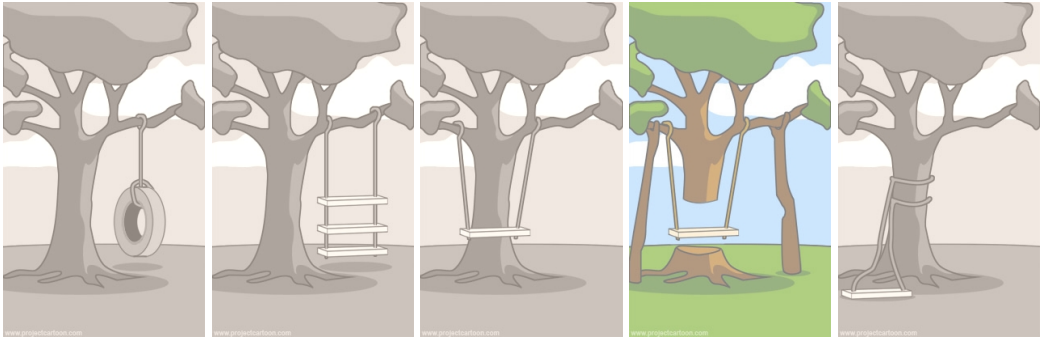
Software Engineering

4. Software Architecture | Thomas Thüm | November 26, 2020

Why Software Architecture?



what the customer really needed | how the customer explained it | how the project leader understood it | how the analyst designed it | how the programmer implemented it

Lecture Overview

1. Introduction to Software Architecture
2. Modeling Structure with Component Diagrams
3. Common Architectural Patterns

Introduction to Software Architecture

On the Role of Architecture

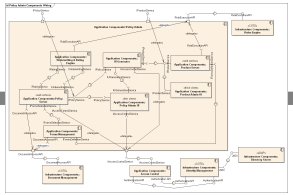
Architecture Bridges the Gap

Large software systems . . .

- have numerous requirements
- require many developers
- need separation of concerns
  (Trennung von Belangen)

"Weeks of coding can save you hours of planning." [anon]

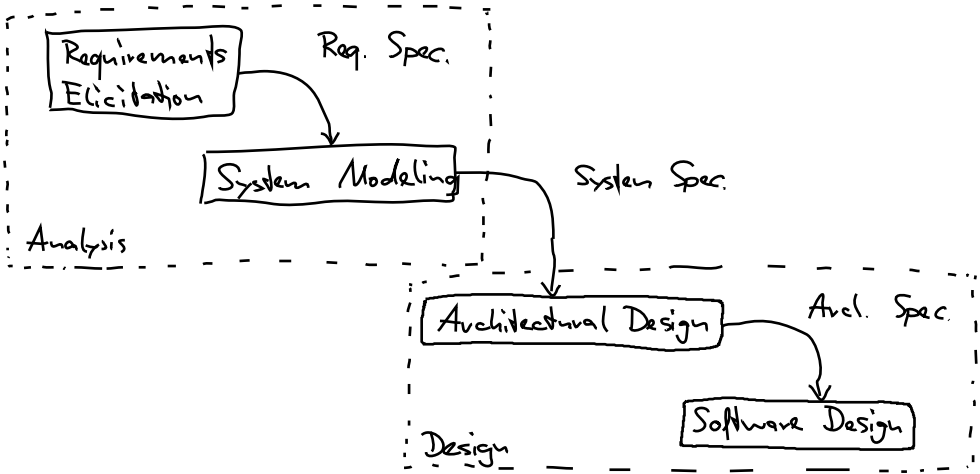
Requirements



Software Architecture

Implementation

Analysis and Design



Software Architecture

Architectural Design (Architekturentwurf)

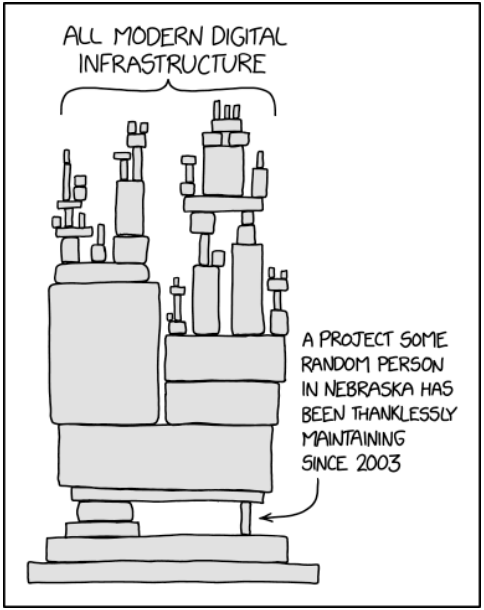
"**Architectural design** is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system." [Sommerville]

Software Architecture

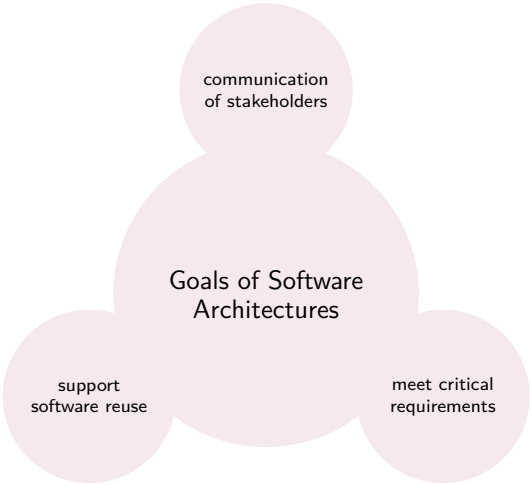
"A **software architecture** is a description of how a software system is organized. Properties of a system such as performance, security, and availability are influenced by the architecture used." [Sommerville]

In Practice: **[Sommerville]**

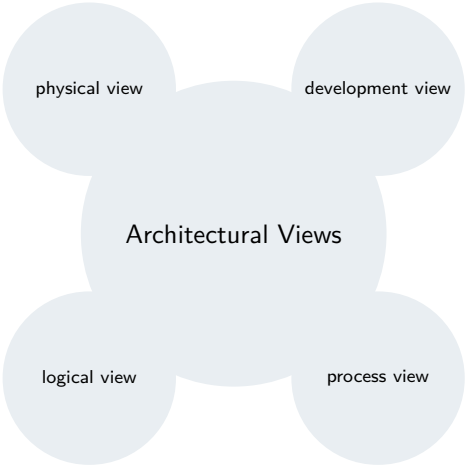
"You might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or subsystems. You then use this decomposition to discuss the requirements and more detailed features of the system with stakeholders."


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

3 Goals of Software Architecture [Sommerville]



4 Views in Software Architecture [Sommerville]



Sommerville:

"A **logical view**, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.
A **process view**, which shows how, at runtime, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
A **development view**, which shows how the software is decomposed for development; that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
A **physical view**, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment."

Conway's Law **[Melvin E. Conway, 1968]**

"Any organization that designs a system [...] will produce a design whose structure is a copy of the organization's communication structure."

Introduction to Software Architecture

Lessons Learned

- What is software architecture?
- Why is software architecture so important?
- Further Reading: Sommerville, Chapter 6.0–6.2 (p. 167–175)

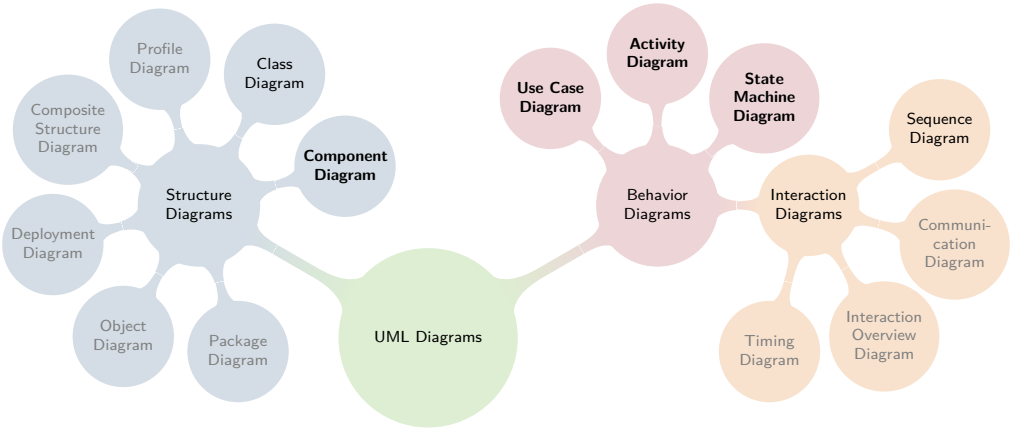
Practice

- Invest five minutes trying to understand the Corona-Warn-App by inspecting the source code of the Android client: `https://github.com/corona-warn-app/cwa-app-android/tree/main/Corona-Warn-App/src/main/java/de/rki/coronawarnapp`
- Summarize what you learned about the app by answering a questionnaire in Moodle: `https://moodle.uni-ulm.de/mod/choice/view.php?id=293405`

Lecture Contents

Modeling Structure with Component Diagrams

Recap: 14 Types of UML Diagrams [UML 2.5.1]



Component Diagrams

Component Diagram (Komponentendiagramm)

A **component** is a replaceable part of a system that conforms to and provides the realization of a set of interfaces. An **interface** is a collection of operations that specify a service that is provided by or requested from a class or component. An interface that a component realizes is called a **provided interface**, meaning an interface that the component provides as a service to other components. The interface that a component uses is called a **required interface**, meaning an interface that the component conforms to when requesting services from other components. (Komponente, angebotene/benötigte Schnittstelle) [adapted from UML User Guide]

Example of a Component Diagram

Hierarchical Component Diagrams

Nesting of Components (Verschachtelung)

Motivation: decompose/structure large systems
Nesting: A component may contain any number of **subcomponents**. (Teilkomponenten)
Ports and Delegates: A **port** is an explicit window into an encapsulated component. A **delegate** connects provided or required interfaces with ports. [adapted from UML User Guide]

Rules for Component Diagrams

Rules for Component Diagrams

- component names are unique
- a component may have any number of required or provided interfaces
- every required interface is connected to provided interface
- every component is directly or indirectly connected to every other component
- subcomponents may be nested to any level
- when subcomponents communicate to a higher-level component, they need to communicate via ports

Gordon Bell:

"The cheapest, fastest, and most reliable components are those that aren't there."

Modeling Structure with Component Diagrams

Lessons Learned

- How to describe architectures with UML component diagrams?
- How to decompose large systems with nesting?
- Further Reading: UML User Guide, Chapter 15

Practice

- Design the architecture of a contract tracing app with a component diagram and submit it in Moodle: `https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=1991`
- Give a positive vote for one other diagram and give feedback to others if you find any potential problems.

Lecture Contents

Common Architectural Patterns

Architectural Patterns

Architectural Pattern (Architekturmuster)

"Architectural patterns capture the essence of an architecture that has been used in different software systems. [...] Architectural patterns are a means of reusing knowledge about generic system architectures." [Sommerville]

Goals

- preserve knowledge of software architects
- reuse of established architectures
- enable efficient communication

Layered Architecture (Schichtenarchitektur)

Layered Architecture **[Sommerville]**

- **Problem**: subsystems are hard to adapt and replace
- **Idea**: decomposition into layers (Schichten)
- layer provides services to layers above
- layer delegates subtasks to layers below
- strict layers: every layer can only access the next layer
- relaxed layers: every layer can access all layers below
- information hiding: layers hide implementation details behind interface

Client-Server Architecture (2-Schichten-Architektur)

Client-Server Architecture (aka. 2-Tier)

- **Problem**: several clients need to access the same data
- **Idea**: separation of application (client) and data management (server)
- clients initiate the communication with a server
- typical: multiple clients of the same kind
- optional: multiple clients of different kinds

[Sommerville]

Example

a browser uses a URL to connect to a server in the world wide web and receives an HTML page

3-Tier Architecture (3-Schichten-Architektur)

3-Tier Architecture

- **Problem**: clients with same functionality but different presentation needed
- **Idea**: separation of data presentation, application logic, and data management
- thin-client application: application logic on the server
- rich-client application: application logic in the client

Rule of Thumb

If you can use the application offline, then it is most likely a rich-client application.

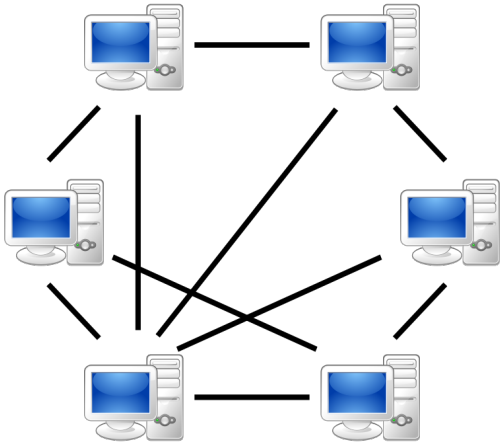
Peer-to-Peer Architecture

Peer-to-Peer Architecture

- **Problem**: high load on server and high risk of failure when transmitting all client data to the server
- **Idea**: decentralized transmission of data
- peers connect to each other and transfer data directly
- peers take over client or server roles
- arbitrary, dynamic topology

In Practice

often combined with a client-server architecture



Peer-to-Peer Architecture in Windows 10

Delivery Optimization

Delivery Optimization provides you with Windows and Store app updates and other Microsoft products quickly and reliably.

Allow downloads from other PCs

If you have an unreliable Internet connection or are updating multiple devices, allowing downloads from other PCs can help speed up the process.

If you turn this on, your PC may send parts of previously downloaded Windows updates and apps to PCs on your local network or on the Internet. Your PC won't upload content to other PCs on the Internet when you're on a metered network.
Learn more

Allow downloads from other PCs

On

PCs on my local network

PCs on my local network, and PCs on the Internet

Advanced options

By default, we're dynamically optimizing the amount of bandwidth your device uses to both download and upload Windows and app updates, and other Microsoft products. But you can set a specific limit if you're worried about data usage.

Download settings

Limit how much bandwidth is used for downloading updates in the background

45%

Limit how much bandwidth is used for downloading updates in the foreground

90%

Upload settings

Limit how much bandwidth is used for uploading updates to other PCs on the Internet

50%

Monthly upload limit

500 GB

Note: when this limit is reached, your device will stop uploading to other PCs on the Internet.

Model-View-Controller Architecture

Model-View-Controller Architecture

- **Context**: data is presented and manipulated over several views
- **Problem**: data inconsistent and new views hard to add
- **Idea**: separation into three components
- model: stores the relevant data independent of their presentation
- view: shows (a part of) the data independent of manipulations
- controller: user interface for the manipulation of data [Sommerville]

Example

In a spreadsheet, data is presented in tables and diagrams. Changing values in a table leads to an update of affected diagrams and tables.

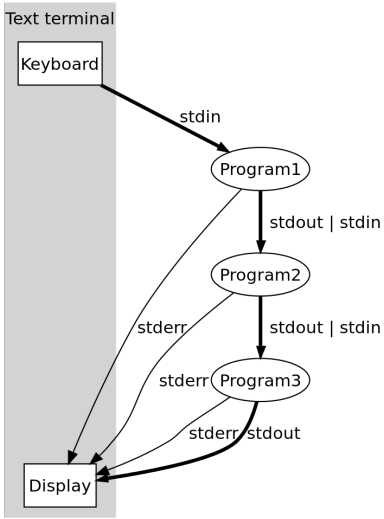
Pipe-and-Filter Architecture

Pipe-and-Filter Architecture **[Sommerville]**

- **Problem**: data is processed in numerous processing steps, which are prone to change
- **Idea**: modularization of each processing step into a component
- filter components process a stream of data continously
- pipes transfer data unchanged from filter output to filter input

Pipe Operator in UNIX

"`ls -al | grep '2020' | grep -v 'Nov' | more`" searches files in a folder from the year 2020 except those from November and delivers the results in pages.



Common Architectural Patterns

Lessons Learned

- What are architectural patterns?
- What is the difference between common architectures? layered architecture, client-server, 3-tier, peer-to-peer, model-view-controller, pipe-and-filter
- Further Reading: Sommerville, Chapter 6.3 (p. 175–184)

Practice

- Describe a further example for one of the discussed architectures (or a combination thereof) in Moodle: `https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=1999`
- Vote for at least one other example.

Lecture Contents