Things to learn:

- Software architecture is a set of structures and also abstractions.
- Structures are elements held together by relations between them.
- There are 3 kinds of structures - modules, components, allocation.
- **Modules** partition the software system into implementation units.
    - Modules are **static** structures.
    - Each module has a responsibility.
    - A module is either code, which must be developed or it already exists and must be acquired and integrated.
    - Modules are basis of work assignment. A module is assigned to a team.
    -
- **Components** are structures, which exist at runtime and they interact with each other.
    - A component can be an instance of a module.
    - Or it can be an instance of multiple modules, which are somehow related.
- **Allocation** structures describe mapping from modules or components to the environment of the software system.
    - For example mapping of modules to developers (who is responsible for developing the module).
    - Or mapping of components to physical infrastructure.
- A software structure is an **architectural structure** if it helps reason about the system the properties of the system.
- A software architecture is an abstraction because we are describing its software elements and the relationships between those elements.
- A software architecture does not include all details about all elements.
    - It only includes those details, which help reason about the system and its properties.
    - Only the necessary level of abstraction is shown in the architecture.
- There is a difference between an architecture of a software system and a representation of that system’s architecture.

Cohesion and coupling

- **Coupled modules** share responsibility. One module has knowledge about the other and vice versa.
    - Coupling manifests in code or in run-time behaviour.
    - We distinguish **loose coupling** and **tight coupling**.
    - If two modules are tightly coupled, change to one most likely requires change to the other.
    - When two modules are loosely coupled, change to one might still require change to the other!
    - To achieve loose coupling, we try to minimize the overl ap of responsibilities.
- Coupling results in dependencies between modules. There are different kinds of dependencies:
    - Data dependency.
    - Control dependency: module B must receive messages from module A.
    - Content dependency: module B expects, that module A does something.
    - Quality dependency: B relies on some level of performance/security of A.
    - Existence dependency: B relies on existence of A.
- **Module cohesion** expresses how responsibilities of a given module belong together (how strongly are the responsibilities related).
    - We distinguish **low cohesion** and **high cohesion**.
        - low cohesion means mixing different unrelated or weakly related responsibilities.
        - When code for one responsibility has to be changed, the change will be made to the other responsibility as well.

Other Types of Cohesion

- **Coincidental cohesion**: Responsibilities are grouped together arbitrarily.
- **Primitive data logic cohesion**: responsibilities manipulate the same kind of data at the level of a primitive data type. E.g. *StringUtils*
- **Temporal cohesion** means that the assigned responsibilities happen at the same time (they are related by time).
    - For example, all responsibilities, which happen at the time of system initiation, are temporally related.
    - Or all responsibilities at the time of system termination.
- **Procedural cohesion**: Responsibilities must be executed in a particular order, but they do not communicate with each other.
- **Communication cohesion**: Responsibilities form a communication chain, where each contributes to some output.
    - A non-software example is a letter-writing kit.
- **Sequential cohesion**: responsibilities are related by the information flow (outputs of one responsibility is the input of another responsibility).
- **Information cohesion**: responsibilities operate on the same business entity.
    - A non-software example is **a personal organizer on a fridge**.
- **Functional cohesion**: responsibilities are related by a single well-defined functionality that can be distinguished from the business point of view..

- The concern of all the responsibilities is to solve a single problem.

Decoupling techniques

Abstraction

- Interface
  - Defines a contract modules must adhere to.
  - It does not prescribe how it should be implemented.
- Abstract module (class)
  - Similar to an interface but can contain some implementation.
  - A base for other modules (classes) that ensures a certain level of contract adherence while allowing for customizable behavior.
- Encapsulation
  - generic principle
  - Internal details of a module must be hidden.
  - A module exposes only what is necessary.

design principles

- Single responsibility principle (SRP)
  - A module should have a single responsibility.
    - A module has only one reason to change.
  - Strongly related to the concept of high cohesion.
  - Reduces coupling on the module because it reduces the reasons why the other modules would need to depend on the module.
- Open-closed principle (OCP)
  - A module should be open for extension and closed for modification.
    - Future requirements should be satisfied by adding new classes, interfaces, or functions to the module.
    - Modifying existing classes, interfaces, or functions is prohibited.
  - Can be achieved by the abstraction techniques (interfaces, abstract modules)
  - Reduces coupling on the module because it enforces coupling only to the abstraction that does not change.
- Liskov Substitution Principle (LSP)
  - Subtypes must be substitutable for their base classes.
  - Parent classes or interfaces must be written so that all their derived classes must have valid implementation for all parent class methods.
  - Supports decoupling with the previous two principles by ensuring they work as intended.
- Interface Segregation Principle (ISP)
  - Many client-specific interfaces are better than one general-purpose interface.
  - In OOP, clients are not forced to implement methods they do not use, which would be unnecessary coupling.
  - In general, clients do not create coupling on unnecessary operations.
- Dependency Inversion Principle (DIP)
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.

Importance of Having Architecture

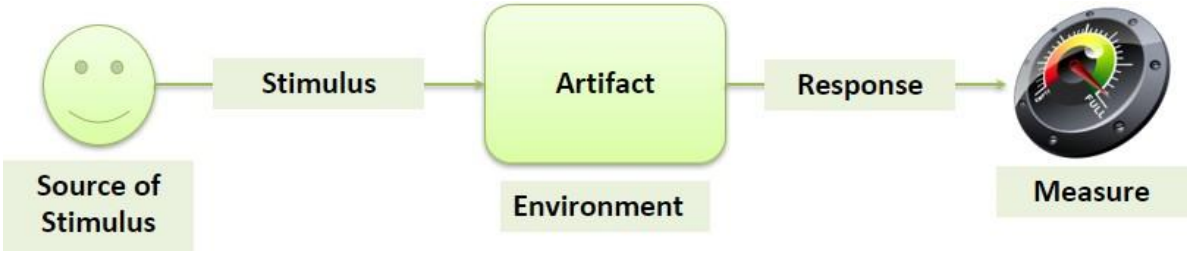
- Every software system has an architecture.
    - But the architecture might not have been documented.
    - The architecture might not have even been properly drafted and the software was developed spontaneously.
- The documentation of an architecture is the architecture's representation.
- The behavior of software elements should also be documented.
    - But only if it is important for reasoning about the system.
- Some reasons for having an architecture:
    - A software system fulfills **missions** for an organization.
    - The system lives in an **environment**.
        - The organization.
        - Customers.
        - Business partners.
        - Laws.
        - These create the environment.
    - **Stakeholders** are people and organizations who have some interest in the system.
    - **Concerns** are stakeholders' interests (performance, security, availability, …).
    - An **architectural description** is a collection of things which documents the architecture.
        - The description must identify the stakeholders and their concerns.
        - It also contains **views** (will be defined later).
        - It only contains the important aspects, which are needed to reason about the system. Other details may be omitted.
        - Note that functional requirements are typically NOT part of the description.
- The architectural description is **complete**
    - ⇔ it addresses all concerns of all stakeholders… <u>often impossible</u>
    - ⇔ it addresses all IDENTIFIED concerns of all stakeholders.
        - This definition is more practical and feasible.
- The description only includes important aspects, but different stakeholders want to see and emphasize different aspects.
    - Because of this, we include views in the architectural description.
    - **A view** is a description of the architecture from the perspective of related set of concerns.
- **A viewpoint** is a set of types of elements and relationships, which are used to describe specific views (*I think?*).
    - It also defines rules about how the elements and relationships should be used.

Requirements on the system

- Requirements on the system can be divided into 3 types: functional, quality, constraints.
- **Functional requirements** describe features that the system offers. How it should interact with the users.
    - These are satisfied by assigning responsibilities to parts of the system.
    - We design structures and assign functional responsibilities to them.
- **Quality requirements** describe how well the system should provide its functionalities
    - How fast, how secure.
    - How long can the service be down.
    - Stakeholders often don't mention quality requirements so we have to ask.
    - We satisfy quality requirements by designing additional structures.
- **Constraints** are facts, which we cannot negotiate about.
    - Laws of physics, legal laws etc.
    - It could be a **system constraint** - we are required to use legacy software or some software, which was already bought.
    - We satisfy these requirements by integrating them into our architecture and changing the architecture where necessary.

Quality attributes

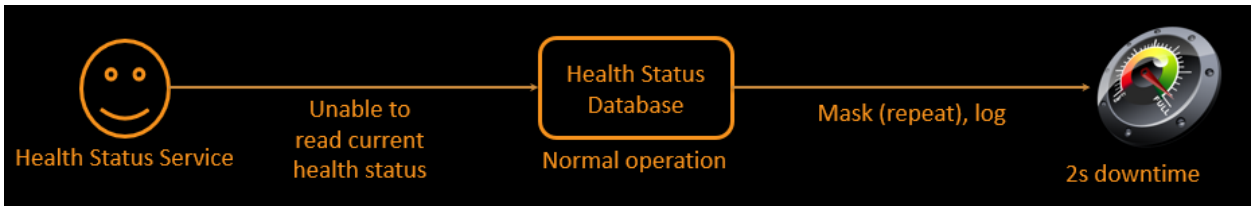
- Quality attributes are **measurable** and **testable** properties of the system.
    - They indicate how well the system satisfies quality requirements.
- A quality attribute can be associated either with the whole system or just a part of the system.
    - The most common case is that it's associated with a functionality.
- There are 3 categories of quality attributes: system, business, architectural.
- *Note: it's probably not important to remember exactly, which attribute belongs to which category.*
- System attributes are the most typical ones. They are further divided into run-time attributes and design-time attributes.
    - Availability, performance, security, …
    - Modifiability, testability, …
- Business attributes are for example: time to market, cost/benefit, …
- Architectural attributes are related to the quality of the architectural design and its documentation.
    - Correctness of the design: no mistakes nor contradictions.
    - Completeness of the design: it covers all requirements.
- We use **quality attribute requirement scenarios** to describe quality requirements.



- **Source of stimulus**: can be a human user, another system.
- **Stimulus**: a condition that requires a response from the system.
- **Artifact**: the system or its part, which receives the stimulus and should respond.
- **Environment**: the conditions, under which the stimulus occurred (for example, the system is in normal state or the system is in an overloaded state).
- **Response**: what the artifact did in response to the stimulus.
- **Measure**: whatever kind of measurement we need to see (for example, how long the operation took).

Runtime-time quality requirements

1. Availability:



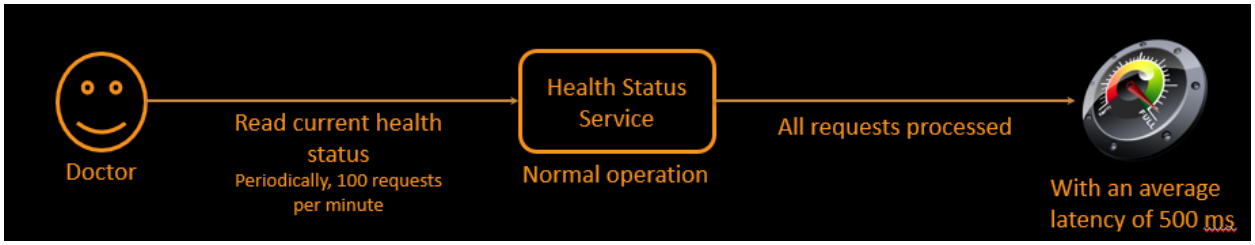
- It is the reliability of the system or some of its parts.
- The ability to recover from faults.
- It means that the system is there and ready to carry out its tasks when it's needed.
    - When the system (or its part) is not needed, then it doesn't make sense to speak about availability.
- What makes the system unable to carry out its tasks?
    - Hardware problems.
    - Software, which it depends on, is out of service.
- When a problem appears, but the system is able to mask it or repair it on its own, then the system remains available.
- Availability also refers to the ability to mask and repair problems.
- Usually, we **measure the downtime** - how long is the system out of service.
- A **failure** occurs, when the system does not provide the service consistent with its specifications and other systems or users observe this.
    - Failure is observable.
- A **fault** is a problem, which occurred but is not observable.
- A fault is not an availability problem. It only becomes one if it becomes a failure.
- Basic techniques for increasing availability: fault recovery and fault repair.
- We want to achieve 2 goals:
    - Mask the fault - so that it doesn't become a failure.
    - Repair the fault - so that it doesn't appear again.
- We have 3 types of tactics: fault detection, fault recovery, fault prevention.
- **Fault detection**:
    - The logic for fault detection can either be in the component or we can introduce a new component, which is responsible for detecting faults.
    - We can either detect fault whenever it occurs or we can make periodic checks.
    - **Monitors** are components used to keep track of the health state of other components. We use monitors when we want to separate fault detection logic from all other components.

    Fault detection tactics:
    - **Ping/echo**: one component pings another and awaits an echo.
    - **Heartbeat**: a component periodically emits heartbeat, which indicates that it is still alive.
    - **Timestamps**: each message or response contains a timestamp. These can be used to check if the messages come in the correct order and at the correct time.

    - **Timeout**: a component waiting for a response only waits until timeout.
    - **Voting**: for detecting incorrect messages (messages come at the correct time and in the correct order, but the content is wrong).
        - We introduce more of the same components. Usually we have 3 components (= **triple modular redundancy**).
        - These components vote about the result.
        - To deploy these extra components, we can use: replication, functional redundancy, analytic redundancy.
        - Replication: components are clones of each other.
        - Functional redundancy: components have the same public interface, but different internal implementation.
        - Analytic redundancy: different public interface and different internal implementation.
- **Fault recovery tactics** have 2 categories: preparation and repair, reintroduction.
- **Preparation and repair tactics** (fault recovery):
    - We use redundancy. There are 3 kinds of redundancy: cold spare, warm spare, hot spare.
    - **Hot spare**: the back-up component perform all the tasks just like the main component.
    - **Warm spare**: only the main component performs operations and then it informs the back-up components so that they can update their states.
    - **Cold spare**: the back-up component is out of service until a fault occurs on the main component, in which case a procedure is initiated to make the back-up into the main component.
    - **Rollback**: checkpoints are created and when a fault occurs, the system is reverted to a previously good state recorded in a checkpoint.
        - Creating checkpoints can be expensive or even impossible.
    - **Sagas**: each action on a component has an associated compensation transaction. When a fault occurs, the compensation transaction is executed.
    - **Retry**: simply try again and see if it works (for example in networks).
    - **Ignore**: the messages from a faulty component are ignored (ignoring messages form a denial-of-service attack).
    - **Degradation**: stop the less critical functions and only maintain the most important functions.
    - **Reconfiguration**: keep as many functionalities as possible, reassign the functionalities to whatever resources are left.
- **Reintroduction tactics** (fault recovery):
    - Helps with recovering a failed component.
    - **Shadow tactic**: the failed component is running “shadow mode” for some time. Then it is back to being the main component.
    - **State resynchronization**: it checks that the back-up components are synchronized with the main component.
    - **Escalating Restart**: the whole system or a component is restarted.
        - There may be different levels of restart - **escalating restart**.
- **Fault prevention tactics**:

- **Removal from service**: a preventive restart or reconfiguration to scrub latent faults like memory leaks etc.
- **Transactions**: operations in the system are executed in transactions to ensure that they are ACID (atomic, consistent, isolated, durable).
- **Predictive model**: a model implemented in a monitor component, which evaluates health and predicts possible faults.
- **Chaos engineering**: introduce turbulent conditions into the system to test it (for example randomly terminate instances in production).

2. **Performance:**



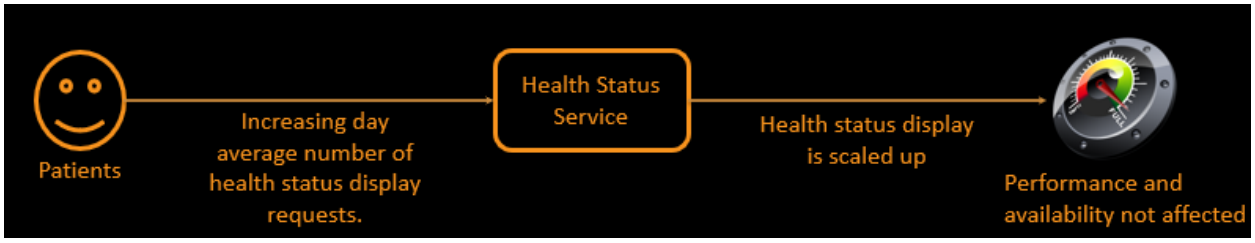
- It is the ability to meet timing requirements. How long it takes to respond to events.
  - Events are for example user requests.
- Example of measurement of performance is "number of processed transactions per minute."
- In a requirement scenario, we have:
  - Source of stimulus: internal (another component) or external (user, other systems).
  - Stimulus: events arriving to the system.
  - Artifact: system or component, which must provide certain performance.
  - Environment: operational mode when the event occurs (normal, overload etc.)
  - Response: the event is processed and the environment might change.
  - Measure: for example latency, throughput, unprocessed requests, …
- We distinguish 3 kinds of events, which can arrive to the system.
  - **Periodic events** arrive periodically (for example every 10 milliseconds).
  - **Stochastic events** arrive randomly based on some probability distribution.
  - **Sporadic events** arrive unpredictably.
- Response time of the system consists of **processing time** and **blocked time**.
  - **Processing time**: the system is consuming resources.
  - **Blocked time**: computation is blocked (waiting for shared resources or waitingfor another component to produce some results).
- There are 2 types of tactics:
  - **Control resource demand tactics**: produce smaller demand on resources.
  - **Manage resources tactics**: make the resource more effective in handlingdemands.
- **Control resource demand tactics:**
  - **Reduce sampling frequency**: reduce the number of arriving events (for example reduce the sampling frequency of a monitor - reminder: monitors keep track of health status of components).
  - **Limit event response**: when too many events arrive, they are put into a queue

and what is done with them is specified in the requirement scenario.
    - For example **only serve some events**, others are not served at all - this would be an availability problem.
    - Or events can have **assigned priorities**. High priority events are served first.
    - Or **limit execution time or resources**. Each event can either be processed until some timeout or only a certain number of iterations is allowed.

- **Produce smaller demand on resources**:
    - Improve various implementations and algorithms to use less resources.
    - Change the architecture - using more components to respond to a single event usually means higher latency, because components need to communicate with each other.
    - Optimize communication between components
        - Optimize serialization and deserialization of data.
        - Reduce the number of requests between components.
        - Remove intermediary components.
    - Utilize edge computing: get the data and computation nodes as close to the customer as possible.

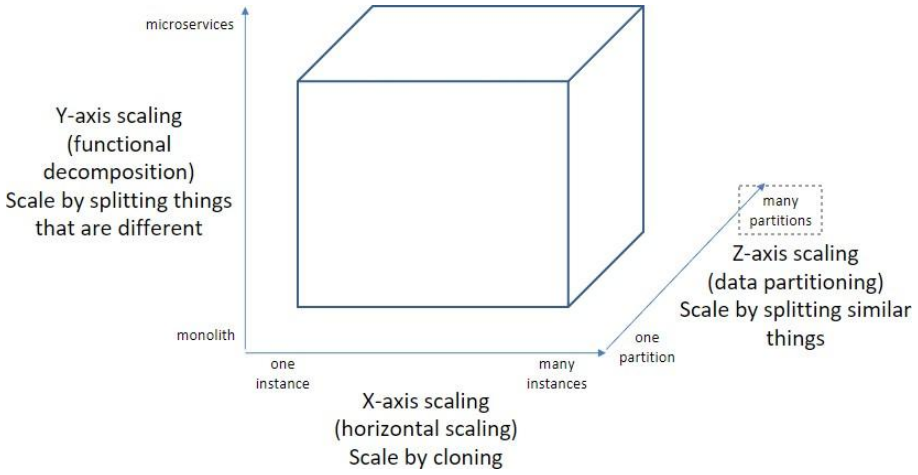
- **Manage resources tactics:**
    - **Increase available resources**: get faster CPUs, more memory etc.
    - **Introduce concurrency**: some events are complex and take a long time to process and while they are being processed, we cannot process other, simple events.
        - We can schedule events in parallel using concurrency so that the simpler events are still processed and don't have to wait.
    - **Replicate components**: use replicas of a component and a load balancer.
    - **Replicate data**: high latency can be caused by accessing data so we can duplicate or cache data.
        - But we need to somehow keep these copies of data synchronized.

3. Scalability:



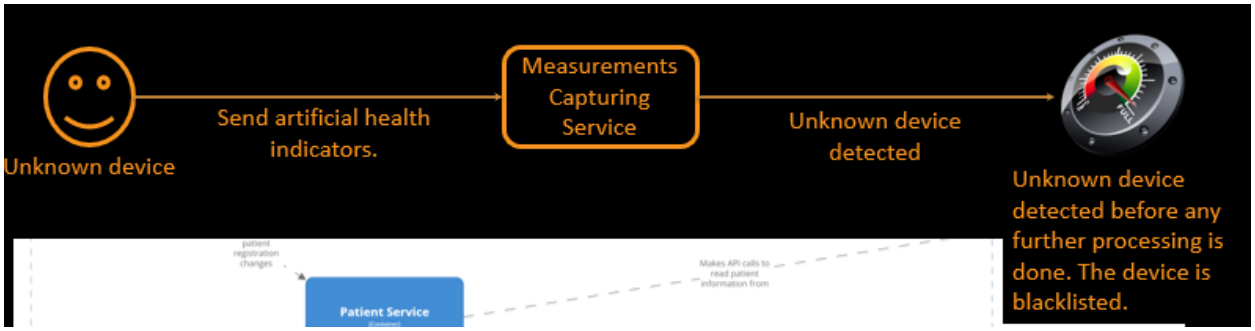
- It is the ability to handle tasks as the system grows in size. Growth is for example:
    - Increased number of users and requests.
    - Additional features requested by stakeholders.
- A narrower definition of scalability:
    - The ability of the system to keep its performance and availability requirements as the number of users, requests and data grow.

    - And the ability to keep its modifiability requirements as the number of required features grow.
- Requirement scenario:
    - Source of stimulus: users or other systems or the reason for the data growth.
    - Stimulus: what grows? (for example, number of requests).
    - Artifact: the component that needs to be scaled.
    - Environment: when is the scaling needed? (during run-time, build time, …)
    - Response: the artifact is scaled accordingly.
    - Measure: time to scale and other availability requirements (whether they must stay the same or can be changed).
- **Scalability tactics** can be visualized in 3 dimensions on a scalability cube.



- **Split similar things**: multiple instances behind a router, each instance is responsible for a subset of data.
    - The router redirects requests to appropriate instances.
- **Split things that are different**: functions are split to separate **microservices**.

4. **Security:**



- The ability to protect data and information from unauthorized access but still provide access to authorized people and systems.
- An **attack** is an attempt for an unauthorized access.
- We have 3 main goals: confidentiality, integrity, availability.
- **Confidentiality**: data and services are protected from unauthorized access (data read).
- **Integrity**: data and services are not subject to unauthorized manipulation (data write).
- **Availability**: system is available for authorized use.
    - For example, DDoS attack won't prevent users from reading some data.

- Other goals: authentication, authorization, nonrepudiation.
    - Nonrepudiation means that if someone sent a message, they cannot deny having sent it. If someone received a message, they cannot deny having received it.
- Requirement scenario:
    - Source of stimulus: an attacker (human or another system).
    - Stimulus: the attack (for example, an attempt to display or change data).
    - Artifact: the system or its component. Environment: conditions, under which the artifact is attacked (for example, if the system is online or offline).
    - Response: confidentiality, integrity and availability is preserved.
    - Measure: how much of the system is compromised, how long it took to detect the attack, how long to recover from the attack etc.
- There are 3 types of security tactics: detecting attacks, resisting attacks, recovering from attacks.

- **Detecting attacks:**
    - **Detect intrusion**: try to recognize, that something unusual is happening by comparing what is normal to what is happening.
    - **Detect service denial**: compare incoming network traffic to historic profiles of known DoS attacks.
    - **Verify incoming message integrity**: check timestamps, signatures, hash values, checksums to make sure that the incoming message is legit.

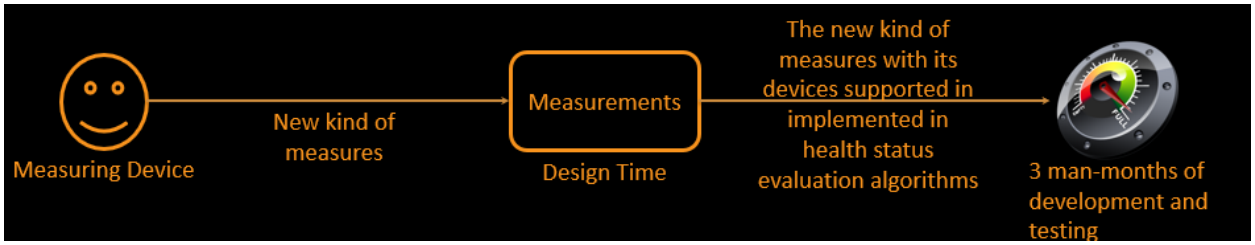
- **Resisting attacks:**
    - **Authentication**: use log-in credentials, two-factor authentication etc.
    - **Authorization**: make sure that the authenticated actor has the right to the requested operations.
    - **Limit access**: control who may access which part of the system (can use demilitarized zone).
    - **Limit exposure**: conceal facts about the system (for example, we don't disclose the physical locations of hosts for services).

    - **Encrypt data**.
    - **Separate data**: sensitive data is stored separately and a different identification mechanism is used for them (so that when attackers access them, they still cannot identify, to who the data belongs).

- **Recovering from attacks:**
    - **Restore services and their states**: use availability tactics to restore states and services.
        - For example cold or hot spares.
        - For data, we can have data back-ups.
    - **Maintain audit trails**: keep records so that we can trace the actions of the attackers.

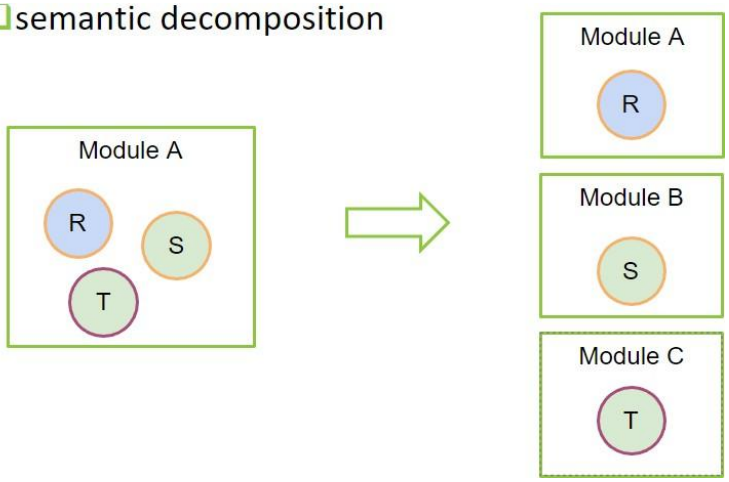
Design-time quality attributes

1. Modifiability:



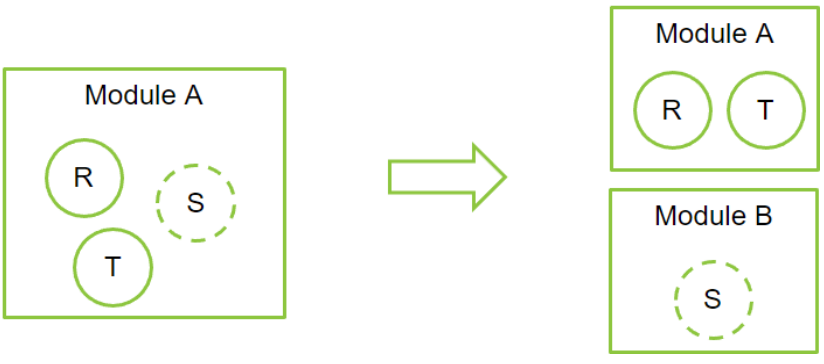
- Is about changes in the system. What can change?
- Add or remove features, fix bugs, improve performance, adapt to new standards and technologies, integrate with other systems, …
- We are interested in the cost and risk of making changes.
- Most common measure: time and money needed to make changes.
- Concepts that affect modifiability: module responsibility, module coupling, module cohesion.
- If two modules overlap in their responsibility, then a change to one likely requires change to the other.
- To increase modifiability, we want to <u>increase cohesion</u> and <u>reduce coupling</u>.
- To increase cohesion, we can use: semantic decomposition, decomposition based on anticipated changes, decomposition based on shared responsibilities.
- **Semantic decomposition**: two responsibilities, which do not serve the same purpose, should be placed in different modules.

❑semantic decomposition

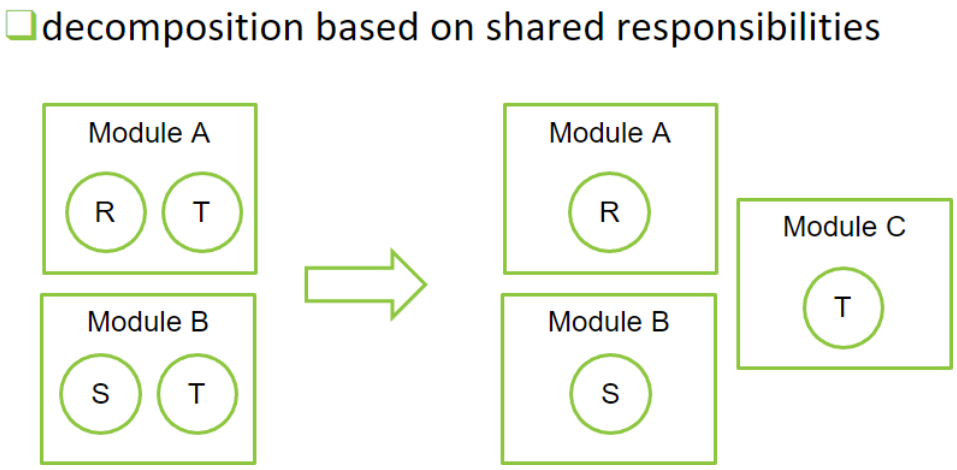


  - Responsibilities R, S and T are placed in different modules.
- **Decomposition based on anticipated changes**: if we expect, that a responsibility is likely to be changed, move it to a separate module.

❑decomposition based on anticipated changes



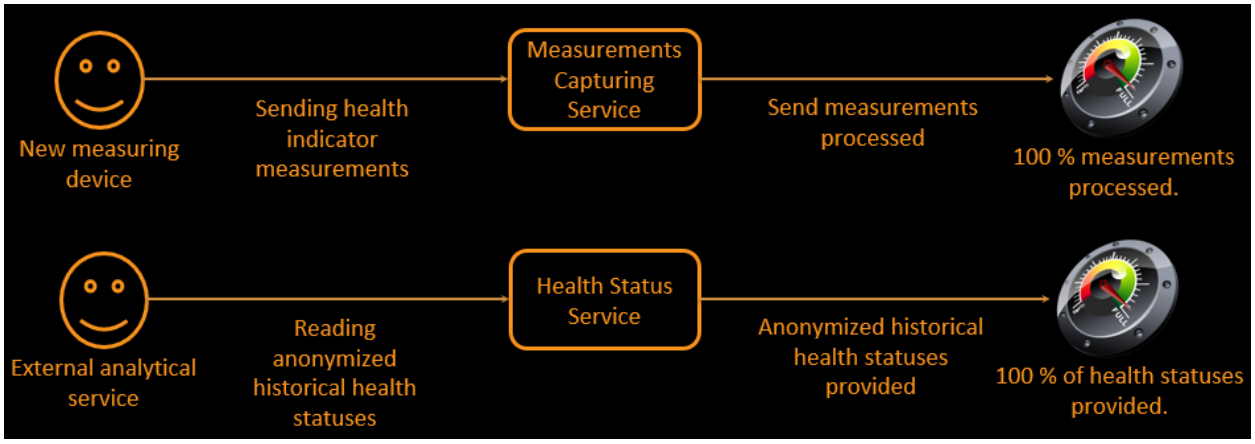
    - R, T, S are all related, but we determined that S is likely to change, so we move it to its own module.
- **Decomposition based on shared responsibilities**: 2 modules sharing a responsibility means that change will affect both modules. Move this responsibility to a separate module.



    - Responsibility T is shared between module A and B so we move it to a separate module C.
- Increasing cohesion by separating responsibilities can lead to dependencies between new modules, which leads to tight coupling. We have to be careful.
- To reduce coupling, we can: restrict dependencies, hide information (encapsulation), use an intermediary translator.
    - prevent a ripple effect = a module is modified only because it depends on another module that was modified.
- **Restrict dependencies**: we say that a module can only use a limited set of other modules.
    - For example in a layered architecture, a module can only communicate with the neighboring layer.
- **Encapsulation**: modules have explicit public interface, which other modules can use and depend on.
    - This eliminates dependency on internal implementation.
- **Intermediary translator** is a new module, which translates messages or calls between two other modules.
    - For example module A has a neat and modern interface but it uses a legacy module B, which is either unstable or doesn't have a nice interface.
    - We don't want to adapt module A so that it can use B and possibly corrupt A in the process. So we introduce a component called the intermediary translator, which forwards messages between A and B.
    - A variant of the translator is the **message broker**. When we have a lot of modules and the communication is not always well defined, then each module sends a message to the broker, which will locate and deliver the message to the correct recipient.
- **Preservation** is a tactic, which supports reducing coupling.
    - When we introduce a new version, the old is not immediately removed, but marked as deprecated. This gives clients time to update.
- **Code refactoring** is another tactic to reduce coupling and increase cohesion.

- **Defer binding**
    - On the highest level of abstraction, we have low-code or no-code approaches.
    - On the lowest level of abstraction, we have parametrized functions.

2. Interoperability:



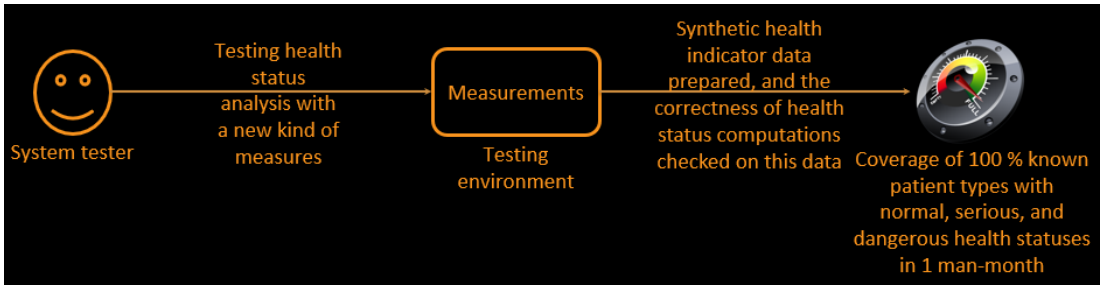
- The degree to which two or more systems can meaningfully exchange messages via their interfaces.
    - Systems exchange services by mutually requesting services of each other.
- **Technical interoperability** means that the systems share protocols, which enable them to exchange information with each other.
    - For example, two systems can use HTTP as the common network communication protocol.
- **Syntactic interoperability** means that one system understands data format from the other.
    - For example, system A sends data in JSON to system B. System B must be able to parse JSON.
- **Semantic interoperability** means that all systems interpret data in the same way.
    - For example, when a number is sent in small endean, then the receiver must also interpret this number in small endean.
- Requirement scenario:
    - Source of stimulus: systems that we must interoperate with.
    - Stimulus: a request to exchange information between systems.
    - Artifact: the system or the part that must be interoperable.
    - Environment: do systems know about each other during design time or do they need to find each other dynamically at runtime?
    - Response: the exchange request is either processed or rejected.
    - Measure: can be the percentage of correctly exchanged information (measure is hard to define).

- **Technical interoperability tactics:**
    - **Enforce a shared communication protocol**: for example, use HTTP.
    - **Intermediary broker**: introduce a component, which will translate messages.
    - **Directory service**: use a directory service to locate other systems (*by locating*

I guess we mean for example to look up the IP addresses of other systems).

- **Syntactic interoperability tactics:**
  - **Enforce a shared data format**: for example DCV.
  - **Intermediary broker**.
- **Semantic interoperability tactics:**
  - **Enforce a shared format**.
  - **Use ontologies**: *I don't know what they are. Just have to remember that it is one of the tactics.*
  - We also need to manage entities and their identities. For example, when one system encodes the InterContinental Hotel Prague, we expect that another system will decode this information and interpret it as the hotel that we meant.
  - For entities and identities, we can use:
    - **Identity broker**.
    - **Shared identifiers** (for example, EAN codes).
    - **Linked identifiers** - each system uses its own identifiers for entities, but these identifiers are linked to a centrally managed database of identifiers.

3. Testability:



- In the software architecture, our goal is to design the architecture in a way that will make it easy to write tests.
- Most commonly, we consider execution-based testing (our tests execute modules and test them as runtime components).
- Decomposition views and usage views help developers and testers see, what must be tested.
- For a system to be **testable**, we need to be able to control each component's inputs and manipulate its internal state.
- Requirement scenario:
  - Source of stimulus: whoever performs the tests.
  - Stimulus: the set of tests being executed.
  - Artifact: the component or system being tested.
  - Environment: time of testing (for example, at development time or at integration time etc.)
  - Response: what we want to system to do so that we can observe the results and perform tests.
  - Measure: how easily the system gives up its faults and how long it took (for example, "the unit tester will be able to write tests, which cover 85% of execution paths in 3 hours").

- There are 2 basic **categories of testability** tactics: add controllability and observability, limit complexity in the system's design.
- **Controllability and observability tactics:** the component maintains some information for testers.
    - **Specialized interfaces**: have special methods only for testing purposes.
        - Turn on verbose mode, getters for some state variables etc.
    - **Record/playback**: faults are sometimes difficult to recreate.
        - Record the state of the component and then we can recreate the fault by playing back the recorded state information.
    - **Localize state storage**: a support tactic for record/playback.
        - We store the recorded states in a persistent storage.
    - **Abstract data source**: make the data source for the component abstract so that we can supply it with a source of testing data (rather than for example having to plug the component into an actual database).
    - **Sandbox**: isolate the component so that we can test and experiment without having to worry about undoing the consequences of the experiments.
- **Limit complexity tactics:** complex systems are harder to test.
    - **Limit structural complexity**: minimize dependencies between components, avoid cyclic dependencies.
        - High cohesion and loose coupling helps here.
    - **Limit nondeterminism**: it is almost impossible to recreate states in a nondeterministic component.
        - Isolate and minimize nondeterminism.

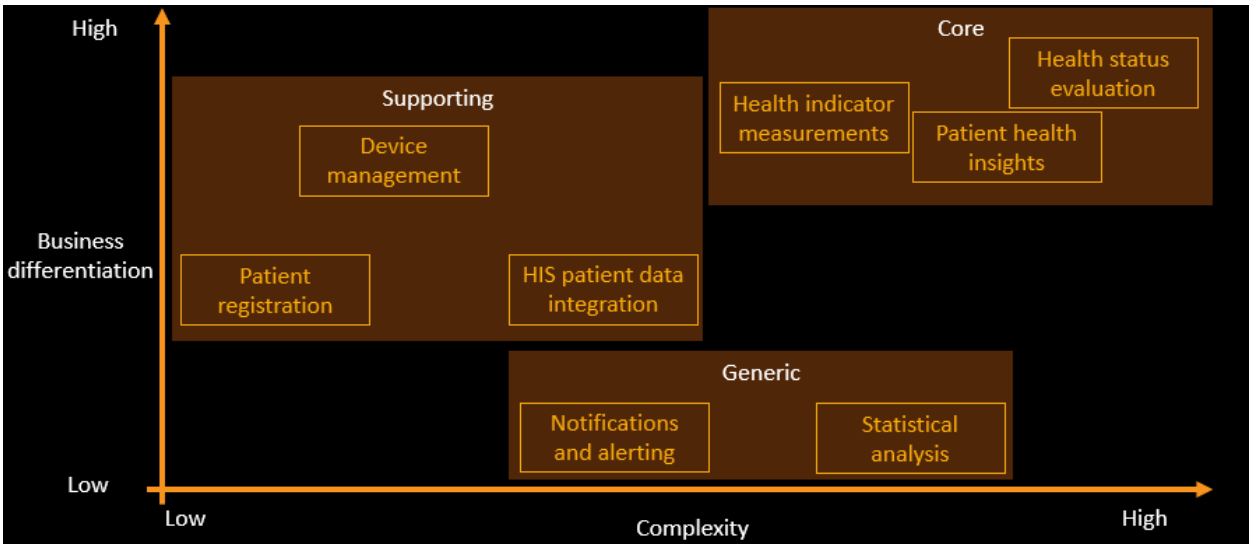
4. **Usability:**
    - Refers to how easy it is for users to accomplish desired tasks, what kind of user support we have.
    - Usability includes:
        - **Learning system features**.
        - **Using the system efficiently**: for example, a diagram editor may offer users the ability to move multiple elements at once (rather than one at a time).
        - **Minimizing the impact of errors**: for example, users want to revert to a previous version of their work.
        - **Adapting to user needs**: for example, automatically fill in the inputs previously entered by the user.
        - **Increasing confidence and satisfaction**: for example, when performing a long task, indicate this to the user.
    - We often need to introduce new modules to achieve these things listed above.
    - Requirement scenario:
        - Source of stimulus: the user interacting with the artifact.
        - Stimulus: whatever the user wants to do (for example, tries to learn the system).
        - Artifact: the system or its component or its feature that the user is using.
        - Environment: most of the time it's runtime (other systems may offer special training modes etc.)
        - Response: provide features or anticipate user needs or support the user in

learning.
    - Measure: various things like task time, number of errors, user satisfaction, the ratio of data loss when an error occurs, …
- There are 2 categories of tactics: support the user's initiative, support the system initiative.
- **Support the user's initiative tactics:** meant to help the user use the system by supporting their activities.
    - **Cancel**: introduce a module that supports cancelling the currently running task upon user's cancellation request.
    - **Pause/resume**: introduce a module that can temporarily free resources when a pause request is received and resume computing when a resume request is received.
    - **Undo**: introduce a module that maintains information about the system's state so that the user can revert to a previous state.
    - **Aggregate**: the ability to apply user operations to groups of objects (for example, editing multiple elements in a diagram editor like mentioned above).
- **Support the system's initiative tactics:** enable the system to assist the user with automated or semi-automated steps.
    - We need models, which enable us to predict user intentions. These require new modules in our architecture.
    - **Task model**: tasks performed by the user (for example, correct typing errors).
    - **User model**: previous behaviours and decisions of the user (for example, we can highlight the most frequently used options in the main menu).
    - **System model**: behaviours of the system (for example, we can show progress bars, which predict the estimated completion time for various tasks).

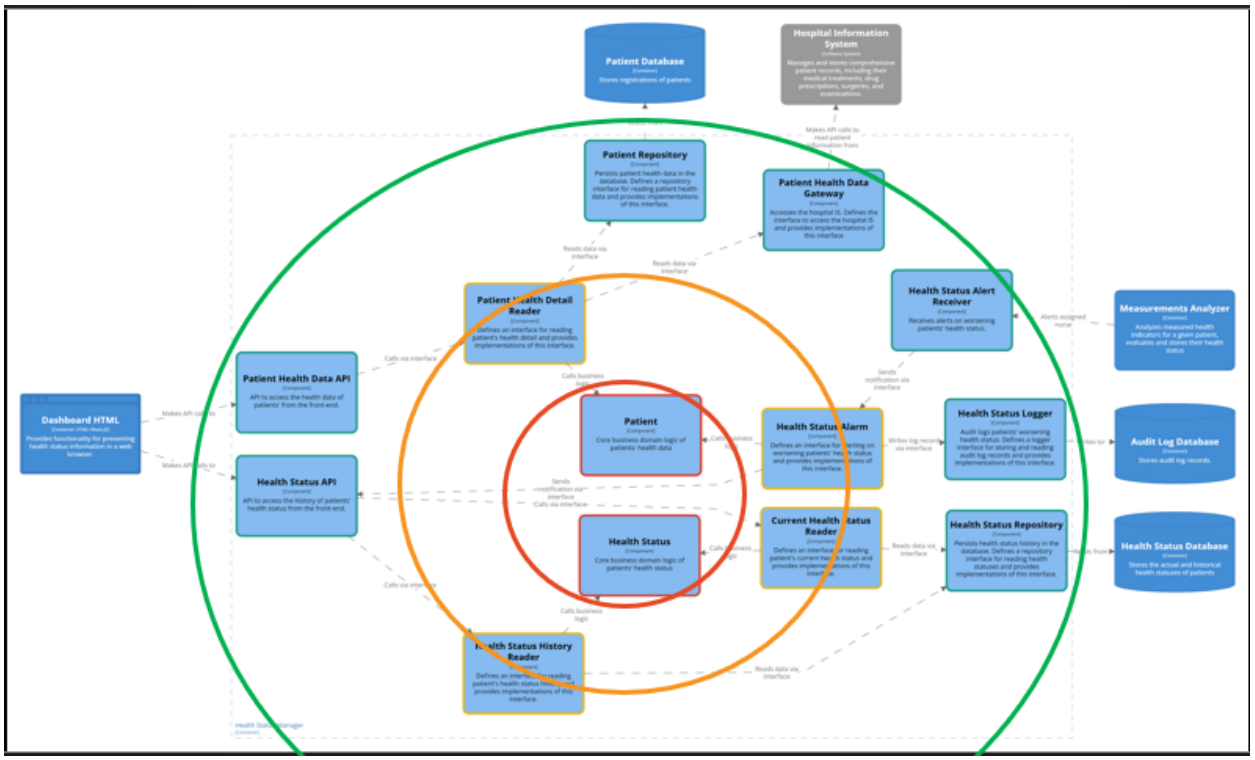
Architectural patterns

- We have seen some patterns: load balancer, broker, router, API.

Domain-driven pattern:

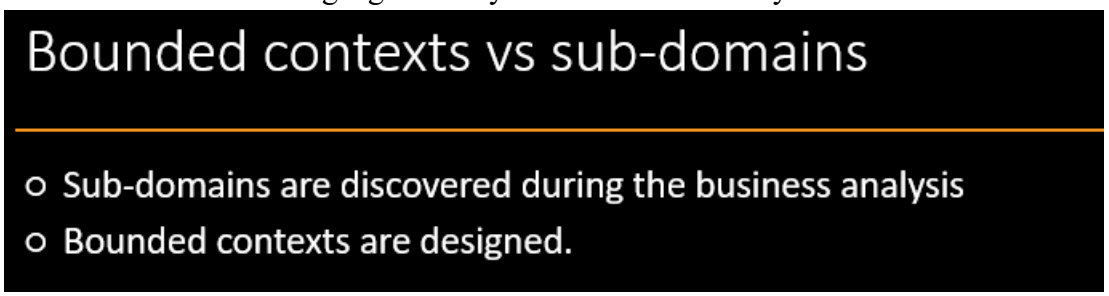


- It is a type of the layers pattern.



- The center of the architecture is the **domain layer**.
  - This layer contains all the important business logic and it is <u>independent</u> of any other modules in the architecture.
- Around the domain layer, we have the **application service layer**.
  - This layer implements use cases as application services.
  - This layer exposes what the system does but it hides how the system does it. This ensures that changes to the domain logic will not affect other parts.
  - This layer does things like: validate user inputs, supply the objects in the domain logic with data from the database, tell domain objects to do some work and collect the results, …
- Around the application service layer, we have the **infrastructural layer**.

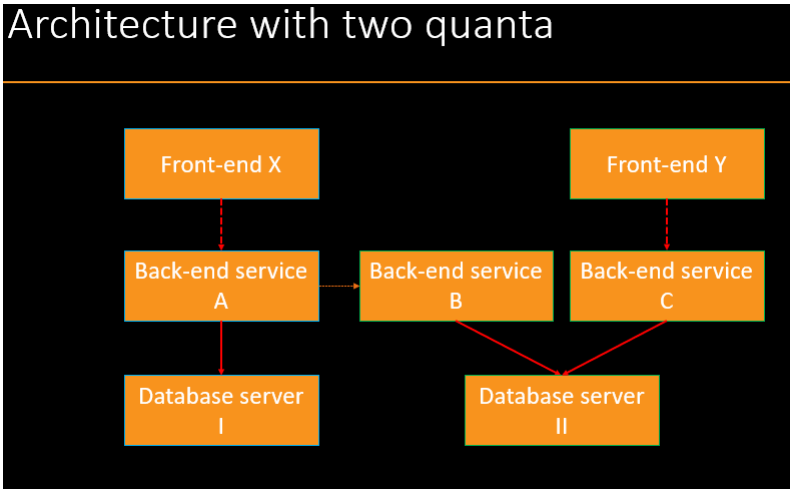
    - This layer enables the application to be consumed by human users via a graphical user interfact or an API.
    - It is also responsible for other things like: persistence of data, logging, notification, security etc.
- All dependencies go inwards. The infrastructural layer depends on the application service layer or the domain layer, the application service layer depends on the domain layer.
    - The application service layer depends on the existence of the infrastructural layer (which is an outwards direction), but it does not depend on any specific framework or language used by the infrastructural layer.



-
- **MDD = model drivern design**
    - A model is a simplified representation of a real-world (sub-)domain that intentionally emphasizes certain aspects while ignoring others.

Modern architecture styles classification

- **Partitioning type**:
  - **T** = primarily partitioned technically, i.e. partitioning logic is based on technical responsibilities (usually leads to layered cohesion)
  - **D** = primarily partitioned by subdomains, i.e. business areas covered by the systém
- An **architecture quantum** is an independently deployable artifact with high static coupling + synchronous dynamic coupling.
  - Static coupling describes how modules are wired together in the code.
  - Dynamic coupling describes how the modules call one another at runtime when deployed as separate run-time artifacts.



- Monolithic architecture styles
  - Layered architecture style
  - Pipeline architecture style
  - Microkernel architecture style
  - Modular monolith architecture style

	Layered	Pipeline	Map-Reduce	Microkernel	Modular Mon.
Partitioning type	T	T	T	T+D	D
Quanta	1	1	1	1	1
Deployability	★	★★	★	★★★	★★
Elasticity	★	★	★	★	★
Fault tolerant	★	★	★	★	★
Modularity	★	★★★	★★★	★★★	★★★★
Overall cost	★★★★★	★★★★★	★★★	★★★★★	★★★★
Performance	★★	★★	★★★★	★★★	★★
Reliability	★★★	★★★	★★★	★★★	★★★
Scalability	★	★	★★★★	★	★
Simplicity	★★★★★	★★★★★	★★★	★★★★	★★★★
Testability	★★	★★★	★★	★★★	★★★

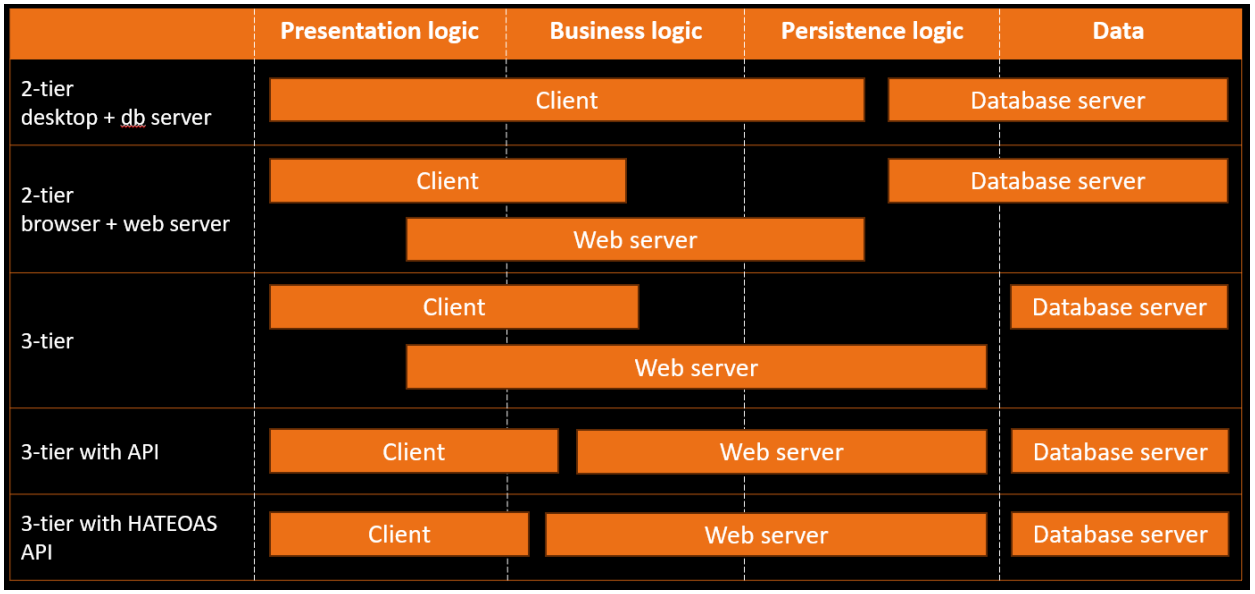
- Distributed architecture styles
  - Service-based architecture style
  - Event-driven architecture style

- Space-based architecture style
- Service-oriented architecture style
- Microservices architecture style

	Modular Mon.	Service-based	Event-driven	Space-based	Service-oriented
Partitioning type	D	D	T/D	T/D	T/D
Quanta	1	N	N	N	1
Deployability	★★	★★★★	★★★	★★★	★
Elasticity	★	★★	★★★	★★★★★	★★
Fault tolerant	★	★★★★	★★★★★	★★★★	★★★
Modularity	★★★★	★★★★	★★★★	★★★★	★★★
Overall cost	★★★★	★★★★	★★★	★★	★
Performance	★★	★★★	★★★★★	★★★★★	★★
Reliability	★★★	★★★★	★★★	★★★★	★★
Scalability	★	★★★	★★★★★	★★★★★	★★★
Simplicity	★★★★	★★★	★	★	★
Testability	★★★	★★★★	★★	★	★

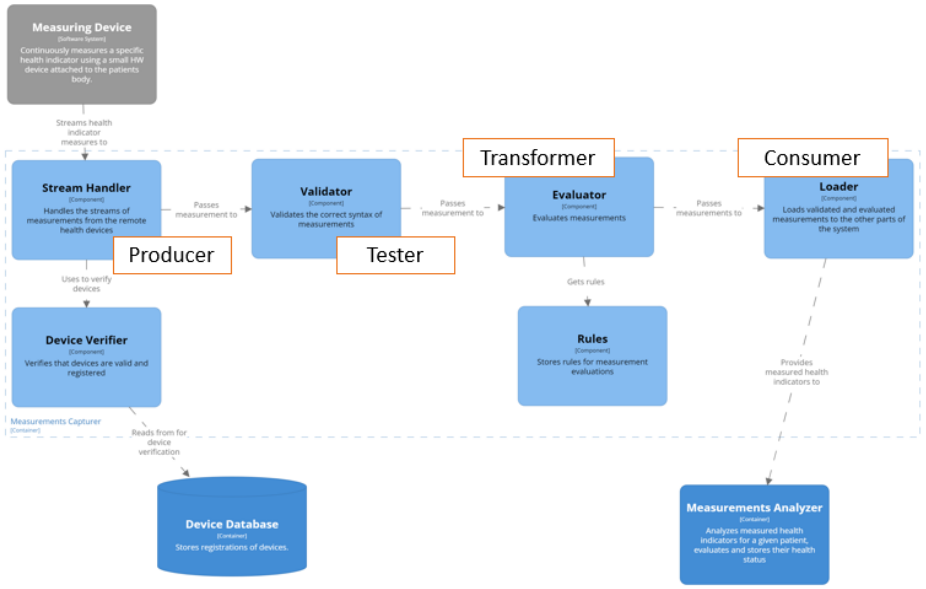
1. Layers:

- When we have many dependencies, evolving one part of the system causes changes to an unknown number of other parts.
- In the layers pattern, the architecture of the system is divided into layers.
- Each layer groups modules with a cohesive set of services.
- Each layer can only use the layer next to it.



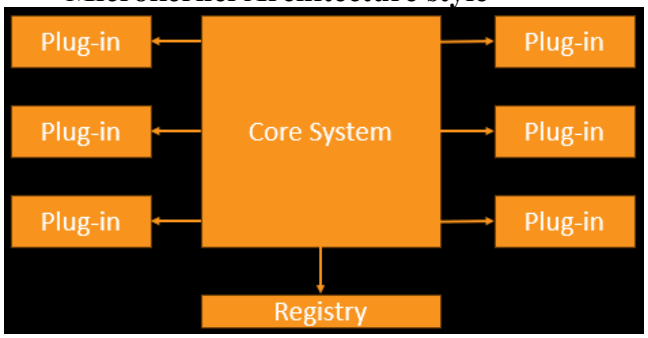
- Case 1: Small web applications with simple business logic and data
- Case 2: When you start, you do not know a lot, but you need to

Pipeline architecture (map-reduce)



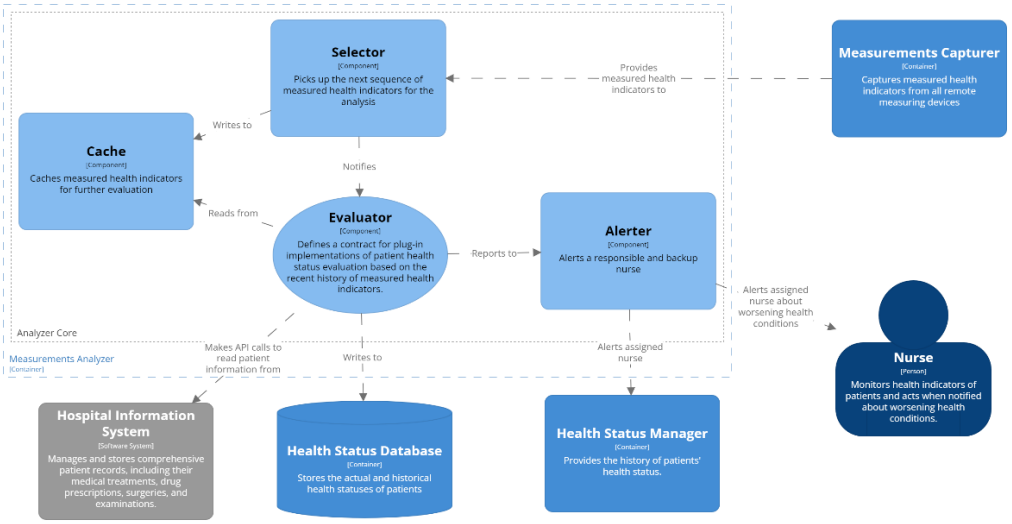
- Producer (also called source, extractor)
  - Starting point of the pipeline
  - Outbound only as it forwards data to the outbound pipe(s)
- Transformer (also called map)
  - Accepts input, performs a transformation (possibly empty), forwards to the outbound pipe(s)
- Tester (also called reduce)
  - Accepts input, tests one or more criteria, produces output (possibly empty) based on the test
- Consumer (also called loader)
  - Termination point of the pipeline
  - Persists the final result to a filesystem or database, sends it somewhere, etc.
- Use cases:
  - **ETL** (Extract-Transform-Load) architectures
  - **Map-Reduce architectures**
  - Electronic Data Interchange (EDI) architectures
  - Multimedia (audio/image/video) processing
  - Manufacturing production lines

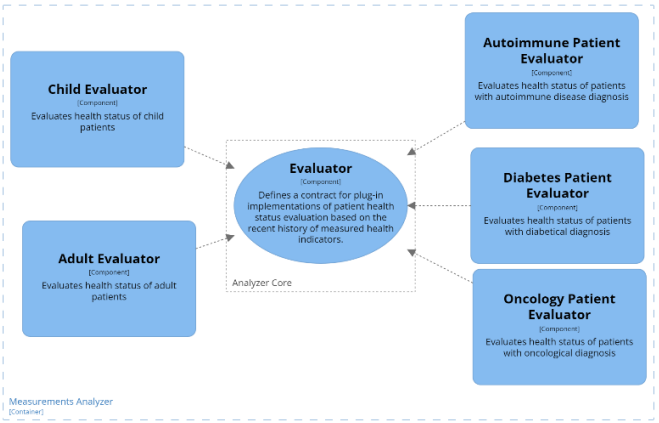
Microkernel Architecture style



- **Plug-In**

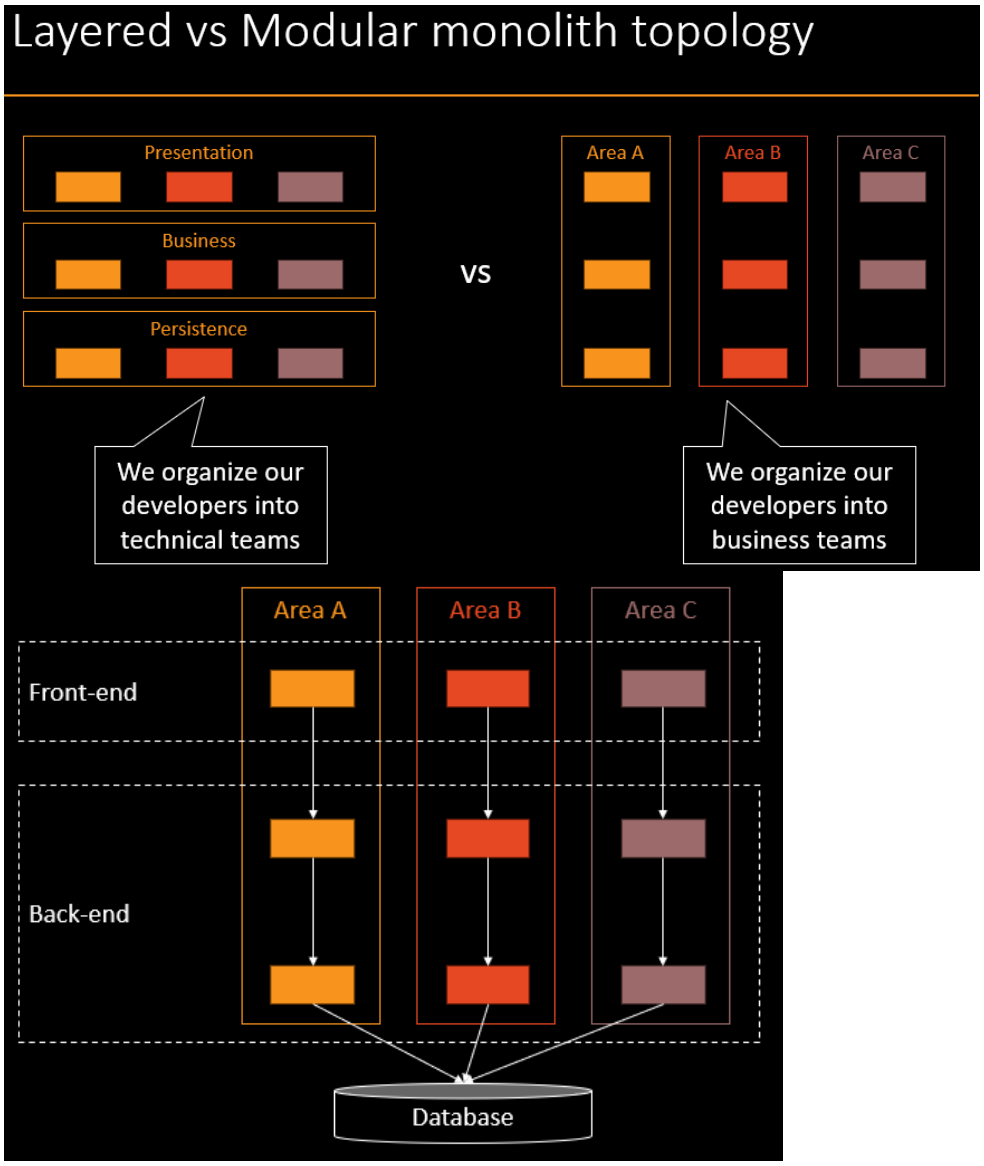
- Standalone module that contains specialized processing and additional features that enhance or extend the core system
- Independent of other plug-ins
- Called by the core system via a method or function invocation to the entry-point implementing a standard contract (interface)





- When to use?
  - IDE (Eclipse, Jira, Visual Studio Code)
  - Web browsers
  - Highly customizable business information systems

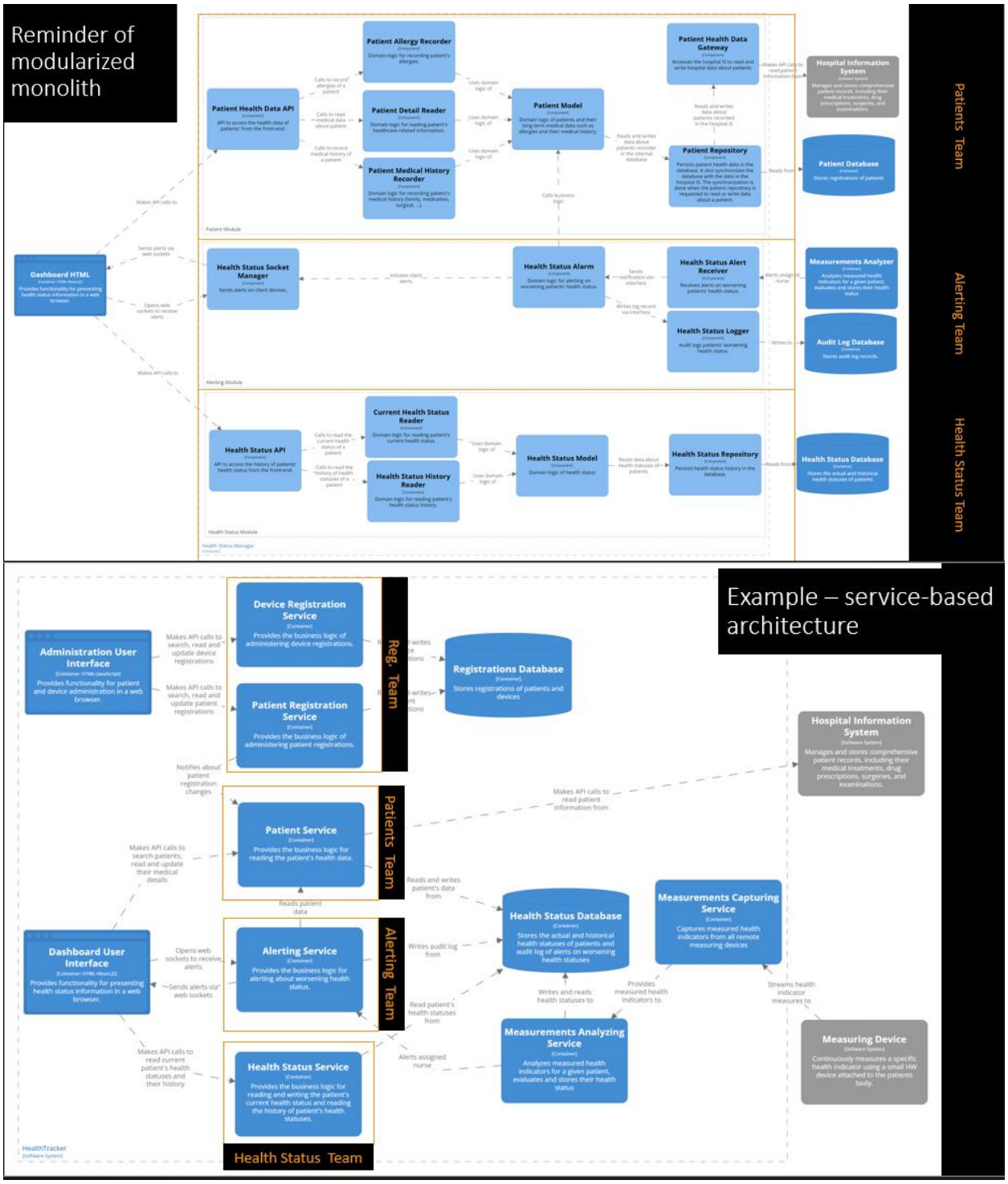
Modular Monolith Architecture Style



- In more complex business domains with independent sub-domains/areas distinguished by the business.

Service-based architecture style

- Consists of separately deployed coarse-grained services
  - representing “portions of application”
  - usually between 4 and 12
  - usually each has a single instance, but multiple instances are possible
- user interface
  - accesses services via their REST/SOAP APIs
- single database
- You can start with modular monolith and divide the modules to services later



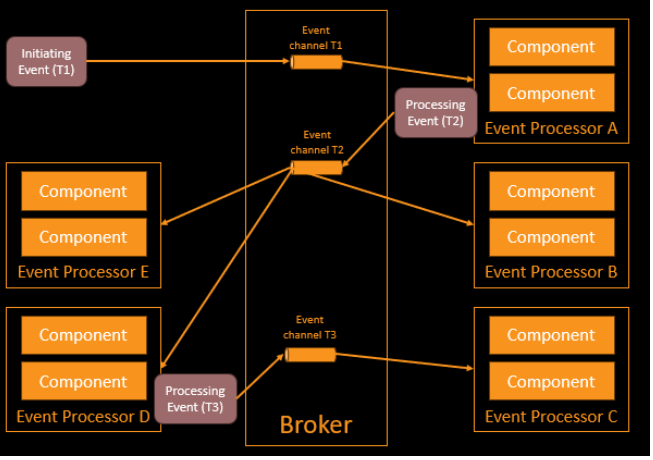
- The flexibility combined with 3-4 star ratings make it one of the most pragmatic architecture styles available.

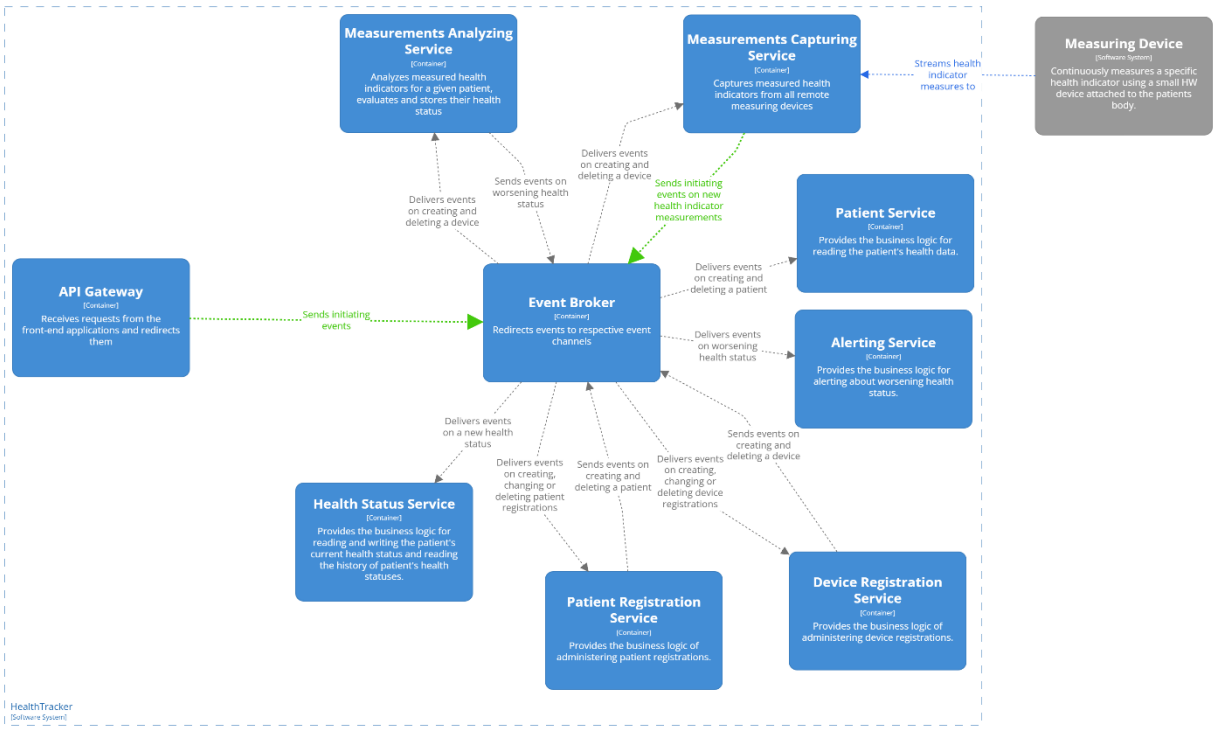
Event-driven architecture style

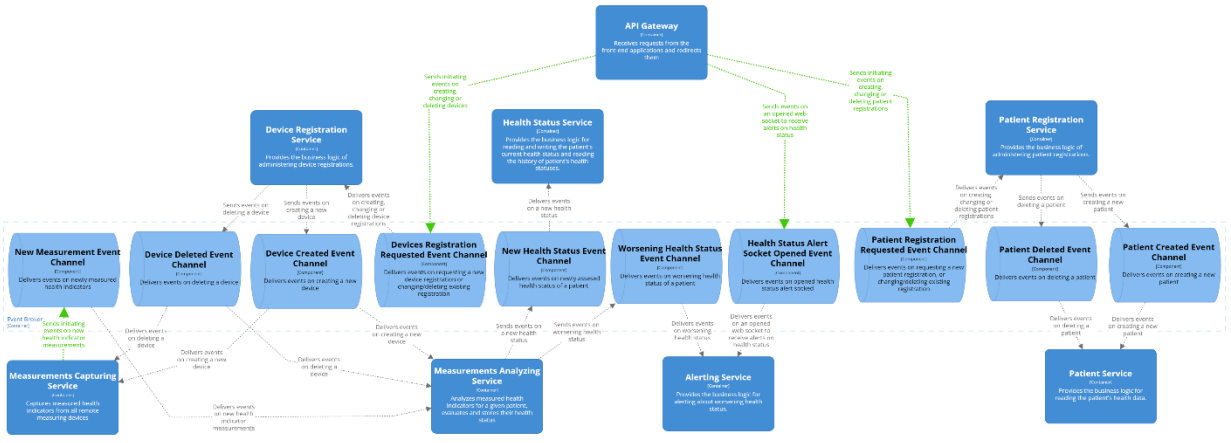
- Distributed fully asynchronous architecture style
- Decoupled event processing containers that asynchronously send, receive and process events

Broker Topology – Initiating Events

- Initial event starts the entire event flow
- Simple event activated in the user interface, e.g., “add new patient”
- Complex event activated by some real-life situation, e.g., “patient in serious condition”





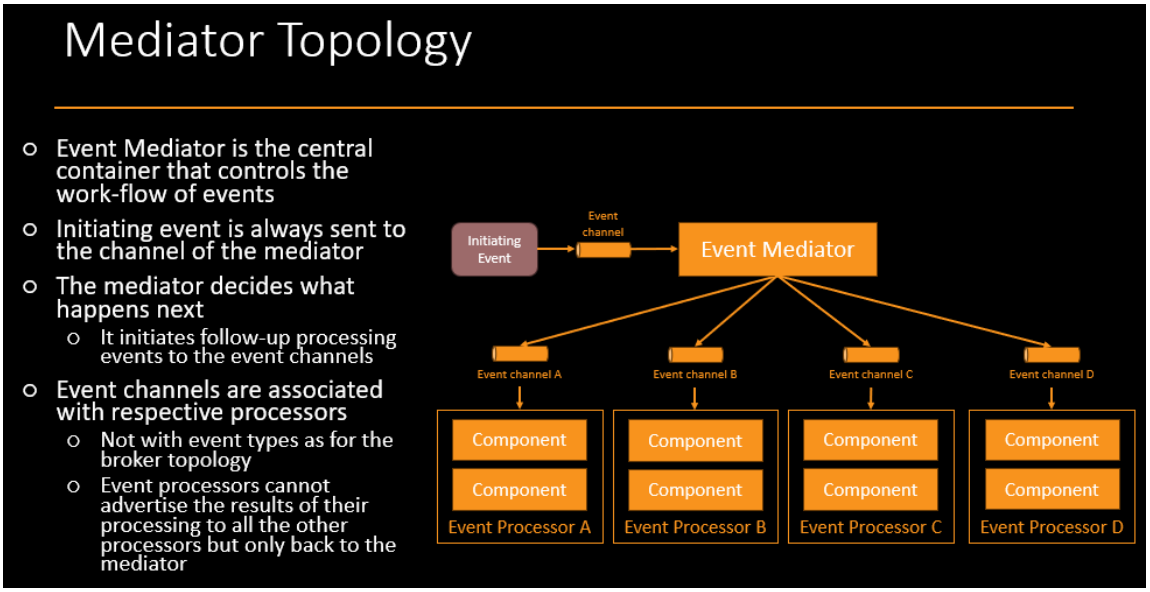


- Used to produce highly scalable and high-performance applications.
- No control over the overall workflow.
  - It is hard to know when a transaction is complete.
  - Nothing holds the complete state.
- Error handling is a big challenge.

- If failure occurs, no one is aware of that crash.
- Restarting a transaction is not supported.

When to use?

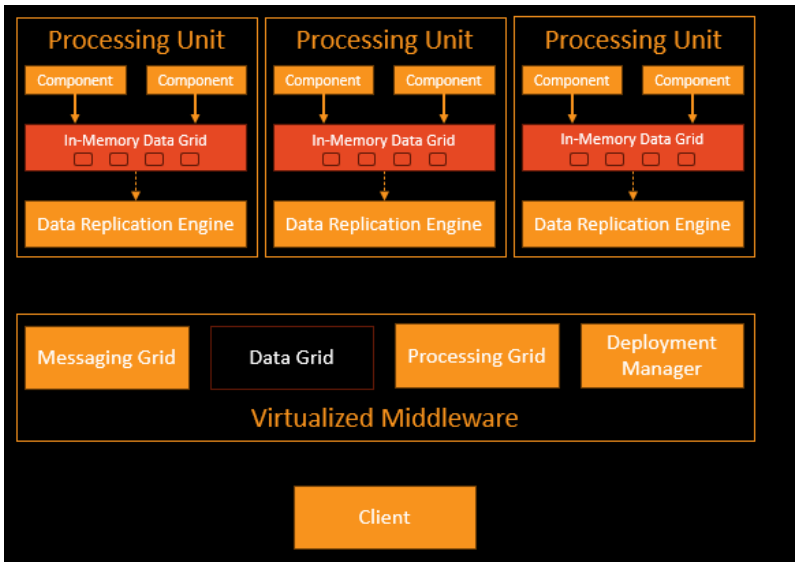
- Case 1: High performance, scalability, fault tolerance and extensibility is of high importance
- Case 2: Events are natural in each problem domain. The user requirements and implemented business processes are analyzed as events and their processing (see Event-Storming requirements analysis technique)



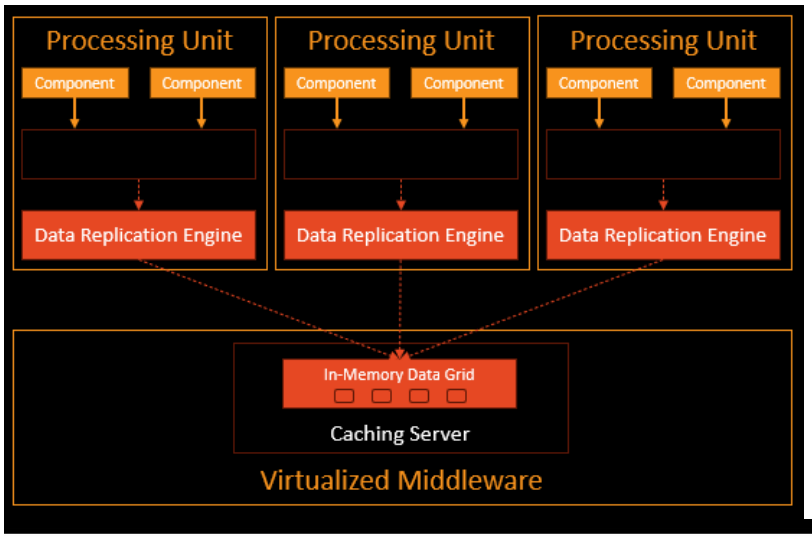
- Solves the problems of the broker topology.
  - Mediator can control the workflow.
  - Mediator can handle the state.
  - Mediator can handle errors.
- Disadvantages
  - Mediator is a single point of failure and a bottleneck.
    - Decreases reliability, performance, scalability.
  - Mediator is a single point of workflow logic.
  - Makes 1 architecture quantum!
    - The mediator has static coupling to each event processor.
- We can break 1 architecture quantum problem by splitting the mediator to more.
  - Usually, for each business area we have a separate mediator.

Space-based architecture style

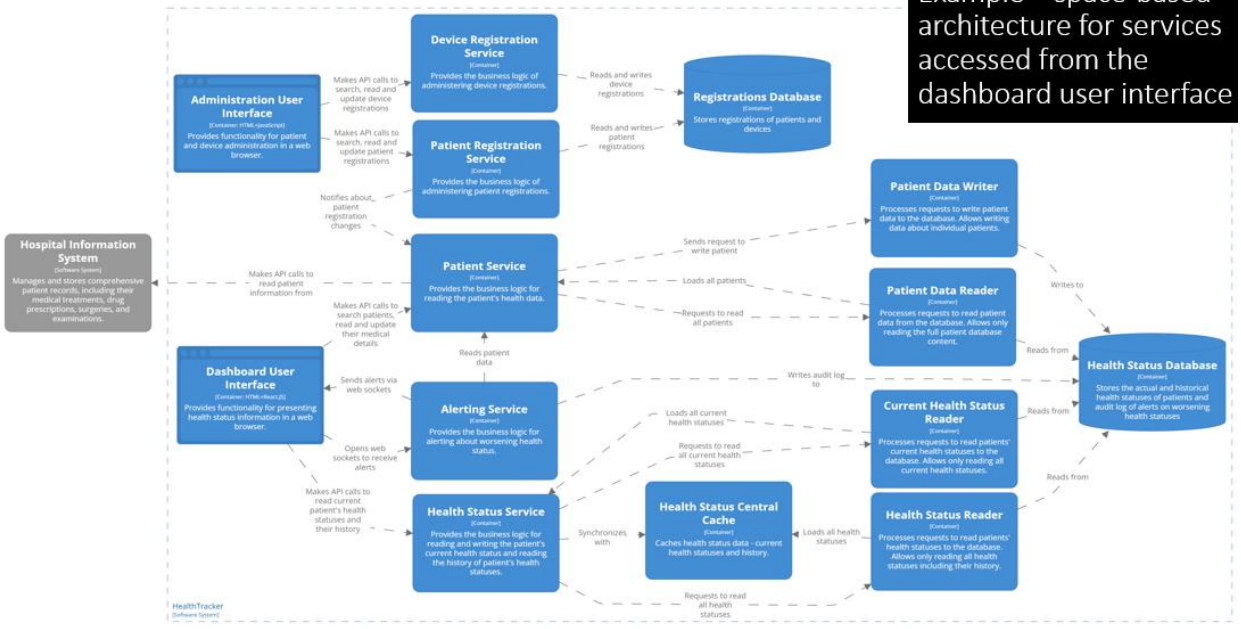
- Focuses on technical bottlenecks in the classical N-tier architectures.
  - with a growing load, bottlenecks appear.
  - you must scale out the layers.
    - scale out = more instances
    - the lower you are, the harder and more expensive to scale out.
- Specifically designed to address problems involving high scalability, elasticity, and concurrency issues.
- Useful for applications with variable and unpredictable concurrent user volumes.
- Can be better than scaling out the database server or introducing caching technologies into a non-scalable N-tier architecture.
- Well suited for applications that experience high spikes in user or request volume
  - e.g., concert ticketing system or online auction system

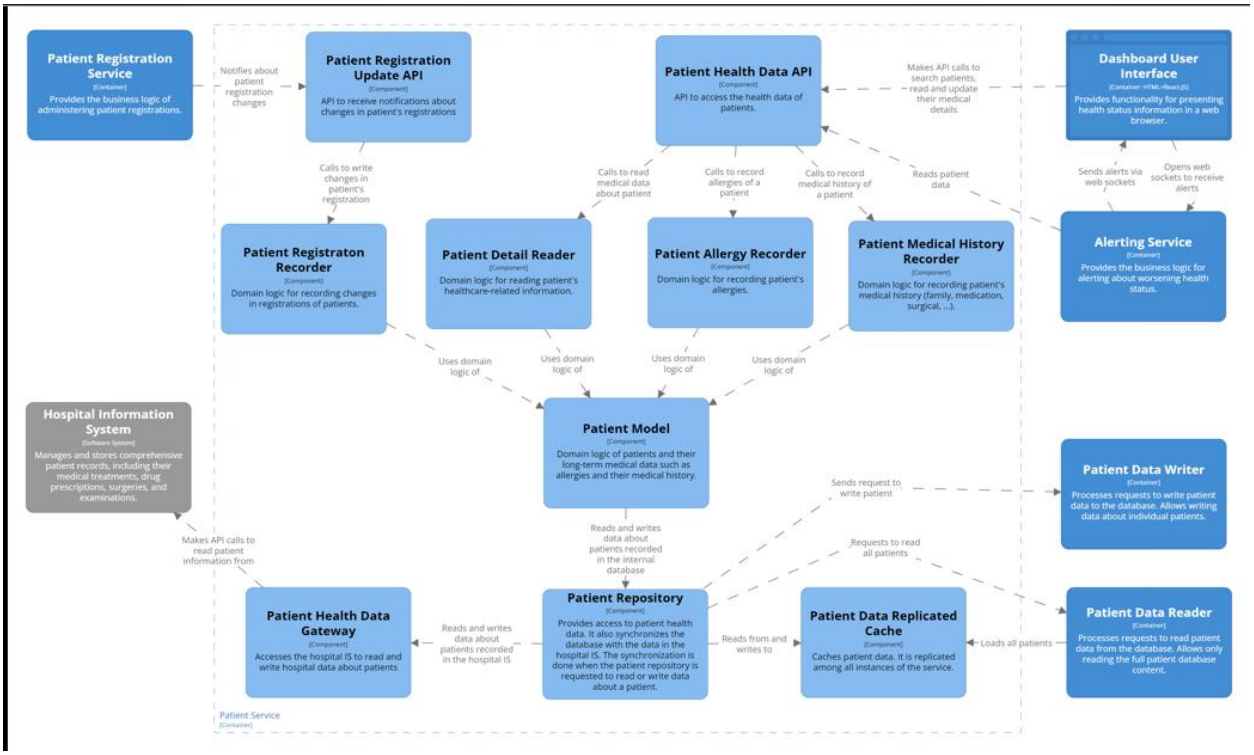


- Or with near cache



Example – space-based architecture for services accessed from the dashboard user interface

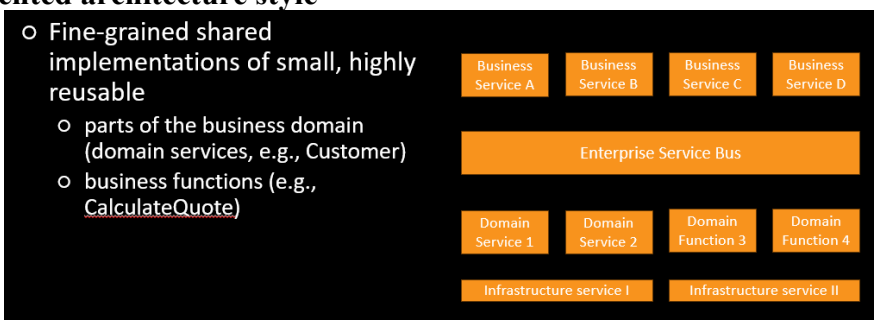




- Space based component diagram, uses replicated cache, data readers and data writers

Service-oriented architecture style

Nepoužívat



Service-oriented architecture:

- A **service** provides some functionality required by the business.
- This architecture revolves around building software by composing and reusing services.
- There are 8 key principles.
- **Standardization:**
  - We need standards for defining service contracts.
  - For example standardized naming conventions and data formats.
- **Loose coupling**.
- **Abstraction:**
  - Only the absolutely necessary information about a service is provided to the consumers, nothing more.
- **Reusability:**
  - At design time, we keep in mind that the service must be and will be reused by other services or systems in the future.

- **Autonomy:**
    - Change to other services does not affect our service and vice versa.
- **Statelessness:**
    - Services don't maintain any states.
- **Discoverability:**
    - Services can be located through a service repository.
- **Composability:**
    - When designing and building a new service, we first try to compose it from other services.

Microservices pattern:

- **Microservices** are similar thing as services defined above, with some additional details.
- Microservices must be **autonomous**, therefore they must have their own database.
- Microservices are usually smaller than services but does not have to be.
- Representation of a bounded context
- Each microservice has its own database so there could be a problem with data consistency.
    - To solve this, we use **sagas** (which were mentioned earlier in I think availability quality requirement).
    - A saga means each action on a mircroservice has an associated compensation transaction. When a fault occurs, the compensation transaction is executed.