the context of the business needs of the organization.
The technological context views the architecture in the context of the technologies used in the organization and known to the development team.

6) Maybe you think about the term “fundamental” in the definition.
Here it means something which is essential and unifying different concepts and principles in a single framework.

7) The basic terminology is defined in the IEEE standard called Conceptual Framework.

8) The central concept of the conceptual framework is a software system.

9) The software system has to fulfill missions of the organization which owns or uses the system.
The organization and its surroundings (partners, legislation, customers, etc.) create an environment in which the software system lives.
Naturally, the environment influences the system.
The system must be built so that it can be used in the environment.
For example, it must meet all required legislation and must be usable by its users.
But, the system also influences the environment.

For example, new technologies bring new possibilities which means that old pieces of business processes may be automated and human power is no longer necessary for them.

10) There live different people in the environment.
These people have some interests in the system.
It is common to call them stakeholders, not only in the field of software architecture but in the field of software engineering in general.
A stakeholder is an individual, team, or organization (or their combinations).
For example, it can be a client, acquirer, owner or user.
It can be an architect, software engineer, project manager, developer, designer, tester or administrator.
And, it can also be a service provider, vendor or subcontractor.
Each of these people have different interests in the system.
A little problem is that their interests are sometimes in a contradiction.
The concept of stakeholder allows us to reflect the reality that many different people are involved in complex systems, and each person has a different perspective.
Many requirements on a system's quality attributes are based on specific stakeholders.
For example, affordability is important for the acquisition people, maintainability is important for maintainers of the system and users want a system that is usable now.

11) The different interests of different stakeholders which pertain to the development, operation, or other key characteristics of the system are called concerns.
A concern may be, e..g, the system’s performance, reliability, security, modifiability, usability, testability, etc.

12) Now it is time for our software architecture and its relationship with the software system.
We discussed this in the previous lecture.
Each software system has its architecture.
However, the question is whether it is known and documented.

13) We need the architecture to be documented.
Therefore, we consider its description.
An architectural description is a collection of products to document an architecture.

14) Architectural description identifies system’s stakeholders and their concerns.
In this relationship, concerns define completeness of the architectural description.
We can say that the architectural description is complete iff it addresses all concerns of all stakeholders.
However, in practice it is very hard or even impossible to identify all stakeholders and their concerns.

Therefore, a more practical definition of completeness of the architectural description says that the architectural description is complete iff it addresses all IDENTIFIED concerns of all stakeholders.

15) We have already discussed in the previous lecture that architecture and its description is an abstraction of the software system.
It emphasizes important aspects of the software and omits unnecessary details.
However, for different people and purposes we need to emphasize different aspects.
Therefore, software architecture is too complex to be described at once.
Therefore, the architectural description consists of views.
A view is a description of the architecture from the perspective of a related set of concerns.
Different views support different stakeholders with their own concerns.
Views reduce complexity of describing an architecture through separation of concerns.

16) It is a good practice to approach the architectural description in a form of views systematically.
There are recommended practices of which kinds of views should be used for which purposes.
We refer to these kinds of views as viewpoints.
A viewpoint defines a set of types of elements and relationships to describe views.
It also defines rules for their usage and presentation.
Concerns then drive the selection of the viewpoints to be used.
Each concern of our stakeholders must be addressed by some architectural view.
However, no fixed set of viewpoints exists.
The development team usually evolves its library of viewpoints which they use to describe architectures.

17) We may express each view using an ad-hoc graphical notation consisting of various boxes and lines.
However, it is a good practice to use the same notation for a set of views conforming to the same viewpoint.
The definition of this model, which may be formal, semi-formal or informal, is called model.
A model gives us a set of modeling constructs expressed with some graphical notation we can use to describe views conforming to the model.

18) So this is the basic terminology of software architectures.
If I use a concept during the semester which is displayed on this slide, I mean by it exactly what was defined today.
Thank you for your attention and see you at the next lecture.

Lecture 3: Architectural Views - Module Viewpoint

1) Welcome to the third lecture about software architectures.
   In this lecture we will go deeper into the topic of architectural views.

2) First, let me remind you what an architectural view is.
   A view is a part of an architectural description.
   It describes the architecture of a software system from a particular viewpoint which is used to cover certain concerns of certain stakeholders.
   For this, it defines kinds of elements and relationships, and rules to use them.
   They are defined as a part of the model which also defines how the elements and relationships are visualized with boxes and lines.
   Usually, the definition is informal and not expressed explicitly.

3) Often, the definition exists in the heads of software architects who design the architecture and document it as a visual diagram with boxes and lines.
   The question is whether such drawing can be called a view of a software architecture.
   Let me show you some examples.
   Several years ago, I was involved in an expert group at the Ministry of Health which was responsible for selecting the best design of an architecture of the future Czech eHealth system.
   Today, some of its parts already exist but at that time nothing existed and the Ministry wanted to have some architecture to start with.
   They asked IT companies to offer their proposals and the best proposal got paid.
   12 architectural proposals were received.
   On the slide, you can see three architectural diagrams which describe three different architectures of the same system from three different proposals.
   You can see some boxes and lines.
   Reading the drawings correctly without the knowledge of the boxes and lines is hard.
   It may seem simple.
   You probably understand the drawings somehow.
   However, are you sure that you understand them correctly?
   To help readers with reading our architectural views, we can do two basic things.
   First, we can provide a textual description of our drawings.
   However, no one likes reading long texts.
   It still holds that a picture is worth a thousand words.
   So the second basic thing is to make our drawings better readable.
   How can we improve the readability of our drawings?

4) One possibility is to use a standardized graphical notation.
   For example, some kinds of UML diagrams such as UML component diagrams or UML package diagrams could be used.
   However, the UML notation itself will not save you.
   It is just a notation.

Even though it has a formal background, it is not exactly prescribed how to use it to express an architectural view.
Another possibility is to follow the practice of designing architectural views recommended in the literature.
There exist different approaches to what viewpoints should be considered when designing a software architecture, what kinds of elements and relationships between them should be considered and when they shall be used.
On top of these recommended viewpoints and principles, some of the approaches also recommend concrete graphical notation to express the views.
On the slide, you can see four different approaches you can find in the literature.
In this course, we will consider the first two.
We will discuss their principles.
A concrete graphical notation is up to you.
You can use the notation used in the slides or your own notation.
You must just ensure that you use the same notation consistently throughout your project or across your projects and that you are able to explain it to the reader.

5) Let me start with the approach of the authors Bass, Clemens and Kazman.

6) They distinguish three basic architectural viewpoints.
The first one is the module viewpoint.
The purpose of the module viewpoint is to answer the following question: how is the system to be structured as a set of code units?
This question is important for the concerns of stakeholders who need to know how complex the system is, how to plan its development and testing, what parts of the system need to be developed and what parts can be bought or where and how the system needs to be modified when requirements change.
The second viewpoint is the component-and-connector viewpoint.
The purpose of the component and connector viewpoint is to answer the following question: how is the system to be structured as a set of elements which have runtime behavior and interactions?
This is important for the concerns of stakeholders who need to reason about runtime properties of the system such as availability, performance, etc.
The third viewpoint is the allocation viewpoint.
The purpose of the allocation viewpoint is to answer the following question: how is the system related to non-software structures in its environment.
It is important for stakeholders who need to know on which hardware infrastructure the system or its parts run or what people are responsible for the development, testing, administration or maintenance of which parts.

7) A module view shows the static implementation units of the system and static relationships between them.
Describing the software architecture using the module viewpoint means a code-based way of considering a system.

Elements of the module viewpoint are called modules.
A module represents an implementation unit that provides a coherent set of responsibilities.
Coherent means logically related.
For example, a module may represent a layer of the system.
It can also be a package of classes, collection of services, a single service, a single class, an interface, a class method, a procedure, etc.
A module has a collection of properties assigned to it.
They are descriptive metadata about the module.
Typically, each module has a name.
Also its functional responsibility may be defined in its metadata in a form of their textual specification.
Other kinds of metadata important for the management of the project may also be specified, e.g. creation date or whole revision history, people who are responsible for the development and/or testing the module, etc.
Last but not least, the architect also links the module to concrete software artifacts where the module is actually implemented.

8) You design a module view to answer questions related to responsibilities of different parts of the software system.
These questions include:
What is the primary functional responsibility assigned to each module?
What other software elements is a module allowed to use?
What other software elements does it actually use and depend on?
What modules are related to other modules by part-of, generalization (specialization) or other kinds of relationships?

9) So a module viewpoint primarily focuses on the decomposition of code into units and it determines assumptions each unit can make about services provided by other units.
Therefore, it serves as a blueprint for the construction of the code.
Software engineers (designers, developers, testers, ...) can see the whole context of the units they work on.
Designers and developers can see what modules they are allowed to use and which they cannot use.
They can also see who depends on them.
This is important because even though, theoretically, modules should not depend on each other's internal implementation, in practice they do.
So when you know who depends on your units and you need to make some change which may impact them, you can communicate with them and prepare them for the change in advance.
The module viewpoint is also important for a project manager who can plan the project with respect to these relationships and people who work on the related modules.
It supports the project manager in work assignments, creating implementation schedules and budget estimations.

10) The module viewpoint helps to plan the development of the software in increments.
You can see what units need to be developed first and what next.
It does not mean that they have to be completed before you can proceed to the others.
It is possible to implement their stubs whose interface works but their internal implementation is trivial.
Then you continue with the dependent units and later you get back to the stubs to improve their internal implementation.
The module viewpoint is also helpful when a given set of features needs to be reused within the software system and even outside of it.
You are able to identify the code unit which provides the feature and all units it depends on.
Then you can take the identified piece of software and reuse it somewhere else.

11) The module viewpoint also supports change-impact analysis.
The overall decomposition structure helps the development team to identify units which need to be changed when a new user requirement appears or an existing one is changed.
And it enables them to see how a change to one part of the system might affect other parts.
You can also use it to trace requirements.
Each module can be associated with requirements it covers or supports.
You may then trace the modules related to a given requirement and analyse, e.g., whether all requirements are covered by the system's parts.

12) The module viewpoint is also important for communication.
When you want to explain the system to a new team member or actually to any stakeholder, you can use the decomposition to communicate its functionality and the structure of the code.
You can do this in the top-down direction according to the decomposition of modules to sub-modules which means that you start with a generic overview and you go to the details as you follow the decomposition.

13) As I have already said, each module is described not only by displaying it as a box connected by lines with other modules but also with its descriptive metadata.
First of all, each module must have its unique name.
The name is the primary means to refer to the module.
It often suggests the role of the module in the system.

It is also useful to specify the visibility of the module's interface.
The interface may be public which means visible to any other module in the architecture.
A sub-module of a module may also have a private interface.
It means that it can serve only to other sub-modules in the same module.

Other kinds of visibility are of course possible.
It depends on your needs which kinds of visibility you define and use in your projects.

We should also describe functional responsibilities of the module.
In other words, we should identify and describe in detail the role of the module in the system, which means what it does in the system.
It means to describe what features the module provides.
And we should also describe who or what uses these features.
Who means human users.
What means other modules in the system and also external systems.

Responsibility is abstracted from implementation details.
These implementation details should be described separately for the module.
It means several things.
First, it means how the module is mapped to source code units.
In other words, in which source files the module is actually implemented?
This is an M:N mapping.
A module can be implemented in different source files, one with interface definition, one with internal implementation and one with unit tests for example.
On the other hand a situation where more modules are implemented in a single source file is also possible.
We will discuss this in a more detail later.
You should also provide test information which includes test plan, test cases and test data.
A management information with the predicted schedule and budget for the module is also specified in some projects.
And, the architect can also require that a module’s implementation meets some restrictions or implementation strategy.
This includes, for example, a required algorithm, code patterns, etc.
Last but not least, we should also record the revision history of the module.
It means who did what with the module regarding at least its design, development and testing.

14) We distinguish several sub-kinds of the module viewpoint.
[SLIDE]

15) Let's start with the decomposition viewpoint.
In this viewpoint, modules are logically decomposed to smaller modules recursively until they are small enough to be easily understood.
As a software architect you enumerate units of software the development team will have to build.
For each module you either further decompose it to smaller modules or you assign it for subsequent detailed design and eventual implementation.

In the decomposition viewpoint, modules are related to each other only by one type of relationships.
It is a part-of relationship and it specifies that a module is a sub-module of another module.

16) A module in the decomposition viewpoint represents a common starting point for the further software design and development.
You usually associate other artifacts with a module.
This is, e.g., interface specification, code of the internal implementation, test plans, development team, etc.
Each module has its interface which is visible outside of the module depending on its visibility specification.
It also has its secrets which are not visible outside of the internal implementation of the module.
This is also called information hiding and the level of information hiding is one of the fundamental architectural concepts.
The less you hide the more dependencies will exist in your system.
This impacts on the system's qualities positively as well as negatively.

17) Why do we need the decomposition viewpoint?
First of all, we use it to divide and conquer the task of designing the internal structure of the software system.
As an architect, you need to divide the system into smaller parts and visible relationships between them which are easily understandable to the development team.
The specified relationships then allow you to manage integration of the smaller parts to bigger pieces.
In other words it helps you with conquering the overall system development.
It does not mean that you have to plan everything in advance.
It does not force you to the waterfall development process.
You may decompose the architecture in iterations continuously as the project grows.
For example, you may postpone detailed decomposition of certain parts of the system for the later development iterations.
You can also start with a simpler decomposition of a module and replace it with a more complex one later as the system becomes more complex.

You know that the other modules use the module only through its interface.
When the interface is not affected by the new decomposition, the outer context of the module is not affected as well.
And when the internal change to the decomposition of the module affects its interface, you know that the other modules are affected by this change and you can act.
To identify what other modules are affected you need the usage viewpoint which we will discuss next.

As an architect, you have also techniques which enable you to minimize the impact of the changes in the module's interface to the other modules.
For example, using interface versioning you can spread the impact out in time.

This also means that you can benefit from the decomposition viewpoint when you need to manage changes.
The decomposition viewpoint helps you to identify which parts of the system are affected by the change.
In the connection with the usage viewpoint it also helps you to identify the other modules which need to be modified because the modules impacted by the change were changed.

Because the decomposition viewpoint shows the structure of the software, helps with iterative development and change management, it is also a useful tool for project managers.
It can support them in iterative as well as agile project management and with the overall organization of the project.
It helps them with structuring and planning the project, with structuring the documentation, with test and integration planning, and also with allocating project resources.

18) Let's discuss the decomposition viewpoint on our running example of the national open data catalogue.

19) It is an information system so we follow the common practice which is to structure the system to three layers - presentation, domain and data sources layer.

20) We first focus on the domain layer.
It is the layer where the business logic of our information system lives.
It is important for us to distinguish the logic of the model of the domain from the specific application logic of the information system.
The domain model is represented by the model submodule.
The application logic is represented by the services module.

21) The model of the domain is independent of the requirements specific for the information system and it tries to reflect the reality.
It comprises datasets and their distributions.
They are important parts of the domain model so we design them as architectural modules.
We will describe in the documentation that they will have some responsibilities, e.g. that a dataset must be able to return its title, description and the list of its distributions.
It must also be able to manipulate with the list of its distributions.
A distribution must be able to return its title, description, download URL and its format.
We do not say whether they will be implemented as classes or their responsibilities will be implemented as methods of some broader class.

We can make this decision if it is important from the architectural point of view.
However, it is not important now so we just say that there are these two implementation units but we do not specify their form yet.

22) The services module represents application specific logic on top of the model of the domain.
The model of the domain provides a fine-grained interface to the domain objects which makes it highly reusable in the future for other applications.
This can be, e.g. an open data quality reporting application which is another application independent of the national open data catalogue.
However, such a fine-grained interface is not necessary for the national open data catalogue.
The catalogue needs a coarse-grained interface which provides application specific services.
This is represented by the service module.
It provides two services.
The first is used for searching datasets on the base of the specified search conditions.
The second is used for getting the detail of a dataset.
The services manipulate with domain objects defined by the domain model module and provide a more coarse-grained interface upon the detailed interface of the domain model.

23) In the data sources layer, we have only one data source which we will call a dataset source.
The other layers will use it to persist and access data about datasets and distributions.
We manage the distributions data together with dataset data through the dataset source because we store them together.
We do not use a relational database which would, because of data normalization, split data about datasets and distributions to separate relational tables.

24) We further decompose the dataset source because we want to show in the architecture that we need the dataset source logic to be separated to two parts.
We need the dataset gateway which enables us to manipulate with the database representation of individual datasets (which means the CRUD operations - create, retrieve, update, delete).
And separately, we need the dataset finder which is for searching datasets.
The reason for this separation is to be able to easily reuse the logic of the manipulation with database representation of individual datasets.

25) And will finish the decomposition for now with decomposing the presentation layer.
The presentation layer is not only about human user interface.
It is also about presenting the system to other machines via APIs for example.
We need both kinds of presentations in our system.

Therefore, we decompose the presentation layer to two submodules - web application and public API.

26) The public API provides two operations - dataset list and dataset detail.
They use the business logic of the domain services instead of implementing their own and ensure the correct presentation as an API on the web according to an existing standard for exchanging dataset metadata records among machines.

27) Regarding the web application, we have decided not to work on it yet.
However, we have it here to show that the web application uses the same domain logic as the public API.
Both do not implement their own business logic but reuse the same.
The public API extends the business logic of the domain services with the technical issues such as correct handling of HTTP requests and correct representation according to the given data standard.
The web application will present the business logic of the domain services to human users.
For this it will offer a rich web user interface built using some of the modern frameworks, e.g., React.
It will extend the basic business logic of the domain services with its own logic of the web user interface.

28) As I have said, a module is an implementation unit.
So it must be represented as a piece of source code somewhere.
It is possible to implement a module in a separate source file.
You can see it on the slide.
The module Dataset Index is implemented in the source code as a Java class in a separate Java source file.
What is not prescribed by the architecture are its public methods and private details.
The decision about the public interface as well as the private details is left to software designers or developers.

29) It is also possible that a module is implemented in more source files.
For example, the module Dataset Gateway is responsible for retrieving and storing datasets and their distributions to the persistent store.
It is implemented in the source code as a Java class in a separate Java source file.
In addition there is another source file with two helper Java classes which we will use to represent metadata about datasets and distributions which is loaded from and stored to the persistent store.
These two additional classes are not represented as modules in the architecture.
They are not important for the architecture and they were introduced to the code by the software designer or developer.

30) It is also possible that two modules are implemented in a single source file.

The slide shows that the two submodules of the public API module are implemented as two different functions in a single source file of the NodeJS application.
The decision about the mapping of the modules to the source code can be done by the architect, software designer or even the developer.
In any case, when the decision is made, it is a good practice to record it to the description of the modules.

31) A module may be associated not only with one or more source files which implement it in the source code.
Other complementary source files may also exist.
For example, a source file with unit tests defined for the module may exist.
Or, we can have an interface definition.

32) Let me now proceed to the other subkind of the module viewpoint.
In the previous example, I sometimes said that a module uses another module.
This relationship between two modules cannot be expressed with the decomposition viewpoint.
The decomposition viewpoint shows only part-of relationships.
To express the usage relationship, we use another sub-kind of the module viewpoint called a usage viewpoint.
It shows how modules are allowed to use other modules.
However, saying that a module uses another module is not enough.
It is too vague and different interpretations are possible.
Therefore, we need to define the semantics of the relationship.
We will use the following definition.
A module uses another if the correctness of the first requires the presence of a correct version of the other.

33) Why do we need the usage viewpoint?
It shows us dependencies among modules important for the development planning.
In other words, it helps with answering the question: What should we develop first?
This is also important for planning when the development team starts and finishes its work on the project and how we can optimize their work and avoid delays.
If you see that a module A uses another module B which is not ready yet, you can reallocate the team for the module A to other parts.
This does not mean that module B has to be finished.
You may require only some of its planned iteration to be finished.
In some cases, you can also decide to develop some trivial or faked implementation of B to enable the team to work on A and iterate the implementation of B later.
Another reason for the usage viewpoint is system extensibility and modifiability.
We have already discussed this with the decomposition viewpoint and we said that we are missing the information about which module uses another module.
This information is provided by the usage viewpoint.
The usage viewpoint also helps with software reuse.

It enables you to identify subsystems which can be reused in different systems.
We have already seen in our running example that we want the domain model to be reused in different applications and that we want the services layer to be reusable for different presentations.
However, we need to identify what other parts of the system these parts need.
This can be identified with the usage viewpoint.

34) Here we have an example of the usage view on our running example.
This is a high-level usage view which shows how the three basic layers use each other in our system.

35) This slide shows a more detailed view.

36) If you read the usage relationships carefully, you could ask why the search datasets service module does not use only the domain model.
As you can see, it also uses the data source layer.
Yes, it would be clear to allow the services modules to use only the domain model layer.
However, there was probably a reason for this in the project.
Maybe, it was that the team was in a hurry and they will refactor this later.
Anyway, the usage view shows clearly that there is such inconsistency in the architecture.

37) Sometimes, it is not enough to have only the usage relationship.
Sometimes, we need to distinguish different semantics of how a module is related to another module.
This slide shows four possible semantics.
[SLIDE]
You can read their definitions and think about them on your own if you want.

38) Sometimes, there are similarities in behavior or capabilities of modules which can be captured by sub-classing.
For this, we can use the last kind of the module viewpoint we will discuss in this lecture.
It is called class viewpoint.
It considers the "is-a" relationship between modules.
It specifies that a module is a special kind of another module.
However, it is not exactly the same as inheritance in OO
The specializing module inherits the behavior, interface and capabilities of the general one.
However, this specialization does not necessarily need to be expressed in code or they are expressed in code but the module is not implemented as a class.
The inheritance of modules is usually only an information for developers who somehow deal with it to ensure that the specializing module shares behavior, interface and capabilities with the generic one.

The class viewpoint is important when we need to reason about software reuse and incremental addition of functionality.

39) To demonstrate the class viewpoint we will extend our running example with a new service which returns the list of datasets related to a given dataset.
It has quite complex business logic which goes deep into the details about the dataset and searches deeply in other datasets for different relationships between them.
The result of the logic is the list of datasets which can be somehow interesting for the consumer of the given dataset.

40) There are two more specific business logics for searching related datasets we want to distinguish and make them callable from the presentation layer.
We have a specific logic for searching relationships with datasets of two specific kinds.
The first are statistical datasets which we call data cubes.
The second are taxonomies which are datasets which represent things in a hierarchical relationship, e.g. the taxonomy of animal species.
For both kinds of datasets we have a specific business logic which searches which datacubes and which taxonomies are related to the given dataset.
We need to model both logics separately but we want to show that they have to share the generic behavior with the generic service Get related datasets.
This is information for the developers independent of how concretely the logic will be implemented.
It can be used in any way which ensures that the implementation of both Get related datacubes and Get related taxonomies share their behavior with the generic service.

41) So this was all from this lecture about the module viewpoint.
See you in the next lecture.

Lecture 4: Architectural Views - C&C Viewpoint

1) Welcome to the next lecture about software architectures.
In this lecture we will continue with architectural viewpoints and views.
In the previous, third, lecture we discussed one of the approaches to the classification of architectural views introduced by the authors Bass, Clements and Kazman.
We discussed its first viewpoint which was the module viewpoint.
We use it to describe a software architecture from the development point of view.
It shows us which code units need to be developed, configured, tested, integrated and delivered.
We call them modules.
In this lecture we will discuss another viewpoint which is called component-and-connector viewpoint.
We use it to describe the software system from the runtime point of view.

2) The component-and-connector viewpoint, shorty C&C viewpoint, shows elements that have some run-time behavior and relationships which exist between them at runtime.
Elements of the C&C viewpoint are called components and connectors.
Components are the principal units of computation and represent the runtime behavior of the software system.
They can be services, their clients, peers, processes, threads, objects, data stores or many other types of runtime elements.
Connectors are the communication vehicles among components and represent some interaction between components at runtime.
Kinds of connectors range from simple call-return communication primitives to complex network connectors.
We can also say that a connector is a pathway of interaction.
It can represent calling an object method from another object or network communication according to a network protocol such as Hypertext Transfer Protocol (HTTP).
Relationships in the C&C viewpoint are attachments which indicate which connectors are attached to which components.
For example, a call-return connector is attached to the calling and called object.
An HTTP connector is attached to the clients and to the server.

3) The basic question the C&C viewpoint helps to answer is: What are the major executing components and how do they interact through which interaction pathways?
The complementary questions are: Which parts of the system are replicated or how they can be replicated?
What parts of the system run in parallel or can run in parallel and how?
The C&C viewpoint also helps to answer questions related to the flow of data in the running system.
It helps to answer the question: What are the major shared data stores in the system?
When the data stores are accessed by different components?
How does the data progress through the system?
And last but not least, the C&C viewpoint also helps to reason about changes to the system's structure at its runtime.
For example, what happens when we want to update a component?

4) The C&C viewpoint helps you to reason about runtime system qualities.
You can reason about the likelihood of failure of a given component or connector and use this property to determine the overall availability of the system.

Or you can reason about the response time of a component or throughput of a connector under different loads.
You can then use these properties to determine system-wide properties related to its performance such as response times or throughput of user's requests.

You can also define how shall a component or connector enforce or provide security features, such as encryption or authentication.

This can be used to determine security vulnerabilities of your system.

These were just examples.
Using the C&C viewpoint you can reason about many other runtime qualities of the system.
We will discuss them in detail later during the semester.

5) Before I show you a sample C&C view, let me remind you of the decomposition module view describing our National Open Data Catalog.
In our decomposition, we had the three top layers - presentation, domain and data sources.
The data sources layer provides access to the persistent data stores.
The domain layer represents the domain logic with its domain model submodel and the application logic with its services submodel.
The presentation layer presents the application logic to the outside world.
For now we consider only other software systems communicating with the National Open Data Catalog through its public API.
This can be, e.g., an external mobile application which enables human users to browse the content of the catalog on their smartphones.

6) This slide shows a high-level C&C view of the National Open Data Catalog.
It shows components as labeled rectangles and connectors as labeled arrows.
The catalog comprises 4 runtime components: public API gateway component, backend service component, Solr component and CouchDB component.
The components interact through the HTTP connectors.
Each HTTP connector is attached to components which interact through the network using the HTTP connector.
The orientation of the connectors shows which component sends HTTP requests and which responses to the requests.
There are also N external components which are applications running on smartphones and other mobile devices of human consumers.
They interact with the public API gateway through the HTTP connectors.
When a consumer’s application sends an HTTP request to the public API gateway, the gateway first requests the backend service through the HTTP protocol.
The backend service sends the result back as the response.
The public API gateway then transforms the result to an external communication format, which is a JSON format respecting an existing international standard for data exchange among data catalogs, and sends it back as the response to the HTTP request.
The backend service uses the Apache Solr component and CouchDB component.
The former is used for searching datasets.
The other is used for persisting metadata about datasets.
Both components are called by HTTP requests.
What is not shown by the presented C&C view are the network communication details.
This will be the task for the third viewpoint, the allocation viewpoint.

We will show here that the HTTP communication of the external consumer applications with the public API gateway is through a lower-performance network which is a public internet.
The HTTP communication between the other components is on a higher-performance internal network.
The allocation viewpoint will also show us that there are some firewalls so the HTTP communication does not flow directly but through the firewalls for the security reasons.

7) Components and connectors can be complex elements.
Sometimes it is enough to show them as black boxes which means that we are not interested, from the architectural point of view, in their internal runtime behavior.
In our previous example, we had the HTTP connectors.
An HTTP connector is a complex system based on the HTTP protocol.
However, its internal runtime behavior is not important for describing the architecture of our software system.
Therefore, we do not describe its internal runtime architecture.
We are interested in its external features.
For example, an HTTP connector is attached to two components but each in a specific role.
One is a client who sends HTTP requests, the other is a server which sends HTTP responses back to the client on the base of its requests.
Therefore, a connector usually has roles for attached components.
A component in a given role must fulfil some conditions.
For example, an HTTP client must be able to send HTTP requests and receive responses.
An HTTP server must serve HTTP requests from its clients.
Both, the client and server, must follow the protocol of the connector which is HTTP in our case.

8) Another question is what is the relationship between components and connectors on one hand and modules on the other.
Modules are static units which need to be developed, configured, deployed, tested and delivered.
When executed, they exist as components, or connectors.
However, there is not a 1:1 mapping.
It is of course possible that a module is executed as a component which is depicted on the slide as the first option.
It shows that the Public API module is executed as the Public API gateway component which is a standalone Node.JS server-side application.

It is also possible that two or more modules run as one component which is depicted on the slide as the second option.
Here, we have the Backend Service component which is a server-side Java application.

Its code is separated to two independent modules - the Domain module and Data Sources module.
We could be interested as architects in the internal runtime architecture of the backend service.
Here, we could see instances of domain classes and data source classes and their runtime communication.
However, for now we are not interested in this detail.

The third and fourth options on the slide show that we also have components for which we do not have corresponding modules in the module viewpoint.
For completeness, it would be good to have them also as modules which are only configured and deployed.
However, we do not have them as modules for simplicity.
It is enough for us to see them in the C&C view.

9) It is also possible that a module or a set of modules is instantiated in more components.
This slide shows that the two modules Domain and Data Sources are instantiated as three Backend Services.
In addition there is a load balancer component which receives requests from the client applications and distributes them to the backend services to balance the incoming load.

10) The C&C view on this slide shows three backend services which are available only for clients sending requests through the Public API gateway 1 component.
Requests sent through the Public API gateway 2 component are not distributed by a load balancer to different instances of the backend service.
In other words, we guarantee a certain level of the quality of the provided services for a set of clients who can access the system through the first public API gateway.
The other clients have a lower quality level because their requests are not load balanced.
Concretely, the gateway 1 is used by government organizations, the gateway 2 by the public.
In other words, the national open data catalog provides different access qualities for the governmental organizations and for the public.

11) Sometimes, it is useful to show interaction between components ordered in time.
For this, we can use UML sequence diagrams.
A sequence diagram considers instances of classes or modules.
The instantiation is denoted by the colon symbol.
The name of the instantiated class or module is shown at the right of the colon.
In our case, the instances are components.
The sequence diagram shows a timeline for each component..
At the beginning of the timeline, the component is created.
The diagram then shows connectors between instances.

UML sequence diagrams are usually used for showing call-return interaction between objects or services.
On the slide, we have the internal runtime architecture of a part of a backend service component.
Namely, we see here how components comprising the backend service interact among each other.
The displayed components here are Java class instances and they interact using simple object method calls.
We show this interaction as a C&C view because it is important for our architecture.
Developers must adhere to this specification.

Concretely, the C&C view on the slide shows what happens when the search datasets service is called.
It first uses the dataset index to search for datasets using the query provided by the caller.
For each found dataset, it creates an instance of the dataset domain model and adds it to the result set.
It then continues with searching other related datasets with the found dataset.
To search related datasets, it asks the domain model for the characteristic keywords and uses them as a new search query which is sent to the dataset index.
For each returned dataset, a new instance of the dataset domain model is created and added to the result set.
The final result set is returned.

You can think about the instances participating in the interaction as class instances, i.e. objects.
However, this is not necessarily true.
Their internal structure can be more complex and they can be a set of instances of different classes from the internal implementation of the respective modules.
This depends on the software designers and developers.
However, from the architectural point of view, this is the necessary level of detail.
We are not interested in the internal implementation details here.

12) So this was all from this lecture about the component-and-connector viewpoint.
See you in the next lecture.

Lecture 5: Architectural Views - Allocation Viewpoint

1) Welcome to the next lecture about software architectures.
In this short lecture we will finish the classification of architectural viewpoints introduced by the authors Bass, Clements and Kazman by discussing the allocation viewpoint.
The previous two lectures were about module and component-and-connector viewpoint.

We use the module viewpoint to show how our software system is structured into units of code which need to be designed, developed, tested and so on.
And we use the component-and-connector viewpoint to show how the system is structured at runtime into components with runtime behavior and connectors which enable runtime interaction between the components.
Structures shown in module and component-and-connector views are usually somehow related to real-world entities.
These entities can be, e.g. people who develop them or hardware infrastructure where they run.
We use the allocation viewpoint to show these relationships between modules, components and connectors on one hand, and the real world on the other.

2) We distinguish three most typical sub-kinds of the allocation viewpoint.
The first is the deployment viewpoint which allocates runtime components and connectors to hardware elements.
The second is the implementation viewpoint which allocates static modules to file structures.
The third is the work assignment viewpoint which allocates static modules to people.

3) Let’s start with the deployment viewpoint.
Its elements are runtime components and connectors on one hand and hardware and communication elements on the other.
Its relationships show how components and connectors from our component-and-connector views are allocated to hardware and communication elements.

4) Here we have, as a reminder, the component-and-connector view from our running example.
We saw it in the previous lecture.
It shows us basic runtime components and connectors between them.
However, it does not show how they are allocated to hardware and communication elements.

5) The allocation may look like this.
This is an allocation view which shows that the components run on two separate virtual servers which are physically connected by a local area network (LAN).
On the higher network layers we have the TCP/IP protocol on top of which we have HTTP.
The virtual servers are behind a firewall.
The local area network is a higher performance network than the public internet.
We receive requests from the public internet through the firewall.

6) The second sub-kind of the allocation viewpoint is the implementation viewpoint.
It shows how software elements (usually modules) are mapped to file structures in the system’s development, integration, or configuration control environment.

This is important for the management of development activities and build processes.

7) Here we have, as a reminder, the decomposition module view from our running example.

8) We already saw allocation of the modules in our running example in the lecture about the module viewpoint.
However, we did not call it in this way.
On this slide, you have a reminder.
It shows how the sub-modules of the dataset source module from our running example are implemented in the file system.
On the right hand side we have source files.
They are of course organized in a directory tree but omit this detail from the slide.

9) The last sub-kind of the allocation viewpoint is the work assignment viewpoint.
This structure assigns responsibility for implementing and integrating the modules from the module viewpoint to the teams who will carry it out.
This is important for project management decisions.
Here, the elements are modules and teams or people.
Relationships are work assignments between modules and teams or people.

10) This slide shows that the modules are assigned to two teams.
In a real project you would probably engage your imagination and you would give the teams some cool names.
I have simply team 1 and team 2.
The work assignment shown here is a pure theory.
In reality, because we do not have many developers in our team and because the system is not so complex, we have only one developer who does all the work.
But, this would not be a very nice example of the allocation view :-).

11) So this was all from this lecture about the allocation viewpoint.
See you in the next lecture.

Lecture 6: Documenting Software Architecture

1) Welcome to the next lecture about software architectures.
This short lecture shows a possible structure of the documentation of software architecture.
The documentation describes the architecture in the textual and visual form.
It does not have to be a single document.
The best way is to write the documentation in a form of wiki.
You have plenty of options today of how to write a wiki.
A wiki tool on Github will serve well for example.

I will now show you a possible structure of the documentation, not the only one possible.
So rather than remembering the exact structure, you should remember what needs to be described in the documentation.

2) The documentation is structured to sections.
The first section in our structure is the documentation roadmap.
It tells the reader what information is in the documentation and where to find it.
There should be an overview of the views with their name, viewpoint and short description.
It should also specify how stakeholders can use the documentation.
It means which stakeholders and concerns are addressed by the views
The best way is to list the stakeholders in a table.

3) The second section provides an overview of the system.
It should be a short description of the system's context, its functions and stakeholders, known constraints and other important background information.

4) The third section provides details about each designed view.
Section 3.1 is a meta-description of the views.
It specifies how you document views.
It means the description of the template for documenting the views and explanation of the notation for the viewpoints you use in your architectural design.
The explanation should specify what each graphical modeling construct means.
Sections 3.2, 3.3, etc. provide documentation for each designed view.

5) A section for a view starts with a subsection which shows the visual representation of the view.
It shows its important elements and relationships but not necessarily all minor elements and relationships.
If the number of elements and relationships is too big, you can split them into more pictures.

6) The next subsection then provides the catalog of all elements and relationships in the view, even the minor ones which are possibly not in the picture.
It provides details for each element and relationship which cannot be in the picture.
This includes all details we have already discussed in the first lecture about architectural views.
It includes the module's name, description, rationale, interface, responsibilities, behavior, etc.

7) So this was how to document your software architectures.
If you use a wiki-like system to write documentation, you can of course benefit from its features.
For example, each view element may have its own linkable page, etc.

You can try this in your seminar work.
See you in the next lecture.

Lecture 7: Architectural Views - 4+1 Views

1) Welcome to the next lecture about software architectures.
In this lecture we will shortly talk about another approach to classifying architectural views.
It is called 4+1 views.

2) Its name is derived from the fact that it distinguishes 4 architectural viewpoints plus one which describes scenarios of how the users use the system.
The purpose of the application scenarios is not to fully describe the required functionalities of the system.
Its purpose is to validate the designed architecture from the user's point of view.
The 4+1 approach is similar to the approach discussed in the previous lectures.
Its viewpoints are similar to the previous approach.
Moreover, it considers the validation using the application scenarios.
It also proposes to use certain kinds of UML diagrams for the introduced viewpoints as a recommended notation to express the views.

3) The logical viewpoint specifies the system's functionality in terms of structural elements.
It is similar to the module viewpoint.
It shows how the system is decomposed to key functional modules such as classes or services.
However, instead of pieces of code, the modules here represent the system's functionality which is decomposed to smaller and smaller modules.
It also shows their important attributes and provided methods.
To show the modules, it is recommended to use UML package diagrams and class diagrams.
As a supplement it is recommended to use UML state machine diagrams to show possible states of the modules and transitions between the states.

4) The behavior viewpoint shows the system's functionality in terms of runtime processes.
Similarly to the component-and-connector viewpoint, it shows the system's architecture at runtime.
However, it is simpler.
Its elements are processes.
A software system is partitioned to processes where a process is a sequence of tasks performed by functional units from the logical viewpoint.
So instead of instances of modules, you consider tasks performed by the modules.
It is recommended to use UML sequence and activity diagrams for this viewpoint.

5) The development viewpoint shows decomposition of the software system to software elements which need to be developed and tested.
The elements have their public well-defined interface provided to other elements and internal implementation.
We also specify usage relationships between the elements.
This is the equivalent view to the decomposition and usage viewpoints we have discussed earlier.
It can be extended with mapping to teams and implementation in source files so it combines the module viewpoint with the implementation and work assignment viewpoints discussed in the previous lectures.
It is also mapped to the logical modules and runtime processes designed in the logical and behavior viewpoints.
The recommended notation for this viewpoint are UML component diagrams.

6) The last viewpoint is the physical viewpoint.
It assigns functional elements from the logical viewpoint and process elements from the behavior viewpoint to physical hardware and network configurations.
The recommended notation here are UML deployment diagrams.

7) The last viewpoint are the application scenarios.
They are expressed as use cases that describe the behavior of the system as seen by its stakeholders.
A use case here is an important functional requirement expressed as a concrete interaction between users and the system.
It shall be created as the first viewpoint during the analysis of requirements prior to the architectural design.
It is a viewpoint which describes the functions of the system.
It is used to explain the architecture documented by the other viewpoints from the functional point of view.
On each use case you can demonstrate what role each module or component of the architecture plays in the use case.
You can also use it to show that each part of the architecture has its part and role in the system and that the architecture is complete from the functional point of view.

8) So this was another approach to describing software architectures using the 4+1 viewpoints.
See you in the next lecture.

Lecture 8: Quality Attributes

1) Welcome to the next lecture about software architectures.
In the previous lectures, we discussed architectural viewpoints.
Software architects use them to describe their architectural decisions.

By this lecture we will start another part of the course which is about these decisions.
Having a given set of requirements we can design different software architectures each more or less fulfilling the requirements.
The resulting architecture depends on the decisions of the software architect.
The architect decides on the base of requirements of different stakeholders.
In this lecture, we will talk about these requirements and how to express them in the structured way.
In the next lectures we will talk about specific kinds of these requirements and typical architectural decisions software architects make to fulfill them.

2) I talked about requirements.
It is important to distinguish three basic kinds of requirements.

First, we have functional requirements.
These are requirements most people will talk about if you asked them what requirements they have.
A functional requirement describes a feature or function the system shall provide to its users.
It describes how the system should behave in interaction with the users.
It describes what work the system is intended to do.
Functional requirements are satisfied by assigning responsibilities to parts of the software system throughout its design.
As an architect you assign architectural structures and you assign functional responsibilities to these structures.

3) Second, there are quality requirements.
These are requirements most people will not talk about but they have them on their minds.
However, they usually think that it is so clear that you have to know about them and it is therefore not important to talk about them.
A quality requirement describes how well the system should provide its functionalities.
This is too broad, we have to concretize this.
So we may talk about how reliable or available or secure are the functionalities provided by the system.
Or we may talk about the performance of the functionalities, e.g., how many requests of that functionality can be served in a given amount of time.
Or we may talk about how easy it is to modify the functionalities of the system or how they are interoperable with other systems and their functionalities.
And we may also talk about how easy it is to test the functionalities.
These requirements are sometimes called non-functional requirements.
This is because they are not about functionalities.
However, it can be misleading to call something by a name referring to what it is not.
These requirements are about the required qualities of the system and functions it provides.

Therefore, we will call them quality requirements.
Quality requirements are satisfied by various structures a software architect designs into the architecture.
These structures do not have functional responsibilities but they assure the required qualities.
For example, to increase performance, you decide to multiply some structure with functional responsibilities and design a load balancer into the architecture.
You do not add anything functional to the architecture.
You just focus on a certain quality and you try to assure it by additional architectural structures.

4) The last kind of requirements are constraints.
A constraint is a specific kind of requirement because it does not come from the individual stakeholders and it is not possible to negotiate about them or assign them a lower priority.
The functional and quality requirements are tractable.
You can discuss them and negotiate about them with stakeholders, prioritize them and ignore them when they have a low priority or your budget is low.
A constraint is a given fact, usually coming from outside.
They are given, e.g. by natural or legal laws, such as gravity, braking speed or accounting rules.
There is also a specific kind of constraints which we may call system constraints.
For example, there can be an existing legacy system your system has to interact with or a requirement to use a certain type of a database management system because licences were already bought.
From the architectural point of view, these system constraints may be considered as architectural decisions which have already been done and you cannot change them.
So you satisfy such constraint by accepting the given design decision and reconciling it with other affected design decisions.

5) Business considerations determine functions as well as qualities that must be accommodated in the system's architecture.
However, functionality often takes not only the front seat in the development scheme but the only seat.
It may easily happen that stakeholders talk only about their functional requirements, not about their quality requirements.
They do not realize that it is important to talk about quality requirements.
Actually, everyone wants to have a quality software system.
Why would you deliver a system of poor quality?
However, the question is not about quality or a poor quality system.
Everything is the question of time and budget constraints of your project.
Doing a perfect system in all ways would be simply too costly.
Therefore, you have to talk with the stakeholders not only about functions but also about qualities of these functions.

And you have to prioritize them as you prioritize the functional requirements.
Not talking about quality requirements or not considering them is often the reason why software projects fail and why software systems need to be redesigned.
Systems are usually redesigned not because they are functionally deficient but because they are difficult to maintain, port, scale, they are too slow, they are hard to integrate with other systems, etc.
While doing so the resulting replacements are often functionally identical.

6) You can not consider functional and quality requirements as independent or complementary requirements.
They are orthogonal.
You can consider functional requirements as one dimension of the requirements space.
Each functional requirement is a point on this dimension.
Each quality is also a dimension.
On that dimension we specify the required level of that quality for each required functionality.
For some functionalities, the quality is irrelevant.
For others, some basic level is sufficient.
And for some functionalities, the high level of that quality may be required.

7) For example, we may consider the security dimension.
A basic level of security is required for the first functionality.
For the second one, the security is irrelevant.
And for the last, third one, a high level of the security is required.
This is just an abstract model of qualities and their relationship to the functionalities.
In practice, you have to describe what exactly means the basic or high level of security.
We will discuss how such a description may look like.

8) The required functionality may be achieved through the use of different architectural structures.
If functionalities were the only requirements, the system could exist as a single monolithic module with no visible structure at all.
Because we need to achieve different qualities and we need to make visible how they were achieved, we need to have the architecture described and we need to be able to reason about the achieved qualities.
Various architectural structures allow for achieving different qualities.
It is up to the software architect to decide about these structures.
However, when trying to achieve the qualities one needs to know that different qualities can never be achieved in isolation.
One quality influences other qualities.
Achievement of one quality may exclude achievement of other qualities.
For example, almost any quality attribute affects performance.

Qualities such as modifiability or security usually mean introducing additional middle modules and components and separation of functionalities and data to different modules and components.
This requires additional communication between the components at runtime which always cost computing power and time.

9) Software system qualities are so important in the field of software architectures that some authors define software architecture using qualities.
One of such definitions is that a software architecture is a mapping of system's functionality onto software structures.
This mapping determines the architecture's support for qualities.
In the previous definitions at the beginning of the semester we had that the architecture is formed by elements, their external behavior and relationships between the elements.
This is implicit in the new definition based on qualities.
Determination of the support for qualities requires that we know the elements, their behavior and relationships between them.

10) After this introduction to software qualities we are ready to introduce some more formal background for describing quality requirements.
First of all, let me introduce a so-called quality attribute.
A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies a quality requirement of a stakeholder.
It is a measure of goodness of the system or its functionality along some dimension of interest of its stakeholders.

11) A quality attribute may be associated with the system as a whole or with its particular part or functionality.
Usually, it does not make sense to associate it with the whole.
Association with a particular functionality is more common and better way because it is clearer and easier to specify.
Let's have some examples.
Let's have a simple functional requirement shown on the slide.
It is more a step in some usage scenario specifying a user requirement but let's have it for simplicity.
A possible quality attribute is a performance quality attribute.
Reminder - a quality attribute is a measurable or testable property of the system.
In case of performance it is a property related to the performance of the system and to the functional requirement which is the subject of our interest.
The performance quality attribute is: how quickly the dialog will appear.
It is a quality property of the system related to the functional requirement.
And it is easily measurable.
The measure is the time the system needs to show the dialog.
Showing the dialog may require, e.g., loading data from a persistent data store or waiting for a response to a request sent to another component of the system.

By the quality attribute, we say that the time to show the dialog is important for our stakeholders.
Later during acceptance testing, they will concentrate on this property of the system probably.

12) We may consider other quality attributes.
For example, an availability quality attribute which says that for our stakeholders it is important how often the function will fail and how quickly it will be repaired.
It may happen that external data for the dialog can not be load and we have to deal with this somehow so that the expectations of the stakeholders represented by this quality attribute are fulfilled.

13) As another example, we have here also a usability quality attribute which says that for our stakeholders it is important how easy it is for the user to use the function, learn it, locate it or revert it when necessary.
In this case, this is probably easy but it is always not that simple and complex architectural decisions are necessary.
For example, it is easy to enable the user to cancel this operation which means putting a button which cancels the dialog.
In this case, it requires almost no architectural decisions.
However, there can be much more complex operations which simply cannot be cancelled without proper architectural structures.
A popular example of this, even if it is not the one you would probably meet in practice, is firing a nuclear missile.

14) How do we achieve quality attributes?
We have said that a software architect achieves them by architectural decisions which incorporate architectural structures to the structure of the software.
However, quality is not related only to the software architecture.
It is necessary to consider quality attributes throughout all phases of the software process, i.e. analysis, design, implementation, and deployment.
No quality attribute is entirely dependent on a single phase exclusively.
Quality attributes may involve architectural but also non-architectural aspects.
Let’s see some examples.

15) The first example is performance.
Performance is a measure of how long it takes the system to respond to requests.
It involves architectural and non-architectural aspects.

For example, it depends on how much communication among components is necessary.
This is the architectural aspect.
The more components with network connectors you have in your architecture, the more serialization and deserialization of data structures is necessary.
This consumes resources and lowers performance.

Performance also depends on what functionality has been allocated to each component.
This is also the architectural aspect.
The more functionality on a single component you have, the less resources can be used for a single request which increases the time to respond to the request.

Moreover, performance depends on where components are allocated.
This is especially true for shared components.
Simply speaking, a shared component, e.g. a database, allocated on a weak virtual does not help with performance.
This is, of course, architectural.

Algorithms chosen to implement selected functionality also influence performance.
However, the selection of an algorithm is not architectural.
It is the matter of the internal implementation of a module and it is, therefore, out of the scope of the architectural decisions.

And it does not depend only on which algorithm is chosen but also how it is coded.
An algorithm with theoretical linear time complexity can be coded with much worse complexity.
This is also the matter of the internal implementation of the module and, therefore, it is not architectural.

16) Another example is modifiability.
It is a measure of how easily a system or its part can be extended, removed or changed.
It is determined by different architectural as well as non-architectural aspects.

For example, it depends on how functionality and other responsibilities are divided among different modules.
The more they are separated among separate modules the easier it is to identify where the source code needs to be changed and how changes at one place of the code influence other parts.
If you merge domain logic or application logic with infrastructural logic such as database access, messaging or network communication, the harder it is to change the code.
This is a pure architectural aspect as you influence the separation of concerns among different code modules.

On the other hand, modifiability is also influenced by coding techniques.
The simplest technique is naming notation.
If each module uses a different naming notation, it is harder for the developers to orientate in the code and maintain it.
In general, it is always possible to make a system difficult to modify by writing obscure code.

Coding techniques are the decision of the development team and it is out of scope of decisions about architectural structures.

17) The last example is usability.
Do not confuse it with user experience which is a broader concept which involves usability but also other quality attributes, e.g. availability, as well as other concepts which are not related to quality attributes.
Usability is a measure of how easily a user can accomplish a desired task.
Of course, the prerequisite is that the system meets all functional requirements.
Even usability involves both architectural and non-architectural aspects.

For example, making the user interface clear and easy to use is non-architectural.
It belongs to the area of user interface design and involves decisions such as:
"Should you provide a radio button or a check box?"
"What screen layout is more intuitive?"

On the other hand, the ability to cancel or undo performed operations, or to re-use data entered previously is architectural.
These usability features involve the cooperation of multiple elements in your system and are, therefore, architectural.

18) On one hand, software architecture is critical to realization of many qualities of interest in the system
These qualities should be designed in and can be evaluated at the architectural level.
On the other hand the software architecture, by itself, is unable to achieve qualities.
It provides the foundation for achieving qualities and reasoning about them.
However, the attention must be also paid to the non-architectural details.
For example, as we have seen, low-quality code may destroy even the best efforts of a software architect.

19) In the next lectures we will discuss different quality attributes and how you can address them as a software architect.
However, we will discuss only some of the attributes.
This slide shows the basic categories of quality attributes which may be considered.

20) System quality attributes are the most typical ones.
A system quality attribute is related to the quality of the software solution itself.
We could also call them "hard" quality attributes as they are the core attributes from the viewpoint of software architects.

21) As a special kind of system quality attributes we distinguish run-time quality attributes.
They are related to runtime quality properties of the system, i.e. how well the system behaves at runtime.
This includes its availability, performance, security or its usability.

22) Another special kind of system quality attributes are design-time quality attributes.
They are related to qualities of the system which are more important at design-time of the system.
This includes, for example, its modifiability, portability, testability, reusability or integrability.
The term design-time is too narrow as this usually includes not only the design of the system but also its development or testing.
We need the system to have these properties when we design, develop or test its parts and we are interested in how easy this work will be.
In practice it is not so easy to distinguish whether the attribute is run-time or design-time.
For example, modifiability is shown as the design-time quality but if we consider user's customizations by users, which is also a kind of modification, we usually think about it at run-time.
So do not understand that the classification is a strict prescription which requires an attribute to be classified somewhere exactly.
Just think about the classification as a tool which helps you to think about different kinds of qualities and not to forget to think about something important.

23) Business quality attributes could be called "soft" quality attributes.
They are not related to the quality of the software solution itself.
They are related to how well they fit to the business environment in which they exist.
For example, from the business point of view, the time to deliver the system to the market can be very important.
No matter how cool your architecture is from the point of view of the hard, system quality attributes, when there is a risk that the company will be beaten by its business rivals because it would take too long to build the system.
It may also happen that the cost-benefit analysis shows that the system is more expensive than the financial benefits which will be received by having the system.
Isn't this because of overly complex software architecture?
Other business qualities may be considered, e.g. whether the architecture allows for reusing legacy, i.e. existing old software, or whether its parts can be outsourced, either development or later support.

This course is not about these soft business quality attributes.
I just wanted to show you that they exist and that you will be probably required to reflect them in practice.

24) Architectural quality attributes could be called meta-attributes.
They are related to the quality of the architectural design and quality of its documentation.
They are more important for the software architect and the development team rather than for the customer and users.
Basic meta-attributes are correctness and completeness of the architectural description.

Correctness means that there are no mistakes or contradictions in the architecture.
Completeness means that it covers all requirements, i.e. functional and quality requirements and constraints.
They are very hard to prove formally.
Completeness can be shown by mapping the requirements to the architectural elements.
Correctness can be shown only indirectly unless you want to get into formal software engineering methods which is very costly.
Indirectly means for example that you let senior software architects or at least other software architects to review your architectural description.
Another meta-attributes can be buildability.
A software architecture is buildable whether it allows the system to be completed by the available team in the available time and within the available budget.
The last one mentioned on the slide is conceptual integrity
It means that there exists an underlying theme or vision that unifies the architecture at all its parts and levels.
A simple example of this is whether some prescribed classification of architectural viewpoints is followed and a unified visual notation is used by all people involved in the architectural design.

25) The question now is how does quality attributes help us to specify quality requirements.
They help us but there are two important points.
First, quality attributes themselves do not specify quality requirements.
They are just measures which can help us to specify the requirements.
What I mean is that it is meaningless to say that a system will be modifiable or available.
It even does not make sense to say that a particular part or function of the system will be modifiable or available.
Every system and each its part is somehow modifiable with respect to one set of changes or available with respect to one set of system failures.
However, it is not modifiable or available with respect to another set.

The second point is that when specifying a quality requirement, the focus is often on which quality attribute a particular aspect belongs to.
For example, is a system failure an aspect of availability, security or usability?
All three “attribute communities” would claim ownership of a system failure.
Different attribute communities have developed their own vocabularies.
Performance community has “events”.
Security community has “attacks”.
Availability community has “failures”.
Usability community has “user input”.

26) These are the reasons why we need a tool to specify quality requirements.
In this course we will use a tool which is a predefined structure of a quality requirement.

We will reuse this structure to specify different kinds of quality requirements which belong to different quality attributes such as performance on one hand and testability on the other.
The structure is shown on the slide.
We call it quality attribute requirement scenario or simply quality requirement scenario.
It is common to describe functional requirements using scenarios.
So we reuse this concept here for specification of quality requirements.

A scenario starts with a stimulus.
It is a condition that requires a response when it arrives at a system.
A source of stimulus is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
The stimulus stimulates an artifact.
This may be the whole system, or some piece or pieces of it.

The stimulus occurs under certain conditions.
The system may be in an overload condition or in normal operation, or some other relevant state of the environment surrounding the artifact.

Then there is a response which is the activity undertaken by the system as the result of the arrival of the stimulus.
It is the required reaction of the system when the stimulus stimulates the artifact.
When the response occurs, it should be measurable in some fashion so that the requirement can be tested.
The way of measuring is specified by the last part of the scenario which is called measure.

27) So this is the end of the lecture about software system qualities, quality attributes, quality requirements and their specification.
I will leave you with the generic concept of quality requirements scenarios without concrete examples.
You will have examples in the next lectures where we will discuss concrete quality attributes, how to specify requirements for these attributes and what decisions you can make as software architects to fulfill these requirements.

Lecture 9: Quality Attributes - Availability

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the availability quality attribute.
In the previous lecture, we introduced the quality attributes in general and we saw how a quality requirement can be specified.
In this lecture, we will concretize it for the availability quality attribute.

2) Availability is usually considered as a reliability of the system or its part and its ability to recover from faults.
In other words, focusing on availability means that you try to minimize faults.
Moreover, you know that there will always be some faults and you try to minimize the impact of these faults to other parts of the system, other systems and to users.
Which means that you also focus on the capability of the system to recover itself from faults.

3) Availability refers to a property of a software system or its part that it is there and ready to carry out its task when it is needed.
What does that mean exactly?
We should ask how it can happen that the system or its part is not able to carry out its tasks when it is needed.
The last part is important.
I said: when it is needed.
In other words, it is not a problem when the system or its part is not able to carry out its tasks when it is not needed.
Even if it is switched off, we do not care.
Speaking about availability is meaningful only for the times when it is needed.

So what does it mean that the system is needed but it is not able to carry out its task?
It means two things.
First, it means that something bad happened in the system.
This may be a problem with hardware such as disk or network.
Or it may be a problem with its software part other parts depend on.
It may also be a problem with another software system our software depends on.

Second, it means that the system was not able to mask or recover from this problem and a user, another part of the system, or another system noticed the problem which caused them their own problem.

The system remains available when a problem appears but it is masked by the system or the system recovers itself from that problem.
So the availability refers to the ability of the system to mask or repair problems.

4) We can also say that availability of a system or its part is a probability that the cumulative service outage period does not exceed a required value over a specified time interval.
The probability can be expressed as ratio of the mean time to failure to the mean time to failure plus the mean time to repair.
This is taken on average.
So the availability of 99 % means that the system works 99 days and then it fails and 1 day is necessary to repair the system on average.

5) We need to specify what is a failure.
   A system failure occurs when the system no longer delivers a service that is consistent with its specification and which is observable by users or other systems.
   So failure is something observable.

   There are also faults.
   A fault is a problem in the system which occurred but is not observable.
   Fault occurrence is not an availability problem.
   A fault is an availability problem when it becomes a failure.
   A situation when a fault causes a sequence of other faults which in the end become failure is also an availability problem.

6) There are two basic kinds of techniques we can use for increasing availability.
   The first is fault recovery.
   It means that faults are acceptable but we keep them from becoming faults.
   In other words, we keep them from becoming visible.

   The second is fault repair.
   It means that we modify the system so that a fault will not appear again.
   We will discuss more concrete techniques later in this lecture.

7) Let's now talk about specification of availability requirements.
   In our previous lecture we introduced the concept of the structured scenario which you can see on the slide.
   We will use it to specify availability requirements in a structured way.

8) The artifact is a component that is required to be available.
   When the component fails it causes problems to other components of the system, other systems or human users and may therefore possibly become a failure.

9) The source of stimulus is someone or something which observes the fault.
   It is the source of fault not because it fails - that is the artifact.
   It is the source of fault because it observes it.
   And the fault actually does not exist until someone or something notices that there is a problem.
   The problem becomes known when the source of stimulus notices.
   The source of stimulus may be internal or external to the system.
   An internal source of stimulus is always a component of the system.
   An external source of stimulus is a human user or another system.
   Scenarios with an internal source of stimulus specify a required reaction of the system to a fault which has not yet become failure.
   Scenarios with an external source of stimulus specify a required reaction to a failure because the problem has already become visible.

10) The stimulus is an observation of the fault.
We consider 4 basic types of stimuluses.
Omission appears when a component does not respond.
Crash appears when a component repeatedly suffers omissions.
Incorrect timing is that a component responds but early or late.
Incorrect response is that a component responds, it responds in time but the response is not correct.
This may include not only incorrect values but also incorrect format or shape of the response.

11) In the environment part of the scenario we specify conditions of the artifact and its surrounding environment under which the fault and its observation is considered.
Usually, we specify the run-time state of the environment such as we are at the system's startup, shutdown, normal operation or overloaded operation.
We can also distinguish whether this is a first fault of any kind in the system or whether there has been another fault already.
In each environment state the system reaction may differ which means that we need different scenarios.

12) The response specifies the reaction of the system to the failure.
The system can mask the fault or try to recover from the fault.
This can also include supportive actions such as logging the fault, notification of appropriate entities (people or systems), disabling the artifact causing the fault or operating in the degraded mode while a repair is performed.
We will talk about this later in the lecture.

13) The response measure specifies how the fault and its repair are measured.
For example, we can specify how often the fault can appear using the probabilistic definition of availability discussed earlier.
I.e. we can specify that the system availability regarding this scenario must be 99.99 %.
We can complement this measure by specifying the time or time interval when the fault is not acceptable.
For example we may say that unavailability of the system described by the scenario is not possible in working hours.
So we may have a scenario for working hours and another scenario for non-working hours.
We can also specify the time required to detect the fault, time to repair the fault or time in which the system or the artifact can be in the degraded mode or down.

14) A scenario should describe a story.
So here we have the story described by an availability scenario.
It shows us how to read the scenario.

15) This slide shows more concrete examples.

We have a simple system whose runtime architecture consists of a central database, several components in the application layer each responsible for some business function.
On top of the business layer, we have the presentation layer which consists of a single component which requests the application components and presents their responses to human users.
Each component at the application layer either reads data from the database or writes data to the database.
Moreover, a reading component reads from the database either for transactional purposes or for analytical purposes.
Transactional read means that the component reading functionality is a part of an interaction with human users so its response needs to be available in some reasonable time so that the users do not have to wait too long.
Analytical read means that the component reading functionality is a part of preparing an analytical report for human users which do not need the report immediately but sometimes in the future, e.g. each business day in the morning.

The first scenario on the slide shows a component A which needs to write data from the internal database.
So the component A, as a source of stimulus, wants the write service from the internal database which is the artifact.
The database, being in a normal operation, fails in providing the service.
It means that it is not possible to write the data.
This fault is an omission which means that the previous write attempts were successful.
The component A observes this fault.
This observation stimulates the system to do something which is specified in the response.
The response specification says that the system must mask this fault so that it does not become a failure by postponing the write and as a supportive action the fault is logged.
The scenario is satisfied when measuring the response corresponds to the specified measures.
There are two measures.
First, there is no downtime which means that the component A continues in its work without any delay.
Second, the data is written to the database within the next 10 minutes.

The other two scenarios are similar.
They are for reading components.
The second scenario is for the transactional read.
It specifies that in case of problems the read is repeated with 5 seconds downtime allowed.
It means that in case of reading problems, the component B may be delayed up to 5 seconds.

The third one is for the analytical read.
It specifies that in case of problems the read can be repeated but the analytical report produced by component C must be available till 6am on the next business day.

16) This slide is a reminder of the component-and-connector viewpoint of our running example.

17) This is an availability scenario related to the Solr search engine.
It specifies how the system shall react when the search is not available for the first time.
It shows that it is expected that the search unavailability is masked without any downtime.
The solution can be, e.g., trying an additional search attempt.

18) This scenario shows what should happen when the search omission appeared 2 times.
It specifies that in that case it is a crash and the reaction of the system is different.
The required reaction is that the crash shall be masked but a 10s downtime is possible.
This gives you some space as a software architect for the decisions you can make.
We will discuss possible decisions later in this lecture.

19) This scenario is similar to the previous one.
Only the allowed downtime is 1 second instead of 10 seconds.
This gives you a smaller space for your decisions.

20) Having the requirements specified, the software architect needs to decide how to fulfill them.
This means that he or she implements new architectural structures into the architecture, either to the module viewpoint, component-and-connector viewpoint or to the allocation viewpoint.
There exist various practices documented in the literature for software architects.
We will call them architectural tactics.
We will discuss some basic tactics for each quality attribute.
It will be an overview without advanced details.
If you are interested in details, you will need specialized literature or you can also take courses at our faculty which focus on concrete quality attributes.

For fulfilling availability scenarios, we usually use different tactics to achieve two goals.
The first goal is to mask the fault.
It means that we ensure that the fault does not become a failure.
In other words, we ensure that it is not observed by a human user or another system.

The second goal is to allow the fault to be repaired so that it no longer appears.

21) To achieve the goals you can use various tactics.
We will classify them to three classes.

The first class are tactics to detect faults.
Without a fault detection you are not able to ensure the reaction of the system specified in the scenarios.
The detection can be done by the component which acts as a source of stimulus in a scenario.
Basically, you can decide as an architect that when an exception is received by the component, the fault is detected.
You can also extend the logic by the component with a fault detection logic according to some availability tactic.
But this is not always enough.
Sometimes you need a systematic detection which is periodical and does not depend on the time when the service of the artifact is actually required.
In that case, you introduce new architectural structures which take this responsibility.
Another reason for introducing new structures is that you do not want to mess the logic of the component with the fault detection logic.
Also when the same fault detection logic would need to be implemented by different components, it can be better to separate the logic to a new architectural structure.

22) The simplest fault detection tactic is ping/echo where one component asynchronously pings another component and awaits echo in some defined amount of time.
This defined time threshold tells the pinging component how long to wait for the echo before considering the pinged component to have failed.
When there are more components which need to mutually check, it may be beneficial to organize fault detection in a hierarchy to reduce the communication bandwidth.
Ping/echo can be used either to check whether a component lives but also to determine reachability and the round-trip delay through an associated network path.
The logic of ping/echo requires that both sides need to implement the necessary logic.
The monitored component must be able to be pinged and send echo.
The other component must be able to send ping and receive echo.
This depends on the nature of components.
For example, if they are network components communicating using the HTTP protocol, you can use the HEAD method.
On the internet protocol (IP), you can use the Internet Control Message Protocol (ICMP) which is used to send control messages to network components.
Basically, ping/echo originated in network environments.
If your components communicate using some other protocols, either on the local network or as processes on a single machine, you need to use specific ping/echo mechanisms.

23) This slide shows an example of the ping/echo tactic.
We implement the ping/echo logic to the backend service component of the national open data catalog.
The backend service needs a logic to send the ping request and when the echo response from the Solr is not received to do some action.

24) This slide provides a deeper detail.
It shows that this logic is only for the search datasets component.
Only one module (from to module viewpoint) uses the Solr search logic so the ping/echo logic is not repeated.
From this point of view, it is OK to implement it in the Search Datasets module.

25) However, the previous architecture means that the logic related to pinging Solr and performing some actions related to its availability complicates the logic of the backend service.
Even though the Solr availability logic will not be duplicated, the backend service logic should be a business logic, not a logic which resolves the availability issues.
If this is a problem for us, we can use the solution shown on this slide where the Solr availability logic is separated to another component called Solr Monitor which monitors the Solr instance (or instances) and performs necessary availability actions.

26) Let’s discuss other availability tactics.
The next one is heartbeat.
It is a fault detection mechanism that similarly to ping/echo employs a periodic message exchange between two components.
One component emits heartbeat messages periodically and another component listens to them.
The big difference between heartbeat and ping/echo is who holds the responsibility for initiating the health check.
Moreover, we usually do not consider that ping/echo messages carry data.
On the other hand, heartbeat messages may carry data, e.g., about the health of the component being monitored.
Here it is the component itself.

27) As we saw in the example of the ping/echo tactic, we can separate the availability monitoring logic to a separate module.
Moreover, it can also run as a separate component which is independent of other components who act as sources of stimuluses in the availability scenarios.
This can be applied not only for the ping/echo tactic but also for heartbeat as well as for other tactics.
We refer to this separate module or component as a monitor.
A monitor is a component that is used to monitor the state of health of various other components and also hardware parts of the system.
A system monitor can detect failure or congestion in the network or other shared resources.
It orchestrates software using other availability tactics to detect malfunctioning components.
You introduce a monitor or monitors to your system when you need to separate the monitoring actions and the follow up availability solutions from the other logics in your system, such as the business domain logic, application logic or database access logic.

28) This slide shows how a system monitor could look like in the component-and-connector architecture of our running example.
It monitors external supporting services - Solr and CouchDb - with the ping/echo tactic and the backend service notifies the monitor with heartbeats which can also carry some health status data.

29) The ping/echo and heartbeat tactics are useful for monitoring whether a component is alive or for monitoring its health status.
As we have seen at the beginning of the lecture there are also other kinds of faults.
A component may be alive and produce its outputs.
However, the messages or responses from the component may come in incorrect order or at incorrect time.
This is common in architectures with a shared persistent data store where different components may incorrectly rewrite data stored by other components.

The most simple but working solution are timestamps.
Each message or response emitted by the component is annotated by a timestamp.
The receiving component then implements the logic which recognizes whether the incoming messages are in the correct order and at the correct time.
If only the order is important, timestamps may be just sequence integers.
If time is important, you need real timestamps.
Again, the logic can be separated to a monitor.

Timestamps are important in distributed systems which consist either from different processes or different network nodes.
In these systems components exchange messages with each other to achieve a common goal.
The message exchange heavily depends on the correct order or timing of the messages.
When the components are on the same local machine, we can order the messages, or events in general, based on the local clock of the system.
In a network environment where components run on different machines, we need different approaches which enable ordering events from different network nodes.
For this we have various logical clock algorithms such as Lamport timestamps.

We also have a simpler tactic called timeout.
It is a tactic that raises an exception when a component detects that another component has failed to meet its timing constraints.
For example, a component awaiting a response from another component can raise an exception if the wait time exceeds a certain value.

30) Another kind of fault is when a component responds, it responds at a correct time but the message is incorrect.
A simple example is when the component sends a message formatted incorrectly.

This is however a problem of interoperability which is another quality attribute.
In the context of availability, another problem is usually considered.
You have a component which, e.g., measures something in the real world.
Let's consider a thermometer which measures a temperature.
It may happen that in a certain time its measures are incorrect.
This is a serious fault.
When the thermometer stops sending measures, you can recognize it easily.
However, recognizing incorrect values is not so easy.
You may have a monitor which has some rule that an outside temperature above 45 degrees of Celsius is incorrect.

If you need a higher precision of recognizing such faults you need the voting tactic.
Voting means that you have more redundant thermometers.
The most common are three redundant components with the same functionality.
This is called Triple Modular Redundancy.
The component which reads the responses is called a voter.
It receives results and applies a voting strategy to detect any inconsistency and report fault.
Usually, the voting strategy is either the majority rule or computing a simple average.
There are three different strategies to implement the redundancy.
Replication means that the components are exact clones of each other.
This is sufficient for hardware faults such as thermometers.

Functional redundancy means that the components have the same public interface but differ in their design and private implementations.
This is useful for software faults which may be caused not by a hardware problem but because the problem was incorrectly designed or implemented.
So you have different implementations of the same problem.
The components receive the same input and should produce the same output but the output is produced by different implementations.

Analytic redundancy means that components have not only different public interfaces but also different private implementations.
So the components receive different inputs and may produce different outputs of different kinds.
For example, this is applied for altitude measurements in aircrafts.
You can measure the altitude using barometric pressure, radar altimeter, GPS, or geometrically using the straight-line distance and look-down angle of a point ahead of the ground.
For different measurements you have different components.
The voter here needs to be much more sophisticated than just the majority rule or computing the average.

31) So this was an overview of fault detection tactics.

Once a fault is detected, the system needs to react somehow to fulfill the availability scenarios.
The scenarios describe not only what should be the observable reaction of the system but they also restrict the time in which the reaction shall happen.
This restricts your possible architectural decisions.
We distinguish two categories of fault recovery tactics.
The first category consists of preparation and repair tactics.
They are based on a variety of combinations of retrying a computation or introducing redundancy.
The other consists of reintroduction tactics.
They are based on reintroducing a failed but rehabilitated component back into normal operation.

32) Let’s start with the preparation and repair tactics.
The most typical is introducing a redundancy.
You prepare for faults by preparing backup copies of a component which needs to be available.
When it fails, you repair the fault by introducing a backup copy.
We distinguish three sub-kinds of redundancy.
The first one is so-called active redundancy, also called hot spare.
Here, all redundant components perform the same tasks on the same inputs.
Outputs from one of them are accepted and when it fails, necessary outputs are ready on the backups so the system can easily switch on one of the backups.
The switch can be performed by the monitor or some other component.
This redundancy can be extended from the components also to the connectors to these components.
In other words, each copy has its own connector, e.g. each redundant component is put on a different communication path.

The next kind of redundancy is so-called stand-by redundancy, also called warm spare.
Here, one component performs the tasks and informs others periodically about state updates.
The other components are active but do not perform the tasks.
They are called warm spares.
When the main component fails, one of the spares must be selected to replace the main component.
When it has not updated its state yet, it needs to be updated first.
The passive redundancy provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy tactic and the less available but significantly less complex cold spare tactic (which is also significantly cheaper).

Cold sparing refers to a configuration where the redundant spares remain out of service until a fault on the main component occurs, at which point a power-on-reset procedure is initiated on a redundant spare.
This of course requires some time and it is not suitable for scenarios here high availability is required.
We can use it for high reliability scenarios where the functionality is required but the time restrictions are not so strict.

33) There are also other preparation and repair tactics.
Let’s discuss some of them.
The rollback tactic permits the system to revert to a previous known good state upon the detection of a fault.
One possibility is to create a copy of a previous good state, so called checkpoint.
Checkpoints can be created at regular intervals or at significant times in the processing (e.g., after completion of a complex task).
Reverting means that the system is reverted back to a state recorded in the checkpoint.

However, creating a checkpoint may be expensive or even impossible.
The rollback based on restoring the previous state is not possible in a distributed environment.
For example if you build your system as a set of small distributed components, which we usually call services, it may not be possible to restore their state back in time.
In that case, there is another approach called sagas.
We have a process consisting of actions performed on distributed components.
Each action performed on a component is a small local transaction which is associated with a compensation transaction.
When a fault appears somewhere in the process, the performed actions are reverted by executing the compensation transactions.

34) A very simple fault recovery tactic is to simply retry an operation.
It assumes that an operation fault is transient and retrying the operation may lead to success.
It is used in networks or other environments where faults are expected and common.
There should be a limit on the number of retries that are attempted before a more serious fault requiring a more complex solution is declared.

Another simple solution is to ignore faulty behavior.
It means that we ignore messages sent from a particular component.
Often we apply this tactic when we determine that those messages are spurious.

For example, when an external component launches a denial-of-service attack, we would like to ignore its messages.
For this, we usually apply an access control list filter solution.

Degradation tactic maintains the most critical system functions in the presence of component faults, dropping less critical functions.

Reconfiguration tactic attempts to recover from component failures by reassigning responsibilities to the (potentially restricted) resources left functioning, while maintaining as much functionality as possible.

35) These were tactics which we denote as preparation and repair tactics.
There is also another category of fault recovery tactics called reintroduction tactics.
They are supporting tactics which help with recovering a failed component.

36) The shadow tactic refers to operating a previously failed component in a “shadow mode” for a predefined duration of time prior to reverting the component back to an active role.
In the shadow mode its internal behavior can be modified so that the component is able to function.
For example, when a component depending on an external data source is not able to function correctly because the external data source is not responding, the component can work in a shadow mode based on the cached content of the external data source.

The state resynchronization tactic supports switching from a failed component to its backup component.
It helps with checking that backups are in the state synchronized with the main component by comparing the states by comparing data samples or checksums.
When they differ and synchronization is necessary in a given moment, resynchronization is executed.

Restarting is a simple reintroduction tactic.
A component, e.g. a process or a whole network node may be reintroduced by restarting it.
In practice, more different restart levels may be defined.
This is called escalating restart.
There can be a level where only unprotected memory is freed, another level where all memory is freed and another where the whole system or its part is switched off and restarted with reloading and reinitializing its executable image.

37) So we discussed tactics to detect faults and to recover from the detected faults.
As a software architect you can expect that components fail sometimes and decide that you will prevent faults.

Removal from service means a preventive restart or reconfiguration of a component in order to scrub latent faults, e.g. memory leaks, before the accumulation of faults affects the services of the system which could result in a system failure.

Transactions is another tactic which ensures that asynchronous messages exchanged between distributed components have ACID properties.
It means that a transaction consisting of a sequence of various messages is atomic, consistent, isolated and durable.
These properties are the basic concepts of the theory of transactions.
They come from the world of relational databases but were generalized for the purposes of distributed systems.
In this world, we have communication protocols which ensure these properties and realize this tactic.
The most well-known is the two-phase commit protocol
operations in the system are executed in transactions which ensure ACID properties
exception prevention

A predictive model is usually implemented in a monitor.
It helps the monitor to evaluate the state of health of a component.
It records its outputs and tries to predict the probability of a future fault.

38) Another prevention tactic is based on chaos engineering.
According to its definition available at https://principlesofchaos.org/, chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production.
The point is that you can simulate turbulent conditions and make it part of the system or integration testing.
A well-known practice are so called chaos monkeys.
As explained on https://netflix.github.io/chaosmonkey/, a chaos Monkey is responsible for randomly terminating instances in production to ensure that engineers implement their services to be resilient to instance faults.
So you can test how well the components of a system are ready for unavailability or other faults of other components in the system.
It allows you to do additional architectural decisions when problems are revealed by this kind of testing.
Using chaos engineering, you can introduce various chaos events such as switching instances or whole subsystems of, introducing different kinds of behavior, introducing incorrect data into message flows, killing message flows, etc.

39) So this is the end of the lecture about the availability quality attribute.
The next lecture will be about another quality attribute called modifiability.

Lecture 10: Quality Attributes - Modifiability

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the modifiability quality attribute.

2) Modifiability is about changes.
It is said that change is the only constant in the universe and this is also true for software.
Change is not only constant but ubiquitous in the software lifecycle.
You need to extend the system's features, modify them or retire old ones.
You need to fix defects, tighten security or improve performance.
Changes also happen to enhance the user’s experience or to embrace new technologies, platforms or standards.
And often it is necessary to integrate the system with other systems, i.e. make the systems work together.

3) Our interest in modifiability as software architects centers on the cost and risk of making changes.

4) Architects design their software architecture for modifiability.
To plan for the modifiability, an architect has to consider 4 questions.

First, you have to ask what can change.
You need to consider any aspect of the system.
You need to consider its functions or features you will need to add, retire or modify.
You need to consider the platform on which the system operates.
This includes a hardware platform, operating system and middleware or the fact whether you run the system on your own or on a platform as a service (PaaS).
You also need to consider the other systems in the system’s environment with which it must interoperate through various protocols.
You also need to consider the qualities of the system which can also be required to change (e.g., performance, availability or security).
And, last but not least, you need to consider the information or domain model in your system which is also the subject to change.
It is not possible to anticipate each exact change.
You do not know in advance what new functions will be required and which will be retired in the future.
You cannot anticipate technological changes, etc.
What you try is to identify future possible changes from the information you currently have from your stakeholders, from your technical knowledge or from the state of your competitors.
The better you know your environment the better you can estimate future changes.
But this ability will never be perfect of course.

5) When you know possible changes, you ask the second question.
What is the likelihood of the change?
As I said, you will never be able to anticipate all possible changes.
Moreover, you even cannot plan the system for all the changes you can anticipate.
The system would never be done, it would be too expensive and it would suffer quality problems in other quality dimensions.
So as an architect you not only try to anticipate changes but you also need to estimate the likelihood of these changes and decide which will be supported in your architecture and which won't.
Of course, any change in the future can be somehow implemented in the system.
The question is how your architecture is ready for this kind of change, how much it will cost to change the system and what is the risk of this change.

6) The third question is when is the change made and who makes it?
The first thing which comes to mind is that a change is made to the source code and that a developer makes this change at the design time where the necessary modifications to the source code are designed, implemented, tested and then deployed as a new release.
However, this is not always true.
Also a system administrator or even a user can do some changes to the system at runtime.

7) The last question then is what is the cost of the change.
Making a system more modifiable involves two types of cost.
The first is the cost of introducing the mechanism to make the system more modifiable.
The second is the cost of making the modification using that mechanism.

8) On one side of the spectrum, we have the simplest mechanism for making changes.
We wait for a change request to come in, then we analyse and design necessary changes in the system, change the source code, we perform tests and deploy the new release.
Actually, there is no sophisticated mechanism.
We just develop software.
The cost of introducing this mechanism is zero.
The cost of making the modification using that mechanism is the cost of all necessary steps from the change analysis to the deployment.

9) At the other end of the spectrum is an application generator.
This makes sense, e.g., in cases of information systems with a data source logic, application logic and presentation logic where the application logic is simple.
A simple application logic is, e.g., an application logic which does only CRUD operations on top of the data source.
CRUD stands form create, retrieve, update and delete of data objects from and to the data source.

Here, the low-code approach can be applied.
It requires you to write only a piece of code and the rest is generated for you.
For example, using tools such as the open source openXava requires you to write your domain model as plain old Java objects and provide several semantic annotations of their properties.
The tools then generate a web application ready for production.
Of course, you cannot get an application with a rich logic.
But these platforms are able to generate a reasonable user interface enabling common users to edit the domain model.
Some software engineers do not like the low code approach.
They argue, e.g., that the generated code is of a low quality and, therefore, hard to extend and integrate with other code.
And really, when this is necessary it may be not a good idea to use the low code approach.

Another case when the low code approach is applied nowadays is application integration.
There are application integration platforms such as Zapier which enable you to integrate the information flow between different applications and services.
These platforms support various web application services such as Google Sheets, Gmail, Slack, Trello, Shopify, various calendar services, social networks, content management systems etc.
You can use these platforms to define an information flow through different applications.
Various information flow actions can be triggered on the base of different kinds of events supported by the platform.
The power of the platform is its ability to integrate many different applications.
Actually, it does not generate any code for you, it is a platform where the integration code is hidden from the user.
This kind of platform is sometimes called no-code platforms.

From our point of view of estimating the cost of change, such platforms are a mechanism for making our system more modifiable.
The cost of introducing such a platform is the cost of developing or buying that platform, its eventual deployment and its learning curve.
Today, various open source low code platforms are available.
In that case the cost is the cost of learning to use this platform which does not need to be high.
The other part of the cost is the cost of making changes using the mechanism.
In case of a low code platform, making changes means changing a small trivial code, e.g. a few plain old Java objects and their annotations.
This has very low costs.
However, when it is necessary to change the system in a way which is not supported by the platform, the costs can be extremely high.
In the extreme you may need to throw away everything and start from the scratch.

10) If you develop a software with a richer application logic, you cannot use application generators, at least not for the whole system.
However, in practice we usually do not stay at the other side of the spectrum.
We try to automate as many software engineering steps as possible and we try to make changes as first class citizens in our software development workflow.
Simply speaking, we count with changes and we expect that they will occur often so we will have to modify and release the system frequently.
That is why we automate software testing, code merging, integration and software releasing.
Some projects even release several times a day.
This includes various mechanisms and techniques collectively referred to as devops.
We still develop code, but we try to automate as many development steps as possible to be able to implement changes often and release frequently.
This is however more a software engineering practice than just a software architecting practice.
From our point of view of estimating the cost of change, devops and similar are a mechanism which has some costs at the beginning.
Even toolchains available for devops are for free, your people have to learn them.
You also need to maintain your toolchain.
But, it saves you the actual effort to make the changes as many routine steps are automated.

11) So this was an introduction to the modifiability quality attribute.
Let’s now proceed to modifiability scenarios.
We will use the same structure of scenarios but its individual parts will have a different meaning.
We will read the scenario as follows.
A source of stimulus needs to change some artifact.
The change to be made is the stimulus.
The system itself or the project team ensures a response which is that the change is made without affecting other artifacts and the result is tested and deployed.
The required parameters of the change are measured and meet given constraints.

12) The artifact in the scenario is a module or component which needs to be changed.
In some cases, the artifact may be something which is external to the software architecture itself, such as the platform on which the system runs or an external software system.

13) The source of stimulus is the one who makes the change.
It may be a developer but also a system administrator or end user.

14) The stimulus is the change to be made.
It can be the addition of a new function or modification or deletion of an existing function.

It can be also a change made to the qualities of the system, e.g. increasing its availability, etc.
There can also be changes in platforms and technologies.

15) The environment specifies when the change can be made.
It can be at design time or runtime.

16) The response to the stimulus is usually making the change, testing the change and deploying the new release.

17) The most common response measures are time and money necessary for the change.
However, they are not simple to measure.
For time, you have to say whether you measure calendar time or staff time.
You also have to say whether you measure the time of your engineers or also the time of control boards and approval authorities.
Cost usually means direct costs which is a function of the time required for the change.
However, you may also want to count in the so-called opportunity cost of having your staff work on the change instead of other tasks.

There are also other kinds of measures.
For example, you can measure the extent of the change, i.e. the number of artifacts affected by the change.
Or you can measure the number of new defects introduced.
While in practice it is sufficient to specify the time and cost measures, in some situations these supplementary costs may be helpful.

18) This slide is a reminder of the decomposition view of our running example.
We will use it for some examples of the modifiability scenarios.

19) Here we have the first example.
For each dataset in the catalog we record its properties such as title, description or publisher.
In this scenario we anticipate a change to the system which requires a new dataset property, for example the date of its last update.
The change will be done by a developer at the design time and the whole national open data catalog will be affected.
We require that all necessary modules will be updated and tested and the new release will be deployed.
All this must be done with the cost of 3 man-days.
Moreover, it is necessary to change the data source and domain logic only.
The other parts of the system must be untouched.

20) This slide is a reminder of the usage view from our running example

The scenario requires that only the dataset source and dataset model modules will be updated.
The usage view shows which other modules can be affected by this change.
We have to ensure that they are not affected.
In other words, we must ensure that all logic related to this new dataset property is contained in the dataset model.
The services and public API modules must be independent of concrete properties of the dataset.
If there would be a dependency on the concrete properties, they would need to be updated as well which is prohibited by the scenario.
So for new properties introduced into the model module, there must be generic functionality of the services and public API modules.
A functionality specific for new dataset properties cannot be added to these modules in this scenario.
To enable such changes, e.g. specific search using the new property, we need additional modifiability scenarios.

21) This example is another kind of a modifiability scenario.
Here, public API consumers require to increase the throughput of the public API component.
So the administrator, who is the source of stimulus, increases the throughput measured in queries per second at run-time.
The result must be that the API is scaled up and the administrator has to manage it in 1 calendar day.
During this day he or she prepares the necessary infrastructure and configurations and then he or she scales the public API up.

22) To understand how to increase modifiability of the system and to fulfill modifiability requirements it is important to understand three concepts - module responsibility, module coupling and module cohesion.
The concepts of coupling and cohesion are based on the concept of module responsibility.
I can forward that loose coupling and high cohesion help with increasing modifiability while tight coupling and low cohesion work against modifiability.

23) We have already discussed responsibilities.
Each module in a module view is designed because we need it for something.
This something is the module’s responsibility.
A module can be responsible for some functionality, part of functionality, supports some quality, etc.
When a change causes a module to be modified, its responsibilities are changed.

24) A change that affects a single module is easier and less expensive than if it changes more than one module.

If two modules overlap in their responsibilities in some way, then a single change may well affect both.
We can measure this overlap by measuring the probability that a modification to one module will propagate to the other.
This is called coupling.
Coupled modules share some responsibility.
This requires one module to have some knowledge about the other module and vice versa.
This often results in various dependencies.
Coupling often manifests itself in the form of visible dependencies either in the code or in run-time behavior of respective components.

We do not measure coupling exactly in terms of some units of coupling.
Usually we distinguish loose coupling and tight coupling.
Tight coupling denotes such coupling where modules are so tied to one another that you cannot change one without changing the other.
In other words, two modules are tightly coupled when the probability that a modification to one module will propagate to the other is close to 100 %.
On the other hand, loose coupling denotes coupling where modules depend on each other the least extent practically possible.
Loose coupling does not mean that there are no dependencies.
So it does not mean that the probability that a modification to one module will propagate to the other is close to 0 %.
It means that we systematically reduce their dependencies by minimizing the overlap of their responsibilities to the level which is practically possible.

25) As I said, coupling results in dependencies between modules.
This slide shows the most typical kinds of dependencies.
We consider two modules - A and B

Data dependency means that data of A is shared with B or passed to B.
This includes dependencies of B on the data syntax and data semantics of A and on signatures of its services (methods, functions, operations, …).
In other words, A controls B in terms of data formats, data semantics and service signatures it must understand and be able to process.

The slide shows an example where the search datasets module uses the dataset gateway module.
It calls the getDatasetMetadata method with a parameter which is the IRI of the dataset.
As a response it receives metadata about the dataset encoded in a JSON file.
There are two data dependencies.
First, the search datasets module depends on the signature of the called method.
Second, the search datasets module depends on the JSON format in which metadata is persisted.

The dependencies in the example exist because of two shared responsibilities of the modules.
The first responsibility is that datasets are identified by IRIs.
The second responsibility is that dataset metadata are persisted in a JSON format which is defined by an existing external European metadata standard.

26) Control dependency means that A controls B in terms of the flow of messages, data or information.
B must receive messages, data or information produced by A, e.g., in the fixed order or under timing constraints given by A.
In general control coupling means that B must receive messages produced by A on the base of some internal logic of A.

The slide shows an example where the search datasets module uses the dataset index module.
Before it searches for datasets, it has to ask on the status of the dataset index.
There is a status which indicates that the index is being updated with new records and therefore the search result may be incomplete.
Another status indicates that the index is up to date and therefore the search result will be complete.
The dependency in the example exists because of the shared responsibility between both modules which is that dataset records are continuously added to the catalogue.

27) Content dependency means that B relies or depends upon the internal workings of A.
In other words, B expects that A does something.

The slide shows an example where the search datasets module expects that the dataset index module validates search query parameters.
The search datasets module relies on the dataset index module that it validates the parameters.

28) There exist various other kinds of dependencies.
For example, quality dependency, or sometimes called quality of service dependency, means that B relies on a certain level of quality of a service provided by A.
This may include, e.g. security or performance of A.

Another example is existence dependency.
It means that B relies on the existence of A.
There are also variants of this dependency where B does not depend on the existence of A itself but it depends on the particular location of A’s interface or identity of the interface.

29) Cohesion in software engineering is the measure of how responsibilities of a given module belong together.
In other words, it is a measure of how strongly are the responsibilities related.
Similarly to coupling, we do not express cohesion as a percentage.
Low cohesion means that the module has unrelated responsibilities.
However, this is a little bit dangerous reasoning about cohesion.
It would be dangerous to say that a module has high cohesion just because its responsibilities are related.
In some sense, there is always a reason to say that responsibilities are related and therefore the module has high cohesion.
We need to distinguish different kinds of relationships between responsibilities.

30) If nothing you can always say that responsibilities are related coincidentally.
So there is a cohesion.
We call it coincidental cohesion.
It means that responsibilities of a module are grouped arbitrarily.
The only relationship between the responsibilities is that they have been grouped together but we cannot say nothing more.
You probably suspect that this is not the cohesion we are talking about.

31) The next level of cohesion is logical cohesion.
It means that responsibilities are grouped because they logically belong to the same category.
For example, responsibilities of validating inputs from users are responsibilities of the same category.
A module with a set of responsibilities of this category which are related only because they validate input has logical cohesion.

The slide shows an example of a module with logical cohesion.
We have here the public API module without any further decomposition.
We just say that the module is responsible for publishing all application logic to network clients.
All the responsibilities of the module are logically related.
They belong to the same category of responsibilities.

32) Temporal cohesion means that responsibilities of the module are related in time.
We assign the responsibilities to a module just because they happen at the same time.

You usually create temporal cohesion by assigning responsibilities which happen at a certain moment of the system or its part, e.g. at its initiation or termination.

In our running example we do not have a module with temporal cohesion.
However, there is one part of the system which I have not spoken about yet.

Each night, the national open data catalog harvests catalog records from local catalogs of open data publishing institutions.
A public institution can run its own local open data catalog and if it provides a predefined API, it can be registered to the national open data catalog which then havests it each night.
This is a certain moment.
The module you can see on the slide is a module which has several responsibilities such as getting the list of catalogues to be harvested, harvesting, validation of harvested data and rebuilding indexes.
All these responsibilities are related temporarily.
That is why the module has temporal cohesion.

33) We have started with the coincidental cohesion which is actually the lowest possible level of cohesion.
You have probably noticed that we have proceeded to higher levels.
As the next levels of cohesion are usually procedural, informational and sequential cohesion.

Procedural cohesion means that responsibilities of the module are related by sequence in which they are executed.
It adds to the temporal cohesion the fact that there is something more than just temporal relation but also a relation which specifies which responsibility follows another possibility.

Informational cohesion means that responsibilities of the module are related by operating on the same data (input, output or some data source).

Sequential cohesion means that responsibilities of the module are related by the information flow, i.e. that one responsibility has as its input an output of another responsibility.

34) Finally, there is functional cohesion.
It is considered as the highest level of cohesion.
It means that the responsibilities of the module are related by a single well-defined task they contribute to and there are no different logics (e.g. application, domain or infrastructural logic).
A given single problem to be solved is the only concern for all the responsibilities of the module.

The slide shows two sub-modules of the services module.
These modules have responsibilities related only to well-defined tasks - searching datasets and getting details of datasets.
Moreover, these responsibilities are only in the scope of the application logic.
There is no other logic mixed such as data source access logic, domain logic or presentation logic.

35) So this was module coupling and cohesion.
How are they related to systems' modifiability?

Low cohesion means mixing different unrelated or weakly related responsibilities.
More responsibilities with weak relationships mean that it is not easy to identify which modules should be changed when a change is required.
Moreover, because the responsibilities are mixed together in the module, their respective code can also be mixed by developers.
You did not decompose the module to submodules which is the reason for low cohesion of the module.
So you let developers make the code for one responsibility dependent on the code for the other responsibility.
When one responsibility needs to be changed, we do not need to change only the code corresponding to this responsibility but also the code for the other responsibilities.
Even though they are weakly related, their code is strongly related because we left them in a single module.

Tight coupling means that when you identify a module or modules which need to be updated to fulfill the change requirement, the change will spread to other dependent modules.

36) Therefore, increasing cohesion and making modules more loose coupled increases modifiability of the system.
All modifiability tactics are based on these principles.
We will discuss them shortly.

37) Before we jump on the modifiability tactics we need to discuss one additional principle which is important for modifiability.
We forgot about a basic principle which does not come from the software engineering bubble.
It is a principle which is the essential principle of nature and universe.
It is time.
Physics shows us that time is nothing else than increasing entropy.
The time flows as the entropy of universe and entropy of things which exist in universe increase.

What does that mean for software?

In the software engineering field, software entropy is disorder or mess in the system.
Simply speaking, according to the second law of thermodynamics entropy of a closed system always increases.
When you spend enough energy, you can decrease the entropy.
A software system is a closed system.

If you do not actively spend energy, its entropy, i.e. disorder, will grow.
From the point of view of modifiability, growing disorder means that cohesion lowers and coupling becomes tighter.
Without energy spent to fight against this, each development iteration will increase the entropy of your system.
Modifiability tactics help you in this fight.

38) We will talk about three categories of modifiability tactics.
The first two are not surprising.
We have tactics to increase cohesion and to reduce coupling.
The third one extends these two with tactics which do not care about cohesion and coupling to make the system better modifiable for people.
It tries to avoid people.

39) Let’s start with tactics to increase cohesion.
These tactics involve moving responsibilities from one module to another.
This helps twice.
First, well defined separation of responsibilities among modules helps us to identify what modules need to be changed when a change requirement appears.

Second, it reduces the likelihood of side effects affecting other responsibilities untouched by a change in the same module.
A module with lower cohesion has different weakly related responsibilities.
However, they are in the single module.
If this module is not decomposed to submodules its internal structure and implementation is unknown from the global point of view.
It is the business of the module’s designers and developers of that module.
The implementation of the responsibilities in the module can be arbitrarily intertwisted together.
Therefore, a change to one responsibility of the module can arbitrarily influence other responsibilities of the module which makes the change implementation more costly.

40) The first tactic we will discuss is semantic decomposition.
It directly targets achieving the functional cohesion which is the highest level of cohesion we have discussed on the previous slides.
It says that if you have two responsibilities in a module which do not serve the same purpose, they should be placed in different modules.
This may include creating a new module or it may involve moving a responsibility to another existing module.
We can view the purpose from two viewpoints.
The first viewpoint is a business function of the responsibility.
For example, searching datasets.
The second viewpoint is the logic which needs to be covered by the responsibility.

Is it the presentation logic, application logic, domain logic or some infrastructural logic such as network message exchange, notification or data persistence.

We have a schematic example on the slide.
We have module A with three responsibilities R, S and T.
Their background color shows a business function.
Their line color shows a logic.
We can see that R and S have the same logic, e.g., the presentation logic.
T has another logic, e.g. application logic.
S and T has the same business function, e.g. search for datasets.
R has another business function, e.g. dataset detail.
The semantic decomposition tactics tells us to assign the responsibilities to different modules.
This will increase the cohesion.

41) The second tactic is decomposition based on anticipated changes.
We have talked about modifiability scenarios at the beginning of the lecture.
These scenarios are hypotheses about likely changes.
We can reason about the modules and their responsibilities affected by the modifiability scenarios.
When a scenario affects a responsibility but not the others, we should move the responsibility to a separate module so that changes in that responsibility do not influence the other responsibilities.

The slide shows a schematic example.
We have module A with three responsibilities R, S and T.
We have a modifiability scenario which describes a likely change.
The dashed line shows the responsibilities which are affected by this change.
We can see that it is the responsibility S.
To be more concrete, imagine that R and T are responsibilities related to the domain logic of datasets, S is a responsibility related to the domain logic of dataset distributions from our running example.
These responsibilities are functionally quite close so we left them in a single module after applying the semantic decomposition tactics.

The scenario describes a likely change related to distributions of datasets.
It says that distributions of datasets will not be only dump files for download but also APIs through which the dataset content is available.
Now we have only distributions which are dump files so this change will require some modifications in our system.

The scenario affects only the responsibility S.
The tactics tell us to separate it to another module as it is shown on the slide.
We separate it into a new module B.

42) Another tactic to increase cohesion is decomposition based on shared responsibilities.
If two modules share a responsibility, a change affecting it will require modification of both modules.
The tactic tells you to move the shared responsibility to a separate module.

You have a schematic example on the slide.
We can see here that the modules A and B share a responsibility T.
The tactics tell us to move T to a separate module.

43) You need to apply these tactics carefully.
Increasing cohesion by moving responsibilities can result in dependencies between modules.
The stronger are the relationship between the responsibilities in a module the stronger are the dependencies between the module and a module when one of the responsibilities was moved.
Decomposition based on anticipated changes and shared responsibilities may lead to moving a responsibility which is strongly related to other responsibilities, even functionally.
So be careful not to increase cohesion in cases where high functional cohesion has already been achieved.
Such decomposition would lead to tight coupling which would be against the overall goal to increase modifiability.

44) This brings us to the second category of modifiability tactics which is reducing coupling.
The goal of reducing coupling is to prevent the ripple effect.
Ripple effect means that a module is modified only because it depends on another module which was modified.
This increases the cost of changes in the system.
In a system with tightly coupled modules you are lucky when you are able to implement changes successfully not looking at the costs.
In practice however, you not only spend a lot of resources but you are even not able to change everything correctly which introduces unexpected bugs to the system.
Systems with tightly coupled components are sometimes called “big bowls of mud” or “spaghetti systems”.
Tactics for reducing coupling help you not to have such systems.
They are based on placing intermediaries of various sorts between coupled modules.

45) The basic tactic is to restrict possible dependencies by restricting the set of modules a module can use.
This is a very common tactic as many architectures are layered.
A layered architecture means that a module on a given layer can use only modules on the layer below.
In other words, modules are restricted in what other modules they can use.

Another basic tactic to reduce coupling is information hiding, also called encapsulation.
It introduces an explicit interface to a module A which separates it from the other modules.
The interface has associated only public responsibilities which are visible by the other modules.
The modules which depend on module A may use only the public responsibilities.
This reduces coupling because it prevents the other modules to build dependencies on the private responsibilities of A.

46) The next tactic is using an intermediary translator.
In comparison to the previous two tactics, this tactic introduces a new module to the architecture.
The intermediary translator is a module with a single responsibility which is translation of messages or calls between modules.

47) An intermediary translator is useful when you do not want to corrupt the model or logic of a module by the model or logic of another module.
The intermediary translator can create so called anticorruption layer.
On the slide we have module A which uses module B.
Module B can be, e.g. a legacy system with old or obscure logic or model and we do want the nice model and logic of module A to be corrupted by B.

48) Therefore, we put an anticorruption layer between A and B.
A will not use B directly but through an intermediary translator in the anticorruption layer.
The translator understands the calls and messages from A and translates them to calls and messages required B and vice versa.

49) We can also view the situation on the previous slides from the other side.
You have a module with a complex or unstable model or logic and you do not want to corrupt other modules.
On the slide, there is module A with an unstable model and logic.
We have three other modules B, C and D which use module A.
These modules either corrupt their internal logic or model with the unstable logic and model of A or they have their own translators in their anticorruption layers.

50) Instead of this, you can build a layer around module A with a translation module inside this layer.
The translator translates the complex or unstable logic and model to something less complex or more stable.
We refer to the layer as a service layer and to the translator as an open host service.
The other modules then use the open host service which has only the translation logic.
The service layer of A then passes the message or information flow between the open host service and module A.

It is possible that we build more open host services in the service layer of module A.
This is necessary when different modules need a different logic or model.

51) In some systems, there are simply too many usage relationships between modules.
In some systems these relationships even appear and disappear dynamically at run time.
In such cases, hardcoding the individual usage relationships in code is not sustainable.
In the code of each module you need to have identities, locations, syntax and semantics of interfaces of the other modules or even several versions of these interfaces.
This is simply too much logic which cannot be a responsibility of each of the modules.
A solution in these situations is a message broker or simply broker.

52) A broker is a new module in the architecture which is used by all other modules.
They use some identities of other modules introduced by the broker but they do not know how to locate and access the other modules.
This is the responsibility of the broker.
It is also possible that the broker presents some shared logic and model and the other modules provide their open host services respecting this shared logic and model.
Sometimes this shared logic and model is called pivot logic or pivot model respectively.

53) Last but not least there are also other tactics which support reduction of couplings.
Either interfaces or intermediary nodes introduced to the architecture according to the previous tactics can change in time.
In these cases, it is necessary not to forget their clients.
If such changes occur and it is not possible to change all clients in a reasonable time, you can prevent the old version of the interface or intermediary and offer also the new version.
The old version can be marked as deprecated.
It provides some time for the clients to be updated.
A specific situation is when the module behind the interface or intermediary is removed from the system.
In that case, we can preserve the interface or intermediary and provide some stub implementation instead of the removed module.
We can mark it as a stub which gives the other modules some time to update to this new situation.

We have also talked about universe, time and entropy.
We saw that it is natural that with each development iteration coupling is tightened.
With this we do nothing else than architecture and code refactoring.
We invest energy not to develop new features of the system but to improve our architecture and code to systematically reduce the coupling introduced by the previous development iterations.
Refactoring is also a tactic to increase cohesion.

54) Work of people is more expensive than the work of computers.
Defer binding means thinking about changes as something which can be handled by a computer and let humans to only specify a change on a higher level of abstraction.
This includes a wide spectrum of tactics.

On the highest level of abstraction we have low code or no code approaches we talked about at the beginning of the lecture.
Here, people use a high level user graphical user interface where they specify their required system and later required changes.
A low code platform generates and deploys everything for them.

On the lowest level of abstraction, we have parametrized functions.
A simple example is a function f(a,b) with two parameters a and b which is more general than a function f(a) that assumes b = 0.
If we have f(a,b), we can parametrize it as necessary.

Between these two extremes we have various tactics which are out of the scope of this lecture.
They include, e.g., ability to replace modules in build scripts of makefiles, configuration files which are read at the initialization of the system or polymorphism in object-oriented programming.

On a higher level of abstraction but not so high level as the low code approaches are systems where other parts can be plugged-in at design time or runtime.
For example, there exist platforms in certain business domains which provide basic functionality which can be extended by plugging-in third party applications a user chooses in an app store.

55) So this is the end of the lecture about the modifiability quality attribute.
We will continue with other quality attributes in the next lectures.

Lecture 11: Quality Attributes - Performance

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the performance quality attribute.

2) Performance is about time and the software system`s ability to meet timing requirements.
It is a measure of how long it takes the system to respond to events.
An event can be a request from a user or other system.
The system must be able to respond to the request in time.
It can also be a clock event marking the passage of time.
The system must respond to these clock events in time.

3) Our running example, the national open data catalog, is a web-based system.
It gets requests from its users via their clients such as web browsers.
These requests include, e.g., to show search datasets or showing their details.
These are simple short transactions which start with a request followed by some processing on the side of the system and end with sending a response.
The desired behavior of the catalog from the performance point of view can be expressed as the number of transactions that can be processed in a minute.
This is a measurable response of the system important from the performance point of view.

4) Another example is an engine control system.
It does not get requests from human users.
It gets requests from the passage of time.
It must control both the firing of the ignition when a cylinder is in the correct position and the mixture of the fuel to maximize power and minimize pollution.
The measurable response in this example can be the variation in the firing time.

5) In both examples, the pattern of arriving requests and the pattern of responses can be characterized.
This characterization will be the language we will use to construct performance scenarios.
We will use the same template we use for the other quality attributes in this course.

6) The artifact of the scenario is either the whole system or some of its runtime components which are required to provide a certain performance level.

7) The stimulus is an arriving request for some service which must be satisfied by the system in time.
Satisfying the request requires resources to be consumed while the system may be simultaneously servicing other requests
From the performance viewpoint the single request is not interesting.
What is important is that there are more requests in a stream and they arrive in that stream in a certain pattern.
This pattern is important for the scenario.

We distinguish periodic, stochastic and sporadic patterns.
Periodic events arrive predictably at regular time intervals.
For example, an event may arrive every 10 milliseconds.

Stochastic arrival means that events arrive according to some probabilistic distribution.

Sporadic events arrive according to a pattern that is not predictable.
It can be still characterizable but not exactly.

For example, when keyboard hits are events for us then we know that there will be at least 100ms between each hit.
Or we may estimate the number of requests in a web-based system on the base of our registered users and their average activity in the history.

It is important to note that it does not matter whether we have 20 requests from a single source in a given period of time or whether two sources each submit 10 requests in that time.
What matters is the arrival pattern.

8) The source of stimulus is either an internal or external source.
An internal source is another component of the system.
The external source can be a user, another system or the passage of time.

9) The environment characterizes the operation mode of the system when the event occurs.
We can consider, e.g. the normal mode, emergency mode, a mode when the system is at the peak or overloaded.
In each environment the requested response of the system may differ.

10) The response is that the system processes the requests arriving in the pattern described by the scenario.
This may also cause a change in the system environment, e.g., from the normal to the overloaded mode.

11) And as we already know we must define the response in a measurable way so that we can later measure the response and prove that the system meets the quality requirements.
For performance, there are several possible measures.

Latency is the time between a request occurrence and a response to it.
It is the most typical performance measure used.
It is often used for requests sent by human users through a human user interface.

System throughput is the number of requests the system can process in a unit of time.
It is often used for APIs and requests sent to this API where we usually specify it as the number of API queries per second.

Jitter to response is the allowable variation in latency.

The number of requests not processed because the system was too busy to respond is also possible.
Sometimes it is called miss rate or data loss.

12) This slide shows an example of a performance scenario.
Human users request the national open data catalog to search for datasets.
The scenario considers 60 search transactions per minute.
The pattern is not periodic - we do not have a transaction exactly each second.
However, there is a continuous uniform distribution of the 60 requests in the minute interval.
So the pattern is stochastic.

The scenario requires that the national open data catalog as a whole processes each request arriving in the specified pattern with the average latency of 1 second.
Processing the requirement means that datasets are found, the response with the list of found datasets is constructed and sent back to the user as a response.

13) This slide shows another example.
It specifies a performance requirement on a new feature of the national open data catalog we have not spoken about yet.
An institution which catalogs its datasets in the national open data catalog may provide its own local catalog which is harvested by the national open data catalog.
The local catalog may request the national open data catalog to be harvested.
If it requests till 2am it will be harvested in the time window between 2am and 5am.

The pattern of these requests is neither periodic nor stochastic.
Therefore it is sporadic.
We only know that potentially all catalogs may require to be harvested during a day.
Only local catalogs of central government and local government institutions may require this feature.
We have approx. 7000 institutions in our country and each may have more catalogs.
But this is very rare so we consider 10000 local catalogs at maximum.
This number is the theoretical maximum, currently we have only 100 local catalogs approximately.
However, in the future the total number of local catalogs may scale to this number.

The expected response is that the national open data catalog sends a response with the confirmation that the request to be harvested was recorded with the average latency of 1 second.
The harvesting itself will happen later in the early morning.
It is required that all requesting catalogs are harvested in the time window between 2am and 5am.
Harvesting means not only reading records from the local catalogs but also storing them to the database of the national open data catalog and reindexing the search index.

14) So this was how to specify performance requirements.
Let's now proceed to tactics to control performance.

They have as their goal to generate a response to an event arriving at the system within some time-based constraint.
At any time instant before the response is complete either the system is working to respond to the event or the processing is blocked for some reason.
The processing time and the blocked time both contribute to the latency.

The processing time means that the system consumes resources.
Even though resources are said to be quite cheap today we should not forget about them.
For example, even if you have a lot of resources, you should think about energy consumption.

The blocked time means that the computation is blocked.
There are several reasons for this.
Contention for some needed resource means that components share a resource which can be used by only one of them at a given time.
The latency caused by the contention depends on how the contention is arbitrated.

Resource availability means that when the resource is unavailable, the latency increases.
This is related to availability quality.

Dependency on the results of other computations means that a component has to wait for the results of other computations that are not yet available.
It means that the component needs to synchronize its computation with other computations.

15) This leads us to two basic categories of performance tactics.
Control resource demand tactics operate on the demand side to produce smaller demand on the available resources.
Manage resources tactics operate on the resources side to make the available resources work more effectively in handling the demand.

16) One possibility to control resource demand is to manage sampling rate.
If it is possible to reduce sampling frequency at which a stream of data is captured, then demand can be reduced, typically with some loss of fidelity.
E.g. You can reduce the sampling frequency in a monitoring system.
In our running example, the national open data catalog periodically checks the quality of the recorded datasets.
There is a requirement to periodically provide a quality report about the state of open data in the country.
It is not possible to check each individual dataset in a given period so we can manage the sampling rate by picking arbitrarily the manageable amount of datasets for the quality check as a sample.

17) When the events cannot be down-sampled, there is a possibility to limit responded events.
When discrete events arrive too rapidly to be processed, you must queue them until they can be processed.
You pick the requests from the queue and process them with available resources.
You set a queue size which, when exceeded, something has to happen with new resources.
Otherwise the performance requirement for the requests which have already arrived will not be satisfied.
It means that the requests on the queue will be served according to the performance scenario and for the other requests the performance scenario will not be satisfied.
The point is that you manage this and you have some control over it.
If you decide to simply drop the other requests, the system or component will not be available for them which is the availability problem.
If you decide to deal with them somehow, you may specify the expected behavior in another performance scenario with the environment set to overloaded.

18) There is also one complementary tactic which is event prioritization.
If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them.
If there are not enough resources, low-priority events might be ignored.

19) The previous three tactics lead to ignoring some events when there are not enough resources to process them all.
All three have in common that they utilize the available resources to fully respond to the event.
We could characterize them as “the winner (a processed event) takes all”.
This can be quite cruel for the events which were not processed.
There is a more merciful tactic to control resource demand.
If we do not have enough resources you can bind the execution time of the resources to process an event.
For example, this is useful for events which require some iterative data-intensive computation.
You can limit the number of iterations to limit the execution time of the resources.
For non-iterative data-intensive computations, such as complex database queries, this tactic simply means setting a query execution timeout.
It means that after the fixed amount of time, the query execution is stopped and either an error message or a partial result is returned.

This tactic can be combined with the other tactics.
For example, you can have a priority queue where you do not skip the low priority events but you give them only a very restricted execution time.

20) All the four introduced tactics were somehow related to controlling the events to control the resource demand.
We can also produce smaller demand on the available resources by optimizing the system.
There are basically two tactics.
First, you can of course improve various algorithms or their implementations in critical areas.
This will decrease latency.

21) Second, you can improve your system's run-time architecture from the performance point of view.
In general the more components are needed to respond to an event the more context switching and inter-component communication is necessary.
This always increases latency.
When this is only inside the memory of a process among different threads the overhead is usually acceptable.
However, when components are different processes on different nodes in a network, the inter-component communication is costly.
This is because you need to turn data structures from one component into bytes on the network, transfer them to the other component and here you need to turn the bytes to data structures in the other component.
Turning data structures to bytes and vice versa costs resources and network transfer is orders of magnitude slower than simple calls inside the memory of a single process.

22) Turning data structures to bytes and vice versa is called data serialization and deserialization, respectively.
To make this work, you need a serialization format understood by both sides.
It is common to use text based formats such as XML or JSON.
Serialization to XML or JSON is costly and here the first tactic helps.
You can optimize the serialization and deserialization by using binary formats to transfer information between nodes.
For example, you could be interested in Apache Thrift (originally developed by Facebook), Protocol Buffers (originally developed by Google) or Apache Avro.

23) Another way to optimize the communication is to reduce the number of necessary requests.
This targets not the serialization but the transfer of serialized data structure across the network.
It is a big difference from the performance point of view when you send 10 requests instead of 1.
This is especially true when you transfer the requests and responses through the internet.

In our running example, when you need an API call to get metadata about a dataset and then another API call for each distribution to get metadata about the distribution you need N + 1 requests and responses sent through the internet.
From the performance point of view is to provide an API which needs to be called just one to get metadata about the dataset and all its distributions.
In the JSON world, the concrete approach to this is GraphQL.

24) You can also reduce the inter-component communication by removing intermediary components.
You can simply remove intermediary components in the communication.
An intermediary component exists in the architecture for some purpose, it has some responsibility which was taken from the other components the intermediary component interconnects.
So you have to put the responsibility on these other components.

25) Another possibility is to put components together.
It is called co-location and it usually means that you put cooperating components on the same processor to avoid the time delay of network communication.
In general, you put components as close as possible.
Also the concept of edge computing belongs to this tactic.
Edge computing is a paradigm of distributed computing which reduces the network communication overhead by bringing the data and computation nodes as close as possible to the consumer.
You can read more about it on the website linked from the lecture notes.
https://www.cloudwards.net/what-is-edge-computing/

26) But be careful with the tactics consisting of improving the architecture.
Text based formats mentioned before have a great advantage - they are human readable and agnostic to platforms, libraries and other technical stuff.
While they can be problematic from the performance point of view, they greatly help with modifiability as it is easier for the developers to track necessary changes in data structures and reduce the ripple effect of changes across components and their respective modules.

Removing intermediaries and component co-location goes strictly against modifiability tactics as well.
Do you remember them?
Their nature is increasing the number of modules and components by splitting them according to different responsibilities and by introducing intermediaries.
All this is called performance/modifiability trade-off and you will always have to somehow deal with it in your projects.

We have mentioned GraphQL as a tool to implement the tactic to reduce the number of requests.

However, this is a nice example of increasing performance while sacrificing modifiability because it presumes building a single monolithic graph of data which is accessed through GraphQL.
To solve this performance/modifiability trade-off problem, integrity principles were specified which instruct you to split the graph to smaller graphs.
The technical platform you can use for this is, e.g., the Apollo Data Graph Platform.
You can read a blogpost by Netflix about how they used this platform.
You will find the links to the principles, platform and blogpost in the lecture notes to this slide.

27) So these were tactics to produce smaller demand on the available resources.
The other tactics are tactics to make the available resources work more effectively in handling the demand.

The first thing you can always do is to increase resources.
This means faster processors, additional memory, faster networks etc.
This also means using appropriate resources.
For example, GPUs are much better for certain kinds of computations such as training machine learning models.
Today this tactic is in many cases the cheapest way to get immediate improvement.

28) However, there are also more sophisticated tactics to manage available resources instead of just increasing them.

Sometimes, latency is not caused by not having enough resources but because there are events which consume resources for a very long time.
They are complex so the long time is acceptable and expected.
There can also be events which are simple and the system is expected to deal with them quickly.
However, this is not possible simply because the system processes the previous complex event.
For these cases we can introduce concurrency.
Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.
Once concurrency has been introduced, scheduling policies can be used to achieve the goals you find desirable.
Different scheduling policies may maximize fairness (all requests get equal time), throughput (requests with the shortest deadline to finish go first), or other goals.

29) To reduce contention that would occur if all computations took place on a single component, e.g. a server, you can introduce replicas of that component.
These replicas are multiple copies of computations and you have a load balancer that assigns new work to one of the available replicas.

There are multiple strategies for the load balancer, e.g. a simple round-robin or assigning the next request to the least busy replica.

30) Another source of higher latency is accessing data.
You can reduce this kind of latency by maintaining multiple copies of data.
A simple tactic is caching.
It involves keeping copies of some, usually recently accessed, data on a storage with a higher access speed than the original storage.
Data replication involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses.
So for example instead of one database server you maintain two to reduce the contention.
In both cases - caching as well as data replication - you need a strategy to keep the copies consistent and synchronized.

31) So this is the end of the lecture about the performance quality attribute.
We will continue with other quality attributes in the next lectures.

Lecture 12: Quality Attributes - Scalability

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the scalability quality attribute.

2) Scalability is the ability of a software system to handle tasks as the system grows in size.
This definition is quite general.
Let's talk about it in a little more detail.

3) First, what is the system growth?

4) The system may grow in different dimensions.
Usually, the system growth means increasing the number or volume of its users, requests per time in a given period or amount of data managed by the system.
More and more it is also necessary to consider another dimension and it is the growth of the features required by stakeholders.

5) So this is what we mean by system growth.
There is also another part of the definition which is not clear.
The system has to handle tasks.
But what is a task?
It can be a lot of things.
It can be a feature of the system.
However, features are a dimension of the system's growth.

6) From the architectural point of view, handling tasks means that the system is able to provide all its features while its quality attributes are preserved.
However, this is still too general.

7) The simplest view of scalability considers only performance.

8) In practice, availability and performance go hand in hand and they are considered usually together in the context of scalability.
The definition then means that scalability is the ability of the system to keep its availability and performance as the numbers of users, requests and data grow.

9) Nowadays, it is more and more important not to consider only these two runtime qualities of the system.
It is also important to consider modifiability of the system.
In other words, scalability is also the ability of the system to keep its modifiability as the number of required features grows.

10) This table provides an overview of the meanings of the term handle tasks discussed on the previous slides.

11) So now we know what scalability is.
How to specify a scalability requirement?
Again, we will use scenarios with the same structure we had in the previous lectures.

12) Artifact in the scenario is a module or component which needs to scale.
It can be the whole system as well.

13) Stimulus is the aspect which grows.
We have already seen that it can be the number of users or requests, the amount of data or required features.

14) Source of stimulus is someone or something who or what is behind the growth.
When the stimulus is that the number of users grows, the source is the users.
When the stimulus is that the number of requests grows, the source is the users or some other system.
When the stimulus is that the amount of data grows, the source is the reason why the amount grows.
When the stimulus is that the number of required features grows, the source is the stakeholder or stakeholders who have these requirements.

15) Environment in case of scalability means when the scaling needs to be done.
It can be at runtime, build time, initiation time or design time.

16) Response simply specifies that the artifact is scaled.

17) Response measure specifies how we will show that the scaling scenario is fulfilled.

First, we want to ensure that the system is scaled while existing availability, performance or modifiability scenarios are still ensured by the system.
Second, we can also measure the time necessary to scale the system.

18) Here is a sample scalability scenario.
The stimulus is the growing day average number of dataset search requests.
We are interested in scaling up the system so that it can serve the increased number of requests as specified by the performance scenarios.
The scenario also specifies that scaling the system up should not take longer than 1 business day.
The environment specifies that the system must be scaled up at runtime.

19) This is another scalability scenario.
Here the stimulus is the growing number of required features of the application services of the national open data catalog.
As we have already said, when we talk about features in the context of scalability, we are interested in not affecting the modifiability.
Our goal is to ensure that as new features are added, the architecture of the system still ensures all specified modifiability scenarios.
In such scenarios, the time measure does not make sense.
They make sense only for the modifiability scenarios, i.e. how long it shall take to make the change.
For the scalability scenario, the measure is just that the modifiability scenarios are not affected.

20) Scalability tactics are based on the so-called scale cube which is a three-dimensional scalability model.

X-axis scaling is a common way to scale components.
You run multiple instances of a component behind a load balancer which is demonstrated on the next slide.

21) This is a common tactic also for availability and performance.
For scalability, you moreover consider adding new instances dynamically as the numbers of requests grow.

22) Z-axis scaling also runs multiple instances but unlike X-axis scaling, each instance is responsible for only a subset of the data.
Here you have a router which routes requests to appropriate instances.
This is demonstrated on the next slide.

23) Again, this is also a common tactic for availability and performance but here you also consider dynamic instantiation.
With z-axis scaling you can solve the problem of the growing amount of data since you have different component instances for different data partitions.
It can also indirectly help with the growing number of requests as data requests are usually distributed uniformly across the data so this naturally distributes these requests to be served by different instances.

24) As we have seen, X-axis and Z-axis scaling improve the performance and availability of a component.
With the growing numbers, we just add new instances of that component.

They however do not solve the problem of increasing complexity of that component.
As new features are added the component modifiability may go down.
This is also related to scalability.
It means that the modifiability must scale as the number of required features grows.
To solve this problem, so-called functional decomposition is applied.
It is based on splitting a monolithic application to a set of loosely-coupled components.
These components are usually self-contained with their own database, domain logic, business/application logic and API.
We usually call them microservices.

X-axis and Z-axis scaling were very similar to availability and performance tactics.
Y-axis scaling is very close to modifiability tactics.
It just focuses on keeping the system modifiable as the number of required features grows in time.
The mentioned microservices approach emerges from the modifiability tactics we have discussed in the modifiability lecture.
However, Y-axis means that you apply these tactics systematically across all parts of the software system which need to scale in terms of the growing number of required features.

25) This slide shows an example of Y-axis scaling.
Individual functions of the system are split to separate microservices each self-contained with their own database, logic and API.

26) This slide combines all three axes together.

27) So this is the end of the lecture about the scalability quality attribute.
We will continue with other quality attributes in the next lectures.

Lecture 13: Quality Attributes - Security

1) Welcome to the next lecture about software architectures.
   In this lecture we will talk in detail about the security quality attribute.

2) Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.
   The attempt for an unauthorized access is called an attack in the security domain.

   The second part of the definition is important.
   It requires that the system still provides its features when it is under attack.
   From this point of view, we cannot say that the system is secure because we can switch it off when it is under attack.

   There are different kinds of attacks.
   For example, it may be an unauthorized attempt to access data, e.g. user names, passwords and emails.
   It may also be an attempt to modify the data.
   Another kind of attack is an intention to deny services to legitimate users which are called denial-of-service attacks.
   The well-known example of this kind of attack is the distributed denial-of-service, shortly DDoS.
   DDoS attacks are known even to the non-IT community as we can often read about them in the media.
   And if I speak about the media, maybe you noticed news about different attempts to attack information systems in hospitals.
   These attacks are usually also a kind of denial-of-service attacks as they usually use ransomware which disables people in the hospital to use the system.
   It usually encrypts system data and the system is then unable to work.

3) There are three basic security goals an architect should have when designing a software architecture focused on security.

   Confidentiality is the property that data or services are protected from unauthorized access.
   For example, an attacker cannot access patient records in a hospital information system.

   Integrity is the property that data or services are not subject to unauthorized manipulation.
   So confidentiality is about reading data, integrity is about writing data.
   For example, an attacker cannot change the result of an examination of a patient.
   Integrity includes not only persisted data but also transferred data.

It means that data transferred through a network is not subject to unauthorized manipulation.
For example, an attacker cannot change the data with the result of an examination when transferred through the internet to the doctor's device.

Availability is the property that the system will be available for legitimate use.
For example, the DDoS won't prevent doctors from writing and reading results of examinations.

4) There are usually also other goals.
Authentication verifies the identities of the parties to a transaction and checks if they are truly who they claim to be.
For example, when the system receives a request for a patient record purporting to come from a doctor, the authentication verifies the identity of the user and that it actually comes from a doctor.
Another example is when the system receives a question about health conditions from a family member, the authentication verifies that it actually comes from a family member.
Or when a doctor receives an email from a patient, the authentication verifies that it actually comes from a patient.

Nonrepudiation guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message.
For example, when the hospital sends a message with patient's health conditions to a family member, the nonrepudiation guarantees that the hospital cannot later deny having sent the message and that the family member cannot deny having received the message.

And finally authorization grants a user the privileges to perform a task.
For example, authorization grants a doctor to access records of patients the doctor cares about.
For authorization, you need authentication which guarantees that the user is a person who he pretends to be.

5) We can now proceed to specifying security requirements.
Again, we will use our structure of a scenario, now we will call it a security scenario.
The artifact is usually a system service.
It can also be more connected to the runtime architecture and then it is a component from a component-and-connector viewpoint.
It can also be a resource the system depends on.
It can also be data within the system or data produced or consumed by the system.

6) The source of stimulus is an attacker.
It can be a human attacker or another system.

It may have been previously identified, correctly or incorrectly, or may be currently unknown.
A human attacker may be from outside the organization or from the inside.

7) The stimulus is the attack.
As the definition says, it is an unauthorized attempt for some service or data of the system.
It can be an attempt to display data, change or delete data, access a system service, change the behavior of the system or reduce its availability.

8) The environment specifies artefact conditions under which the attack happens.
What is usually most important to specify is whether we consider the scenario when the system is online or offline.
It is also important to specify whether the artefact is in the internal network behind a firewall which does not allow for requests from a public network, e.g. from the internet, or in the demilitarized zone behind a firewall which allows for requests from the public network, or even fully open to the public network.

9) The response usually is that confidentiality, integrity and availability of the artifact is preserved.
Moreover, the response may be that attackers are identified or deterred through monitoring their activities.
In a more detail, the response usually is that transactions performed by the system are carried out in a fashion such that data or services are protected from unauthorized access while not being manipulated without authorization.
It can be required that the response parties to a transaction are identified and that they cannot repudiate their involvements.
Also some supporting actions may be required such as creating the audit trail and notifying appropriate entities.

10) The measures of the response may include how much of a system is compromised when a particular component or data is compromised.
It may also include how much time passed before an attack was detected, how many attacks were resisted and how long it took to recover from a successful attack.

11) This slide shows a sample security scenario.
It specifies that a disgruntled employee from a remote location attempts to modify the pay rate table during normal operations.
The system maintains an audit trail and the correct data is restored within a day.

12) Here is another scenario.
It specifies that an unknown human attacker tries to change system data and make the hospital system unavailable by sending an email to an employee with a link to a ransomware which encrypts all files on network disks in the internal network.

The expected response of the system is that the ransomware email is detected and the email is deleted.
It is required that the ransomware is immediately detected in 90 % of attacks.
When the attack is successful, the ransomware must be detected and removed and corrupted files restored from backups in 48 hours.

13) The last sample scenario specifies a security requirement for our running example with the national open data catalog.
It describes a denial of service attack to the service of the national catalog which harvests the content of external catalogs registered in the national catalog.
The attack is that a known external catalog requests to catalog a high number of odd dataset records to make the harvesting service unavailable for other external catalogs.
The expected response is that the harvesting service will not become unavailable and the measure is that the DoS attack is detected directly during the harvesting task, the external catalog is recorded in the audit trail and both the administrator of the national catalog and the administrator of the external catalog are notified about the incident.

14) Let's now move to security tactics.
It is good to think about how to achieve security in a software system as about physical security.
In your house you may use motion sensors to detect attacks, you use locks to resist attacks and you may have insurance to recover from attacks.

15) Detecting attacks means tactics to detect that the attack happens.

Detecting intrusion means that you compare network traffic or service request patterns within the system to a set of signatures or known patterns of malicious behavior.
When the actual pattern does not match the known patterns a possible attack is alerted.
In simpler words, you try to recognize that something unusual happens inside your system by comparing what is normal to what is currently going on inside the system.

Detecting service denial is similar to detecting intrusion.
Here you compare the traffic coming into the system to historic profiles of known DoS attacks.

For both, detecting intrusion and service denial there exist various solutions you can buy.
However, they are quite expensive.

A cheaper tactic is verification of message integrity.
This employs techniques such as checksums, hash values, electronic stamps or signatures to verify the integrity of incoming messages.
A message or request without a valid verification is considered as an attack.

Detecting possible attacks is a complex area and these are only some examples of these techniques.
Please refer to the security literature if you are interested in more details.

16) Resisting attacks means tactics which prevent the system from certain attacks.

The most common techniques used by almost all systems are actor authentication and authorization.
Authentication presumes that actors are identified, usually through their user names in case of human users or IP addresses in case of external systems or services.
Authentication means ensuring that an actor is actually who or what it purports to be.
Authentication is based on using a unique possession of the actor known only to the actor, e.g. a password, authentication code generated by a unique authentication sequence generator, code sent via secondary communication channel, digital certificate or biometric identification.

Authorization means ensuring that an authenticated actor has the rights to access or modify data or services.

These basic tactics are then combined with other supporting tactics.
Limiting access involves controlling what and who may access which part of the system.
A common concrete technique based on this tactic is a demilitarized zone mentioned earlier.
It is a subnetwork between two firewalls - one facing the public internet, the other facing the intranet of the organization.
It is used when the organization wants to let external actors access services available in the system whose services are available and communicate on the intranet.
The actors have limited access to these services.

Limiting exposure refers to reducing the probability of a successful attack or restricting the amount of potential damage.
The basic principle of this tactic is to limit services available on a single host to restrict the amount of potential damage.
Reducing the probability of a successful attack can be achieved by concealing facts about the system or its part to be protected.
For example, when you distribute hosts when you run your services among geographically dispersed data centers, you limit exposure when you conceal facts about where the hosts are located.

Another commonly used tactic is data encryption.
You protect data from unauthorized access by applying some form of encryption either to persisted data or to communication through which data flows.

Separate data entities is a tactic to prevent attacks where the attacker tries to get holistic information about an entity which consists of sensitive and non-sensitive data.
More actors have access to non-sensitive data about the entities and only some of them may also access the sensitive data.
You need to reduce the attack possibilities to the sensitive data from those who can access the non sensitive data.
An example is sensitive health data about patients besides other less sensitive data about patients.
To solve this problem you can separate the data about patients to two databases.
What you need to ensure is that when an attacker accesses the sensitive patient data and has access to individual records, he is not able to identify respective patients.
So you use a different identification mechanism here which disallows the attacker to connect a record to a respective patient.
The pairing between the two data sources is stored in the third database.
This makes accessing the holistic information about patients much harder for the attacker.

17) Recovering from attacks means that once the system has detected an attack and attempted to resist it, it needs to recover.

First, it means restoration of services and their state.
This can be considered a kind of failure so here we refer to the availability tactics.
Regarding data, we do not usually need such strong tactics such as hot spares or similar.
Just simple data backups stored separately from the attacked system are usually sufficient.

Second, we need to maintain an audit trail.
It means keeping a record of actor actions and their effects to help trace the actions of the attackers.

18) So this is the end of the lecture about the security quality attribute.
See you at the other lecture.

Lecture 14: Quality Attributes - Interoperability

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the interoperability quality attribute.

2) Interoperability is the degree to which two or more systems can usefully exchange meaningful information via their interfaces in a given context.

So it is not a quality of a single system but a shared quality of two or more independent systems which need to exchange information.
Moreover, it is necessary to specify in which context the systems need to be interoperable.
It means which information for which purposes shall be exchanged.

Information exchange means that a system requests a service of another system.
In other words, systems exchange information by mutually requesting services of each other.
As a part of a service request, parameters specifying the request or richer information about some entities can be exchanged as well.
For simplicity, we will consider systems communicating through a network connection.

3) There are several levels of interoperability.

   First of all, when a system needs to send information to another system, it must be able to serialize the information as data and then send this serialization via the network.
   It must follow a network protocol which must be also followed by the other system as well.
   The other system needs to deserialize the received expression.
   This ability of the systems is called technical interoperability.
   Technical interoperability means that the systems share a network protocol or protocols which enable them to exchange information.
   Technical interoperability is usually not a problem today as we have well accepted network protocols.

4) Our running example, the national open data catalog, must be interoperable with local open data catalogs of individual open data publishers for the purpose of exchanging information about cataloged datasets.

   Technical interoperability between the national open data catalog and the local open data catalogs is ensured by sharing HTTP as the common network communication protocol.

5) The next level is about how the information is represented in data.
   In general, data is a sequence of bytes structured according to some structural pattern.
   The system receiving information from another system needs to be able to recognize and process structural patterns in the received data so that it is able to deserialize them and map them to its internal structures.
   For example, when one system sends a message in JSON, the other system must be able to parse the JSON message and recognize key-value pairs in it so that it is able to map them to its internal data structures and operation calls.
   This may also include message validation which means checking that the message contains the expected key-value pairs.

This is called syntactic interoperability.
Syntactic interoperability means that the systems share structural patterns in which the exchanged data representing exchanged information is structured.
We usually refer to these structural patterns as the syntax of the exchanged messages.
The syntax defines the so-called exchange format.

6) Syntactic interoperability between the national open data catalog and local open data catalogs in our running example is ensured by sharing a common exchange format DCAT (Data Catalog Vocabulary) and its extension DCAT-AP - DCAT Application Profile for data portals in Europe.
It is a data format based on the RDF data model with a prescribed structure.
It has several possible serializations.
On the slide, you can see one possible serialization format called Turtle.
Data in RDF data model can be expressed in other serialization formats, e.g. JSON.
National open data catalog understands all possible RDF serialization formats which follow the DCAT and DCAT-AP specification.
They prescribe certain properties of datasets and their expression in the RDF data model.

7) In the definition of interoperability, we speak about information not about data.
Data has no meaning on its own.
It must be interpreted by software and then by a human user which makes it information.
The correct interpretation by software on both sides is called semantic interoperability.
It means that a piece of information represented in data by the sending system is interpreted as the same piece of information by the system receiving the data.
Semantic interoperability means that the systems share the same meaning and interpretation of the data.

8) Semantic interoperability between the national open data catalog and local open data catalogs in our running example is ensured by sharing the common semantics defined by DCAT and its extension DCAT-AP.
For example, it defines that all instances of class Dataset shall be interpreted as datasets, not publishers.
It also defines that the property title represents the title of the dataset and not its description and that the property spatial represents the geographical area covered by the dataset, not the spatial resolution between geographical entities in the dataset.

9) Similarly to the other quality attributes we will use our structured scenarios to express interoperability requirements.
The artifact in the scenario is our system or its part which needs to be interoperable with other systems.

10) The source of stimulus is the other system or systems our system needs to be interoperable with.

So the artifact and source of stimulus are the systems required to be interoperable.

11) We are missing the interoperability context which is the purpose of the required interoperability.
It is specified by the stimulus.
The stimulus is a request to exchange information between the system.

12) The context is further specified by the environment.
Usually, the environment specifies whether the systems know about each other's existence and location at design time or need to find this dynamically later at runtime.

13) The response is that the request to interoperate results in the required exchange of information.
The information is understood by the receiving party both syntactically and semantically.
Alternatively, the request is rejected and appropriate entities are notified.
In either case, the request may be logged.

14) It is not easy to measure interoperability objectively and exactly.
Possible responses include either correctly processed information exchanges and correctly rejected information exchanges.
Therefore, we can require a certain percentage of information exchanges correctly processed and percentage of information exchanges correctly rejected.

15) This slide shows an example of an interoperability scenario.
Let's assume that we work on a vehicle information system.
The artifact is our vehicle information system.
The source of stimulus is a central traffic monitoring system.
It is the other system our vehicle information system needs to be interoperable with.
The purpose of the interoperability is to exchange current location of a vehicle through the vehicle information system which needs to send this information to the traffic monitoring system.

The expected response is that the traffic monitoring system combines the current location from the vehicle with other locations of other vehicles, overlays it on a map and broadcasts the aggregate information to all subscribers.

We measure the percentage of information exchanges correctly processed.
We require that the location information is included with a probability of 99.9 %.
It means that there are some unsuccessful attempts allowed.
These can be cases, for example, when the traffic monitoring system changes its message exchange requirements and we do not manage to update our vehicle information system.
This is not only about updating the system but also installing it to all vehicles.

16) This is another example of an interoperability scenario.
It is from our running example with the national open data catalog.
Here, the artifact is the national open data catalog which needs to be interoperable with local open data catalogs to be able to harvest their catalog records.
The individual local open data catalogs are unknown prior to the national catalog run-time.
They need to be discovered at run-time.
The expected response is that the national catalog harvests dataset records from the local catalog, integrates them with other records from other harvested local catalogs and enables users to search through the records.
The measure is the number of successfully harvested records which needs to be 100 %.
Of course, assuming that the system is available, i.e. that it works.
It must be ensured that possible changes in the shared assets between the national and local catalogs which ensure interoperability are changed in a way does not affect interoperability in any way.

17) Let's now talk about interoperability tactics.
You can think about possible architectural decisions at different interoperability levels - technical interoperability, syntactic interoperability and semantic interoperability.

Regarding technical interoperability, you may enforce the same network protocol, for example HTTP.
This is quite a commonly used pattern.
HTTP is not only about how information is transferred but it also standardizes operations which can be called on a network resource, e.g. GET, PUT, POST, DELETE, etc.
This is the core of the REST principles for web services which is the basic technique for achieving technical interoperability between services on the web.
Many applications or platforms on the web today provide their services to third parties as REST services.

18) Sometimes, a common protocol and principles cannot be enforced.
For example, there may be a system which communicates through the HTTP protocol but it does not provide REST services.
It provides W3C style web services based on standards such as SOAP and WSDL.
Maybe you have heard about these standards and maybe you also heard an opinion that they are already dead.
However, this is not true.
Even though the majority of services on the web today are REST services, most of the services in the internal enterprise environments are SOAP services.
From our architectural point of view, both are just technologies and we need to ensure interoperability.
If you need REST but have SOAP, you can introduce an intermediary technical interoperability broker which is able to translate between both worlds.
It is very similar to the modifiability tactics we discussed in one of our previous lectures.

In addition to brokers, we also spoke about anticorruption layers and open host services. Get back to them please and think about how to use them to provide a translation service between both worlds.

19) There is also another problem at the technical level we have not discussed yet.
When the systems know about each other in advance, you can prepare communication channels in advance where you also preconfigure locations of the APIs provided by the systems.
However, often you do not know locations of the systems which need to be interoperable in advance.
The location must be determined at runtime.
For this you need to introduce a directory service.
It enables systems to search a directory of known systems and their available interfaces.
For the interfaces, the directory service provides technical information about where to locate them.

20) A directory service is a part of the architecture of our running example.
The national open data catalog harvests local open data catalogs of individual open data providers.
The national catalog does not know the local catalogs in advance.
Each night, when the harvesting is performed, the list of local catalogs may differ.
There is a local open data catalogs directory service where local catalogs register and unregister themselves.
The national catalog first requests the list of local catalogs and HTTPS URLs of their REST APIs.
It then harvests each obtained local catalog.

21) Regarding the syntactic interoperability, you can enforce a shared syntax between systems by using an existing standard format which is broadly accepted in a given domain.
We have already seen the standard DCAT-AP which is the standard created by european commission which shall be used by european open data catalogs for sharing their catalog records about datasets.
This standard is also used by the national open data catalog which enforces this standard to be used by local open data catalogs in Czechia.

SKOS (simple knowledge organization system) is a standard for sharing knowledge organizations systems, such as taxonomies.

DCV (data cube vocabulary) is a standard for sharing statistical observations organized in statistical data cubes.

HL7 FHIR is a standard for health care data exchange.

DATEX II is a standard for exchanging traffic related data.

Schema.org is a collaborative, community activity which produces a common format for publishing structured data on web pages.
It is driven mainly by Google and other search engine producers whose interest is to have entities represented on web pages machine readable.
This is also the interest of many web site owners so they use this common web language.

22) Similarly to the technical interoperability, it is often not possible to enforce a shared syntax.
Again, in these cases we introduce brokers which translate between different syntaxes.
The capability of a broker is to translate between different syntaxes.
For example, there exists a local czech standard for electronic health records called DASTA (you have a link in the lecture notes).
The interoperability of systems supporting DASTA with systems supporting HL7 FHIR can be ensured by introducing a syntactic interoperability broker which translates between both syntaxes.

23) For semantic interoperability, it is also possible to use standards.
All standards mentioned as the examples do not specify only the common syntax but they also specify concrete semantics of individual syntactical elements.
This is true for almost all existing standards.

24) For example, this slide shows a sample representation of a hotel expressed according to schema.org.
From the syntactical point of view, it is JSON data with key-value pairs.
A machine can recognize them but they mean nothing on their own.
Schema.org specifies their meaning.
In informatics we do not usually use the term meaning but semantics instead.

You can see the key @type set to Hotel.
This is one of the basic concepts of schema.org.
Using schema.org you express information about entities as JSON data.
A JSON expression is typed.
Actually, it is not JSON, but its extension JSON-LD.
JSON-LD is one of the formats to express data in the RDF data model.
Typing in schema.org comes from the RDF data model.
The core principle of the RDF data model is that everything is named by an IRI.
So here in the example, the value Hotel is not the string Hotel but IRI https://schema.org/Hotel.
I will give a lecture where we will need RDF in more detail at the end of the semester.
In this lecture we will not go deeper to RDF.

25) In this example, just visit that IRI.
You can see a page which does not contain only the definition of the syntax but mainly the semantics.
The syntax is defined in a form of examples down on the page.
Actually, the syntax is the trivial part of schema.org.
This is true for any standard.
What is important and hard to define is the semantics.

You can see here a description of what a hotel means.
It is a description for humans.
If you are a developer responsible for writing code to process schema.org data from another system and the data is typed as schema.org Hotel, this description tells you what the hotel is.
You can also see the long list of possible properties you can expect with their semantic descriptions.

As I said, defining semantics is hard.
For example, you can ask whether a hostel is a hotel according to schema.org or not.
Hostels provide shared-room lodging which is a special case of providing lodging.
So, according to the semantic specification for hotels, a hostel is a hotel according to schema.org.
Or you can ask whether a motel is a hotel.
A motel provides lodging paid on a short-term basis for motorists.
So a motel is also a hotel.
So if you want to represent information about a hostel or motel, you can express it as a hotel.

What you may also see in the definition of the hotel type is a structured representation of an IS-A hierarchy.
If you do not know what an IS-A hierarchy is, you can consider it as a subclassing on types.
You can see that a hotel is a lodging business.
Let’s take a look at it.

At the bottom of the page, you can see that schema.org distinguishes several subtypes of a lodging business, hostel, motel and hotel among them.
You can see that they are three different types besides each other.
This can be understood as a specification that a hotel is not a hostel or motel.
In other words, a lodging business is either a hotel or a hostel or a motel.
This would be in contradiction with the fact that a hostel or a motel is a hotel.

Here, the semantics of schema.org is not clear.
And there are more problems.
A lodging business is both an organization and a place.

It is written at the top of the page.
Do you know a place which is an organization, or an organization which is a place?

These inconsistencies in semantics make achieving interoperability hard.
Let’s consider two different systems which need to be interoperable.

The first has hotels, hostels and motels as places where accommodation is offered.
These types are considered as disjoint sets of places on a map.
It uses schema.org and expects entities properly typed as a hotel, hostel or motel.
It uses their address property and puts them on a map as points of service where an accommodation can be found.

The other has hostels which are considered as hostel chain organizations.
It uses schema.org in a different way.
It expects hostels typed as hotels because its architects are not aware of the hostel class or they simply ignore it.
Moreover, they put the headquarters to the address property.

So what happens when the second system shares information with the other system?
They both use the same syntax, so the syntactic interoperability is achieved.
However, they do not share semantics.
As a result, the first system displays headquarters of hostel chains as points of service where hotel accommodation is offered.

26) Other mentioned standards are more properly defined industrial standards.
Their semantic specification is usually much better.
This is achieved by strict formalization and validation of the defined semantics.
Properly defined semantics is more and more important as the importance of interoperability grows.
That is why today a quite old but overlooked approach called ontology engineering becomes a hot topic again.
They appeared as one of the emerging technologies in the recent Gartner’s hype cycle for 2020.

Ontology is a shared conceptualization of a given domain expressed in a machine readable form as a formal graph of semantic concepts and semantic relationships between them.
One of the reasons why ontologies are necessary are requirements to achieve better semantic interoperability and to solve the problems illustrated above with the hotel example.
I will not explain here the techniques of ontology engineering.
I can just say that using ontology engineering properly, you could explicitly define semantic relationships between places, organizations, hotels, hostels and motels so that the semantic inconsistencies would not happen.

So just remember that ontologies are another semantic interoperability tactic in the repertoire of a software architect.

27) Up to now we have talked about data structures to represent information and about their syntactic and semantic interoperability.
We should not forget about the fact that data represents information about entities.
These entities are identified.
A part of semantic interoperability is also the task to interpret the entities encoded in the exchanged data as the same entities.
For example, when a system encodes information about the InterContinental Hotel Prague, we expect that another system decodes this as information about the same hotel and not the other.

The easiest situation is when you can ensure a shared identity system where each system uses the same unique identifier for an entity.
In such a case, the InterContinental Hotel Prague has an identifier shared and understood by all interoperating systems.

However, this is often hard to achieve when you are in an open environment where interoperating systems change dynamically.
Here, each system may have its own identity management.
In that case, you have two options.

First, you may introduce an identity broker which has a capability of converting identifiers between different identity systems.
This is applicable in cases where a limited number of well-known identity systems exist.

Something between this option and shared identities is that you leave each system to use its own identity system but they also provide links between their identifiers and some other identifiers, preferably global and centrally managed.

This is the case of the GS1 digital link, you have a link in the lecture notes.
GS1 is a global organization which standardizes EAN codes, barcodes and RFID chips producers and sellers use to identify their products and services.
Their recent activity is the mentioned digital link.
The EAN codes is also an example of a global and centrally managed identity system.
You can use your own identities for products you produce or sell but you also map them to EAN codes through which you exchange information about products with your partners and other systems.
The digital link takes this a step further.
While EAN codes are just numbers which must be interpreted by each system, digital link introduces web IRIs to identify products.
These identifiers can be accessed by machines using the well-known HTTP protocol.

Any machine can automatically get data about a product using its GS1 digital link IRI directly from the producer or seller without knowing its internal identity used by the producer or seller and even without knowing who is the producer or seller and in which system the information is available.
This is the highest level of semantic interoperability at the level of individual entities.
These principles are generally called Linked Data principles which will be a topic of another lecture later.

28) So this is the end of the lecture about the interoperbility quality attribute.
See you at the other lecture.

Lecture 15: Quality Attributes - Testability

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the testability quality attribute.

2) Software testability refers to the ease with which software can be made to demonstrate its faults through testing.
We typically mean execution-based testing.
Specifically, testability refers to the probability that the software system with a fault will fail on its next test execution
If a fault is present in the system, then we want it to fail during testing as quickly as possible.

We try to achieve these quick failures by a system of automated tests which can be executed for each change in the system.
Writing the automated tests is not an architectural activity.
It is a kind of development activity.
Developers of modules at the lowest levels of decomposition write unit tests.
And testers who are responsible for testing modules at the higher level of decomposition write integration tests.

What is architectural is how to design the architecture so that it is easy to write tests which reveal problems easily and quickly.
If you design an architecture which helps with this, you save the work of developers and testers which makes the whole software project cheaper.

3) How can you help with this as a software architect?
What architectural decisions can you make to make a software system better testable?

First of all, you contribute by designing the module views such as decomposition and usage views.

Developers and testers use these views to see what modules need unit tests and what combinations of modules should be tested by integration tests.

You can do even more as a software architect.
As we have said, we typically consider execution-based testing.
Tests execute modules and test them as run-time components.
For a system to be properly testable, it must be possible to control each component’s inputs and possibly manipulate its internal state.
Then it must be possible to observe its outputs and possibly its internal state.

4) First, let’s discuss how we can specify testability requirements.
Again, we will use our structured scenarios.
In a scenario, the artifact is the portion of the system being tested.
It is usually a unit of code which corresponds to a module in the module viewpoint at some level of decomposition.
It can even be the whole system.

5) The source of stimulus is the one who performs the test.
It can be a tester on the developing organization side, i.e. a unit tester, integration tester or system tester.
It can also be a tester on the customer side, i.e. acceptance tester or end user.
We can also consider human or automated testers.

6) The stimulus is the set of tests which are executed by the source on to test the artifact.
It can be executed due to various reasons.
These reasons include completion of a coding increment such as a class, layer or service.
They may also include the completed integration, the completed implementation of the system or the delivery of the system to the customer.

7) The environment specifies when the test happens.
It can happen at development time, at integration testing time, at system testing time, at deployment time, or while the system is running.
It can also specify the test environments in use to specify and execute the tests.

8) The expected response of the testability scenario is that when the tester tests the artifact using the stimulus in the environment, the system is expected to be controllable so that it is possible to perform desired tests, the results from the tests can be observed and failures easily identified when there are some.

9) The response measure is aimed at representing how easily the system under the tests shall give up its faults.
This may include the effort to find a fault or class of faults or the effort to achieve a given percentage of state space coverage.

This may also include the probability of fault being revealed by the next test.

10) This is a generic sample testability scenario, typical for unit testing.
A unit tester completes a code unit during development which is the stimulus.
As a reaction, the code unit is unit tested.
It is expected that the unit test will capture the results of the test and that the unit tester is able to write a unit test which covers 85 % of all possible execution paths of the module in 3 man-hours.

11) This is a more concrete sample from our running example.
A system tester shall test the harvester in the national open data catalog when a new version of the DCAT standard is released and the new version of the national catalog which supports it is prepared.
The harvester harvests local open data catalogs.
The system tester has to prepare test cases and execute them.
It is expected that the results of the test are captured.
It is required that the system tester is able to write and execute test cases in 1 man-day.
The test cases must cover 100 % of cases with missing values of mandatory properties which appeared in the new version of the DCAT standard and 100 % of cases with invalid values of properties which appeared in the new version.

12) The previous scenario does not provide much time to the system tester.
So it is necessary that the architecture helps as much as possible.
How the architecture may help?
For example, it may enable to easily isolate the harvester so that it can be executed as a stand-alone runtime component and easily provide inputs in a form of individual incorrect catalog records prepared for testing.
This makes testing more transparent and easier to prepare for the system tester.

This takes us to testability tactics.

There are two categories of tactics for testability.

The first category deals with adding controllability and observability to the system.

The second deals with limiting complexity in the system's design.

13) The control and observe system state tactics cause a component to maintain some sort of state information, allow a tester to assign a value to that state information and make that information accessible to the tester on demand.
The state information might be of different kinds.
For example, it can be the value of some key variable, performance load or resetting to some state described in a UML state diagram of that component.

14) The first tactic in this category is so called specialized interfaces.
Its goal is to control or capture variable values for a component through a specialized interface which is used only for testing purposes.
This interface provides specialized test routines.
For example, this includes set and get methods for important variables, a report method that returns the full state of the component, a reset method to set the internal state to a specified internal state or a method which turns on verbose output, various levels of logging or resource monitoring.
This specialized testing interface can be removed if needed.

15) Another tactic here is record/playback.
It presumes that the state that caused a fault is often difficult to recreate.
So the tactic is to record the state when it crosses an interface of a component.
You can then recreate the fault by playing the system back using the recorded state information.
For this you need also the previous tactic because you need a testing method to recreate the state of a component.

16) And the supporting tactic is to localize state storage.
It proposes to store recorded states in a single persistent store.

17) Abstract data sources tactic allows for controlling component’s input data.
It proposes to abstract a data source interface, e.g., a database, which you can substitute a data source with another source with testing data.

18) Sandboxing refers to isolating a component from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.

19) The second category are the limit complexity tactics.
They are based on the presumption that complex software is harder to test.
A complex software has very large operating state space so it is more difficult to re-create an exact state in that large space.

20) The first limit complexity tactic is the limit structural complexity tactic.
It is based on avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components and modules in general.
Having high cohesion and loose coupling, which are the modifiability tactics, can also help with testability in this sense.
They are a form of limiting the complexity of the architectural elements by giving each element a focused task with limited interaction with other elements.

21) The second tactic in this category is the limit nondeterminism tactic.
Nondeterministic systems are harder to test than deterministic systems because it may be almost impossible to re-create its exact state.
Sources of nondeterminism can be, for example, a multi-threaded component that responds to unpredictable events or an unconstrained parallelism.
These cases must be identified, localized and isolated so that they are small as possible so that their testing is easier.
This is called limit nondeterminism.

22) So this is the end of the lecture about the testability quality attribute.
See you at the other lecture.

Lecture 16: Quality Attributes - Usability

1) Welcome to the next lecture about software architectures.
In this lecture we will talk in detail about the usability quality attribute.

2) Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.

3) It comprises the areas shown on this slide.
First, it comprises learning system features.
If the user is unfamiliar with a system or its particular feature, what can the system do to make the task of learning easier?
This may include, for example, providing help features or a guide which guides the user through the task.

4) Second, it comprises using the system efficiently.
What can the system do to make the user more efficient in its operation?
For example, when users perform complex tasks, the system may help them by enabling them to leave the task, do something else for a while and then return back to the task.
Or it may include a kind of group operations.
For example, a diagram editor may help a user to move a group of diagram elements instead of moving each one element separately.

5) The third is minimizing the impact of errors.
What can the system do so that a user error has minimal impact?
For example, the user may wish to cancel a command issued incorrectly.
Or the user may wish to restore a lost version of his work.

6) The fourth is adapting the system to user needs.
How can the system adapt itself to make the user's task easier?
This is similar to the second one but here the system evolves to help the user.

A basic example is that the system automatically fills in inputs previously entered by the user.

7) The last but not least is increasing confidence and satisfaction.
What does the system do to give the user confidence that the correct action is being taken?
For example, for a long running computation, the system may indicate that it is performing a long-running task and the extent to which the task is completed.
Our national open data catalog may confirm that a registration of a dataset is correct and accepted.
Later it may send a message that the dataset was registered.

All the five aspects have in common that they are hard to achieve inside individual modules or components of the system.
They need to be designed by a software architect as a part of the architecture, usually as separate modules or components independent of the modules and components with functional responsibilities.

For example, for automatic filling of previously entered inputs you need a module which remembers what users filled in and provides the remembered inputs to all modules which need to provide such behavior.
Undo and redo operations must also be supported by a separate module which stores the states of the system to which the user may undo or redo.

Therefore, these usability aspects are purely architectural.

8) Let’s now talk about usability scenarios.
The artifact of the scenario is the whole system, its component or its feature the user is interacting with.

9) The source of stimulus is the user interacting with the artifact.

10) The stimulus is that the user tries to use the system efficiently or learns to use the system.

11) Here the environment usually is the runtime environment.
Some systems provide a special learning mode or training mode.
For these systems, this could also be an environment.

12) The system should either provide the user with the features needed or it should anticipate the user's needs and do what the user needs automatically.
If the stimulus is to learn to use the system, then the system should support the user in learning.

13) The response is measured by task time, number of errors, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs.

14) Here is a sample usability scenario in our running example of the national open data catalog.
The user uses the dataset search feature.
He or she uses a specific part of that feature where the user selects a dataset and the system shows the list of possibly related datasets.
For example, for a dataset about public authorities it will show a dataset with their budgets and public contracts.
The response of the system regarding the usability of this feature targets the last usability aspect - the confidence of the user.
It is necessary to increase the confidence by explaining why the shown datasets are related to the original one.
We want to measure this in a way that the user gains the knowledge of the semantic relationships.
This can be measured quite easily.
You prepare a test case with a predefined dataset and a list of more or less related datasets.
You also know the correct relationship between them in advance.
You let the human tester run this test case, later you ask her to explain why the returned datasets are similar and you compare the answers with the explanations known to you.

15) This is another example where the user deletes registrations of datasets from the catalog.
Here we want the system to enable the user to delete registrations corresponding to entered search criteria within 30 seconds.
It is clear that it would not be possible if the user would have to click on each record, click a delete button and confirm the deletion.

16) The last example is from another domain - electronic drugs prescription.
It is mandatory for medical doctors to prescribe drugs to their patients electronically through a central system.
To prescribe a drug, the doctor needs his or her security personal certificate.
This must be renewed each 2 years.
It is required that the doctor is able to renew the certificate on his own with up to 3 clicks without reading any documentation.
This can be measured easily by asking some doctors.
Maybe this scenario sounds clear to you.
However, the system in its state at the year 2020 does not fulfill this scenario and certificate renewal is a very complex task, especially for doctors.

Do you remember the modifiability/performance tradeoff?
This example hides another tradeoff, the security/usability tradeoff.
Of course, you can make a perfectly secured system with n-steps of authentications and various obscure certificates.
However, the system would be useless.

17) So this was how to specify usability requirements with scenarios.
Let's now shortly talk about tactics.

Basically, there are two categories of tactics.
The first category supports the user's initiative in the human-computer interaction.
These tactics help the user to use the system by supporting his activities.

The second category supports the system's initiative in human-computer interaction.
These tactics enable the system to assist the user with automated or semi-automated steps.

18) Let's first talk about the user's initiative.
The cancel tactic means that the system must listen for a cancel request.
The command being canceled must be terminated and allocated resources used must be freed.
Also collaborating components must be informed.
So this needs a special module in the system for cancelling commands as it may go across different modules of the system.

The pause/resume tactic is similar.
It means that the system must be able to temporarily free resources so that they may be re-allocated to other tasks.
And then return them back to the task which needs to be finished.
Again, you need a module with specific responsibilities for this.

The undo tactic means that you introduce a module with a responsibility to maintain a sufficient amount of information about the system state so that an earlier state may be restored, at the user's request.
This also requires that other modules have specific interfaces which are used by resetting their state.
We had this already for testability.

And the last tactic is aggregate.
It is the ability of the system to aggregate lower-level objects into a group, so that a user operation may be applied to the group, freeing the user from the drudgery.
This usually also needs a specific module with a responsibility to execute operations for individual objects in the group.

19) The second category of tactics is about supporting the system's initiative.
They enable the system to anticipate either its own behavior or the user's intention.
This anticipation must be based on some model which enables the predictions.

You can have models of three kinds.

The first is the task model.
It is a representation of the task performed by the user.
For example, the model of grammatical rules helps to correct typing errors.
Or there exists a model trained by machine learning to recognize vague textual descriptions which can be used by the national open data catalog to warn the user that the provided dataset description is vague.

The second is the user model.
It is the model of the previous behavior and decisions of the user.
The system may use it to adapt itself to be better personalized to the user.
The basic example is prefilling forms with previously entered values.
However, the system may do more complex things.
For example, it may adapt its main menu by emphasizing most frequently used commands.

The third is the system model.
It is the model of the behavior of the system itself.
The model is used to determine expected system behavior so that appropriate feedback can be given to the user.
For example, it can be a progress bar that predicts the time needed to complete the current activity.

Again, all these tactics need a new module or modules in the architecture which maintain the model and make some decisions based on the model across different modules with functional responsibilities.
So you need to design them in your architectural documentation.

20) So this is the end of the lecture about the usability quality attribute.
See you at the other lecture.

Lecture 17: Architectural patterns - Domain-driven architecture

1) Welcome to the next lecture about software architectures.
In this lecture we will start the last part of the course where we will talk some interesting architectural patterns.

2) Architectural patterns are similar to design patterns.
A design pattern is a recommended coding practice which is a well-known solution to well-known problems caused by some recurring common situation.

3) For example, you probably know iterators.
The problem is that often you need to be able to iterate collections of things.
The problem is how to do this when you do not know anything about the internal structure of the iterated collection.
So there is a recommended practice - a design pattern called iterator.
It describes a solution where an entity called iterator is introduced separately from a collection which shall be traversed.
The iterator encapsulates accessing and traversing the entities in the collection.
Clients then use the iterator to access and traverse entities in the collection without any dependence on the internal data structure of the collection.
The collection itself can be organized as an array, a matrix, a tree, a graph or anything else.
Independently of the internal structure, the iterator provides a well-known interface to access the current entity in the collection and move to the next item in the collection.

4) An architectural pattern is similar.
It is a recommended architectural practice which is a well-known solution to well-known architectural problems caused by some recurring common situation.
So what is a difference between design and architectural patterns?

5) A design pattern is a more fine grained solution to a known particular programming problem.
An architectural pattern targets one or more quality attributes and the whole architecture of a software system or its part.
There is not a strict border between design and architectural patterns.
For us, we will call a pattern an architectural pattern if it introduces more modules and relationships between modules or more components and connectors into architecture design.
So even patterns called design patterns can be considered as architectural patterns.
The iterator pattern is not this case.

It talks about a specific coding practice which is applied inside the code of the modules but does not impact the architecture unless you do not specify architectural modules at a detailed level where you would have an iterator as an architectural module.

6) A well-known observer pattern can be applied at the code level as well as at the architectural level.
It is applicable in cases where observers need to be notified by a subject when something significant happens.
You can apply it in the code inside a module or at the architectural level where you introduce subject and observer modules.

7) In the previous lectures we already talked about some simpler patterns.
For example, we talked about routers, load balancers, brokers and application programming interfaces (APIs).

A load balancer helps with balancing load on a component by forwarding incoming requests to multiple copies of that component.

A router is similar but it has some additional logic for forwarding requests on the base of the content of the requests.
Both are useful for fulfilling performance and partly also availability requirements.

A broker helps with reducing coupling between modules or components by forbidding direct dependencies among them and allowing only dependencies on the broker.
In enterprise environments a specific kind of broker is used called enterprise service bus.
An enterprise service bus is a central point of an architecture.
All other services in the architecture are registered to the bus.
When a service needs something, it requests the bus.
The bus orchestrates the other registered services to fulfill the request and responds to the calling service.

An application programming interface is a public interface published by a component.
Other components or systems may use the services of the component only through the interface.
This helps with reducing coupling but also with integrability of the system and partly also with its security.

Each pattern solves some problems but also has disadvantages of course.
For example, a broker or its extended variant enterprise service bus introduces communication and processing overhead so it results in slower communication speed.
Maybe you had components which have already been able to communicate before.
What you cause by introducing a bus is that their communication is slower.

This is not an improvement from the point of view of the people responsible for these components.
They may not be interested that now components are more loosely coupled thanks to the broker.
What they care for is the speed.
Therefore, they may become opponents to the new concept which may cause future socio-technical problems.
So you need to be careful with patterns.
Think not only about their advantages but also try to analyse their negative impact to the system qualities and also how it will affect stakeholders, especially the technical ones.

8) In this lecture we will talk about a more abstract architectural pattern.
It defines a software architecture as a whole so it could be called an architectural style.
Let’s start with something simple.
It is an architectural pattern called layer pattern.

The recurring common situation is that in a complex system we often need to develop and evolve portions of the system independently of each other.

The problem here is that when we do this in a wrong way, we create dependencies between different parts.
As a result, developing and evolving one part has undefined impact on several other parts and the development becomes hard to manage and it is harder and harder to develop, evolve, test and deploy the system.
We have discussed this when we talked about the modifiability quality attribute.
To solve this problem we need to increase cohesion and reduce coupling.

A common solution to this problem is to decompose the system to layers.

9) The layer pattern divides the architecture into modules called layers.
Each layer is a grouping of modules that offers a cohesive set of services.
Layers completely partition the architecture and each partition is exposed through a well-defined public interface.
The usage between layers is unidirectional and it is depicted by layouting the layers in the decomposition view.

On the slide, you can see a sample layered architecture.
It is our running example, the national open data catalog.
We read it from the top left corner to the bottom right corner and in this direction we also interpret the usage between the layers.
Each layer may use the layer directly below it or on its right hand side.
So the presentation layer is allowed to use the API layer, the API layer is allowed to use the Application services layer, and so on.

The Application services, Domain model and Data sources layer can also use the Infrastructure layer.

Actually, many software systems have a layered architecture.
It is simple, clear and easy to design and explain.
However, it also has drawbacks.

One of them, important for this lecture, is that it usually forces us to put technical and infrastructural responsibilities to the lower levels.
When something needs to be changed here it may have an undesirable impact to the higher levels even though we do our best to separate concerns.

10) One small note - layer pattern is a pattern for the module viewpoint.
It specifies that the source code is structured to layers.
However, it does not speak about the component-and-connectors viewpoint.
From this runtime point of view, it is still possible to have an architecture with one big monolithic component where all layers are instantiated as parts of a single executable unit or an architecture where each layer is instantiated as one or more stand-alone network components communicating through their network interfaces.
These are two extremes and everything between them is also possible.

11) There is another approach to layering called a domain-driven pattern.
It proposes another kind of layering which is centered around the problem domain for which we build a software solution.
The basic principle of this pattern is the separation of technical complexities from the complexities of the problem domain.
The assumption is that presentation, persistence and domain logic responsibilities usually change at different rates and for different reasons.
An architecture that separates these concerns can accommodate change without causing an undesired effect to unrelated modules.

12) The domain-driven pattern puts the domain layer at the center of the architecture.
It contains all the logic important for the business.
It is a conceptual abstract view of the problem domain created to fulfill the needs of the application use-cases.
It does not depend on anything else.
It is agnostic to the technicalities of the other parts of the system which are responsible for communication with human users or with other systems or for data stores.
Domain logic represented at the domain layer is focused only on domain rules, domain concepts and workflows.
It is free from any technical details, including model persistence.

It would not be correct to call the domain layer as the lowest level.

It is really at the center of the architecture and it is independent of the other modules in the architecture.
This is why we depict it as a circle in the centre.

13) On top of the domain layer we have the application service layer.
It represents application use cases and application behavior.
Use cases are implemented as application services at this layer.
An application service should be ignorant to what consumes its functionality.
It should not bend to meet the needs of its clients.
Instead, it should expose an application use case.
An application service contains application logic which coordinates the fulfillment of the use case by delegating to the domain layer and the other layers.
Domain layer provides the detailed domain logic.
It has all responsibilities important for the business domain.
It's fine grained logic is however exposed only to the application layer which encapsulates the business logic as a coarse-grained set of services which hide the details of the domain layer.
The application layer exposes what the system does but hides how it does it.
The application layer creates a façade around the domain logic.
The domain logic can evolve independently of other parts and the façade ensures that changes to the domain logic do not affect the other parts.

The application layer does not contain any business logic.
It contains only the application logic.
The application logic contains the workflow steps required to fulfill application use cases.
These steps include hydrating of domain objects from the database, validating user input, mapping it to domain objects, delegating to domain objects to perform some domain work and collecting business decisions made at the domain layer.
The steps may also include delegating to other layers as you will see later.

So the application logic is all about coordination and orchestration through delegation to domain layer and other layers.
The application services do not do any work, but they understand who to talk to to complete the task.
They should be procedural and thin.

14) The application layer cannot persist the domain objects, it cannot present information to human users and it cannot notify anyone nor anything on its own.
For this we have other layers in the domain-driven architecture which are called infrastructural layers.
To perform its work, the application layer also uses these layers.

The infrastructural layers provide the technical logic which enables the application to function.

The application and domain layers are focused on modeling behavior and business logic, respectively, the infrastructural layers are concerned with purely technical capabilities which, for example, enable the application to be consumed by human users via a graphical user interface or by applications via an API.
The infrastructural layers are also responsible for the persisting the state of domain objects, logging, security, notification and integration.

15) As has been said, the domain and application layers at the center of the architecture should not depend on any other layers.
All dependencies in the architecture face inwards.
Of course, the application layer depends on the existence of the infrastructural layers and their responsibilities.
However, it does not depend on any particular technology, framework or language used by the infrastructural layers.
It also does not depend on their interfaces.
Instead it defines its own interfaces which are implemented by the infrastructural layers.

For example, it defines an interface that enables domain objects to be hydrated and persisted.
The interface is written from the perspective of the application layer.
The infrastructural persistence layer then implements and adapts to this interface.

The application service layer uses the infrastructural layers (for example, for persistence or logging) and the infrastructural layers use the application service layer (for example, the presentation layer uses the application service layer to call some application service use cases).
In any case, the application service interfaces are implemented on the side of the infrastructural layers but not vice versa.

16) Let me try to explain this in our running example of the national open data catalog.
Let’s have a simple use case - showing a dataset detail.
The human user in the web front-end of the national open data catalog chooses a dataset from a given list.
The web front-end sends a request to the presentation API layer.
This is where our domain-driven architecture starts.
The request is an HTTP request sent through the internet.

17) The presentation API layer is responsible for receiving and transforming the received request into a form expected by the application service layer, and for calling the respective application service for getting the detail of the dataset.
In the runtime component-and-connector viewpoint that would be transformation of the HTTP request and either calling an application service method inside a single process, or interprocess communication on a single node or calling another microservice providing the application service on another network node.

However, this runtime aspect is not important for the domain-driven point of view.

18) The called application service then orchestrates the processing to prepare the dataset detail.
First, it delegates to the persistence layer to retrieve a catalog record about the dataset.
The delegation is performed by the application service by calling an operation provided by the persistence infrastructural layer.
The operation conforms to the persistence interface defined by the application service layer, not the persistence layer.

19) After the application service obtains metadata about the dataset from the persistence layer it delegates it to the domain layer.
A corresponding domain object representing the dataset is created.

20) The application service then delegates necessary decisions and business actions to this domain object.
Usually, for showing a detail of something it is not necessary to do any business work.
However, this is not the case of showing a dataset detail in the national open data catalog.
The catalog also presents to the users the list of related datasets.
The logic of getting related datasets consists of the business logic which constructs the list of metadata search parameters.
This is the business logic so it is delegated to the domain layer.
The dataset domain object performs this construction.

21) The application service takes the parameters, transforms them to a new query to the persistence layer and calls the persistence layer once more.

22) The persistence layer returns metadata about found datasets.
The application layer calls the domain layer to construct domain objects also for these new datasets.

23) In the end the application layer transforms the metadata about the datasets from their domain representation to the response for the presentation API layer.
The presentation API layer transforms this to the HTTP response and sends the response to the front end.

Here the work ends.

24) The presented layered architecture tries to prevent exposing the details of the domain model in the domain layer to the outside world.
The application layer hides the domain model.
However, the existence of the application layer itself cannot ensure this.

If the application layer would provide domain objects to the infrastructural layers, the programmers of the infrastructural layers would create dependencies on the domain model.
This would make the domain-driven architecture useless.
Therefore, it is important not to pass domain objects across the boundary between the application layer and the infrastructural layers.
On the other hand, you should never send raw data or user input to the domain layer.

Instead you use simple data transfer objects which just encapsulate data in its simplest possible form.
The simplest possible form practically means no logic.
Either they are plain value objects or JSON messages, higher layers adapt to the data structures defined by the lower layers.

25) So, in our running example, when the presentation API layer calls the application service which shows a dataset detail, it passes a simple value object which carries the dataset ID but no logic.
The value object has no methods except value getters and setters.
After the application logic for constructing a dataset detail is finished, the result is the dataset detail including also previews of other related datasets.
This is returned as another use-case specific value object.

Communication with the persistence layer is similar.
The application service calls the persistence layer by a method and with a data structure defined by the application layer.
It defines value objects through which a request to hydrate a dataset is sent and a response with the hydrated dataset is returned.

In both cases, the application layer transforms the value objects to data structures defined by the domain layer where true domain objects are constructed and returned to the application layer.
But the domain objects are never sent to the infrastructural layers.

26) You can apply the domain-driven pattern in any software system.
However, it cannot be recommended generally and in all cases.
The dependency inversion which puts the domain model into the center of the architecture without dependencies on the other parts of the architecture may make things more complicated than necessary.

27) Let’s discuss maybe the predominant software architecture pattern for client-server applications.
It is called a three-tier architectural pattern.
It is also a kind of a layer architecture.
It has three parts - presentation tier, application tier, data tier.

They are called tiers but actually they are layers.
But we will stay with the terminology of this pattern so we will call them tiers.
The principle is simple.

28) The presentation tier is the front-end layer.
It provides the user interface.
Its main responsibility is to display information to and collect information from the user.
If we talk about web applications, the presentation tier is usually developed using HTML, CSS and JavaScript, either natively or using some web frontend framework such as React JS or Svelte JS.

29) The application tier contains the business logic.
It is responsible for application use cases which consist of collecting information from the presentation tier, processing it according to the business logic and returning information for presentation to the presentation tier.
As a part of these responsibilities, the application tier may need to load data from the data tier or modify the data.
The application tier can be developed in any programming language incl. PHP, Java, C#, … .
Various frameworks exist for these languages.

30) The data tier, also known as a database tier, is a data storage system, usually a database system and a data access code around it.
The database system can be a relational database management system such as PostreSQL, MySQL or commercial one such as Oracle or Microsoft SQL Server.
It can also be a NoSQL database server or a multimodal database server such as MongoDB or Cosmos DB.
Or it can be a graph database such as Neo4J, Virtuoso or GraphDB.
More different database systems can be combined in the data tier.

31) The three-tier architecture has many benefits.
It provides a good separation of concerns from the technical and development point of view.
Each tier can be developed by a separate team of specialists.
Each tier can run on a separate operating system or server platform.
And each tier can run on at least one dedicated hardware or virtual server.
These properties make the development faster as teams are separated.
They improve scalability as the tiers can be scaled independently of the others.
They improve availability as an outage in one tier can be masked more easily.
They also improve security because the presentation and data tier cannot communicate directly.
Modifiability and integrability is also improved because the separation increases cohesion and reduces coupling.

The application tier can be object oriented but it does not have to be.
Object orientation has also some overhead as it has additional requirements on the skills of the programmers.
(We should note that not each code with classes and methods is object oriented.)
The application server code mixes the domain logic, application logic and infrastructural logic.
And this is OK in many cases as it is easy and straightforward for the majority of developers.
The code is built quickly and testing is also straightforward.

However, the benefits based on the simplicity have their price.
The three-tier architecture concentrates only on the technical challenges of software development, deployment and maintenance.
This may quickly lead to a mismatch between the terminology and logic of the code invented by the developers and the terminology and logic of the business people who use the system, who have requirements on the system or who pay for the system.
As the time goes and the system grows it is harder and harder to identify what needs to be changed as new business requirements appear.
Just because the terminology and the logic of the business is not visible in the code.
It is present in the code but in its own technical way invented by the developers.
The mapping between the two worlds blurs as the time passes and in the end it may even become incompatible.

The domain-driven architecture, and the domain-driven development approach in general, tries to solve this problem, by isolating the domain logic (business logic and terminology) from the application logic (application use cases and terminology) and from the infrastructural logic (purely technical thing not interesting for the business).
The main principle is not to mess the domain logic with the logic of application use cases and with the infrastructural (technical) logic.

32) Orientation to the business terminology and logic has however another consequence.
As we have already seen, the domain model living at the domain layer is an abstraction of the business domain.
To build the domain model we have analysts in our team who interview the stakeholders and analyse their needs and business processes.
The result may be a quite big and complex domain model which reflects the business reality and simplifies it for the needs of the application.
However, it happens that different business stakeholders emphasize different needs.
They may talk about the same business concept but for each of them a different part of the concept reality is important.
It may even happen that they talk about the same business concept but they use a different term for it in their language.
Or they may use the same term but they use it to denote a different semantic concept.

In other words, business stakeholders do not share their business terminology and business logic.
There are different business stakeholders each emphasizing a different part of reality and using a different terminology.
The domain-driven development approach targets this problem by introducing the concept of subdomains.

On this slide we have the problem domain from our running example - Open Data Catalog.
From this slide, it seems that the problem domain is homogeneous and that its abstraction can be modeled as a single domain model.

33) When we take a closer look, we will find out that the domain is not homogeneous at all.
There are different parts, subdomains, which are quite different.
This slide shows how the problem domain of open data catalog is partitioned to subdomains.
Search represents the area where data consumers search for datasets registered in the catalog.
Registration represents the area where data publishers register their datasets to the catalog and manage their existing registrations.
Notification is the area of the problem domain where data consumers may register themselves to notifications about changes in the datasets and receive these notifications.
Quality is the area of the problem domain which includes all rules and processes for measuring quality of the recorded datasets.
Preview is the area of the problem domain where data consumers may preview datasets they find in the catalog.
Reporting is the area of the problem domain where the status of the catalog is reported in a form of reports for the management.
Semantic annotations is the last area of the problem domain where datasets are annotated by concepts from various semantic taxonomies and vocabularies which specify the semantics of the datasets.
The areas shown on the slide are subdomains of the problem domain.
Domain analysts should identify them in collaboration with business stakeholders.

34) The analysts should also identify so-called core domains.
A core domain is a subdomain which is crucial for the business.
In this subdomain, the business creates added values and makes money.
The other subdomains are only supportive or not important.

Why is this important for you as a software architect?
First of all, they give you high-level information about what is critically important and what is less important in the business domain.
The other subdomains are less important.

You do not need to build a unique and perfect solution for these less important subdomains.
You do not need to invest many resources here.
What about reusing existing open-source solutions here?
The result will not be unique here - it will offer the same features as other solutions using the same open-source.
But who cares?
What is important is that we save resources for the core domains.

35) For each subdomain, the analysts give you the subdomain model.
It consists of a conceptual information model and a use case model.
It can be expressed semi formally in UML, pseudo formally in a UML-like notation, or informally in a plain text or even verbosely.
All forms are acceptable depending on a given project.
We could talk about advantages and disadvantages of each form but it is not the point of this lecture.

What goes to the point of this lecture is that now you have different models for each subdomain.
It does not tell you that you should design 4 software architectures for 4 separate systems.
It tells you that there are differences in how business understands itself.
It tells you how responsibilities of the whole system are separated from the business point of view.
The domain-driven architectural pattern tells you that you should reflect this in your architecture when you think about how responsibilities should be decomposed to different modules in your decomposition viewpoint.
This helps you with keeping cohesion and coupling at acceptable levels.

36) Let’s consider the concept of a dataset in our problem domain of open data catalog.
Every business stakeholder speaks about datasets.
From what they say our analysts conclude that it is a shared concept among the stakeholders.
They understand it as a logical collection of data about entities of the same kind.
For example, a collection of data about all addresses in Czechia is a dataset.
As the analysts continue in discussions with the stakeholders they get a list of various properties of a dataset and its relationships with other concepts in the problem domain.

37) They may end up with a very complex conceptual model.
On this slide it is expressed as a simplified pseudo-UML class diagram without unnecessary details.
The model on the slide covers the whole problem domain, all views of the business stakeholders.

When the problem domain corresponds to the domain of the whole enterprise (i.e. whole business of a company), it is called enterprise model.
The model in our example is simple.
Real enterprise models may contain hundreds and often thousands of concepts.
The problem with such models is that they are hard to read, understand, maintain and evolve.
Therefore, sooner or later they become useless as the difference between the enterprise model and the software, its source code and data structures, grows.
At some point of time, it does not tell us much about the software.
We cannot use it to explain our software, to introduce novice developers to the project, to analyse impact of new feature requests, etc.
Moreover, all the details are not important to all people involved.
A developer working on quality reporting does not need to know the details of dataset previews.

38) If you follow the classical approach of putting the business and application logic to a monolithic application tier you end up with all the complexity at one place.
Different application use-cases need a different part of the reality of datasets and different business functions.
But we mix them together in a single shared domain model.
As a result, code can become excessively complex and collaboration overhead between teams can become excessively costly.

39) Switching the architecture from the three-tier pattern to the domain-driven pattern does not help here to solve the problem.
Yes, the domain model now is isolated but this is not the problem being discussed.

40) As we have seen in the previous slides, we can divide the problem domain to subdomains and try to conquer the conceptual model in parts corresponding to the subdomains.
The subdomains which were identified as not core domains do not have a model at all.
The core domains have their own conceptual model.

The slide shows three models for our three core subdomains.
Instead of having a single concept of dataset, we have three different dataset concepts for each subdomain

Modeling each subdomain separately is easier.
Less concepts have to be captured in a conceptual model of a subdomain.
Each separate model is simpler, easier to read and learn.
A change in one model does not affect the other models, therefore the models are loosely coupled, which makes their evolution easier.
Moreover, it is not necessary to resolve inconsistencies and conflicts which exist in the problem domain as a whole but not inside the subdomains.

41) Each model defines a so-called bounded context which is another important concept of domain driven development.
It is not necessary that each subdomain has its own conceptual model.
Some subdomains may share their conceptual model and differ only in the behavior.
Similarly, when a model of a subdomain is still too complex it may be further decomposed to more models.
The resulting bounded contexts provide the architect with information for deciding about the architectural decomposition.
Basically, the domain-driven pattern tells you to make a separate module for each bounded context.
These modules are then each organized according to the domain-driven pattern.
As a result you have clearly separated modules with high cohesion and low coupling.
This has an overhead of course because the architecture is not as simple as it would be when we applied the well-known three-tier pattern.
But when you have a more complex and evolving business domain where business stakeholders do not share their terminology and business needs, you should consider this kind of architecture.

42) This is still not the end of the lecture.
If you are thinking now that I exaggerated the reduction of coupling, you are right.
Now we have totally isolated bounded contexts which need to be integrated - they will be part of a single application in the end.
Each has its own domain model and is responsible for different application use cases from different subdomains.
However, the use cases from different subdomains cannot be isolated from each other.
For example, when a new dataset is registered to the national open data catalog, the registration cannot happen only in the bounded context responsible for registrations.
It also has to be propagated to the other bounded contexts.
So there are dependencies between the bounded contexts.
The domain-driven pattern tells you that you have to minimize these dependencies to keep them loosely coupled.

43) One possibility is to integrate the bounded contexts through a shared database.
Each bounded context has an infrastructural layer responsible for data persistence using this shared database.
However, this solution introduces strong coupling.
When a team responsible for one of the bounded contexts needs to change something in the database schema, it can unexpectedly influence the other bounded contexts which depend on the current database structure.
This architecture is not prohibited but it is not a recommended practice for the domain-driven pattern.

44) The recommended approach to achieve a pure domain-driven architecture is to use a so-called shared-nothing architecture where each bounded context has its own codebase, its own datastore and even its own development team.
The bounded contexts communicate via well defined and explicit contracts.
For example, a new dataset registration is a use case within one of the bounded contexts.
The application layer forwards to the domain layer for the business logic and to the persistence layer to store the dataset metadata to a database dedicated for this bounded context.
Moreover, the bounded context also contains an infrastructural layer responsible for notifying other bounded contexts.
The application layer forwards the information about the new dataset to the infrastructural layer.
The infrastructural layer calls the other bounded contexts, respectively their infrastructural layers, through an explicitly defined contract.
It depends on the designed component-and-connector architecture whether this means method calls inside a single process or some kind of interprocess or network calls.

45) The separation shown on the previous one is an ideal from the domain-driven point of view.
However, it will take more effort to implement for the development team who is required to have advanced skills and high discipline.
If this is a problem, you can choose a database integration.
On the slide, we have the already discussed solution with the shared database.
However, this is not a database integration.
If you need the shared database and you are able to manage the problems caused by the strong dependency introduced by the shared database, you probably do not need more bounded contexts and maybe the domain-driven pattern at all.
It either means that your domain is simple without subdomains or that your domain is complex but highly consistent and there is a strong consensus between various business stakeholders about the terminology, business needs, goals and processes.

46) Database integration means that the infrastructural layer responsible for informing other business contexts does not call the other bounded context through their contracts but it directly writes to its database.

And one small note, we are talking about the decomposition viewpoint not about the component-and-connector viewpoint.
Therefore, at runtime the databases may be located in a single database system but as separate databases with their own separated database schemas.

47) The above mentioned integration possibilities were applicable for data integration.
When a bounded context needs a functionality of another bounded context, the database integration does not help.

The functionality of the other context needs to be called.
Here, we need the solution where the contexts are integrated through the defined contracts.

48) The direct contracts between pairs of bounded contexts is also a kind of coupling.
In case of higher number of bounded contexts or in case of bounded contexts which are external and which are not under your control, this coupling could also be a problem.
If this is a problem it is recommended to introduce additional structures to the architecture to reduce coupling between the bounded contexts according to the modifiability tactics we have discussed in the lecture devoted to modifiability.
For example, you can introduce an anticorruption layers and open host services.

And you can go even further and integrate bounded contexts using the event driven architectural pattern.
We do not have space to discuss it in more detail.
But the basic concept is not hard.
Instead of direct calls of defined contracts, each bounded context has an infrastructural layer which translates business as well as technical events which happened in the bounded context as asynchronous messages about these events and puts them on a queue, either central or individual queues of other contexts.
Here the message remains until it is successfully processed or discarded according to some business logic.

49) Up to now we have talked about decomposition viewpoint.
All the examples you have seen in the previous slides about the domain drive pattern were decomposition views.
What about runtime?
What about the component-and-connector viewpoint?
We have talked about it a little at the beginning of the lecture.
This slide is a reminder.
It was about how a layered decomposition architecture may exist at runtime.
We have seen that it may be a runtime monolith on one hand or distributed network components on the other hand.

50) For domain-driven architecture, it can be the same.
In the component-and-connector viewpoint we may decide to design only two runtime components - a monolithic application server and a monolithic database server.
It means that all bounded contexts and their layers are finally compiled and built to a single executable unit which is running on a single network node.
The databases are managed by one monolithic database server which is another network node called by the application server.
It is possible to scale only the application server as a whole or a database server as a whole.

However, with detailed project management it is possible to exploit the other advantages of the domain-driven pattern, especially separation of concerns between the bounded contexts.

51) You can break the database monolith by having a separate database server for each database.
This enables you to scale individual databases as necessary.
Also separation of concerns is more easily achieved for the data of different bounded contexts as it is harder to create dependencies.

52) If you are not happy with the monolithic application server, you can break it to a separate network service for each bounded context.
The connector between services is usually HTTP and interfaces are exposed to the network as HTTP REST services.

53) We are almost at the end of this lecture.
Just the last two architectural patterns, only briefly.
The first is service-oriented architecture.
[DEFINITION ON THE SLIDE]

54) There are 8 key principles which need to be followed.

Standardization means that there are organizational standards for defining service contracts starting in naming conventions, data formats and exchanged data semantics.
It is possible to incorporate external data standards, e.g. schema.org or develop own standards.

Loose-coupling is the concept we are already familiar with from the lecture about maintainability.

Abstraction means that only information about a service is published which is absolutely necessary for the consumers.
This is also what we already know from the modifiability lecture.
It is a tactic to reduce coupling.

Reusability means that you design a service with its possible future reusability in mind.
In other words, you have to count with the fact that the service can be reused by any other service or other system anytimes in the future.

Autonomy means that services are autonomous in controlling their runtime and design.
So a change in another service does not influence the runtime and design of the service and vice versa when the runtime or design of the service is changed, other services are not affected.

Statelessness means that services cannot maintain state.

Discoverability means that services can be located through a service repository.

Composability means that when a new service is being designed and built we first try to compose it from other existing services.

55) If you go back to the domain-driven pattern, you can see that the result where bounded contexts exist as separate services in the runtime architecture is almost a service-oriented architecture.
The services are loosely-coupled and abstract.
The SOA principles of standardization and statelessness are useful principles which can be followed by a domain-driven architecture as well.
Discoverability can be useful in some cases but it is not crucial.
If each service has its own database as we have on the slide, services are autonomous.

What is problematic from the domain-driven point of view are reusability and composability.
Theoretically, they could be useful.
However, reusing a service A in a service B means that the domain model of A can leak to the domain model of B.
So reuse is possible but you need to be careful.

56) The last architectural pattern we will shortly discuss here are microservices.
We do not need to add much to the previous slides to get to the microservices architecture.
Microservices are services from the previous slide with all the characteristics.
There are some additional details.

First, microservices must be strictly autonomous so having their own database is a must.

Second, they are typically smaller than services corresponding to whole bounded contexts.
An unwritten rule is that they should be less than a thousand lines of code.

Third, because they are so small, data consistency becomes really a problem.
In a monolith it is easy as every transaction is a simple database transaction.
However, with microservices there is no single database but many.
2PC commit protocol is not a solution because with microservices we have distributed and long running transactions.
In this case we need sagas.
A saga is a sequence of local transactions each running on a separate microservice.
If a local transaction fails then saga executes a series of compensating transactions that undo the changes that were already made by the finished local transactions.

This requires that microservices do only compensable transactions.
For example, the effects to the real world cannot happen earlier than the saga confirms that all local transactions are finished.

There are many other benefits and drawbacks of the microservices architectural pattern but they are out of scope of this lecture.
If you are interested I recommend you to start your exploration of microservices with an article linked from the lecture notes.

57) So this is all from the lecture about the domain-driven pattern and also other architectural patterns.