Architecture in general
▶ general term to describe buildings and other physical/ non-physical structures▶ art and science of designing buildings/nonbuilding structures▶ style of design/method of construction of buildings and other physical/non-physical structures
1. durability: last for long time without falling apart 2. utility: artefact should be useful 3. beauty: artefact should look good ("slender design")

SOFTWARE ARCHITECTURE
- Design Process with selection and composition activities ▶ process rather different to simply writing and testing code - Systems is a composition of interacting and related components ▶ Large systems are complex and can't be built from one piece ▶ Principle to manage complexity: "divide et impera" - Components can be made up to other components ▶ Architecture describes what components are used and how they are organized and the behavior of components (in collab) ISO/IEC 42010 ▶Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution Cesare Pautasso 2021 ▶A software system's architecture is the set of principal design decisions made about the system. Martin Fowler 2015 ▶Software architecture is those decisions which are both important and hard to change. ▶4 Examples SW Architecture Styles: 1. Layered Architecture, 2. Client-Server Architecture, 3. Microservice Architecture, 4. Event-Driven Architecture
Why SW Architecture?: ▶Manage complexity through abstraction ▶Communicate, remember, and share global design decisions among the team ▶Visualize and represent relevant aspects of a SW system ▶Understand, predict, and control how the design impacts quality attributes of the system ▶Define a flexible foundation for the maintenance and future evolution of the system

Capturing the Architecture
Every system has an architecture. Some architectures are manifest and visible, many others are not. A systems descriptive architecture ultimately resides in its executable code. Before a system is built, its prescriptive architecture should be made explicit. A system architecture may be visualized and represented using models that are somehow related to the code (existing or yet to be written).

Twin peaks model
the modeling process is non-linear. Analysis and synthesis cannot wait for each other. Architectural relevant requirements are discovered from the domain while making progress with the design. ▶emphasizes two main objectives: ▶identifying the right requirements ▶identifying the correct interpretation of the requirements The model stresses continuous collaboration and communication between stakeholders.

Software Development Process 1. Collect requirements 2. Create high level solution concept ▶ Iterate to 1 until stakeholder and producer have gained a common understanding of the solution 3. Refine solution concept creating software architecture 4. Detail software architecture into software design 5. Create unit tests and program software 6. Integrate components and run integration tests

Software Architecture in Development
▶ Software architecture is result of the software design activity ▶Software architecture connects requirements with a solution ▶ Anforderungen ▶ SW-Architekt ▶ Architektur ▶ Programmierer ▶ Software ▶Solution Concept: High level software architecture
▶Software architect: is a role, not a person

Software Architecture Quality Attributes
▶ are non functional Requirements BSP: The online store must be available to user requests at least 99.9% of the time (availability)1.Functional suitability: completeness, correctness 2. Performance efficiency: time behavior, resource utilization 3.Compatibility: Co-existence, interoperability 4.Usability: appropriateness, accessibility, learnability 5.Reliability: availability, stability, fault tolerance 6.Security: integrity, authenticity, confidentiality 7. Maintainability: modularity, reusability, testability 8. Portability: adaptability, installability, replaceability

External and Internal Quality: ▶ external quality: perceived from system end user perspective ▶ internal quality: perceived from software dev perspective

▶ Quality attributes need to be: ▶ measurable, testable ▶ If we can't prove that our system satisfies a required quality attribute we don't know if the system performs well or poorly ▶ No single software architecture can provide all quality attributes ▶ Some quality attributes are in conflict with others, making it hard or even impossible to combine them in a single software architecture ▶ Software architects therefore have to make tradeoffs.

Frameworks
▶ contain reusable and individual parts of a software architecture for a specific domain or purpose ▶ templates for actual software architectures ▶ Examples: Hibernate, Spring, AngularJS(framework for JS applications)

Unterschiedliche Arten Pattern (Architectural/Design/Software Design):
Architectural Pattern ▶ generic architectural solutions to generic problems ▶ describe the decomposition of a system into sub-systems, based on a specific strategy ▶ Architectural pattern can contain design pattern ▶ Examples: Layer, Pipes and filters, Model-view-controller Design Pattern ▶ represent a composition of classes that collaborate to solve a specific problem ▶ Examples: State, Factory, Singleton Software Design Pattern ▶ Patterns are proven solutions for specific problems ▶ have been tried and tested many times ▶ patterns are being discovered not developed ▶ pattern works with abstractions: interfaced / abstract classes ▶ real classes are unknown to the pattern and are derived classes

Pattern versuchen folgende Architekturmerkmale zu verbessern:
▪Verständlichkeit ▪Erweiterbarkeit

Objektorientiere Pattern folgen folgenden Konstruktionsprinzipien:
▪Loosely coupled systems ▪Abstraktion ▪Information hiding ▪Clear responsibilities ▪Dependency inversion Prinzip

Using Patterns
▶ objective: making experience accessible ▶ check if the pattern fits to the problem to be solved ▶ fosters robustness of a design, relying on proven concepts
▼ pattern can impose a performance penalty ▼ increase complexity of an architecture ▶ since they often rely on delegation patterns improve: ▲comprehensibility ▲ extensibility OO patterns follow: loosely coupled systems, abstraction, information hiding, clear responsibilities, dependency inversion

Design Decisions
▶ The initial set of design decisions is empty ▶ Decisions are made for a reason ▶ Decisions depend on other decisions ▶ Decisions are made by more than one person ▶ Decisions are made over time ▶ Decisions are changed over time

System as components
▶ Large systems are complex and cannot be built from one piece ▶ One of the oldest principles to manage complexity is "divide et impera" ▶ We understand systems as a composition of interacting and related components ▶ Components themselves can be made up of other components ▶ Architecture describes what components are used and how they are organized ▶ Architecture describes the behaviour of components only as far as their collaboration is concerned

Models (UML... etc)
▶ to describe things in the real world ▶ models are abstractions of the real world, simplifying reality Architectural model: artifact that captures some/all design choices Architectural modeling: reification ▶ documentation of design ch.

Architectural models
1. static: structure, decomposition, interfaces, components, connectors, dependencies 2. dynamic: behavior, deployment 3. styles and pattern 4. design process: constraints, rationale, quality assurance, team org. 5. target audience: technical devs, marketing/ customers, managem.

UML Diagramme Kategorisierung:
1. Strukturdiagramme ▪Klassendiagramme ▪Component diagram ▪Object diagram ▪ Package diagram
2. Verhaltensdiagramme ▪Activity diagram Verwendet in Echtzeitsystemen ▪State diagram Verwendet in Echtzeitsystemen ▪Sequence diagram ▪Use Case diagram

Descriptive vs Prescriptive Architecture
Prescriptive: explicit architecture before system is built, serving as a blueprint, defines how system should be build Descriptive: architecture resides in its executable code, representing how the system is actually built (exists in every system (brownfield and later in greenfield) green field development: system just has prescriptive architecture ▶ because system is built from scratch brown field development: system has both descriptive and prescriptive architecture. prescriptive architecture may evolve over time or require updates to reflect new changes.

Domain and design models
domain model: collects decisions which are outside of your control ▶ system will be evaluated against it ▶ architecturally significant requirements ▶ use case scenarios, feature models ▶ describes the reality that exists independently of your system.
Use Case Scenarios
The model of the architecture can be broken down into scenarios, each illustrated using the other views. Scenarios help to ensure that the architectural model is complete with respect to requirements. Scenarios can be prioritized to help driving the development of the system according to different stakeholders expectations.
design model: collects your main design decisions and is fully under your responsibility ▶ system will be built based upon it and is represented by it: ▶prescriptive specifying how the system should be implemented. This includes interfaces, quality attributes, and the structural decomposition that dictate the architecture and design rules ▶ also descriptive represents the system as it is designed, documenting interfaces, quality attributes, structural decomposition, and the actual implementation details to reflect the system's architecture.

Solution vs Product
Solution architect: solves problems of 1 customer by designing solution architecture made of multiple products Product architect: designed one product architecture that can serve multiple customers ▶Programming system has to consider interfaces and system integration▶Programming product has to consider generalization, testing, doc▶Programming system product: REST API, operating system

Views
System context: user (roles, personas), dependencies to other sys. Containers: main logical execution environments in which sys. runs Components: structural decomposition of SW and interf. / dependen. Connectors: How are components interconnected, what connector Classes: structure of code inside each component (UML class diag.) ⇒ From overview to detailed view
What is a view? ▶We cannot capture all aspects of an architecture in a single model. We decompose a model according to different viewpoints or perspectives. Example for building analogy: electrical system, plumbing structure. Views are not always orthogonal and should not become inconsistent.

UML Diagrams
class diagrams are well suited to describe structural aspects of sw-architecture and helpful in documenting design patterns ▶to describe structure and behavior
Structural: class diagram, component diagram, package, deploym., object
Behavioral: activity diagram, sequence diagram, state machine

Direct Association The fan knows his club, but the club doesn't know all its fans

Solid Aggregation/Composition the part doesn't exist on their own

Associations ▶dependencies on interfaces are preferable to depend. on classes ▶Inheritance and generalization have stronger dependency

Class Diagram
Attributes: describe the appearance and knowledge of a class of objects.
Operations: define the behaviour that a class of objects can manifest. Stereotypes – help you understand this type of object in the context of other classes of objects with similar roles within the system's design. Properties – provide a way to track the maintenance and status of the class definition. Association – is just a formal term for a type of relationship that this type of object may participate in. Associations may come in many variations, including simple, aggregate and composite, qualified, and reflexive. Inheritance: allows you to organize the class definitions to simplify and facilitate their implementation

Architectural Styles
style in software system is determined by: ▶ set of component types that perform some function ▶ topological layout of these components indicating their relationship ▶ set of connectors that implement communication/coordination among components ▶ set of semantic constraints, like state Examples:
1. Data centered style: Mehrere Clients greifen auf eine Datenbank zu
2.Batch sequential style: Einzelne Prozessschritte verarbeiten Daten und übergeben diese dem nächsten Prozessschritt, der mit den vom vorherigen Prozessschritt bearbeiteten Daten weiterarbeitet.3.Pipe and filter style: Auf den Daten wird sequenziell eine Kette von Operationen ausgeführt. Dies geschieht in einer Pipe, die Schritt für Schritt die Operationen ausführt.4.Call and return style: Funktionen rufen andere Funktionen auf und arbeiten mit dessen Rückgabewerten.5.Layered style: Die Software wird in Schichten zerlegt: Presentation Layer, Application Logic Layer, Persistence Layer. Die unterschiedlichen Schichten greifen auf die darüber oder darunterliegenden Schichten zu. 7.independent component style: Klassen tauschen untereinander Informationen aus. Z.B. wie das Observer Pattern 8.Event style: Abhängig von den Zuständen und den Events werden bestimmte Funktionen ausgeführt die bei einem Event getriggert werden.

Build Management
▶ Software products are usually build from source code and libraries ▶ tools like compilers, linkers etc. are needed ▶ number of commands are executed to build software product ▶ commands are part of the tool chain you use to create sw product ▶ real sw product requires repeatable, documented build process ▶ Creating software products from within IDEs is not reproducible BSP: Maven, Gradle, Ant, webpack, make

Gradle Build System
▶ advanced general purpose build management sys. based on Groovy ▶ supports the automatic download and configuration of dependencies or other libraries ▶ Gradle has the notion of projects and tasks in build.gradle ▶ Gradle build consists of one or more projects ▶ Each project consists of tasks ▶ build system uses plug-ins (extensions) to extend core functionality ▶ apply plugin: 'pluginname'

Gradle Dependencies
▶ Gradle allows managing the classpath of your projects ▶ can add JAR files, directories or other projects to the build path ▶ supports automatic download of Java library dependencies ▶ Java library is identified via its project's groupId:artifactId:version

Configuration Management
configuration: collection of artefacts in a specific version configuration management: encompasses all processes dealing with the management of configurations ▶ foundation on which any SW development process is built ▶ can't develop software without proper configuration management

Repository
▶ repository contains the history, all versions, branches and tags ▶ bare repositories don't contain a working tree and end with .git ▶ typically used on a server (Github) ▶ repository locations are defined by URLs ▶ distinguished between fetchurl and pushurl

Staging Area ▶ place to store changes from the working tree before the commit ▶ staging area contains a snapshot of the changes ▶ working tree contains the set of working files for the repository

Commits ▶ Each commit operation creates a new commit obj. in the repository ▶ commit object uniquely identifies a revision of the repository via an SHA-1 hash and knows its parents ▶ The commit object contains the author and the committer

Branches ▶ branch is a named pointer to a commit ▶ you select a branch via git checkout command

HEAD
▶symbolic reference pointing to the currently checked out branch ▶ sometimes it points to a commit object: detached HEAD mode ▶ If you switch branches, the HEAD pointer points to the branch pointer, which in turn points to a commit ▶ If you checkout a specific commit, HEAD points to commit directly

Tag
▶ points to a commit, which uniquely identifies a version of Git repo ▶ tag is a named point to which you can revert to any time ▶ to mark the repository for a specific reason, e.g. with release ▶tags can have a timestamp and a message associated with them

Components
can be made up of other components ▶ system as a composition of interacting and related components ▶ Application-specific BSP: [MP3 Codec, Payment Service, Song Classifier, Customer Data]: ▶ directly involved in implementing the functionality ▶ requires specific domain knowledge to build correctly ▶ may be difficult to reuse across other application domains ▶ reusable, buy may provide much more than what is necessary ▶ Infrastructure BSP:[Media Player, Math Library, GUI Toolkit, Web Server, Database]: ▶ independent of a specific application domain, addresses the needs of multiple classes of applications ▶ highly reusable, may require some configuration to satisfy req. ▶ deliver support for extra-functional requirements

Black Box
1. Encapsulation: Clearly separate content from rest of system 2. Abstraction: Hide implementation details behind interface 3. Modularity: Reusable unit of assembly / deployment 4. Composability: Components are meant to be interconnected and be made out of other components

Component lifecycle decisions
1. Development part: Plan, Code, Build, Test, Release 2. Operational part: Deploy, Operate, Monitor If externally sourced: Discover, Select, Integrate If stateful: migrate, backup, recover (in Operation)

Component vs Object
Abstraction: Components: Encapsulation (hiding of data and behavior), modularity (independent modules with clear interfaces), reusability (reuse of entire modules). Objects: granularity (finer subdivision), interface (defined methods for interaction with other objects). Architecture: Components: State and functionality (contain both), coarse-grained (larger than individual objects), unit of composition and deployment (can be deployed as independent building blocks), well-defined and documented (clearly specified), explicit dependencies (clearly defined dependencies). Objects: State and functionality (also contain both), fine-grained (smaller and more specific than components), identifiable unit of instantiation (clearly identifiable as an instance at runtime). Code: Components: larger units that stand alone and are explicitly defined. Objects :optional (do not always need to exist), entangled with other (strongly interconnected with other objects).

Open-closed principle
▶ Modules shall be open, that they are extensible ▶ Modules shall be closed, that their source code shall not be altered ▶ inheritance (static relationship, no new things during runtime)

Servlet
Java class that is called by a servlet container (Tomcat) ▶ to process requests from clients and generate responses dynamically ▶ which servlet is called is determined by the URL requested from the servlet container ▶Runs within a servlet container ▶Processes HTTP requests from clients ▶Generates dynamic responses ▶Is called by the servlet container based on URL mappings defined in web.xml or annotations ▶Follows a specific lifecycle managed by the container

Web Application Container Structure
▶directory and file structure is standardized ▶ all applications reside under servlet containers "webapp" folder ▶ each web application resides in its own sub folder ▶ in the application folder there is a WEB-INF folder containing the applications classes and libraries ▶ the web.xml file defines the applications URL mappings, servlets, ...

Java Server Pages JSP Lifecycle
▶ make it easier to assemble HTML pages with dynamic content ▶ with Servlets you embed HTML in Java ▶ with JSPs you embed Java in HTML first time use: servlet container creates Java servlet from JSP, generated code is then compiled by servlet container to Java class ▶ afterwards the servlet methods are called with each request ▶ preferred over direct HTML in servlets because: they separate presentation from logic, allow embedding Java code within HTML rather than embedding HTML in Java, more maintainable for designers who work primarily with HTML, easier to modify the presentation layer without touching Java code, automatically compile into servlets by the container, more natural for creating dynamic web content

Building Three Tier Web Architecture
▶client-server architecture that separates and application into three logical and physical computing tiers Top (presentation) tier: client running in browser Middle (application) tier: application logic, ORM Bottom (data) tier: database system 1. Create data model ▶ UML diagram, create UI mockups ▶ develop entity model in Java/Spring JPA, create DB 2. Repository Layer ▶ for each entity create a java class with CRUD oper. 3. Service Layer ▶ contains application logic and handles requests pertaining >1 entities ▶ contains method that transforms model classes into data transfer objects DTO 4. Creating DTO ▶ represent serializable objects that are transferred between service layer and user interface or REST API 5. Controller Rest API ▶ controller layer classes connect browser client to server app 6. Security ▶ to prevent unauthorized users from accessing anything but login ▶ filters permit to investigate requests before they reach any servlet 7. Creating the client ▶ client runs in browser and communicates with server using HTTP req.

Relational Database Management System
designed to store large volumes of data consistently and structured ▶ BSP most powerful and popular systems: Oracle, IBM DB2, MySQL ▶ RDBMS are based on relational model as introduced by E.F.Codd Tables: The data in an RDBMS is stored in ""tables"". A table is a collection of related data entries. A table consists of columns and rows Fields: Every table is broken up into ""fields"". The fields in the example table of the previous slide are ID, LASTNAME, FIRSTNAME, BIRTHDATE and ACTIVE. A field is a column in a table that is designed to maintain specific information about every record in the table. The field has a data type. All entries in the column of this field must have the fields data type Columns: A column a vertical entity in a table. It contains all information associated with a specific field in a table Rows: A row is a horizontal table entity. A row contains all data pertaining to a single entry of the entity described by the table. NULL Values: A NULL value in a table designates that this field has no value. A NULL is different from a zero integer or a Null string (""). NULL is a valid value and can be searched for Constraints: Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints can either be column level or table level. Examples: NOT NULL – Ensures that a column cannot have a NULL value; DEFAULT – Provides a default value for a column when none is specified; UNIQUE – Ensures that all the values in a column are different; PRIMARY Key – Uniquely identifies each row/record in a database table; FOREIGN Key – Uniquely identifies a row/record in any another database table.

Examples for constraints
rules enforced on data columns on a table ▶ limits type of data, to ensure accuracy and reliability ▶ on column or table level **BSP:** Not Null, Default, Unique, Primary key, Foreign key

Data Integrity
Entity Integrity: there are no duplicate rows in a table Domain Integrity: enforces valid entries for a given column by restricting the type, the format, or the range of values Referential Integrity: rows cannot be deleted as long as they are used by other records User-Defined Integrity: enforces some specific business rules

Database Normalization
process of efficiently organizing data in database ▲ helps eliminate redundant data 1st normal form 1NF: ensure there are no repeating groups of data ▶ also primary key, attributes, related data in one table 2nd normal form 2NF: ensure there are no partial dependencies of any of the columns on the primary key 3rd normal form 3NF: ensure all non-primary fields are dependent on the primary key

Why ORM (Object Relational Mapping)?
OOP works with classes and objects, RDBMS with entities and tables ▶ mismatch is resolved using an Object Relational Mapper (ORM)▶ Java Persistence API JPA specification describes the management of relational data in java programs DataTypes A property data type can be: any Java primitive (int, long, float, double, etc.); any Java primitive wrapper (Integer, Long, Float, Double, etc.); Strings; Dates; Big integer and decimals; Byte arrays (for BLOBs); Character arrays (for CLOBs); Enum values Entity Keys (Primary key specification) Every entity must have a primary key. Key is specified with the @Id annotation. Only one key defined for a given entity class hierarchy. Keys can be simple or composite. Keys need to be generated. JPA supports four generator types

Entity Classes
entity class: represents single table of connected database ▶ class must not be final, nor methods or persistence instance vars ▶ may be serializable and be passed across remote calls (RMI) ▶entities may exist in a class hierarchy EntityManager: central interface through which JPA provides operations for managing entities in the persistence context BSP: create, retrieve, update, delete, persist ▶ flush: saves EM context state to database ▶ detach(entity): detach entity from EM contex ▶ refresh(entity): refresh entity state from database ▶ getReference(id): retrieve lazy entity from database
Primary Key Generation GenerationType
▶.AUTO: JPA chooses generator dependent on database type ▶ .IDENTITY: performance issues ▶.SEQUENCE: requires data sequence support by DBS or simulation ▶ .TABLE: always works, requires extra table to manage keys

Inheritance (JPA)
flexible inheritance: entity may be abstract ▶ entity can extend non-entity and vice versa inheritance mapping strategies: ▶ single_table: one table for superclass and its subclasses ▶all entities in one Table ▲ if performance more important than storage space▶ joined: subclasses have FK to superclass ▲ ideal when data normalization and efficient storage are more important than query performance. ▶ table_per_class: one singular table per class

Associations
Entities can be associated to other entities and embeddable objects ▶ OneToOne, OneToMany, ManyToOne, ManyToMany ▶ Uni- / bi-directional relationships ▶ Lazy and Eager fetching techniques

Why Junit tests?
▶ test software before publishing, write tests before writing code ▶100% block coverage: each line of code is executed at least once ▶regression testing: making sure quality doesn't get worse over time
Properties
1.Distribution: a service is available in a network 2.Component appearance: can be called independently 3.Stateless: service starts in the same state with each call 4.Loosely coupled: services do not depend on each other 5. Exchangeability: standardized interfaces promote this property 6. Location transparency: the actual hardware is immaterial 7. Platform independence: operating sys. and hardware are immaterial 8. Access via interface: knowledge of interface suffices to use service
Types of Services
▶ business process is characterized by a number of use cases ▶ use case is supported by a composite service ▶ composite services are comprised of basic services and may use other
Pros and Cons
▲ Mapping of business processes to services gives a good overview of all services required and interfaces used ▲ Complexity of distributed systems is reduced due to partitioning ▲ Services and basic services can be reused ▲ Services can be exchanged as long as interface remains compatible▼If services are made too small complexity increases ▼ overhead due to required communication through several layers ▼commonly used protocols require considerable network bandwidth ▼ SOA is only effective if associated busin. processes are well defined

What is REST? ▶ architectural style based on web-standards and the HTTP protocol ▶ everything is a resource, resource is accessed via a common interface based on HTTP standard methods ▶ REST server: provides access to the resources ▶ REST client: accesses and modifies the REST resources ▶Every resource should support the HTTP common operations ▶ Resources are identified by global IDs (typically URIs) ▶ REST allows that resources have different representations: XML, JSON.... ▶ REST client can ask for a specific representation via HTTP protocol

HTTP Methods
Die PUT, GET, POST und DELETE Methoden werden normalerweise in REST basierten Architekturen verwendet. GET: reading access of the resource without side-effects POST: creates a new resource DELETE: removes the resource PUT: updates an existing resource or creates a new resource

RESTful web services
▶ RESTful Webservices basieren auf den http Methoden und dem REST Konzept. ▶ Ein RESTful Webservice definiert eine Standard URI für Services, die unterstützten MIME-typen (XML,JSON) and the set of operations (POST, GET, PUT, DELETE) which are supported

INTERFACES
provided interface: specify and document how to use externally visible features offered by component 1. Data types: define component data model 2. Operations: call functionality, modify state 3. Properties: read/write visible state attributes 4. Events: call-backs, notify about state changes
component reusability: component can only be reused if dependencies are satisfied component communication: components always communicate through interfaces ▶ no out-of-band communication or shared state ▶ interfaces should behave consistently with clients expectations ▶ obvious, consistent, easy to guess, natural, predicable
Principles
▶ few interfaces principle: every comp. communicates with as few others as possible ▶clear interfaces principle: don't publish interfaces with useless or unused functionality
Encapsulation
reduce the cognitive burden for using a component. Well designed interfaces are simple to understand. Every interface should hide one secret (avoid leakage: do not reveal unnecessary details). Changes to the implementation hidden behind an interface do not affect clients. Difficult to hide the impact of all changes.

Explicit Interface Principle:
Das Explicit Interface Principle (EIP) besagt, dass eine Klasse oder ein Modul seine öffentlichen Schnittstellen klar und eindeutig definieren sollte. Dadurch wird sichergestellt, dass nur explizit freigegebene Methoden von außen aufrufbar sind, was die Kapselung verbessert und unbeabsichtigte Abhängigkeiten vermeidet. BSP: In C# oder Java kann dies durch explizite Schnittstellendefinitionen (Interfaces) erreicht werden, sodass nur die vorgesehenen Methoden für andere Komponenten sichtbar sind.

Interface Description Languages (IDLs):
plattformunabhängige Definition von Schnittstellen, um die Interoperabilität zwischen verschiedenen Systemen sicherzustellen. BSP two real world interface description languages: Protocol Buffers (Google entwickelte IDL zur effizienten Serialisierung strukturierter Daten für die Kommunikation zwischen Diensten.), Web Services Description Language (WSDL) (XML-basierte IDL zur Beschreibung von Webservices und deren Schnittstellen.)

API
▶objective: make it easy for as many clients to interact with system ▶ should be described from perspective of application devs using it ▶What is an API? Application Programming Interface: is a software interface that allows two applications to interact with each other without any user intervention. APIs are standard contracts that define how developers communicate with a service, and the kind of output those developers should except to receive back ▶Where to find APIs? Operating systems, programming language runtimes, hardware access, user interfaces, databases, web and cloud services ▶API Design the objective of a good API design is to make it easy for as many clients as possible to interact with the system. BSP Commonly known APIs: RESTful API, Google Maps API, Twitter API, Firebase API, PayPal API ▶Usability vs. reusability A general interface helps to reuse a component in many architectures. A general interface is more complex and difficult to use in a single architecture ▶Performance vs. reusability A general, reusable component may be less optimized and provide worse performance than a specific, non-reusable one. For the same (reusable) interface, there can be multiple implementation components optimized for different hardware platforms/execution environments ▶Uniform access principles Facilities managed by a component are accessible to its clients in the same way whether implemented by computation or by storage. Helps to introduce caching or memorization without affecting clients.

Design Advice
▶keep it simple, keep as small as possible but not smaller ▶ maximize information hiding ▶ names matter, internally consistent, externally consistent ▶ document everything ▶ make it easy to learn and easy to use, make it hard to misuse

JAX-RS
Java defines REST support via the Java Specification Request JSR JAX-RS: Java API for RESTful Web Services ▶ JAX-RS uses annotations to define REST relevance of Java classes

Jersey
▶Jersey is the reference implementation for the JRS 311 specification. ▶It provides a library for implementing RESTful web services in Java Servlet containers. ▶On the server side Jersey provides a servlet implementation which scans predefined classes to identify RESTful resources. In your web.xml configuration file or via annotations you register this servlet for your web application. The Jersey implementation also provides a client library to communicate with a RESTful webservice. ▶Jersey includes a client library for communicating with RESTful web services. ▶The base URL format is: http://your_domain:port/display-name/urlpattern/path:from_rest_class. ▶It processes HTTP requests by selecting the appropriate class and method based on annotations. ▶REST applications consist of resource (data) classes and services, typically managed in separate packages. ▶JAX-RS supports XML and JSON creation via Java Architecture for XML Binding (JAXB).

Complexity
▶determined by: number of items, item types, item relationships ▶ additive complexity vs multiplicative complexity ▶ to reduce complexity: reduce dependencies as much as possible

ReactClient
React props and state
The props variable is a dictionary containing all properties passed to a component. Props are read-only and are always passed from top to bottom. You can pass functions from higher level components for callback purposes. State represents the components state. State must exclusively be changed by the setState() function, never ever directly. Changing any state variable results in a call of the render() function. The render() function renders the user interface and comprises of HTML like elements, own components, and JavaScript code. Arrow functions and this: The handling of this is different in arrow functions compared to regular functions. In regular functions the this keyword represents the object that calls the function, which could be the window, the document, a button or whatever. With arrow functions the this keyword always represents the object that defined the arrow function.

Architecture vs. Code
▶ Every system has an architecture ▶ Some architectures are manifest and visible, many others are not ▶ A system's architecture ultimately resides in its executable code ▶ A system's architecture may be visualized and represented using models somehow related to the code

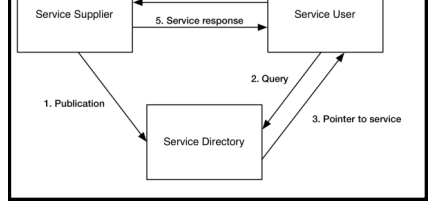
Architecture vs. Technology
▶ A technology only serves to implement some dimension of an architecture ▶ A network can be part of an architecture ▶ A database can be part of an architecture ▶ Some framework X may be part of the architecture ▶ Architecture is more than just a list of products used in a project ▶ Technology can shape an architecture, but an architecture should not just be bound to the technologies that form it ▶ Keep in mind: Technology changes rather fast

System Boundary and Interfaces
▶ The system boundary is one of the first things that need to be established ▶ A closed system is isolated from the rest of the world and does not exchange any information with the outside ▶ An open system is interdependent and interconnected to other systems outside of your control ▶ Today, most systems must be open systems to be useful ▶ Carefully crafted, stable interfaces are required at the system boundaries ▶ Interfaces make components to useful building blocks for constructing something larger, something whose value is greater than the sum of its parts.

Wann zieht man eine abstrakte Basisklasse einem Interface vor? Wenn man eine Aggregation oder eine Default-Implementierung bereitstellen muss.

Was ist der Unterschied zwischen einem aktiven und einem passiven Filter? Ein aktiver Filter stellt einen eigenständigen parallelen Prozess dar. Ein aktiver Filter ist der typische Fall. Er wird von einem übergeordneten Programm aufgerufen und läuft immer als parallele Einheit. Es gibt auch passive Filter, die von einem dem passiven Filter benachbarten Filterelement aufgerufen und damit aktiviert werden.

Service Oriented Architecture Component Diagram



DESIGN PRINCIPLES:

Design Principles are well-established guidelines in software development that help make code more understandable, maintainable, and flexible by promoting concepts such as encapsulation, abstraction, separation of concerns, and loose coupling.
1. Encapsulation: Methods and data are contained within an object
▶ type of object is called a class ▶ data and the method logic are not visible from the outside
2. Abstraction: object exposes behavior solely via interface methods ▶ method declaration defines interface towards the outside ▶ method implementation shall be of no concern to method clients
3. Information Hiding(IH): Similar to Abstraction, implementation of objects behavior is hidden ▶ only the interface methods are exposed to the outside ▶ internal data cannot be accessed, except via interface methods **4. Separation of Concerns (SoC)**: focusing attention on one specific aspect ▶ helps order thoughts and structure the system into a variety of parts with a clear responsibility
Why: If a code unit lacks a clear task, it becomes hard to understand, use, fix, or extend. If you describe a class with "and," it likely does too much and violates good design principles.
5. Interface Segregation Principle (ISP): split up large interfaces into smaller, so it only meets the requirements of a particular client ▶ clients shall not depend on interfaces they don't need -Same idea as sin-gle responsibility for classes - Example: Multifunctional printer is splitted into Printer, Copier and Scanner) **6. Single Responsibility Principle (SRP):** dedicate each class to a specific task, every module or class should have responsibility over a single part of the functionality provided by software, task encapsulated by class ▶ don't create all-round classes that take care of unrelated problems
BSP: For date-related functionality: java.util.Date **7. Dependency Lookup:** object looks up its association with another object during runtime by looking for this object at pre-agreed places
▶ objects kept in a central class can be accessed by all objects ▶ objects kept in a registry, where anybody looking for it can ask ▶ objects kept in a container that is filled with objects during system initialization or runtime **8. Dependency Injection:** separates construction and use of objects, association between objects is created during runtime from own instance and not during compile time
▶ client who wants to call a service delegates the responsibility of providing that service to external code (injector) during runtime ▶ client isn't allowed to call injector code, injector constructs service ▶ injector injects the service into the client ▶ client does not need to know about the injector, or other things
Advantages of Dependency Injection:
▲ Improves testability – Dependencies can be easily mocked or swapped during unit testing.
▲ Enhances maintainability – Decouples object creation from business logic, making code more modular.
▲ Promotes reusability – Components can be reused in different contexts without changes.
▲ Supports scalability – Simplifies management of large applications by centralizing dependency
- Dependency Injection Disadvantages:
▼ creates clients that demand configuration details be supplied by construction code
▼ can make code difficult to trace because it separates behavior from construction
▼ forces complexity to move out of classes into linkages between classes, which might not be desirable / easily managed
3.1 Implementations of Dependency Injection:
Pros and Cons
▶ Constructor Injection: Dependencies are passed as parameters to the constructor.
▲ Advantage: Ensures that the object is always fully initialized.
▼ Disadvantage: Can lead to long constructor signatures with many dependencies.
▶ Setter Injection: Dependencies are added at runtime through setter methods.
▲ Advantage: Allows optional or late dependency assignments.
▼ Disadvantage: Risk of using the object before it is fully initialized.
▶ Interface Injection: Dependencies are added through interfaces.
▲ Advantage: Enforces a clear separation between dependency and consumer.
▼ Disadvantage: Less common and often dependent on specific frameworks.
Open-closed Principle means: -Modules shall be open in a sense that they are extensible -Modules shall be closed in a sense that their source code shall not be altered, but allows new functionality to be added to an application -How can we extend something without changing it? By Inheritance -can help reduce the risk of introducing bugs or other issue into the codebase when new features are added
Design Principles to support loose coupling
when designing complex systems, first step is to break them down into simpler parts which are subsystems or component systems → identifying subsystems using divide and conquer is challenging quality measure for a good design the following aspects are considered: - coupling: measure of amount of interaction between subsystems ▶ **loose coupling:** few dependencies between different system parts - cohesion: measure of the cohesion of a subsystem ▶ **strong cohesion:** within each system part strong dependencies
Design by contract - interface specifications for software components are considered like "contracts"
▶ caller/buyer must abide by the contract, just like called class/seller ▶ contracts are specified in terms of associations - Assertions: can be one of multiple types: ▶ Preconditions: must be checked by caller prior to call, check for valid input when called ▶ Postconditions: must be checked by called prior to responding, conditions must be true after a method has completed, check for valid output ▶ Invariants: must be adhered to by all objects of a class, conditions must be true before and after a method is called, state of object is always valid - contract of a method comprises pre- ▲ postconditions of that meth. ▶ contract of a class comprises the contracts of its methods ▲ invariants of the class
- when overriding, contracts need to be kept: ▶ overridden meth. must not tighten, but may loosen preconditions ▶ overridden meth. must not loosen, buy may tighten postcondition
Classes must follow the following rules regarding the superclass in "Design by Contract":
▶ Same or stronger invariant ▶ Same or weaker preconditions (concerning methods) ▶ Same or stronger postconditions (concerning methods)
Are preconditions and postconditions related to a method or a class? The contract of a method includes its preconditions and postconditions. Who has the responsibility and benefit for a precondition?
The caller is responsible for fulfilling the precondition, and the callee benefits.
Who has the responsibility and benefit for a postcondition? The callee is responsible for the postcondition, and the caller benefits.
Do invariants apply to a method or a class? An invariant applies to a class and must be upheld by all objects of that class.
Can preconditions be relaxed in a subclass? A precondition can only be relaxed in a subclass.
Can postconditions be relaxed in a subclass? A postcondition can only be made stricter in a subclass.
Liskov substitution principle
he Liskov Substitution Principle states that subtypes must be replaceable for their base types.▶ If you want polymorphism of objects in a program, methods that use pointers or references to base classes must be able to work with objects of derived classes without noticing it - let Φ(x) be a property provable about objects x of type T ▶ then Φ(y) should be true for obj. y of type S where S is subtype of T
- subclass should extend the behavior of its superclass without changing its contract or expected behavior ▶ should adhere to the "is-a" relationship, ensuring that objects of subclass can be used interchangeably with objects of the superclass

ARCHITEKTURAL PATTERNS

Layers Pattern
problem before: partition complex systems to minimize dependencies between their parts
▶ structure the system horizontally so that a higher layer acts as a client that is being served by a lower layer server ▶ lower level servers may act as client to their lower layers ▶ each tier represent some abstraction - solution: components are arranged in layers / tiers ▶ each tier serves dedicated purpose, e.g. communication, ... ▶ Client-Server model: lower layers offer services to higher layers - Examples: Computer, Client Server (3) Tier Model, Communication
Layer Pattern Call Order: - Higher layers can call services of a lower layer ▶ Lower layers must not call services of a higher layer - layer bridging: higher layers can call services several layers lower
- strict layering: higher layer just call services from layer direct. below

Pipes and Filters Pattern
- problem: data stream oriented systems shall be partitioned into a number of steps ▶ system shall be easily extendible and support parallel processing - solution: sys. task is being decomposed into separ. processing steps ▶ each step is being implemented as a filter ▶ each filter is connected to the following via a pipe ▶ filters read data, transform them and output the transformed data ▶ Can remove data from their input data stream ▶ Can add data to their input data stream ▶ Can modify data read from their input data stream - Examples: Compiler - Filters: can remove, add, modify data from their input data stream - Pipe: transfers and buffers data, decouples the entities it connects ▶ writing into and reading from a pipe can be asynchronous - Pipeline: arrangement of filters connected with pipes ▶ pipeline connects data source with data sink
Pros and Cons▲ easy to add new filters or to remove them ▲ easy to change the order of filters in a pipeline ▲ filters can be developed independently ▲ rapid prototyping ▲ only adjacent filters share data, all others are decoupled ▲ storing intermediate information is unnecessary ▲ Filters can run in parallel ▲ be reused easily ▼ difficult to handle errors since there is no single system state
▼ slowest filter in the pipeline determines the throughput ▼ data format transfor. may be required if filter formats don't match
Plug-In Pattern - problem: shall be possible to add new functionality to existing software without having to modify it ▶ new functionality shall be addable by third parties ▶ system shall work with or without add-ons - solution: interfaces defined as extension points ▶ plug-ins implement these interfaces and are registered with core sys. ▶ during runtime plug-ins replace the interface with real implem. ▲highly modular▲ flexibility, easy to extend ▲ scalability with new plugins ▲ easy to maintain
▼ high complexity, management between core and plugins ▼ low performance because of overhead
Monolithic architecture is a traditional software design where all components of an application (UI, business logic, database access, etc.) are tightly integrated into a single codebase. The entire system is deployed as a single unit, making it simpler but less flexible when scaling or updating features.
▼ low modularity low flexibility, ▼ difficult to extend ▲ less complex, ▲ easier to develop initially because of simpler structure ▲ more efficient as all components are tightly integrated
Broker Pattern - problem: large systems must be scalable ▶ system components must be distributable to several computers, so components must thus be loosely coupled ▶ wouldn't be feasible to have every component know all the others - solution: components are classified by their role in inter-component communication ▶ server components provide one or more services ▶ client components consume one or more server comp. services ▶ roles can change during operations, i.e. server can become client - broker pattern puts intermediate layer between client and server components in an distributed system ▶ server register their services with the broker and then wait for req. ▶ clients request a service from the broker ▶ broker transfers the request to one of the servers ▶ server responds to broker, who in turn transfers response to client - client does not need to know the physical location of the server ▶ immaterial which physical server instance it is served by ▶ broker is only one that has to have knowledge of phys. instances - Examples: internet with domain name server
Broker Pattern Proxies - client class calls a method of a server class ▶ works only if server and client classes reside in the same computer ▶ in distributed system, the method calls need to be serialized for transmission over a communication network (marshalling) - at receiving side the serialized method call needs to be converted back into standard method call (unmarshalling) ▶ for this purpose we introduce a client- and a server-side proxy - client calls client-side proxy method which interacts with the broker
▶ broker transfers the clients call to the server-side proxy, which interacts with the server
▶ Participants: Client, Client- & Server-side proxy, Server, Broker
Pros and Cons▲ broker pattern separates communication between client and server from functionality (separation of concerns) ▲ server implem. can be changed as long as interface remains stable
▲ client only needs to know local broker ▲ logical service it requests ▲ same is true for the server
▲ client and server are platform-independent, can be implemented in different programming languages
▲ enables very large sw systems distributed across many computers ▼ local broker constitutes central point of failure if anyth. goes wrong ▼ performance is penalized my marshalling and unmarshalling op.▼ Broker can become a performance bottleneck ▼ Brokers and proxies are tightly coupled
BSP:The internet with a Domain Name Server (DNS).
Model-View-Controller Pattern - problem: user interface is a component that often changes
▶ same information may have to be presented in different ways
▶ presentation shall be updated when the underlaying info changes
BSP: Spreadsheet with graphical and tabular representation of the data. - solution: each assigned to dedicated components ▶ data storage and processing (model) ▶ Presentation logic (view)
▶ User input processing (controller)The mentioned points are each assigned to individual components.
- MVC-Model: stores the data to be presented in a view ▶ In simple systems the model may contain business logic - model shall be independent of views and controllers ▶ passive: In case model data changes, controller may inform views ▶ active: In case model data changes, model informs the views ▶ In **push mode** the model actively sends the data to the view ▶ In **pull mode** the model informs the view that there is new data - MVC-View: serves to present model data to a user ▶ can be many views on the same model data ▶ all views are updated in case of a model data change
- view may furthermore present interactive elements like buttons ▶ interaction creates events that are handled by controller - MVC-Controller: controls the model and view state based on user input
▶ controller transforms events caused by user actions into method calls of the model
▶ controller controls the state of the view, e.g. to activate or deactivate control elements
Dependencies: The Controller receives data from the View and must understand its format. If the structure of the View's data changes, the Controller must be updated, making it dependent on the View. If the View can be modified by the Controller, then the View is dependent on the Controller. If the Controller updates data in the Model, it depends on the Model. The View must display the Model's data, making it dependent on the Model.
Service-Oriented-Architecture Pattern
Problem: Business processes need to be supported by IT ▶ as business processes change frequently, so does the IT system ▶ If business process architecture is tied too closely to IT architect. ▶ modifications in former may require architect. changes on latter ▶ Architectural changes in sw systems are expensive and take long time
Solution: encapsulate business processes / parts in components ▶ A service / application service implements a use case
Patterns - Structural Pattern: deal with composition and granularity of classes and objects
- Behavioral Pattern: treat responsibilities of and cooperation between classes or objects, object interaction - **Generational Pattern:** hide generation of objects within an operation / class, isolate a system from how its objects are created

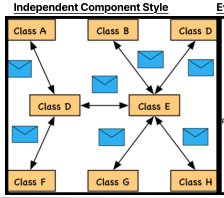
STRUCTURAL DESIGN PATTERN

Adapter Pattern
- problem: class shall be reused and offers the correct data, but the interface to access this data does not match the clients expectations ▶ existing classes can't be modified, like classes from a class library
▶ adapter pattern adapts "bad", existing interface of an existing class to a form requested by a client
- solution: adapter translates methods of an existing class to new methods as expected by the client
▶ outside it offers a new interface, in the existing class it uses the existing methods ▶ The translation is: This pattern is used when existing classes cannot be directly modified, such as classes from a class library.
- class adapter: implementation as class based adapter ▶ adapter inherits from existing class and extension interface ▶ new methods are collected from interface, client uses interface
▶ Without private inheritance, the adapter provides both methods: those from the existing class and the new ones. - object adapter: implementation as object based adapter ▶ adapter calls methods of existing class. ▶ leaner variant **Pros and Cons** ▲ enables communication between two independent sw components ▲ adapters can be extended with additional functionality like filtering ▲ adapters facilitate exchanging of classes. It just requires a modified adapter to use a different existing class ▲ object adapter can be applied to obj. of subclasses of adapted class ▼ object adapter introduces min. one more operation, slowing things ▼ adapters are particular to the classes to be adapted, hardly reusable
Bridge Pattern - problem: usually impl. is linked with its abstraction via inheritance ▶ tight coupling of base class to derived class is disadvantageous
▶ better to have more separation through bridges Solution: ▶ The Bridge pattern separates an abstraction from its implementation. This allows both parts to be modified and extended independently
Decorator Pattern - problem: possibility to add new functionality to obj, not entire class
▶ add properties like borders/behavior like scrolling to UI compon.
▶ Inheriting border from another class is inflexible, the choice of a border is static, fixed during compile time
▶ client couldn't control when and how to add border to a component - solution: enclose component in another object that adds border ▶ this object is a decorator ▶ decorator conforms to the components interface so that it's presence is transparent to the components clients ▶ decorator forwards requests to the component and may perform additional actions like drawing a border
Pros and Cons▲ component does not know its decorator ▲ dynamic extension: functional. can be added/ removed during runt. ▲ possible to extend several classes of an inheritance hierarchy, even classes that didn't exist when the decorator was written ▲ easy to combine any number of decorators: decorate recursively ▼ if class has a lot of methods and decorator extends a few of them, most decorator methods are just overhead ▼ difficult to locate errors in combined decorators ▼ decorator inherits from its component, but this part is never used. **How does the Bridge Pattern work?** The Bridge Pattern separates an implementation from its interface (abstraction) as much as possible. The class hierarchies of an abstraction and an implementation are connected through the so-called "bridge." This allows both parts to be modified and extended independently. **Where is the bridge visible?** The abstraction aggregates the interface of the implementation. This aggregation relationship acts as a bridge, linking the two class hierarchies on the abstraction and implementation sides.
Facade Pattern - problem: subsystem exposes complex interface to its clients
▶ only a few methods are really being used ▶ client needs to now many classes of subsystem to access methods of interest - solution: facade class collects all meth. of interest in single interface ▶ clients need to be aware of the facade class and its methods only ▶ facade hides all subsystem implementation details
▶ facade class delegates method calls to classes of the subsystem ▶ pattern leaves open if it is still possible to access subsystem methods directly **Pros and Cons** ▲ subsystems do not know their façade ▲ using the subsystem can become easier ▲ clients only weakly coupled to subsys, facilitates exchanging subsys
▲ facades often encapsulate legacy systems, even non-object oriented ones, can make legacy systems appear to be object-oriented ▼ facade adds an additional method call ▼ exposes only a part of the subsystems functionality ▼ changing the subsystem interface requires changing the facade ▼ If direct access is permitted, there is no info hiding by the facade ▼ The facade exposes only part of the functionality of the subsystems.
Composite Pattern - problem: compose objects into tree structures to represent part-whole hierarchies ▶ treat individual objects and composition of objects uniformly ▶ components can be grouped to form larger components ▶ primitives should be treated in the same way as large components
- solution: composite pattern provides an abstract class "node" that is both primitive and container
▶ leaves and composites inherit from "node" | Components: ▶ Nodes: defines the interface and behavior for the derived classes Composite and Leaf, it provides a default behavior for child operations ▶ Leaf: represents a terminal element in the tree structure that does not aggregate further nodes and can only be a child node itself ▶ Composition: represents a node in the tree structure that can aggregate other nodes
Pros and Cons ▲ simplified handling of tree structure since leaves and composites share the same abstract base class ▲ creating large nested structures is easy ▼ all nodes are treated equal, if they need to be treated differently there must be type checking during runtime ▼ changing abstract base class may require changing all derived classes as well
Proxy Pattern problem: placeholder object shall control access to real object
▶ placeholder may implement additional functionality like logging - solution: proxy class provides same interface to client as the original ▶ proxy delegates all method calls to the original class ▶ before delegation additional functionality can be added - types: ▶ virtual: postpone creation of real object until its needed
▶ protective: control access permissions ▶ Remote Proxy: Stellt lokalen ersatz für das remote reale Objekt zur Verfügung ▶ synchronizing: organize concurrent access to real object ▶ firewall, counting, cache
Pros and Cons ▲ existing applications can be extended with proxy pattern▲ proxy functionality can be added by demand ▲ client needs to know the original class interface only ▲ original class needs to be created only when the proxy cannot provide client request by itself ▼ locating errors become more difficult ▼ can be a performance penalty due to an additional method call

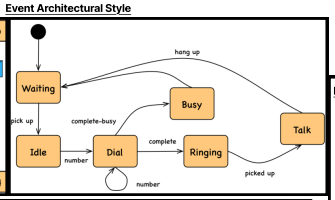
BEHAVIORAL DESIGN PATTERNS

Template Method Pattern
▶ Intent: define skeleton of an algorithm in an operation ▶ defer some steps to subclasses
▶ subclasses can redefine certain steps of an algorithm without changing the algorithms structure
▶ solution: invariant parts of an algorithm are implemented once ▶ varying behavior is implemented in subclasses
▶ permits to collect common behavior in a common class during refactoring to avoid code duplication
Pros and Cons
▲ Base class calls method of subclasses without becoming dependent
▲ algorithm is already implemented in base class with some details left open (abstract class)
▲ good code reuse; only variant parts are deferred to subclasses
Command Pattern
▶ Intent: object-oriented replacement for callbacks ▶ encapsulate a request as an object, supporting parameterization and undo functionality ▶ caller shouldn't have to know receiver of the request
Pros and Cons
▲ creation of command ▲ its execution are temporally decoupled ▲ caller doesn't have to know any command details (exchanging ez)▲ commands can be exchanged during runtime ▲ command objects can be reused, e.g. hfor different receiver objects
▼ packaging a command in an object creates overhead ▼ for each type of concrete command, there must be dedicated class
▼ adding additional parameters requires extra effort since usually the command interface is identical for all commands ▼ Implementation of this in distributed system can be quite complex
Goal: The Command Pattern is meant to hide the details of a command from the caller.
It should allow the creation time and execution time of the command to be separated.
Observer Pattern
▶ problem: dependent objects shall be automatically informed of an state changes in the object they depend on
▶ set of dependent obj shall be dynamic. changeable during runtime
▶ solution: create two classes, Observer and Observable ▶ Observer can register with the Observable
▶ Observer provides a method where it can be updated
▶ Pull variant: Observer pulls data from observable after it has been informed of a change
▶ Push variant: Observer gets data from observable with update msg
Pros and Cons
▲ loose coupling between observer and observable ▲ observable doesn't have to know observers at compile time ▲ new observers can be added any time
▼ updating many observers can be time consuming ▼ possibility of endless loops when updating an observer causes a state change in the observable ▼ each observer is always informed regardless of its state
3.7 Erklären Sie das Prinzip einer Callback-Schnittstelle anhand des Beobachter-Musters. Beim Beobachter-Muster werden Nachrichten in beide Richtungen zwischen Beobachter Objekt beobachteten Objektausgetauscht.Das beobachtbare Objekt verpflichtet den Aufrufer, eine spezielle Schnittstelle (Callback-Schnittstelle) zu implementieren, die vom beobachtbaren Objekt vorgegeben wird. Ändert sich der Beobachter, so hat dies keinen Einfluss auf dasbeobachtbareObjekt.DagegenwirkensichÄnderungenanderCallback-Schnittstelleoder den angebotenen Daten des beobachtbaren Objekts auf den Beobachter aus.
Strategy Pattern
Das Strategie-Muster soll es erlauben, dass ein ganzer Algorithmus ausgetauscht wird, um die Wiederverwendbarkeit zusteigern. ▶ problem: define a family of algorithms, encapsulate each one, and make them interchangeable, most suitable is selected at runtime ▶ strategy permits to vary algo. independent from clients that use it ▶ example: line breaking a stream of text, sorting algorithms in a collection, pathfinding in a navigation, payment methods in e-commerce system ▶ solution: strategy pattern encapsulates each alternative algorithm in a dedicated object
Pros and Cons
▲ more flexible than implementing each algorithm in a subclass ▲ avoids branching for different algorithms, less code ▲less complexity
▼ application needs to know when to apply each strategy ▼ more complex communication compared with implementation of strategy in context class
BSP: Sorting Algorithms(a program can support multiple sorting algorithms), Payment Processing System (shopping cart system allows users to select different payment methods dynamically), Travel Route Selection (navigation system allowsusers to select different travel strategies)
Visitor Pattern
▶ problem: be able to execute an operation across heterogeneous collection of objects of a class hierarchy
▶ be able to define new operations without having to change the class of any of the objects in the collection
▶ solution: pattern suggests defining the operation in a separate class referred to as a visitator class
▶ for every new operation to be defined, new visitor class is created
▶ since the operation is to be performed across a set of objects, the visitor needs a way of
vaccessing the public members
of these objects
When use Visitor Pattern?
▶ when object structure contains many classes of objects with differing interfaces, and you want to perform operations on these ▶ when many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations ▶ when classes defining the object structure rarely change, but you often want to define new operations over the structure
Java Persistence Architecture(JPA)
▶ describes the management of relational data in java programs
▶ Benefits: reduced development time (provides high-level API that simplifies the development of java applications that interact with a database), portability (standard API that can be used with any relational database), improved performance (jpa provides caching mechanisms that can improve the performance of db operations) ▶ **Examples for annotations used with JPA:** @Entity (used to mark a java class as an entity, lightweight persistence domain object that represents a table in a relational database), @Id (used to mark a field as primary key of an entity), @Column (used to map a field a column in a database table)

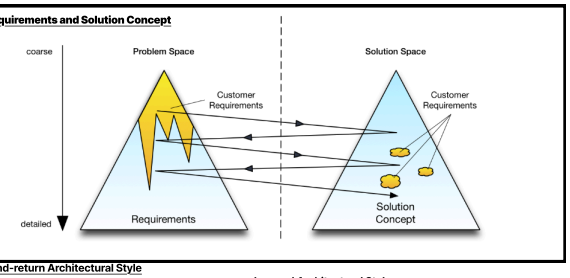
Independent Component Style



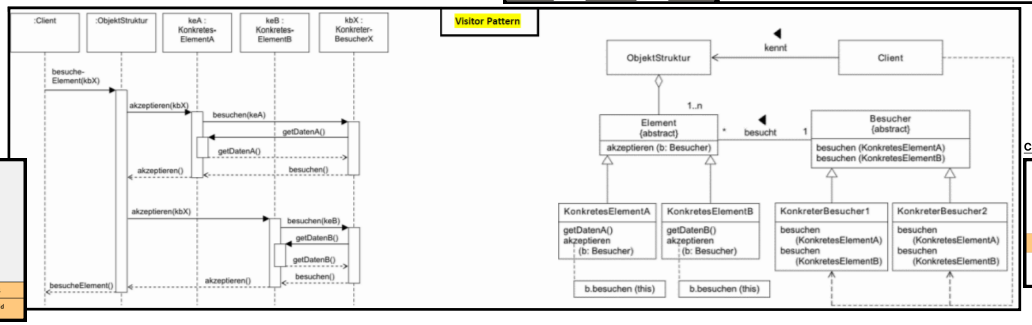
Event Architectural Style



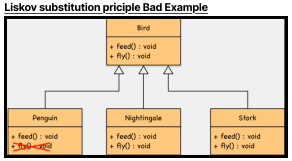
Requirements and Solution Concept



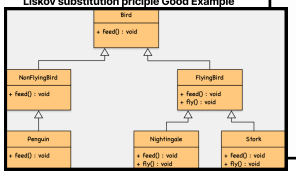
Visitor Pattern



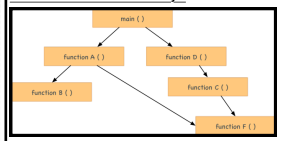
Liskov substitution priciple Bad Example



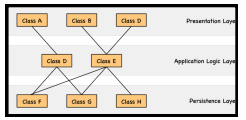
Liskov substitution priciple Good Example



Call-and-return Architectural Style



Layered Architectural Style



KLAUSURAUFGABEN

1.5 Explain briefly the Single Responsibility principle.Give a textual example.
→Spickzettel Seite 2 oben links, Unterpunkt Design Principles
1.8 In the Design by Contract principle contracts are specified in terms of assertions. Briefly explain the three different types of assertions.
→ Spickzettel Seite 2, 1. Spalte, mittig
2.3 Explain and discuss the Service Oriented Architecture pattern
→ Spickzettel Seite 1, letzte Spalte ganz unten
1. Which Idioms has the same class diagram as the Decorator pattern, except for a different multiplicity?
The Composite pattern
2.Are Separation of Concerns and the Single Responsibility Principle equivalent with regard to classes?
The Single Responsibility Principle (SRP) applies only to classes, whereas Separation of Concerns (SoC) is a general principle. SoC can be seen as a fundamental process of decomposition, while SRP is an object-oriented design principle specifically for classes.
With regard to classes, they are equivalent.
3.Which activities take place immediately before and after the development of software architecture?
Before development:
▶ Requirements analysis: A detailed examination of requirements is conducted to understand the functionalities and behavior of the system. ▶ System design: Based on the gathered requirements, a system design is created that describes the system's functional and non-functional architecture.
▶ Technology selection: Decisions are made regarding the technologies, frameworks, and platforms that will be used for software architecture development. ▶ Architecture planning: A detailed plan is created for how the software architecture will be developed. ▶ Requirements collection and creation of high-level solution concept (iterating between these until stakeholders and producers have a common understanding)
After development:
▶Component and module design: Based on the developed software architecture, individual software components and modules are designed. ▶ Implementation: The actual development of the software begins, with components and modules being implemented according to the defined design. ▶ Code reviews and quality assurance: Regular code reviews are conducted to ensure that the code meets standards and best practices. ▶ Documentation: Throughout the entire process, the software architecture and developed code are documented. ▶ Detailed software architecture refinement into software design, followed by unit test creation and programming
4.What is the difference between a solution concept and a software architecture?
A solution concept describes the higher-level strategic direction and approach and provides initial understanding between stakeholders and producers, whereas software architecture defines the technical structure and organization of software components. The solution concept considers a broader range of factors, while software architecture focuses specifically on software development, representing the fundamental organization of a system embodied in its components and their relationships
5.What is the difference between an architecture pattern and a framework?
An architecture pattern is a proven concept for designing a software architecture.
A framework is a structured collection of components and tools that supports application development.
▶ Architecture patterns define the fundamental principles and structures of an architecture.
▶ Frameworks provide a prebuilt infrastructure for building applications.
6.Which aspects must software architecture primarily consider?
Functionality, Scalability, Maintainability, Reliability, Performance, Security, Usability
7.What is the difference between inheritance and generalization?
▶ Inheritance is a concept in object-oriented programming (OOP) focused on code reuse and class relationships in software implementation, allows a class to inherit properties and behavior from another class ▶ Generalization is a modeling concept in UML, representing a hierarchical organization of classes and the abstraction/specialization of relationships,
represents an "is-a" relationship between a more general element (superclass) and a more specific element (subclass) Generalization is the conceptual relationship, while inheritance is its implementation in object-oriented programming.
8.Describe the different types of visibility in UML class modeling.
▶ Public (+): Attributes and methods with public visibility are accessible from anywhere.
▶ Private (-): Attributes and methods with private visibility are accessible only within the class itself.
▶ Protected (#): Attributes and methods with protected visibility are accessible within the class and its subclasses.
▶ Package (default visibility): Attributes and methods with package visibility are accessible within the same package/module, but not outside it.
9.What is the Java Virtual Machine (JVM) good for?
The JVM provides memory management for: Bytecode execution, Objects, Parameters and local variables, Return values, Intermediate computation results
The JVM consists of three main components:1. Runtime Data Areas, 2. Class Loader Subsystem, 3. Execution Engine
10.What is the Java classpath used for?
The Java classpath specifies a list of directories, JAR files, and ZIP files where the JVM searches for and loads class files when running a Java program.
11.Is it possible to add new functionality to a system during runtime using the plug-in pattern? Is the same possible with inheritance? If so, how does that work?
Yes, it is possible.
▶ The plug-in architecture is the best way to add functionality not originally planned in a system. This architecture removes limitations on an application's features. ▶ Unlimited plug-ins can be added.
12.What are Microarchitectures?
Design patterns are used as proven construction principles on a small scale. They are also referred to as microarchitectures. hWhile design patterns represent fine-grained patterns, architectural patterns are coarse-grained patterns.
13.Two Use Cases for the Command Pattern
▶ Logging – Simplified Logging: Command objects can be queued in a stream, enabling a logging mechanism for commands ▶ Undo/Redo Functionality – Reversing Commands: An Undo/Redo mechanism can be implemented by storing executed commands, Each concrete command class must implement undo() and redo() methods, The references to executed commands are stored in a command stack, allowing tracking of which command should be undone or redone next
14.Can the Facade Pattern Prevent Access to Subsystem Classes?
The Facade Pattern provides a simplified interface for accessing a subsystem, but it does not technically prevent direct access to the subsystem classes. ▶ Direct Access Remains Possible:
The facade acts as a central access point, but subsystem classes are still accessible, Developers can bypass the facade interface, either intentionally or unintentionally ▶ Only Conventional Restrictions: Naming conventions and documentation can discourage direct access, but they do not enforce it, Structural restrictions can be enforced using package-private visibility (Java) or module-level access controls. ▶ Lack of Information Hiding: As long as direct access is possible, the system cannot fully benefit from information hiding.
15.What Is the Difference Between the Proxy and Adapter Pattern?
Both the Proxy Pattern and Adapter Pattern extend objects by adding functionality, but they differ in purpose and behavior: ▶ A Proxy uses the same interface as the object it represents. It acts as a placeholder or intermediary without modifying the interface. ▶ An Adapter modifies an existing interface to fit a new requirement. Unlike a proxy, an adapter changes the way an object is accessed or used.
16.How Should a Leaf Class Handle Child Functions?
A Leaf class inherits child-related functions from the Node base class but has no children:
▶ Methods such as getChildNode() or addChild() typically throw exceptions in a leaf object.
▶ This ensures that calling child-related methods on a Leaf object results in an error, enforcing the structural integrity of the Composite Pattern.
▶ The Composite class overrides these child-related methods to provide actual functionality.
17.Does the Template Method Pattern Work on a Class or Object Basis?
Since the Template Method Pattern relies on static inheritance, it is a class-based design pattern:
▶ Hook methods (overridable steps in the algorithm) are implemented in subclasses.
▶ Each subclass provides a unique implementation for these methods.

PART 1 SOFTWARE ARCHITECTURE OVERVIEW

1. In a software development process, which activities come immediately before and after the development of the software architecture?
→Klausuraufgaben 3 Seite, erste Spalte
2. Please explain the "Twin Peaks" model for requirement engineering.
→1. Spalte Spickzettel Seite 1, erste Spalte
3. Explain the difference between a solution concept and a software architecture.
→Klausuraufgaben 3 Seite, erste Spalte
4. Name four software architecture quality attributes.
→ 7.Thema Spickzettel Seite 1 erste Spalte
5. What is an architectural pattern? Give an example.
→ 11. Thema Spickzettel Seite 1 erste Spalte
6. What is a Idioms? Give an example.
Represent a composition of classes that collaborate to solve a specific problem ▶Are language-specific patterns Examples include: State, Factory, Singleton
7. Explain the difference between an architectural pattern and a framework
→ 10/11. Thema Spickzettel Seite 1 erste Spalte

PART 2 MODELING SOFTWARE ARCHITECTURE

1. What aspects does architecture have to consider mostly?
▶Structural aspects (components, interfaces, relationships) ▶Behavioral aspects (interactions, dynamics) ▶Quality attributes (performance, security, maintainability) ▶System context and constraints ▶Stakeholder requirements
2. What determines an architectural style in a software systems?
▶Set of component types that perform specific functions
▶Topological layout of components indicating their relationships
▶Set of connectors that implement communication/coordination
▶Semantic constraints ▶Design rules and patterns
3. Give at least three examples of architectural styles for software systems.
▶Data-centered style (clients accessing a central database)
▶Layered style (hierarchical layers with defined dependencies)
▶Pipe and filter style (sequential data processing) ▶Event-driven style (components reacting to events) ▶Service-oriented architecture (SOA)
4. Explain the difference between a prescriptive and a descriptive architecture.
→Spickzettel Seite 1, 2. Spalte ganz oben
5. Name two types of UML diagrams each for modeling structure and behavior.
→ Spickzettel Seite 1 unten links
8. What is the difference between inheritance and generalization?
→ Klausuraufgaben
9. Describe the different types of visibility in UML class modeling.
→ Klausuraufgaben

PART 6 EXTENDING SOFTWARE:

1. Please explain the open-closed principle in software architecture.
→3. Spalte Spickzettel
2. With inheritance you can add new functionality to a system without having to modify existing components. Why wasn't that possible in the Graphical Editor example?
Inheritance couldn't be used because the editor was poorly structured (lack of separation of concerns, tight coupling).
3. How can a plug-in manager find new plugins?
New plug-ins are found by registration or scanning predefined directories. Plug-ins implement interfaces and are integrated at runtime.
4. Is it possible to add new functionality to a system during run-time using the plug-in pattern? Is the same possible with inheritance? If so, how does that work?
Plug-in Pattern: Yes, new functionalities can be added at runtime.
Inheritance: Not possible because inheritance is statically defined at compile-time.

PART 7 WEB APPLICATION DEVELOPMENT WITH ECLIPSE:

1. What is a servlet in the context of a Java web application?
→3. Spalte Spickzettel
2. Why would you use JSPs rather than writing HTML directly from servlets?
→3. Spalte Spickzettel
3. In the JSP example project, go into the work/catalina/... directory and search for compiled JSPs.Study them. who has created the JSP class files? For the JSP compilation process:
▶The servlet container (e.g., Tomcat) creates Java servlet source files from JSPs ▶These are compiled into class files automatically ▶Located in the work/catalina/... directory ▶Generated during first access to the JSP ▶Container handles all compilation hand class loading

PART 9 RELATIONAL DBMS

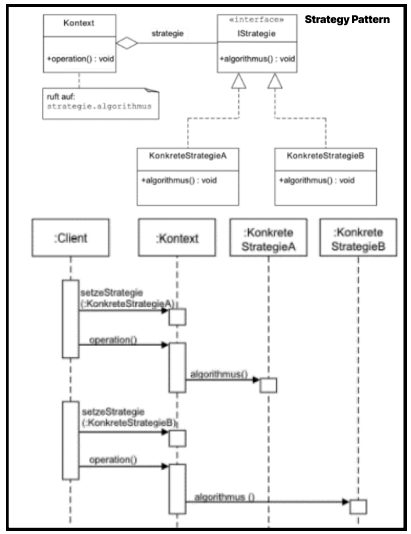
1. Give three examples for constraints in a relational database management system.
1.NOT NULL: Ensures a column must always have a value 2.UNIQUE: Ensures all values in a column are different 3.PRIMARY KEY: Uniquely identifies each row in a table Other examples include DEFAULT and FOREIGN KEY constraints.
2. What is "referential integrity"?
▶Referential Integrity: This is a database concept that ensures relationships between tables remain consistent. It means that:
▶Foreign key references must point to existing records in the referenced table ▶Records cannot be deleted if they are referenced by other records ▶Ensures data consistency and prevents orphaned records
3. What is "normalization" good for?
▶Eliminates data redundancy▶Ensures data consistency▶Reduces update anomalies▶Makes the database more efficient▶Organizes data in a logical manner It follows different forms (1NF, 2NF, 3NF) to progressively improve data organization.
4. What is a "foreign key"?
A foreign key is a column or set of columns that creates a link between data in two tables. It references the primary key of another table.
5. Create three entities named "article", "comment", and "user". A user can write articles and add comments to articles. A comment always belongs to an article. Create the UML diagram. Then create the entity relationship diagram. DIAGRAMM VERWEIS auf?

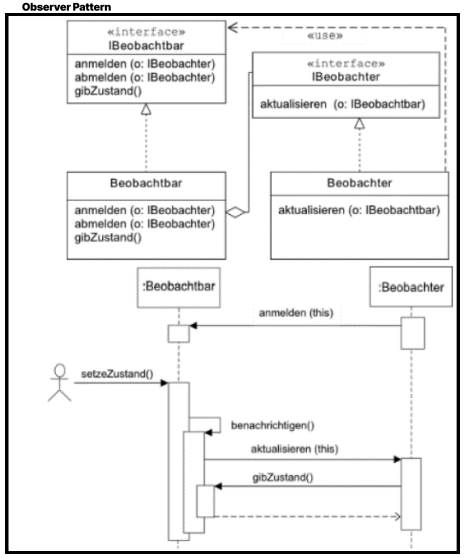
PART 10 JAVA PERSISTANCE ARCHITECTURE

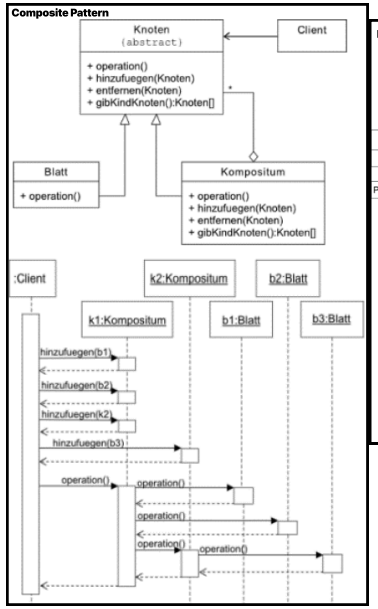
1. What is an ORM good for?
▶Bridging the gap between object-oriented programming and relational databases ▶Automatically converting data between database tables and Java objects ▶Reducing boilerplate database access code ▶Providing database independence ▶Simplifying data access operations ▶Managing relationships between objects
2. In the JPA stack, is the ORM positioned hierarchically above or below the JDBC layer? Please explain.
In the JPA stack, ORM is positioned above the JDBC layer:
oben:Application Layer→JPA/ORM Layer (Higher level abstraction)→JDBC Layer (Lower level database access)→Database, unten
ORM uses JDBC underneath but provides a higher-level abstraction for developers to work with objects rather than SQL queries directly.
3. What is an entity class in the jPA?
▶A Java class mapped to a database table▶Marked with @Entity annotation ▶Must have a no-arg constructor ▶Must not be final ▶Must have a primary key (marked with @Id) ▶Can be serializable ▶Represents a row in the database table
4. What is the function of the EntityManager in the JPA?
▶Is the central interface for managing persistence operations ▶Manages the lifecycle of entity instances ▶Provides operations for persisting, merging, removing, and finding entities ▶Maintains a persistence context ▶Handles transactions ▶Creates and executes queries
5. What does the EntityManager flush() command do?
▶Synchronizes the persistence context with the database ▶Writes pending changes to the database ▶Executes SQL statements for pending operations ▶Does not commit the transaction ▶Updates the database with in-memory changes
6. Explain te different types to map inheritance in the JPA.
JPA inheritance mapping strategies:
a) Single Table Strategy (@Inheritance(strategy = InheritanceType.SINGLE_TABLE)): ▶All classes in hierarchy mapped to one table ▶Discriminator column identifies specific subclass ▶Fast but can have unused columns **b)** Joined Table Strategy (@Inheritance(strategy = InheritanceType.JOINED)): ▶Each class has its own table ▶Child tables only contain additional fields ▶Uses joins to retrieve complete objects ▶More normalized but slower queries **c)** Table Per Class Strategy (@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)): ▶Each concrete class gets its own table ▶No discriminator needed ▶Tables contain all fields (inherited and own) ▶Can lead to data redundancy
7. Explain the different ways to generate primary keys in the JPA
a) GenerationType.AUTO: ▶JPA provider chooses appropriate strategy ▶Most flexible option ▶Provider-dependent b) GenerationType.IDENTITY: ▶Uses database identity column ▶Database generates key on insert ▶Can have performance implications ▶Common with MySQL auto_increment c) GenerationType.SEQUENCE: ▶Uses database sequence object ▶Requires database sequence support ▶Better performance for batch operations ▶Common with Oracle d) GenerationType.TABLE: ▶Uses a separate table to store key values ▶Works with any database ▶Poor performance but highly portable
8. JPA Eager and Lazy Loading explain the concepts
▶eager strategy is a requirement on the persistence provider runtime that data must be eagerly fetched, when related data is needed almost always
▶lazy strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first time accessed
▶lazy loading: used to defer loading of the attributes + associations of entities until the point at which they are needed ▲ more efficient ▼ exceptions no loaded
▶eager loading: opposite concept, where attributes + associations of entities are fetched explicitly + without need for pointing them ▼ impact performance

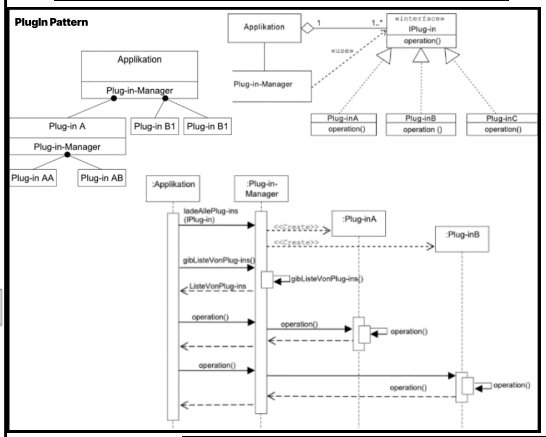
WS 23/24:

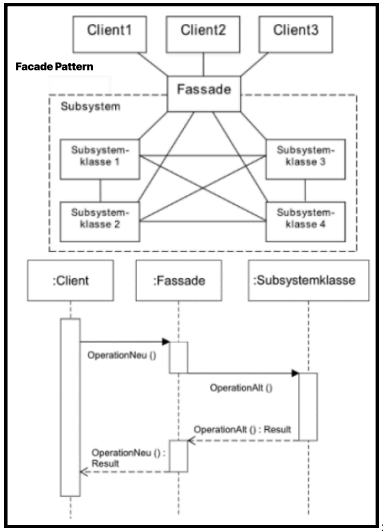
2.4 Give an example of where a reusable components interface characteristics are in conflict with that of a usable component.
Reusable vs. Usable Components ▶Reusable Component: A reusable component can be used in multiple projects or contexts. It is often generic, configurable, and flexible to meet different requirements ▶Usable Component: A usable component is optimized for a specific application or purpose. It is usually easy to use, with a clearly defined interface, but less flexible for other applications.
Difference ▶ Reusable components are versatile but often more complex to use.
▶ Usable components are easy to use but optimized for a specific task and less adaptable.
Example: UI Widget Design
▶ A reusable UI form framework could be highly configurable, supporting different input fields, validations, and layouts. This flexibility makes it reusable in many projects but results in a complex API with numerous configuration options.
▶ A usable UI component, on the other hand, could be a simple, pre-defined login form that can be used directly but offers limited customization options.
Conflict:
▶ The reusable component requires more configuration effort, which reduces usability.
▶ The usable component is ready to use but not adaptable to other use cases.
2.5 Explain the explicit interface principle.
→Spickzettel Seite 1, Letzte Spalte, ganz oben
3.3 Which design pattern would you best use to implement a binary tree? Please explain.
The Composite Design Pattern is best suited for implementing a binary tree. ▶ A binary tree consists of nodes, which can either be leaves (no children) or parent nodes with two child nodes.
▶ The Composite Pattern enables a recursive structure where both leaves and inner nodes implement the same interface, allowing the entire tree structure to be handled uniformly.
3.4 What can you do in a softwarearchitecture to keep softwarecomplexity low?
To keep software complexity low, the following measures should be considered in software architecture: ▶ Modularization – Divide the system into small, independent modules to improve maintainability. ▶ Clear Layered Model – Separate presentation, business logic, and data management for better structuring. ▶ Loose Coupling & High Cohesion – Components should be as independent as possible but strongly connected internally. ▶ Consistent Interfaces – Well-defined APIs prevent unnecessary dependencies. ▶ Use of Idioms – Proven design patterns help improve structure and maintainability. ▶ Automated Testing & CI/CD – Detect and prevent errors early to reduce complex debugging processes. These principles facilitate scalability, maintainability, and extensibility of the system.
3.5 Plugins (Plugin-Arch.) ▶Plug-ins are managed by a **plug-in manager** ▶The application defines interfaces for extensions ▶Using a component of the runtime environment, the plug-in manager instantiates and manages the plug-ins.
▶ Through the manager, the application gains access to the instances of available plug-ins without instantiating them itself ▶A plugin architecture is a software design pattern that allows an application to be extended dynamically by adding or removing components (plugins) at runtime.
▶A plugin is a self-contained software component that adds new functionality or modifies existing behavior in an application without changing its core code



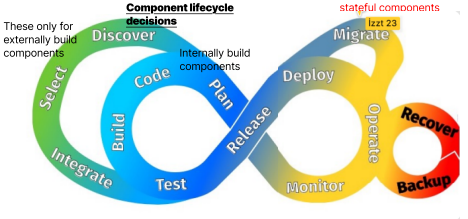




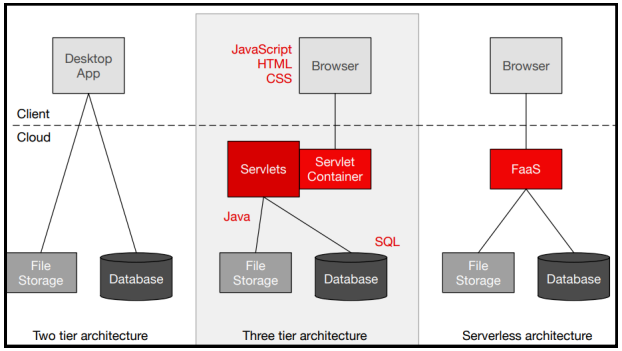




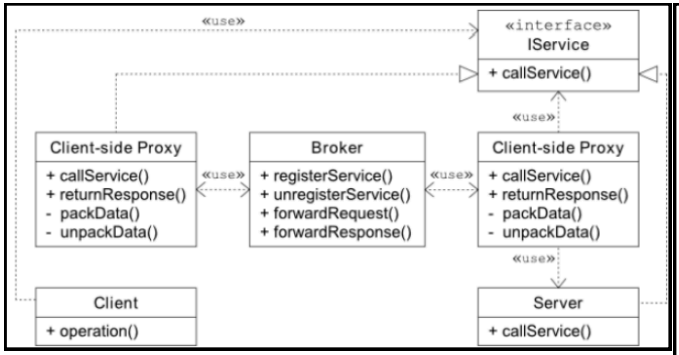
2.5 Plugin Architecture vs Monolithic Architectures Advantages
The main purpose of a plug-in architecture is to provide flexibility, extensibility, and modularity by allowing the system to dynamically load and integrate new features without modifying the core application. Compared to a monolithic architecture, a plug-in architecture offers greater adaptability and maintainability, making it ideal for applications that require frequent updates, customization, or third-party integrations (e.g., browsers, IDEs, CMS platforms). However, monolithic architectures can be simpler, more performant, and reliable for applications with stable requirements. ▶Use a plug-in architecture if flexibility, extensibility, and modularity are critical. ▶Use a monolithic architecture if simplicity, performance, and reliability are the primary concerns.



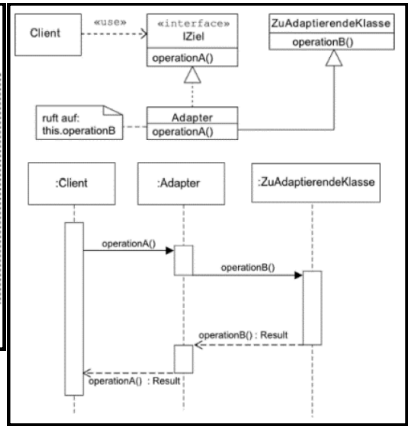
Popular Application Architectures (3-Tier Architecture)



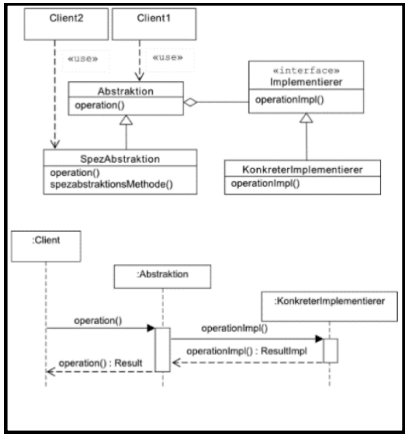
Broker Pattern



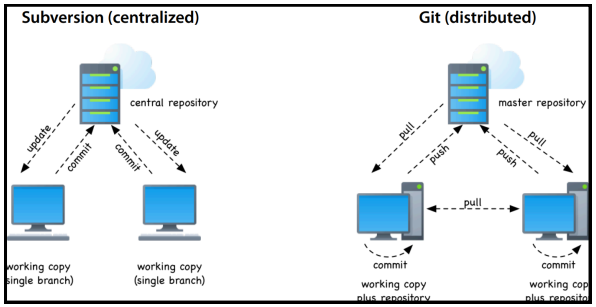
Adapter Pattern



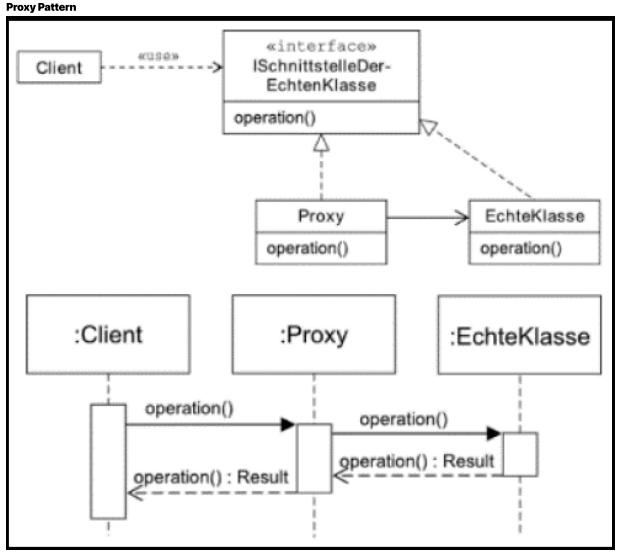
Bridge Pattern



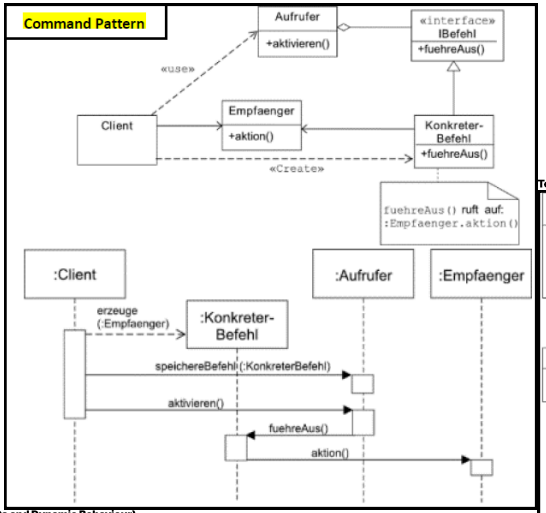
Centralizes vs. Distributed version control



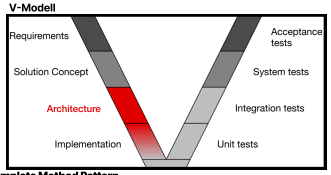
Proxy Pattern



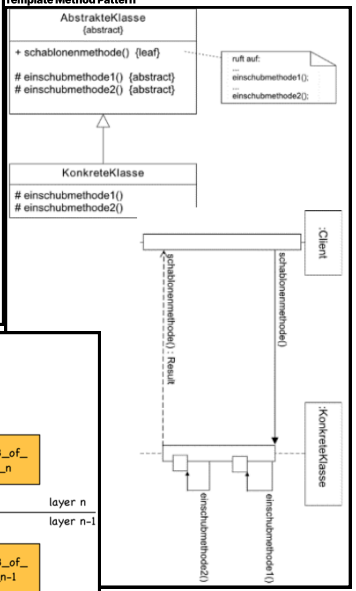
Command Pattern



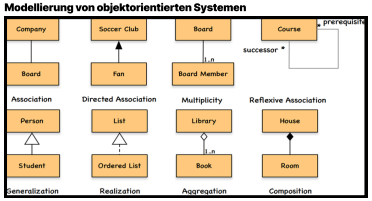
V-Modell



Template Method Pattern

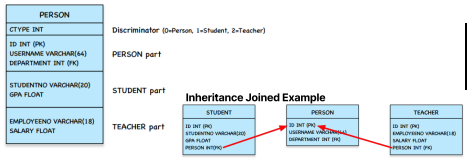


Modellierung von objektorientierten Systemen



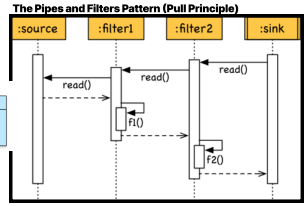
3-Tier web Architecutre

Inheritance Single Table Example

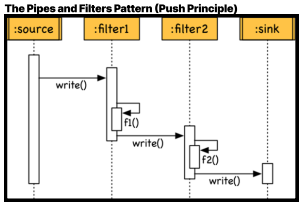


Inheritance Joined Example

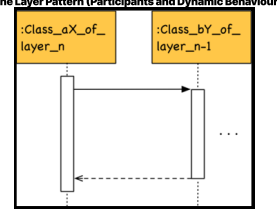
The Pipes and Filters Pattern (Pull Principle)



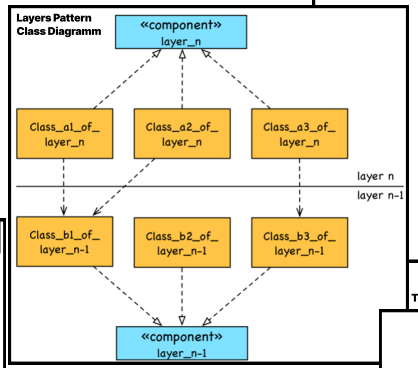
The Pipes and Filters Pattern (Push Principle)



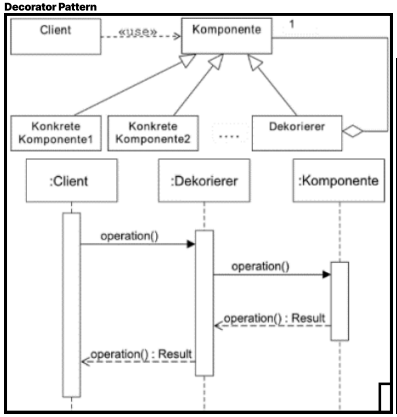
The Layer Pattern (Participants and Dynamic Behaviour)



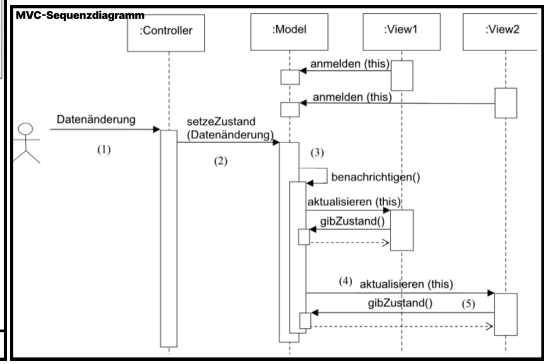
Layers Pattern Class Diagramm



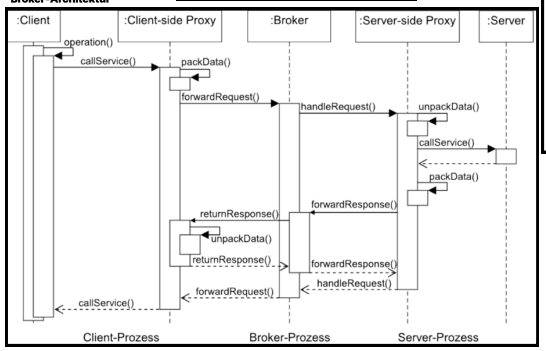
Decorator Pattern



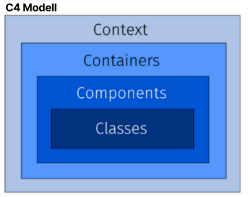
MVC-Sequenzdiagramm



Broker-Architektur



C4 Modell



Template Method Pattern Stack Beispiel