Chapter 3: Software Architecture

Artifacts and Quality Attributes

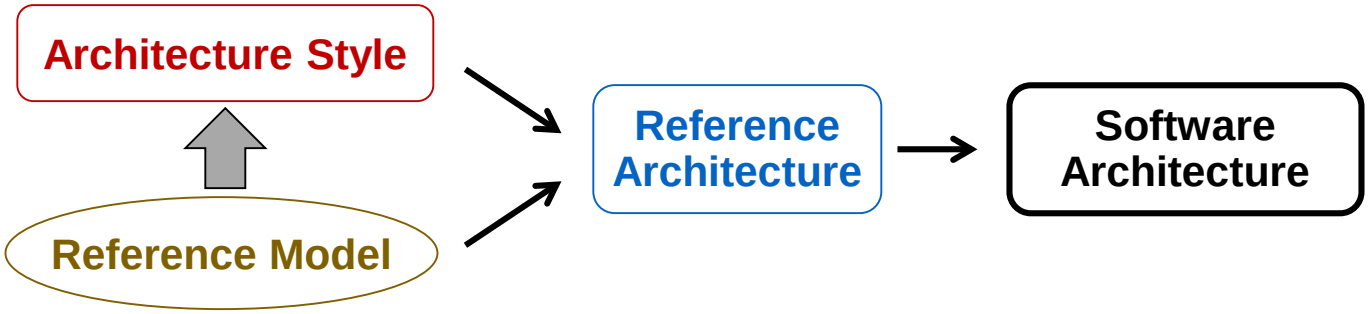
H.S. Sarjoughian

CSE 460: Software Analysis and Design

School of Computing, Informatics and Decision Systems Engineering
Fulton Schools of Engineering

Arizona State University, Tempe, AZ, USA

Specifying Software Architecture



- ***Architecture Style (aka Pattern)***
  - is a description of component types and a pattern of their runtime control and/or data transfer
  - a style places general constraints on components and the kinds of interactions among them – a family of architectures such as Client/Server and Data-Centered
  - architecture style does not deal with "standard" domains such as the collaboratory environments (see "Ch2 ArchBusCycle.pptx")

Key Artifacts for Specifying Software Architecture

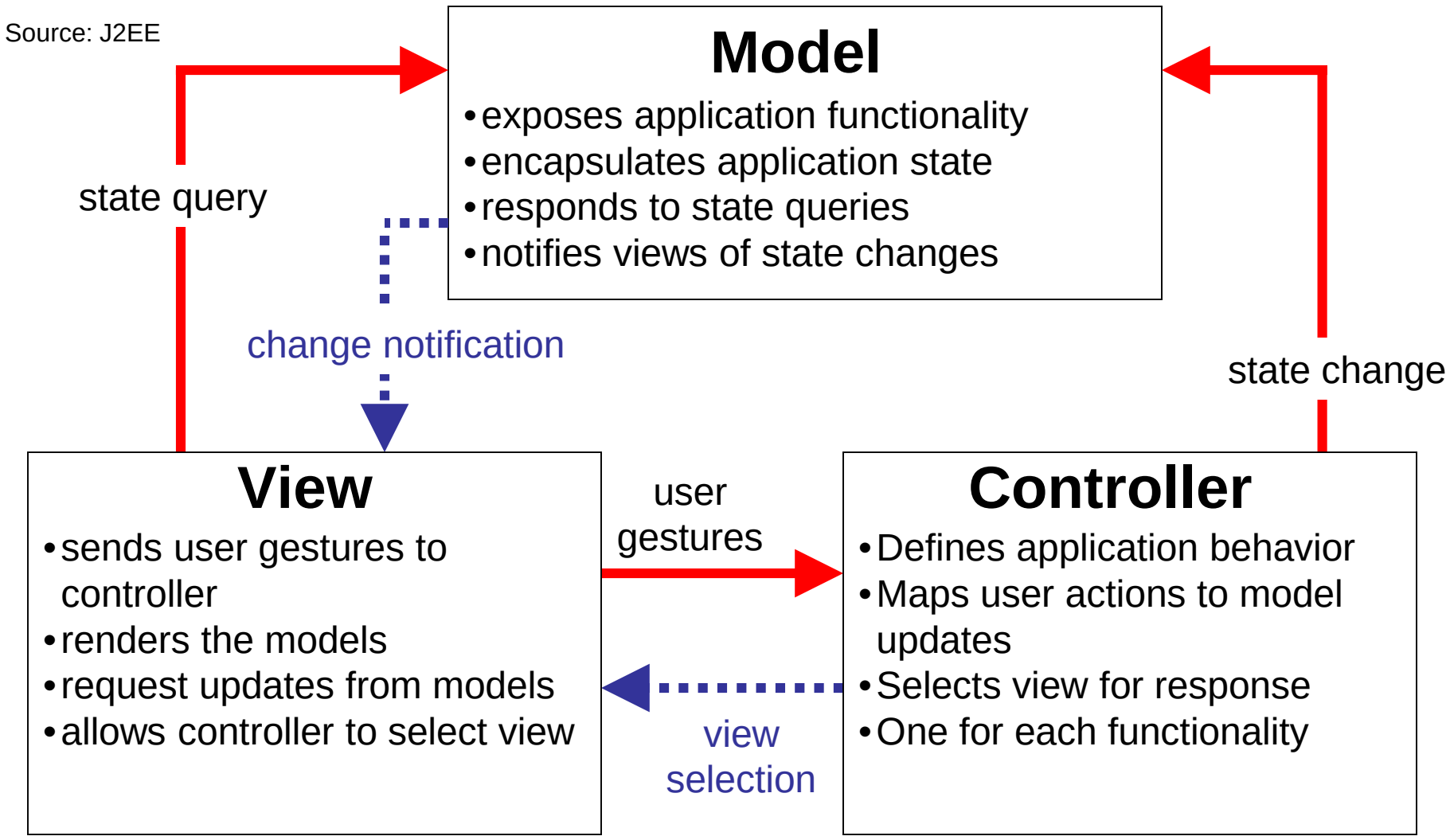
- **Reference Model**
  - represents a division of functionality together with data flow between the parts
  - is a standard decomposition of a known problem (domain) into parts that cooperatively provide a solution to the problem – e.g., Model-View-Controller
- **Reference Architecture**
  - is a reference model mapped onto software components along with the data flows between the components – software components cooperatively implement the functionality defined in the reference model
  - mapping of reference model to reference architecture is not necessarily one-to-one

Architecture Style Template (Documentation)

Architectural style establishes how the following things relate to one another

- **Context** – A recurring, common situation experienced in real world.
- **Problem** – A generalized problem that is seen in the context. The problem and its major variants are outlined. Major parts of the problem (complementary and opposing) aspects are described. Required quality attributes are included.
- **Solution** – A successful architectural resolution to the problem. The solution provides appropriately abstracted structures.
  - Element types
  - Interaction mechanisms or connectors
  - Topological layout of the components
  - Semantics constraints
  - Weaknesses

Model-View-Controller



Model-View-Controller (cont.)

TABLE 13.3 Model-View-Controller Pattern Solution

Overview	The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
Elements	The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic. The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both. The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.
Relations	The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	There must be at least one instance each of model, view, and controller. The model component should not interact directly with the controller.
Weaknesses	The complexity may not be worth it for simple user interfaces. The model, view, and controller abstractions may not be good fits for some user interface toolkits.

Source: Software Architecture in Practice, 3rd Ed.

Architecture Style Template (Documentation)

Architecture style establishes how the following things relate to one another

- **Context** – A recurring, common situation experienced in real world.
- **Problem** – A generalized problem that is seen in the context. The problem and its major variants are outlined. Major parts of the problem (complementary and opposing) aspects are described. Required quality attributes are included.
- **Solution** – A successful architectural resolution to the problem. The solution provides appropriately abstracted structures.
  - Element types
  - Interaction mechanisms or connectors
  - Topological layout of the components
  - Semantics constraints
  - Weaknesses

Architecture Style Solution Template

Architecture style (also referred to as system pattern) specifies a family of architectures. An architecture style as a solution consists of:

- **Overview –** provides a description for the pattern
- **Elements –** components (e.g., databases, processes, …) performing some form of computation (functions)
- **Relations** – interaction mechanisms such as procedure call, events, message bus, …
- **Semantic constraints** – specifies important and necessary constraints covering topology, element behavior, and interaction mechanisms (e.g., use of a particular communication protocol or data storage medium)
- **Weaknesses –** highlights structural and behavioral limitations

Horizontal Structural Partitioning

Defines separate branches of the modular hierarchy for each major program function (e.g., input/output activities and kernel data processing)

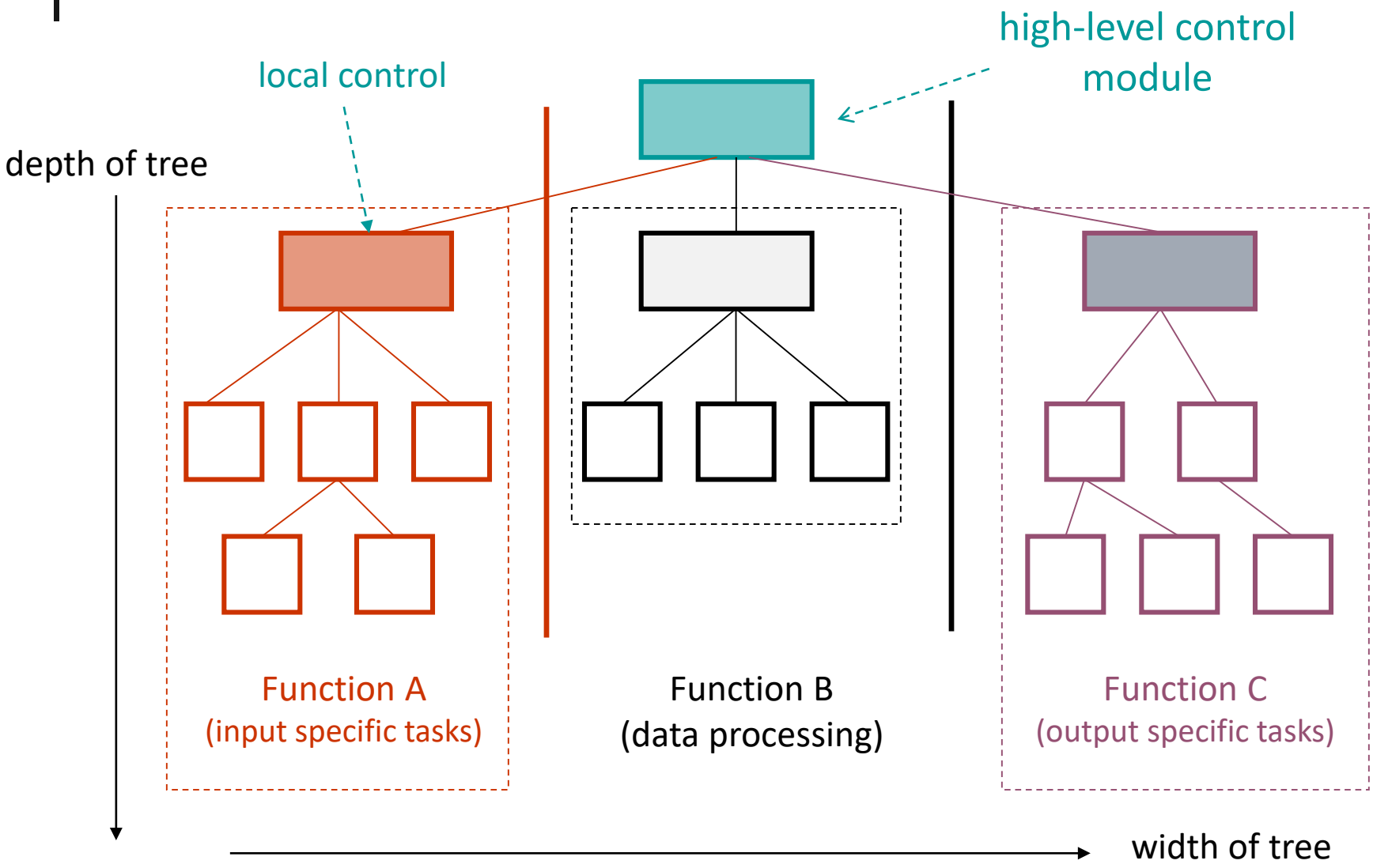
Advantages

- easier to test and maintain each overall function
- propagation of fewer side effects from one major function to another
- easier to extend

Disadvantages

- Control across the entire software (overall control) can be complex (e.g., control of a module placed in bottom of hierarchy in one horizontal partition by a decision-making module belonging to another horizontal partition).
- More data movement may be necessary

Architecture Horizontal Partitioning



Vertical Structural Partitioning

Focuses on allocating decision-making (control) to top of the hierarchy with I/O and data processing assigned toward the bottom of the hierarchy

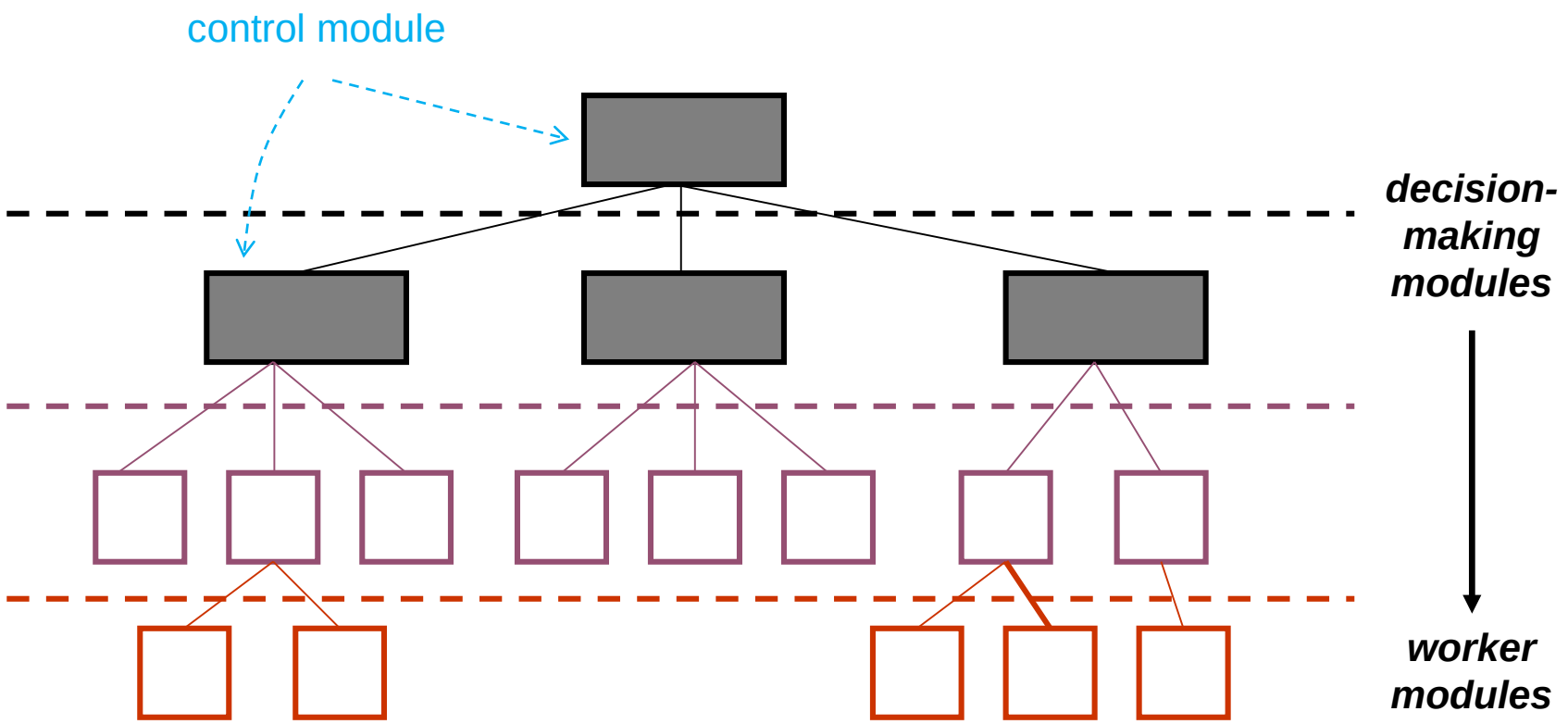
Advantages

- less degree of side effect for “worker” modules

Disadvantages

- higher degree of side effect for decision-making modules and modules further down the hierarchy

Architecture Vertical Partitioning



⇒ middle layer modules are responsible for partial control and processing
⇒ good software architecture structure usually is neither too wide nor too deep!

Architecture Styles: Vertical & Horizontal Partitioning

- Vertical and horizontal partitioning are two distinct architecture choices – the selection of one vs. another within a program architecture should be evaluated and justified

- Normal and expected behavior of the software (as specified in architecture analysis requirements) can provide important insights for partitioning choices. E.g., nature of change might be primarily “control oriented” from a high-level point of view with little change expected on “worker” modules

Quality Attributes

- Establishing **quality attributes** (e.g., system) is an essential step toward specifying a **software/system architecture** – an architecture can facilitate or impede many necessary and desired quality attributes (e.g., extensibility)
- System quality attributes provide a basis to **measure** a software architecture from distinct, complementary, viewpoints. Quality attributes can be categorized as those which are
  - observable during execution (e.g., availability)
  - not observable during execution (e.g., reusability)
- **Functionality** of a software system may be assured without necessarily satisfying some key architectural quality attributes
  - architectural attributes must be developed in addition to required/desired functionalities
  - the same functionality can be provided using alternative architectures – e.g., client-server vs. data-centric

System: Execution Observable Quality Attributes

- **Performance:** specifies how well (i.e., response time) a system responds to stimulus (internally and externally)
  - architecture issues/techniques
    - minimize communication
    - minimize number of transactions
  - performance has been (and still is) one of the most desired attributes

- **Availability:** indicates the proportion of time the system's services are available during some time period
  - reliability is related to availability
  - architecture issues/techniques
    - introduce redundancy
    - minimize patterns of interactions (error reporting & handling)
    - use customized monitoring components

System: Execution Observable Quality Attributes (cont.)

- **Functionality:** specifies how well the system provides the services it is supposed/expected to provide

- **Security:** specifies a system's ability to prevent unauthorized use (e.g., denial of service and IP source address spoofing)
  - architecture issues/techniques
    - use authentication server
    - placing the system services behind a firewall
    - …

- **Usability:** indicates the degree to which the system supports error avoidance, error handling, learnability, memorability, etc.

System: Non-Execution Observable Quality Attributes

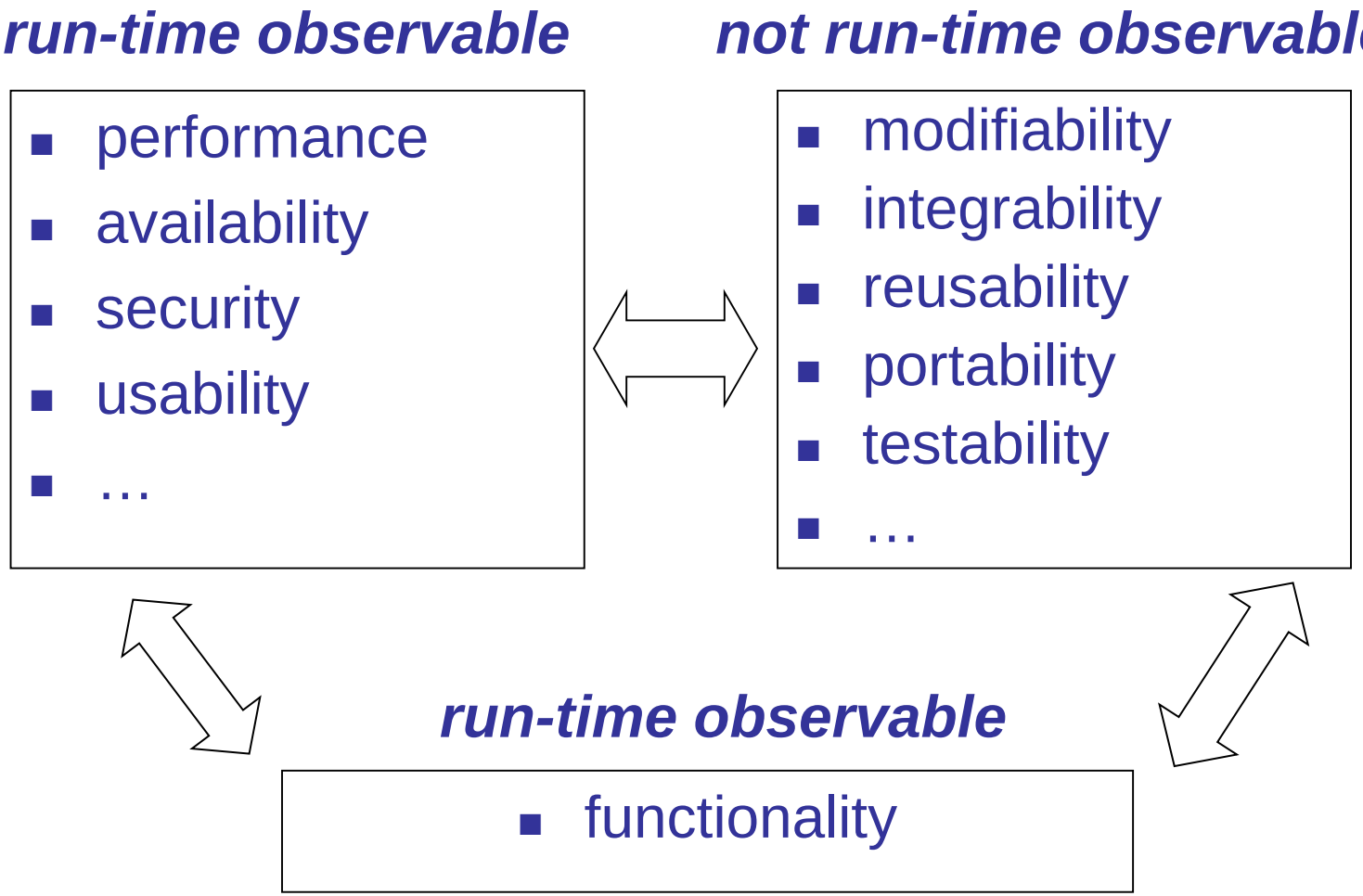
- **Modifiability:** indicates how well an architecture responds to making changes to it – amount of effort required (cost), the length of time it might take, …
  - local vs. global change
  - direct impact on software development process

- **Integrability:** indicates to what extent an architecture can be made to work with independently developed components
  - architecture issues/techniques
    - complexity of components
    - separation of responsibilities
    - interaction mechanisms and protocols

System: Non-Execution Observable Quality Attributes (cont.)

- **Reusability:** specifies to what degree an architecture can be reused – ease with which structural partitioning and use of individual or collection of components can be carried out
  - reusability is closely related to modifiability
- **Portability:** specifies a system's ability to run using alternative computing environments
- **Testability:** specifies how well (ease and simplicity) the system lends itself to testing

System Quality Attributes



qualities influence one another, some more than others!

Business Qualities

- Cost and schedule
  - Cost
  - Time to market
  - Anticipated lifetime of the system
- Market and marketing
  - Targeted market
  - Rollout schedule
  - Use of existing (legacy) systems

Architecture Quality Attributes

- Qualities that are integral to a well-defined architecture are
  - Conceptual integrity: the degree to which there exist a unified design theme and vision
  - Correctness and completeness: the degree to which a system's behavior can be assured to satisfy requirements given some finite run-time resources
  - Buildability: specifies to what extent the architecture promotes and supports completion of the project for a given team, resources, and time period

References

- *Software Architecture in Practice, 3rd Edition, SAP2, Bass, L. Clements, P., and Kazman, R., Addison Wesley, SEI Series in Software Engineering, 2012*
- *Software Architecture in Practice, 2nd Edition, SAP2, Bass, L. Clements, P., and Kazman, R., Addison Wesley, SEI Series in Software Engineering, 2003*
- *Software Architecture in Practice, SAP, Bass, L. Clements, P., and Kazman, R., Addison Wesley, SEI Series in Software Engineering, 1998*
- *Shaw, M. and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proc. COMPSAC97, 21st Int'l Computer Software and Applications Conference, August 1997, pp. 6-13*
- *OMG Unified Modeling Language, http://www.omg.org/spec/UML/2.5/, 2016*