

---

# **Software Architectures**

**definition, design, qualitative attributes, tuning**

**© R. Ošlejšek and B. Bühnová**  
**Faculty of Informatics**  
**Masaryk University**  
**oslejsek@fi.muni.cz**

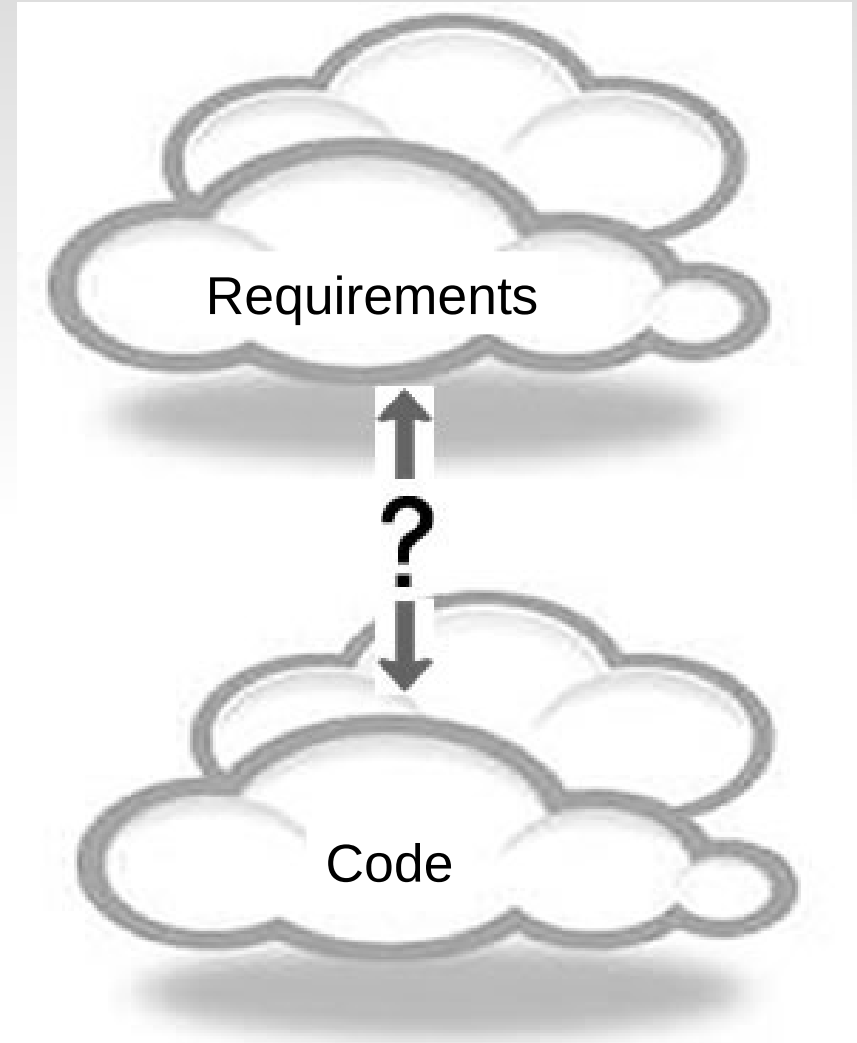
# Motivation

---

Why software architectures?

**How to fill the gap between requirements and code?**

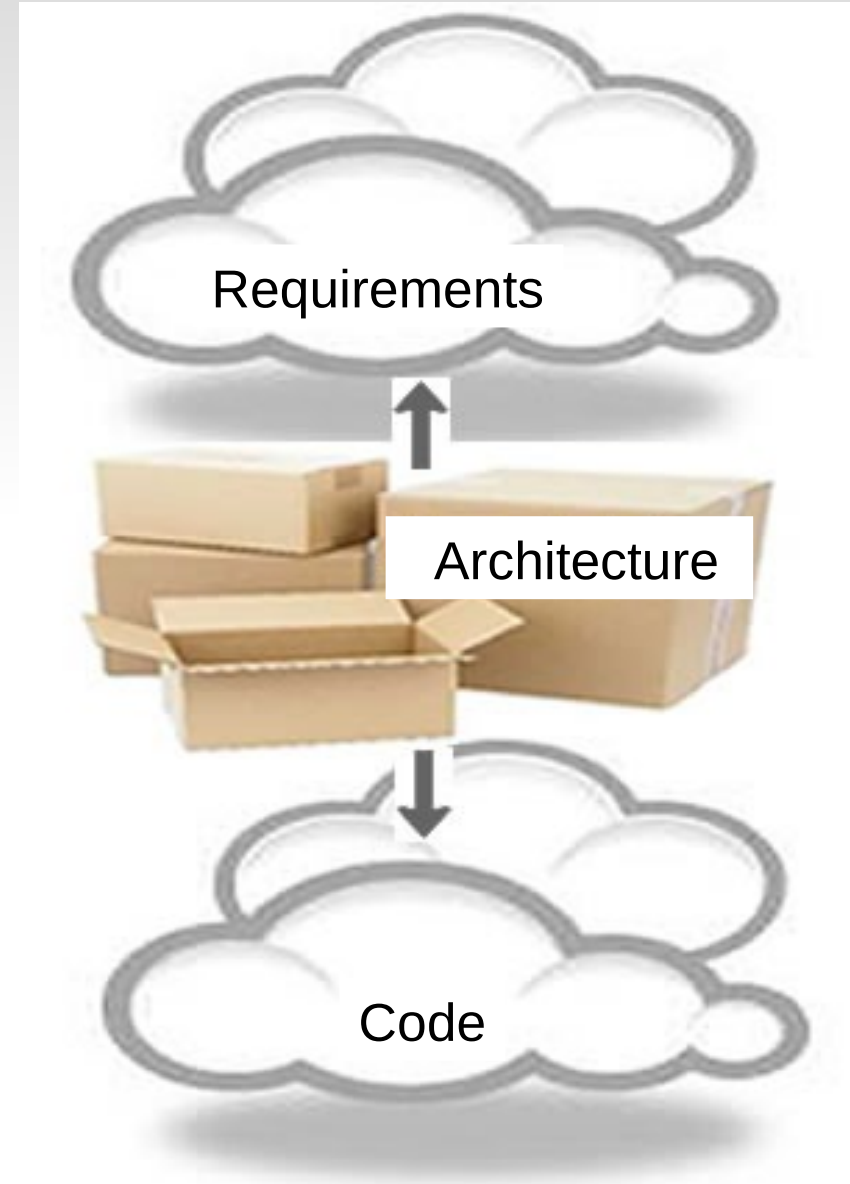
- How to reduce the risk of the code deflection from user requirements?
- How to reduce the risk of unsuitable structured code?
- How to reduce the risk of making a code which is hardly comprehensible or modifiable?



# Motivace – Řešení

## The role of software architecture

- Coarse-grained structure of the system
  - Well-defined, distributable
- Abstraction on the system level
  - Dividing the functionality to responsibilities distributed among modules/components of the system



# Lecture outline

---

- Definition of software architecture
- The role of software architect
- Software architecture design
  - Requirements
  - Architecture design
    - The process of development
    - Techniques of qualitative design
  - Evaluation of SW architectures
    - Qualitative attributes of SA
    - Tuning tactics
- Overview of relevant topics

# Definition of software architecture

---

To date there is still no agreement on the precise definition of software architecture.

Architecture of a software system is composed of a set of **fundamental design decisions** about the system. [Taylor et al. 2009]

Software architecture of deployed software is defined by **aspects that are hardly to change**. [Klusener et al. 2005]

Software architecture of program or computational system is a system structure which is composed of **software elements**, externally **visible features of these elements** and mutual **relationships between them**. [Bass et al. 2003]

Architecture is a basic structure of a system incorporated into **components** and their **mutual and external relationships**, and a set of principles covering the design and development. [ANSI/IEEE Standard 1471/2000]

# Fundamental parts of software architecture

---

## Three basic elements:

- 1. Modules** = “components” comprising the functionality of the system (often run on parallel)
  - encapsulated parts of software, packages, processes, threads, components, etc.
- 2. Connectors** = communication links and channels (often with its own inner logic)
  - procedure and method calls, channels for message distribution (publish-subscribe style), etc.
- 3. Deployment** = mapping of modules and connectors to hardware (or software) sources
  - physical resources and servers (with parameters like CPU frequency, HD size, communication speed, etc.).
  - operating systems or application servers.

# Advantages of well-defined system architecture

---

## **Mutual communication**

- Unified view on the system from the perspective of various roles participating on the development, including the roles on the customer side.

## **System analysis**

- Prediction of qualitative attributes of the architecture

## **Re-usability**

- Single architecture proposed to fulfill concrete non-functional requirements can become a basic design structure for many systems.
- Integration of external modules
- Design of a module usable in wide range of systems

## **Project planning**

- Price estimation, scheduling milestones in the development process, dependency analysis

# Roles of software architect

---

## Software designer

- Must be able to recognize, re-use or find new effective design approaches and apply them.

## Domain expert

- Must have detail knowledge and understanding of the application domain, its significant features and strangenesses.

## Technical engineer

- Must know technical aspects of proposed solution.

## Expert on SW/HW standards

- Must be well-informed about relevant standards in order to estimate and communicate their benefits and impacts.

## Economist of software engineering

- Must balance the project and the its development process so that the required features are implemented effectively.



# Software architecture design

---

Three main activities of architectural design:

- **Requirements specification**
  - Functional and extra-functional
- **Architecture design**
  - Basic design decisions
  - Architectural models
  - Development process
  - Techniques of high-quality design
- **Evaluation of SW architecture**
  - Qualitative attributes of SA
  - Methods of quality evaluation
  - Tuning tactics

# Software architecture design

---

Three main activities of architectural design:

- **Requirements specification**
  - Functional and extra-functional
- **Architecture design**
  - Basic design decisions
  - Architectural models
  - Development process
  - Techniques of high-quality design
- **Evaluation of SW architecture**
  - Qualitative attributes of SA
  - Methods of quality evaluation
  - Tuning tactics

# Requirements on SW architecture

---

**Functional requirements** = restrictions put on the functionality of the system. It concerns internal functionality of components as well as the inter-component collaboration:

- The `serviceA()` service requires at most 10 cooperating components.
- Every open transaction (a service provided by another component) should be closed before the the component opens a new transaction.
- Component must not be blocked during the evaluation of their services (when waiting to the result of external components).

**Non-functional (extra-functional) requirements** = restrictions on the implementation and behavior of the functionality.

- Guaranteed response time in X% of calls.
- Availability X% every month.
- HW/SW compatibility.

# Software architecture design

---

- **Requirements specification**
  - Functional and extra-functional
- **Architecture design**
  - Basic design decisions
  - Architectural models
  - Development process
  - Techniques of high-quality design
- **Evaluation of SW architecture**
  - Qualitative attributes of SA
  - Methods of quality evaluation
  - Tuning tactics

# Fundamental design questions

---

Typical questions asked by software architect:

- Is there some **architecture prescript** that should be taken into account for the system?
- Which kind of **structuring (decomposition) process** will be used?
- How to **decompose the system** into sub-systems (modules, components)?
- What can be **re-used from previous projects**?
- What would be **re-usable in future projects**?
- Which components can or cannot be **bought**?
- Which **architectural styles** best suits the system?
- Which kind of **distribution** is possible and suitable?
- How to communicate with **existing software**?
- How to access **existing data**?

# Architectural models

---

They define

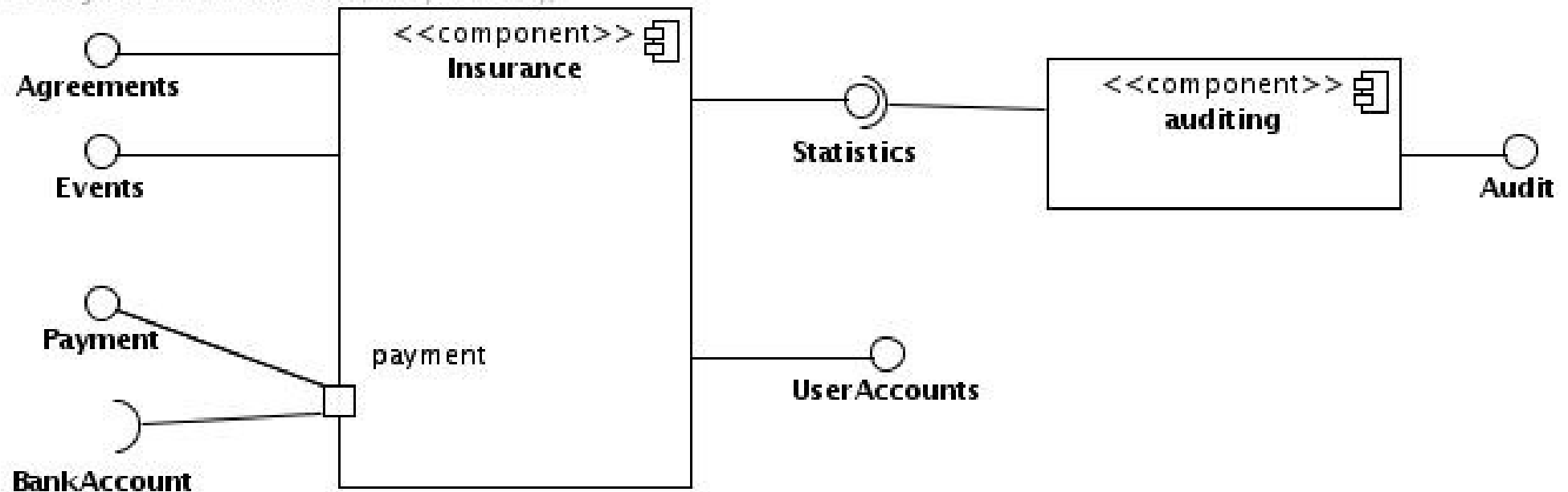
- Modules – system components
- Connectors – communication styles
- Deployment – mapping onto HW/SW sources

# Architectural models – modules

## Modules = system components

- **Model of static structure** – defines internal structure of the system. Internal elements are either *composite* (composed from sub-elements) or *primitive* (include code).
- **UML** – component diagram for composite elements, class diagram for primitive elements

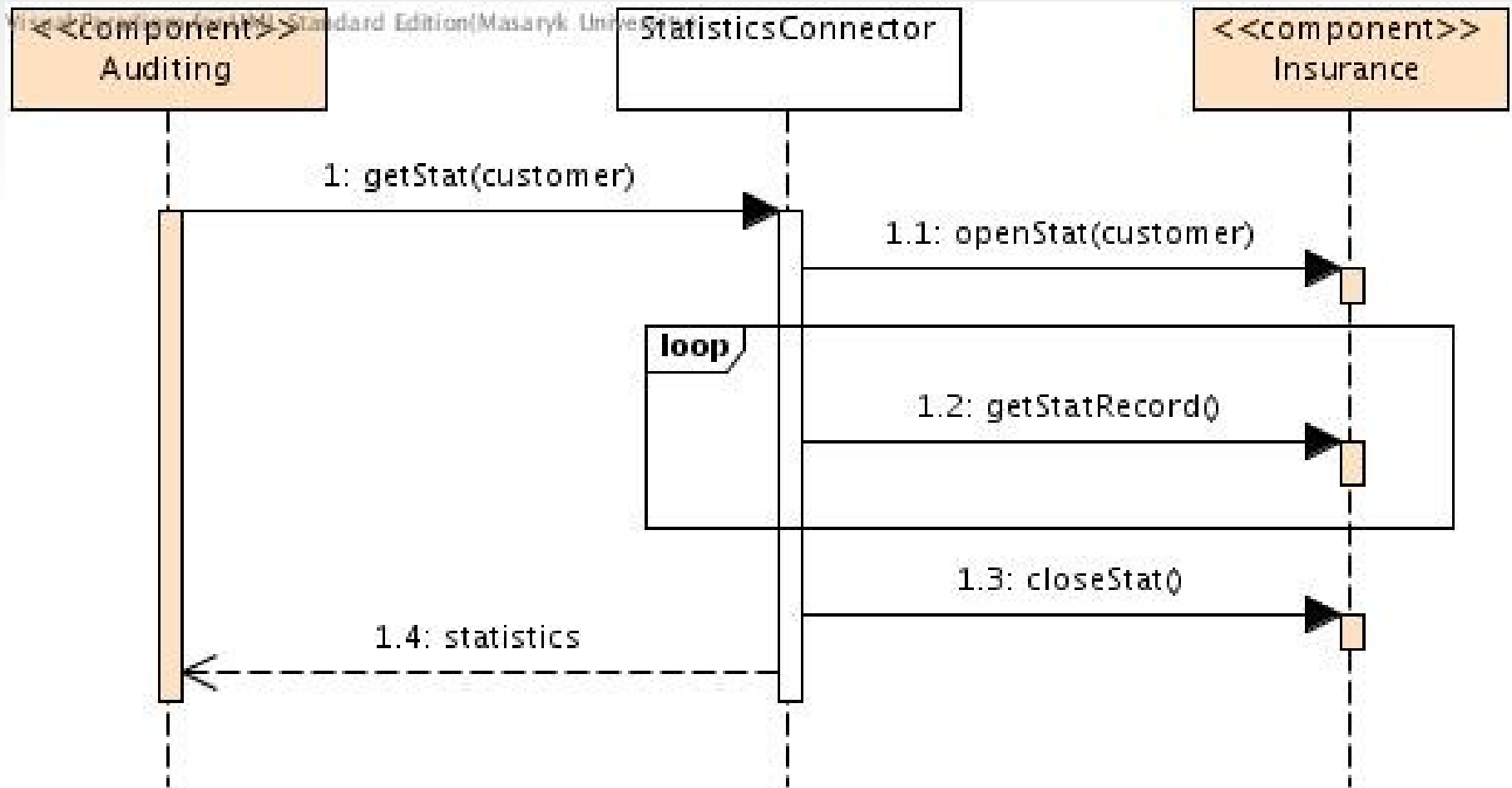
Visual Paradigm for UML Standard Edition (Masaryk University)



# Architectural models – connectors

## Connectors = communication styles

- **Procedural (dynamic) model** – defines mutual interaction between components of the system
- **UML** – sequence diagram, communication diagram, activity diagram

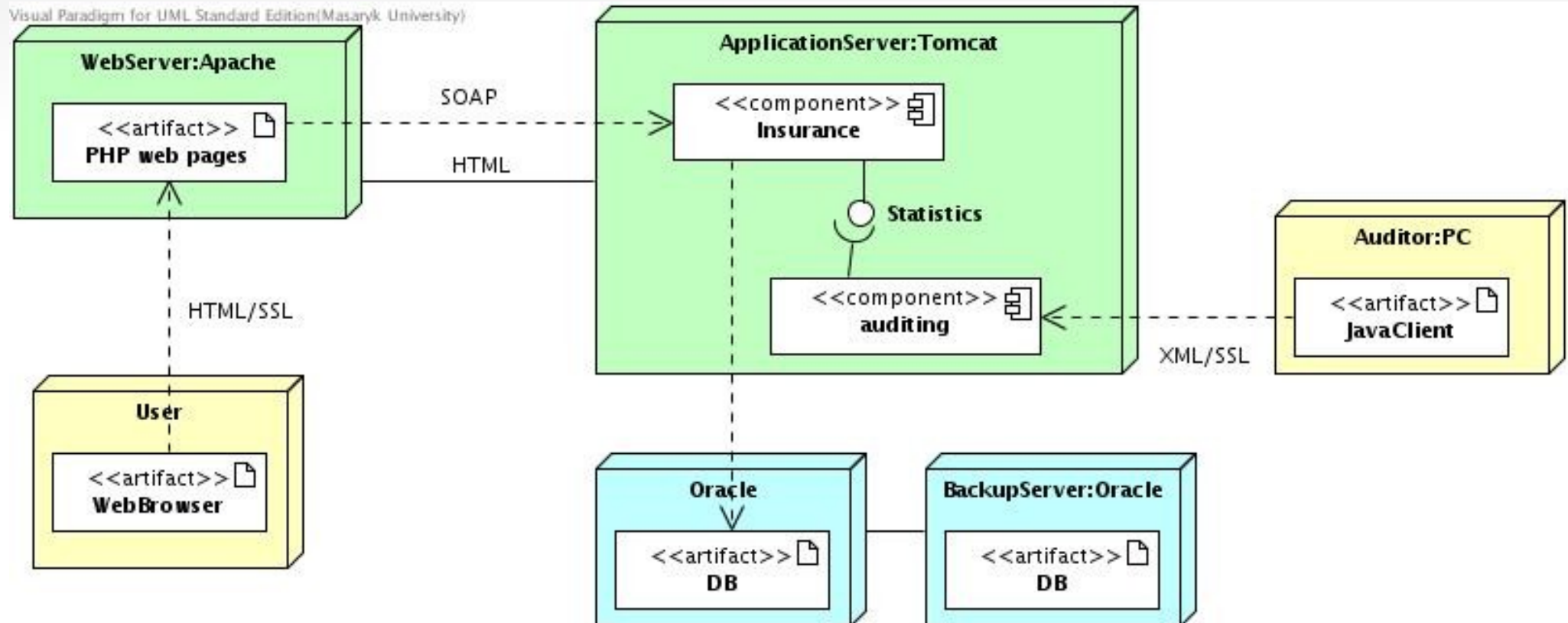




# Architectural models – deployment

## Deployment = the mapping onto HW/SW sources

- **Deployment model** – defines structure of the system (including its characteristics) and the mapping of modules and communication links onto these sources
- **UML** – deployment diagram



# Development process

---

## The main tasks of the development process:

1. Identify system **components**.
2. Identify **interfaces** of system components.
3. Design **connectors** between components (interfaces).
4. Identify which parts of the system should be **allocated** on shared nodes.
5. Evaluate quality of the proposed architecture.

# Architectural design tactics

---

Tasks 1.-3. can be performed in different order depending on selected methodology

## **Top-down approach – gradual refinement of the architecture**

- Design the whole system as a single component with defined interfaces
- Break up the component into the first-level sub-components and their connections
- Break up sub-components into lower levels, stop at sufficiently primitive components
- Implement, find or buy components corresponding with the proposed primitive components

## **Bottom-up approach – assembling the architecture from predefined components**

- Select candidate components from library
- Try to assemble components to hi-level components, use only the interfaces that are necessary for required functionality
- Proceed step by step until the top-level component (= the system)
- Remove unused components (those that was not finally used)

# Techniques of a hi-quality design

---

## **Architectural patterns**

- Techniques of the architecture design proven by practice
- Re-usable for various systems

## **Quality metrics**

- Quantitative evaluation of the quality of a design based on objective criteria (number of inter-module links, module size, etc.)

## **Correctness by construction**

- Design techniques composed of identifiable steps. Each step guarantees a quality of the result.

# Architectural Patterns

---

**Architectural pattern** is named collection of architectural design decisions that are applicable to design problems appearing over and over again, which are parametrized under various contexts of software development. [Taylor et al.]

# Software architecture design

---

- **Requirements specification**
  - Functional and extra-functional
- **Architecture design**
  - Basic design decisions
  - Architectural models
  - Development process
  - Techniques of high-quality design
- **Evaluation of SW architecture**
  - Qualitative attributes of SA
  - Methods of quality evaluation
  - Tuning tactics

# Qualitative attributes of SA

---

Related mainly to non-functional (extra-functional) requirements:

**Non-functional (extra-functional) requirement** of software system defines restrictions on the implementation.

**Extra-functional attribute** is concrete qualitative aspect related to extra-functional requirement.

Examples of basic extra-functional qualitative attributes:

- **Performance** – throughput, response time, efficiency of utilization of various sources
- **Reliability** – faultless running, availability, robustness, ability to recover
- **Security** – confidentiality, integrity, availability
- **Scalability** – load, concurrent communication, data scaling
- **Maintainability** – modifiability, adaptability

# Methods for quality evaluation

---

## **Monitoring and testing = verification of the quality of existing system**

- Cheap and popular method
- Applicable on implemented system
- Imprecise results (e.g. depend on the number of testing cycles)

## **Prediction from model = quality estimation of the system under development**

- Requires (simplified) model of the system, which is created and elaborated during the design time.
- Usable during the whole process of the architectural design
- Quality of results strongly depends on the model and its attention to detail

## **Formal verification = verification of assumptions formulated about the system**

- The most expensive but most accurate method
- Evaluation is performed on the model which is well-elaborated for the verification purposes (can be partially generated from code or from design models)
- To guarantee precision of results, it is necessary to make big effort to fine tune the model (and its levels of detail)



# Tuning tactics

---

**Tuning of software architecture** is the optimization of selected qualitative attribute by adjusting the architecture.

## **Advantages:**

- Practice-proven techniques used to increase the quality of the system from the point of view of selected qualitative attribute

## **Attention:**

- Optimization is not guaranteed. Often only a minor optimization is achieved.
- There is the danger of making another attributes worse.

**It is necessary to apply tuning tactics in common sense and to understand mutual connections between attributes as well as the impact of tactics to them.**

# Tactics – Performance

---

**Performance** reflects the ability of a software system to fulfill the requirements for fast response and high throughput of the system together with the minimization of computational resources.

## Minimize the number of adapters and wrappers by adjusting interfaces

- **How:** Cleaning up interfaces, changes of signatures of provided services
- **Effect:** The reduction of resources that handle a single service call

## Simplify communication handled by interface

- **How:** Offer more interfaces for the same functionality
- **Why:** Different interfaces can be designated for different runtime context (e.g. clients of different platforms using different data formats). In their specific context they could operate more efficiently.

# Tactics – Performance

---

## Separate data from the computation

- **Why:** Data can be better optimized without the need to make changes in software (computational) components and, vice versa, computational algorithms can be optimized without the changes in data.

## Revise the utilization of broadcast connectors

- **Why:** Message broadcasting increases reliability of message delivery but also increases the load of the system. Therefore we should omit its needless usage.

## Replace synchronous communication with asynchronous whenever possible

- **Why:** With synchronized communication, the slowest component slows down all other components involved in the sequence of the call.

## Often mutually communicating components should be allocated close to each other

- **Why:** To minimize network communication which is slower than in-memory communication

# Tactics – Reliability

---

**Reliability** of a software system is probability that the system will provide expected functionality (with respect to design restrictions) without errors and failures for the given period of time.

## Carefully check external dependencies of components

- **Why:** It's necessary to ensure that wrong behavior of single component has only minimal impact on the faultlessness of other components.

## Allow selected components to expose their state, and define state variants

- **Why:** If it is not possible to guarantee the reliability of a component, then access to its internal state enables its clients to check and evaluate the “state of health” at runtime by means of predefined state invariants.

## Employ suitable error reporting mechanisms

- **Why:** If component fails, it should be able to inform the rest of the system about the reason, e.g. via exceptions.

# Tactics – Reliability

---

## **Check reliability of components in their connectors**

- **Why:** Reduces the probability of the failure propagation beyond the component border

## **Avoid the existence of critical parts (single points of failure)**

- **How:** Components replication, decomposition of a component to multiple components according to their responsibilities, strengthening control abilities of connectors located around these components.
- **Why:** Failure occurred in such a critical part will highly probably paralyze the whole system.

## **Integrate auto-backup and recovery mechanisms of critical functionality and data into the system**

- **Why:** Reducing the risk of reliability breach due to data loss or disruption of functionality.

## **Integrate “health state” monitoring to the system**

- **Why:** Possibility of fast reaction.

# Tactics – Scalability

---

**Scalability** is the ability of software system to adapt itself to new requirements related to the system size and scope.

## **Make components sufficiently integral, with clear purpose and easy-to-understand interface**

- **Why:** Adding new components or their replication will have minimal impact on the other parts of the system.

## **Distribute data sources**

- **Why:** To avoid common bottleneck which often erases during the system expansion (many components accessing a single data source).

## **Identify data suitable for replication**

- **Why:** The possibility to serve more clients accessing the data concurrently.

# Tactics – Scalability

---

## **Ensure sufficient integrity and well-defined purpose to every connector**

- **Why:** The same reason as in the case of components

## **Consider replacing direct dependencies with indirect**

- **Why:** Direct dependencies (through associations, i.e. synchronous calls) are not convenient during the system expansion because they require multiplication of the relationships. Less restrictive connection by means of indirect dependencies (through sending messages via broadcasting, for instance) is much more suitable in this case.

## **Eliminate bottlenecks of the system (components and connectors used by more clients concurrently)**

- **Why:** Avoiding the slowing down the system when number of such clients is increased.

## **Use parallel processing on suitable parts of the system**

- **Why:** Acceleration of expensive calculations required by increasing number of clients.

# Tactics – Maintainability

---

**Maintainability** represents difficulty to change a software system in response to new requirements, changes in environment or debugging.

## **Split different responsibilities to different components / merge the same responsibilities to the same components**

- **Why:** Faster localization of parts requiring some changes.

## **Remove operations related to interaction from components, keep only operations related to functionality**

- **Why:** Interaction logic should be located in connectors.

## **Keep component small and compact**

- **Why:** You can easily modify small functionality of the system by replacing selected component.

## **Isolate data from computation**

- **Why:** It increases the probability that changes in data will not require changes in computation and vice versa.



# Tactics – Maintainability

---

## **Separate different communication principles from different connectors**

- **Why:** Easy allocation of part requiring a change, including components which can be affected by this change.

## **Remove operations related to functionality from connectors, keep only operations related to interaction**

- **Why:** Functionality should be located in components.

## **Eliminate unnecessary dependences**

- **Why:** Plenty of dependencies decreases understandability of the system.

## **Make the architecture hierarchical**

- **Why:** Possibility of multiple views on the system on different levels of abstraction.

# Stručný přehled navazujících témat

---

- **Komponentové softwarové inženýrství**
  - Vývoj systémů z COTS (Components of The Shelf)
- **Servisně-orientované architektury (SOA)**
  - Vývoj systémů integrací autonomních služeb
- **Aspektově orientované architektury**
  - Integrace průřezových koncernů (crosscutting concerns)
- **Softwarové produktové řady**
  - Rodiny produktů s jednotným jádrem a variačními body
- **Dynamické a adaptivní architektury**
  - Architektury schopné adaptovat se na run-time změny prostředí
- **Model-driven architektury (MDA)**
  - M2M transformace a zjemňování modelů

# Shrnutí

---

- Definice softwarové architektury
- Role softwarového architekta
- Návrh softwarové architektury
  - Specifikace požadavků
    - Funkční a extra-funkční
  - Návrh architektury
    - Základní návrhové otázky
    - Architektonické modely
    - Vývojový proces
    - Techniky kvalitního návrhu
  - Ohodnocení SW architektury
    - Kvalitativní atributy SA
    - Metody hodnocení kvality
    - Taktiky ladění SA
- Stručný přehled navazujících témat

# Questions?

---

