Introduction to software architecture

Denis Conan

September 2024

Foreword

- The content of these slides is extracted from the following references:
  - L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 3rd Edition*. Addison-Wesley, 2012.
  - P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architecture: Views and Beyond, 2nd Edition*. Addison-Wesley, 2011.
  - P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.
  - EIT Digital, "*Software Architecture for the Internet of Things*", Coursera MOOC, 2015
  - A. Sunyaev. *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Springer, 2020.

1 Subject of this series of slides: Software architecture

1.1 Definition of "Software architecture"

- "The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [Bass et al., 2012]
- Software architecture = an abstraction, i.e. it omits certain information
  - Elements interact with each other by means of interfaces that partition details into public and private parts
  - Architecture focuses on the public side of this division
- Desirable properties of software architectures:
  - Can be constructed, evaluated, and documented
  - Answer to requirements to satisfy stakeholders
  - Have a repertoire of patterns and description languages (Architecture Description Language, ADL)
- In this course, we consider distributed architectures

1.2 Examples of sets of software structures

- Module decomposition structures = Implementation units
  - What is the primary functional responsibility, e.g. assigned to each element?
  - What other elements is an element allowed to use?
  - What other software does it actually use and depend on?
- Component-and-connector structures = runtime entities, e.g.
  - What are the major runtime elements and how do they interact?
  - What are the major shared data stores?
  - Which parts of the system are replicated? Can run in parallel?
  - Can the system's structure change as it executes and, if so, how?
- Allocation structures = mapping from software structures to organizational, developmental, installation, e.g.
  - What is the assignment of each software element to development teams?
    - Brook's Mythical Man-Month: group inter-communication = $O(n^2)$
    - Conway's law: design structure = a copy of the organization's communication structure

1.3 Other architectures: System and Enterprise

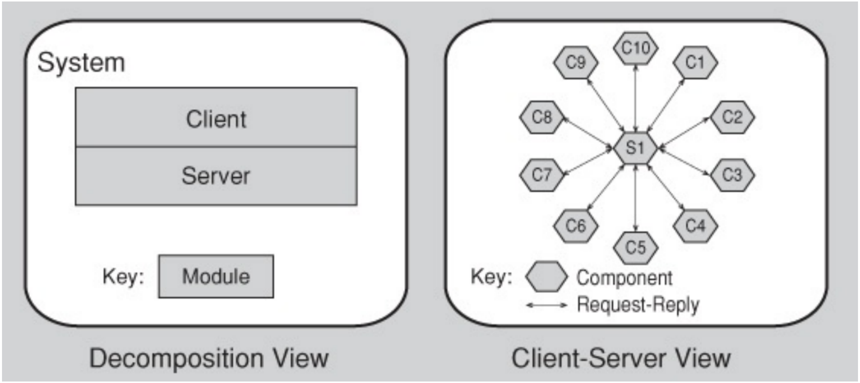
- System architecture
  - “Total system” = hardware + software
    - E.g., hardware-software co-design of embedded systems
  - In this course, we limit ourselves to software-intensive systems
- Enterprise architecture
  - “System of systems” with also other concerns:
    - How the software is used by humans to perform business processes?
    - How the subunits are aligned with the organization’s core goals and strategic direction?

- Each type of architecture has its own specialized vocabulary and techniques

1.4 Views of a software architecture

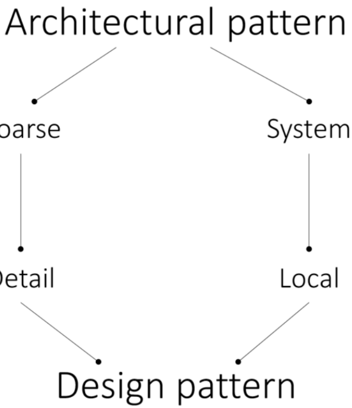
- Each of the software structures provides a different perspective
  - E.g. module decomposition, component-and-connector, allocation
- Although they give different system perspectives, they are not independent
  - Elements of one structure will be related to elements of other structures
  - We need to reason about the relations
- A view is a representation of a set of elements and relations among them
  - Not all system elements, but those of a particular type
- Documenting an architecture is
  1. Documenting the relevant views
  2. Adding documentation that applies to more than one view

1.4.1 Example of Client–Server with two views



2 Motivations for and objectives of studying Software Architecture in this course

2.1 Architecture patterns Vs. Design patterns



. In this lecture, we do not make the distinction between the terms "architecture pattern" and "architecture style". The two pages that are referenced plus the fact that there is a definition of the "architecture style" concept in the "architecture pattern" page indicate that the two terms are close.

2.2 Software architecture and Middleware

- Middleware is
  - Software glue
  - Software that connects software components in the context complex, distributed applications
  - Any software that allows other software to interact
  - The "connector" part in the "component-and-connector" view
- Design patterns exist for the design of the connectors
  - As home work, study "Introduction to design patterns for middleware":
    - Asynchronous call, Synchronous call, Buffered messages, Inversion of control, Proxy, Wrapper, Interceptor, Component/Container, etc.
- Middleware and architecture patterns are strongly related

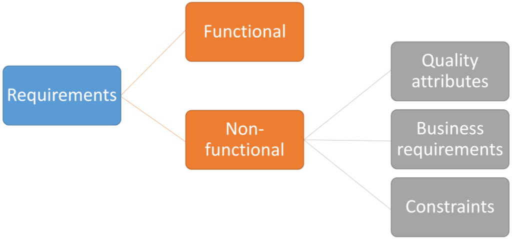
3 Attribute-Driven Design (ADD)

. Interested readers should also take a look at “Domain Driven Design” [Evans, 2004] and [Vernon, 2013].

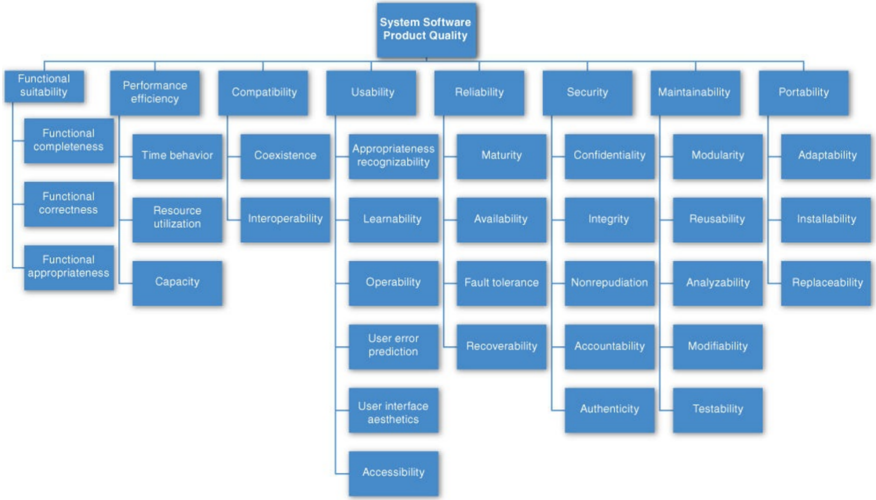
3.1 Quality attribute requirements

3.1.1 Definition of “Quality attribute”

- Quality is about the extra-functional characteristics: modifiability, usability, testability, scalability, availability, security, etc.
  - Related to the functions as perceived by the user or customer
- “A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders” [Bass et al., 2012]
- Quality attributes (when architecting) $\neq$
  - Business requirements and Constraints (taken into account before architecting)



3.1.2 ISO/IEC 25010 product quality standard I



3.1.2 ISO/IEC 25010 product quality standard II

- Functional suitability: The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
- Performance efficiency: Performance relative to the amount of resources used under stated conditions
- Compatibility: The degree to which a product, system, or component can exchange information with other products, systems, or components, and/or perform its required functions, while sharing the same hardware or software environment
- Usability: The degree to which a product or system can be used by users to achieve goals with effectiveness, efficiency, and satisfaction in a context of use
- Reliability: The degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time
- Security: The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
- Maintainability: The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
- Portability: The degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or other operational or usage environment to another

3.1.3 Modelling quality attribute requirements I

- Quality attribute requirements are modelled into scenarios

- Let's take an example from the document that describes the micro-project:
  - Section 2.3, at page 7: "Each participant publishes his/her location. Each participant receives locations of other participants in the group."
  - Section 3, at page 10: "The system should be able to handle up to 3000 groups of tourists at a time, and handle up to 3000 $\times$ 10 tourists. The system notifies all the members of the group of the tourists within a maximum of 1 second after having received a location from a tourist."

- In Section 3, at page 10:
  - "In the context of the project, we ask you to take in consideration those requirements for the report. In the report, we ask you to analyse the architecture of the application with regard to two quality attributes chosen [...]"

3.1.3 Modelling quality attribute requirements II

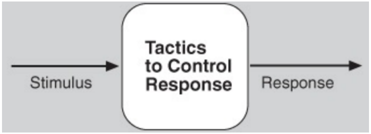
Source	who or what	A tourist
Stimulus	does something	...broadcasts a location message to the members of their group
Environment	under certain conditions	...during normal operations, with at most 3000×10 tourists that are broadcasting at most one message per second,
Artifact	to the system or part of it	...to the group communication system.
Response	the system reacts with these actions	The group communication system notifies (sends notifications to) all the members of the group of the tourists
Resp. measure	which can be measured by these metrics	...within a maximum of 1 second after having received the broadcast message from the tourist.

3.1.3 Modelling quality attribute requirements III

- What to specify in stimulus-oriented requirements modelling
  1. Source of stimulus: This is some entity (a human, a computer system, or any other actuator) that generated the stimulus
  2. Stimulus: The stimulus is a condition that requires a response when it arrives at a system
  3. Environment: The stimulus occurs under certain conditions (environment state). The system may be in an overload condition or in normal operation, or some other relevant state.
  4. Artifact: Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it
  5. Response: The response is the activity undertaken by the system as the result of the arrival of the stimulus
  6. Response measure: When the response occurs, it should be measurable in some fashion so that the requirement can be tested
- See Slide 4 in the appendices for some other illustrative scenarios

3.2 Tactic to Deal with a Quality Attribute I

- The quality attribute requirements specify the responses of the system that realize the goals of the business
- Techniques to achieve quality attributes are called architecture tactics
- "A tactic is a design decision that influences the achievement of **a** quality attribute response" [Bass et al., 2012]



- Within a tactic, there is no consideration of tradeoffs
  - In this respect, tactics differ from architecture patterns, where tradeoffs are built into the pattern

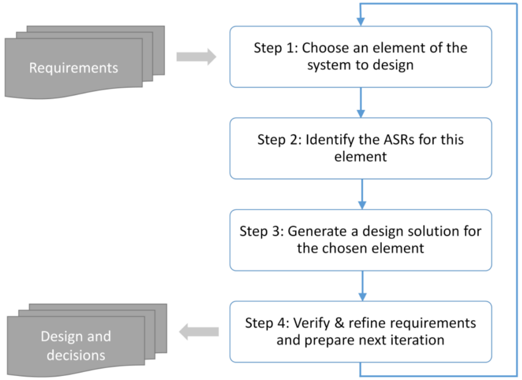
3.2 Tactic to Deal with a Quality Attribute II

- We propose to work on tactics by studying chapters of Bass, Clements, and Kazman's book [Bass et al., 2012]
  - Availability
  - Security
  - Interoperability
  - Modifiability
  - Scalability
  - Performance
- See, in the appendices, Slide 12 onwards for some other examples of decisions/tactics

3.3 Architecture Pattern to deal with several Tactics

- Architecture pattern = composition of architecture elements
  - Is a bundle of design decisions that is found repeatedly in practice
    - It brings to play a set of tactics
  - Has known properties that permit reuse
  - Describes a class or style of architectures
- Pattern cataloguers strive to understand how the characteristics lead to different behaviors and different responses to environmental conditions
  - As long as those conditions change, new patterns will emerge
- Examples: layered architecture, microkernel architecture, event-driven architecture, microservices architecture, space-based architecture
  - See the corresponding slides in the next sections

3.4 ADD Methodology


Requirements
Step 1: Choose an element of the system to design
Step 2: Identify the ASRs for this element
Step 3: Generate a design solution for the chosen element
Step 4: Verify & refine requirements and prepare next iteration
Design and decisions

$\Longleftarrow$ *Element is mostly a functional "component"*

$\Longleftarrow$ *Design solution introduces "connectors" (i.e. middleware), that is bring to play an architecture pattern*

ADD = Attribute-driven design

ASR = Architecture Significant Requirement: e.g. a quality attribute

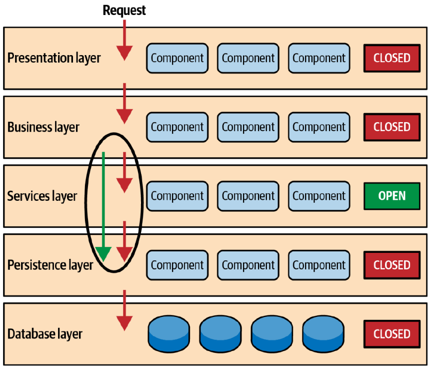
Element = the whole system, a subsystem, or a component

4 Architecture Patterns

. In [Richards, 2022], the author also presents the “Space-Based Architecture”.
. Architecture pattern images are extracted from [Richards, 2022].

4.1 Layered Architecture I

- A.k.a. "n-tier architecture"
- Separation of concerns: e.g. presentation layer deal only with user interface
- Closed layer: a request moves from layer to layer; easier software evolution

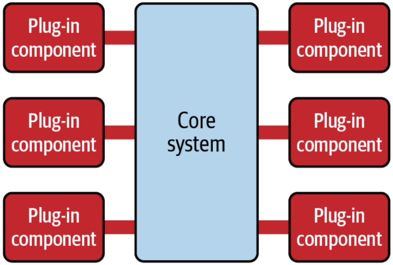


4.1 Layered Architecture II

- Advantages
  - Well-understood
  - Simple when your changes are isolated to specific layers
  - W.r.t. Conway's Law: Interesting when teams are organized by technical domains
- Drawbacks
  - Today, considered as a monolithic architecture
  - Many layers to cross and perhaps requests as simple pass-through processing
  - Operational concerns w.r.t. scalability (all layers must scale), elasticity, fault-tolerance, performance, etc.
  - The opposite of domain driven design: all layers must evolve when adding a new feature, e.g. new attribute to a business object

4.2 Microkernel Architecture I

- A.k.a. "plug-in architecture"
- Examples: Eclipse, browsers, Jenkins, and ESB (Enterprise Service Bus) like JBoss ESB, OpenESB, WSO2 ESB
- The core system provides extensibility, flexibility, and isolation
- Plug-ins = libraries, modules (e.g. OSGi), remote services

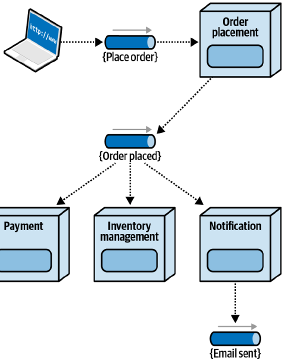


4.2 Microkernel Architecture II

- Advantages
  - Very flexible, simple, and cost-effective
  - The core system can control which users get which features
  - Great support for evolutionary design and incremental development
  - W.r.t. Conway's Law: can be considered as either a technically partitioned or a domain partitioned architecture
- Drawbacks
  - The core system may become a bottleneck for scalable and elastic systems
  - The core system is a single point of failure
  - Deployed as a monolithic (single deployment) architecture
  - Not interesting when most of your changes are within the core system

4.3 Event-Driven Architecture I

- A.k.a. “publish-subscribe based”
- Examples: AMQP, MQTT, Kafka
- Asynchronous processing using highly decoupled event processors
- Processor: From large, complex process to a single-purpose function
  - Microservice appli. and function as a service syst. (FaaS)
- Event processing generates events, thus forming an asynch. workflow
  - Services advertise their state changes

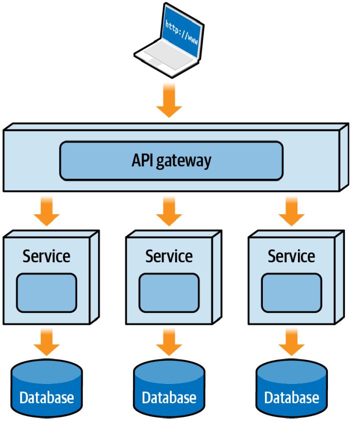


4.3 Event-Driven Architecture II

- Advantages
  - Asynchonous and decoupled (producers and consumers do not know each other)
  - Excellent for fault-tolerance, scalability, high-performance, and extensibility
    - Dynamically add/remove new event types and their corresponding producers and consumers
  - W.r.t. Conway's Law: since events can be technical evts or business evts, can be considered as either a technically partitioned or a domain partitioned archi.
- Drawbacks
  - Not when (business) interactions basically follow the request-response model, e.g. CRUD with synchronous processing
  - Not a good option for business problems that require high levels of data consistency, i.e. little or no guerantee of when processing will occur
  - No central workflow orchestrator, e.g. more complex handling of workflow errors

4.4 Microservices Architecture I

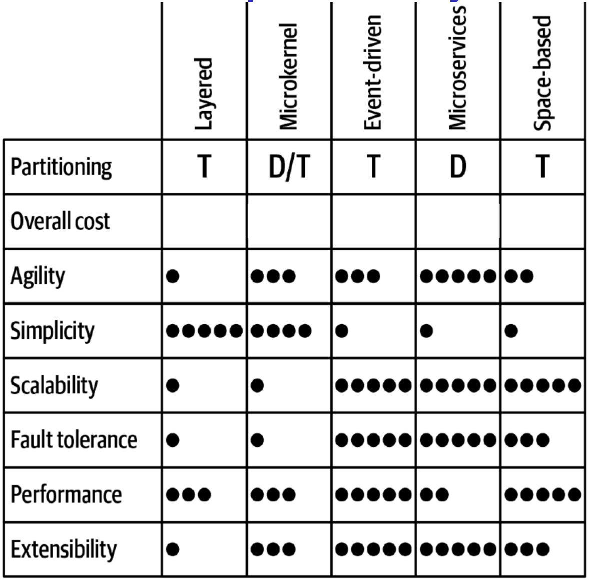
- μservice = single-purpose, separately deployed, accessed through an API gateway
- Each μservice accesses its own data, or makes requests to other μservices for "their" data
  - Concept of "bounded context", which is named after the Domain-Driven Design approach
- Numerous μservices with containerization, orchestration, DevOps —i.e., teams own services and the corresponding testing and release of those services
  - Docker Compose, Docker Swarm, Kubernetes
- Examples 1, 2, and 3



4.4 Microservices Architecture II

- Advantages
  - Architectural agility (ability to respond quickly to change)
    - Great support for evolutionary design and incremental development
    - Testing and deployment are easier to control
  - W.r.t. Conway's Law: domain partitioned architecture with teams owning microservices and the corresponding testing and release of those microservices
- Drawbacks
  - Perhaps the hardest architecture style to get right
    - Service granularity with single responsible principle is difficult to master
    - Asynch. *versus* synch. communication has numerous trade-offs
    - Complex workflows and lots of inter-service communication or orchestration
    - Distributed transaction management
  - Network latency, security latency, and data latency

4.5 Architecture pattern analysis



	Layered	Microkernel	Event-driven	Microservices	Space-based
Partitioning	T	D/T	T	D	T
Overall cost					
Agility	●	●●●	●●●	●●●●●	●●
Simplicity	●●●●●	●●●●	●	●	●
Scalability	●	●	●●●●●	●●●●●	●●●●●
Fault tolerance	●	●	●●●●●	●●●●●	●●●
Performance	●●●	●●●	●●●●●	●●	●●●●●
Extensibility	●	●●●	●●●●●	●●●●●	●●●

References I

arc42 (accessed September 2024).

`https://arc42.org`.

Bass, L., Clements, P., and Kazman, R. (2012).

Software Architecture in Practice, 3rd Edition.

Addison-Wesley.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2011).

Documenting Software Architecture: Views and Beyond, 2nd Edition.

Addison-Wesley.

Clements, P., Kazman, R., and Klein, M. (2002).

Evaluating Software Architectures: Methods and Case Studies.

Addison-Wesley.

Evans, E. (2004).

Domain-Driven Design: Tackling Complexity in the Heart of Software.

Addison-Wesley.

References II

ISO/IEC (1991).

Information technology — Software product evaluation — Quality characteristics and guidelines for their use.

International Standard ISO/IEC-9126, ISO/IEC Joint Technical Committee.

https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42010:ed-1:v1:en.

ISO/IEC/IEEE (2011).

Systems and software engineering — Architecture description.

International Standard ISO/IEC/IEEE-42010:2011, ISO/IEC/IEEE Joint Technical Committee.

Richards, M. (2022).

Software Architecture Patterns, 2nd Edition.

O'Reilly.

Sunyaev, A. (2020).

Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies.

Vernon, V. (2013).

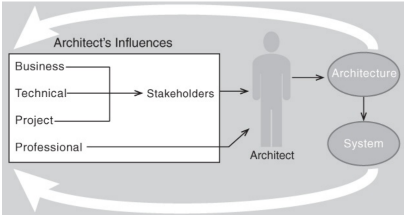
Implementing Domain-Driven Design.

Addison-Wesley.

Appendices

5 Software architecture: business and technical perspectives I

- On a business perspectives
  - Communication among stakeholders: business, technical, management, etc.
    - An architecture channels the creativity by reducing design complexity
  - Architecture-based development focuses on finding
    1. A stable design and
    2. a stable (predictable) development plan



  - Documentation of the architecture: See [ISO/IEC/IEEE, 2011] and [arc42, 2024]

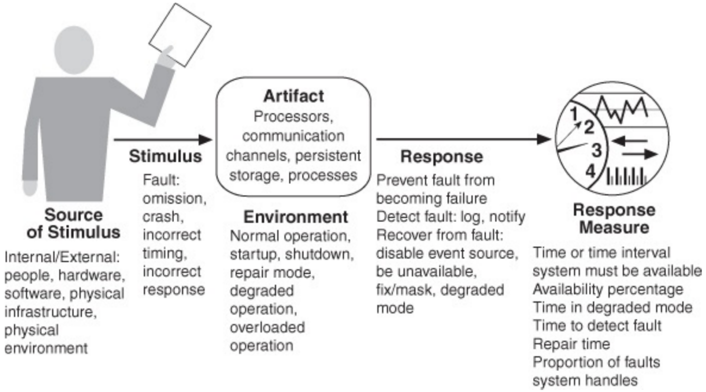
5 Software architecture: business and technical perspectives II

- On a technical perspective
- Software architecture is about preliminary design
  - Each stakeholder (customer, user, project manager, coder, tester, and so on) is concerned with different characteristics of the system
- Software architecture is about design at large
  - Provides a language in which different concerns can be expressed, negotiated, and resolved at a level that is manageable (by one person) for large, complex systems
- The early design decisions carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life
- It is the earliest point at which these design decisions can be scrutinized

6 Examples of illustrative scenarios of quality attribute requirements I

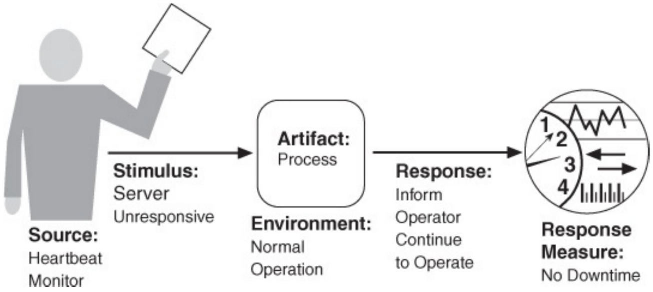
In [Bass et al., 2012], quality attribute requirements are modelled in graphics

- Availability


Source of Stimulus
Internal/External: people, hardware, software, physical infrastructure, physical environment
Stimulus
Fault: omission, crash, incorrect timing, incorrect response
Artifact
Processors, communication channels, persistent storage, processes
Environment
Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response
Prevent fault from becoming failure
Detect fault: log, notify
Recover from fault: disable event source, be unavailable, fix/mask, degraded mode
Response Measure
Time or time interval system must be available
Availability percentage
Time in degraded mode
Time to detect fault
Repair time
Proportion of faults system handles
1
2
3
4

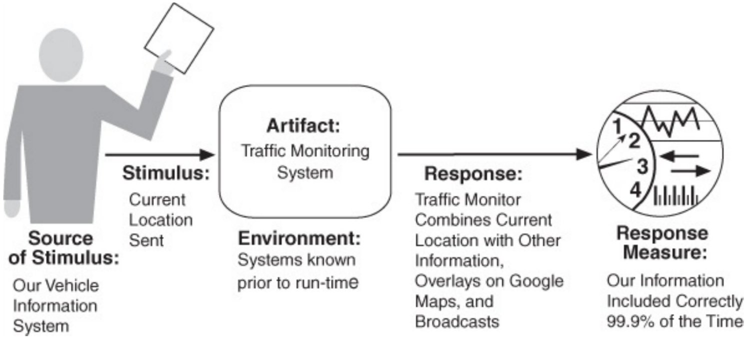
6 Examples of illustrative scenarios of quality attribute requirements II

- Availability



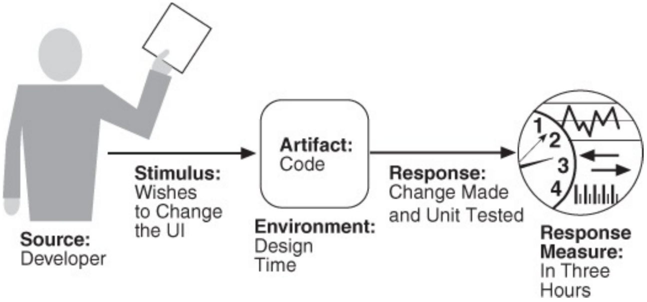
6 Examples of illustrative scenarios of quality attribute requirements III

- Interoperability


Source of Stimulus:
Our Vehicle Information System
Stimulus:
Current Location Sent
Artifact:
Traffic Monitoring System
Environment:
Systems known prior to run-time
Response:
Traffic Monitor Combines Current Location with Other Information, Overlays on Google Maps, and Broadcasts
Response Measure:
Our Information Included Correctly 99.9% of the Time
1
2
3
4

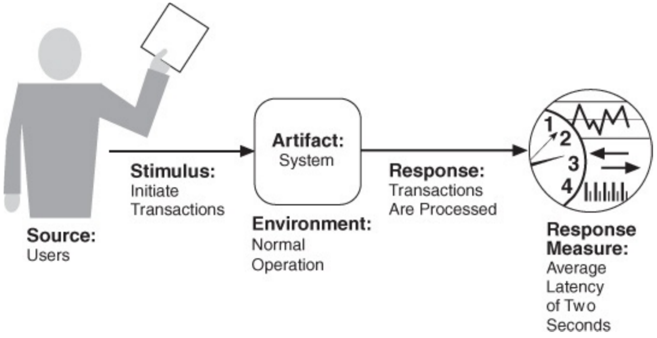
6 Examples of illustrative scenarios of quality attribute requirements IV

- Modifiability


Source: Developer
Stimulus: Wishes to Change the UI
Artifact: Code
Environment: Design Time
Response: Change Made and Unit Tested
Response Measure: In Three Hours
1
2
3
4

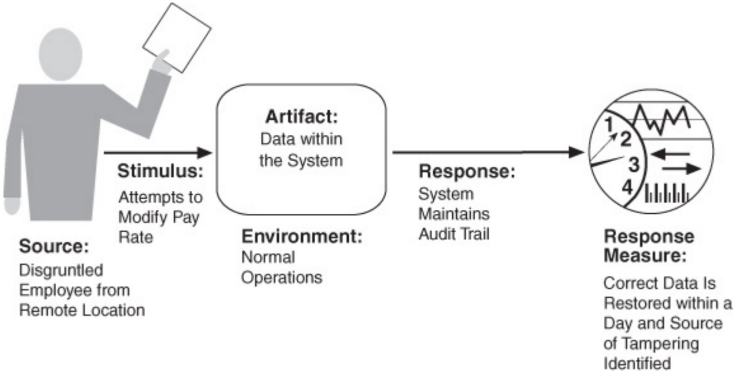
6 Examples of illustrative scenarios of quality attribute requirements V

- Performance



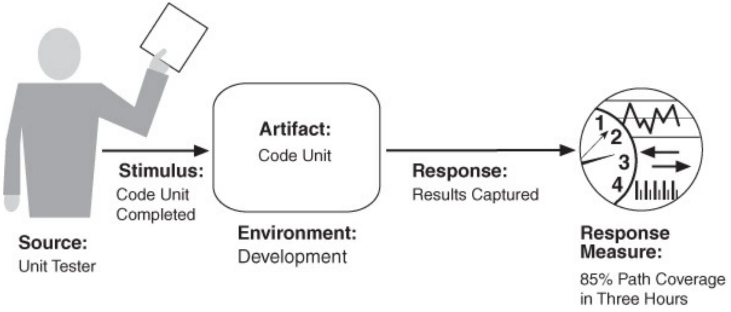
6 Examples of illustrative scenarios of quality attribute requirements VI

- Security


Source:
Disgruntled Employee from Remote Location
Stimulus:
Attempts to Modify Pay Rate
Artifact:
Data within the System
Environment:
Normal Operations
Response:
System Maintains Audit Trail
Response Measure:
Correct Data Is Restored within a Day and Source of Tampering Identified
1
2
3
4

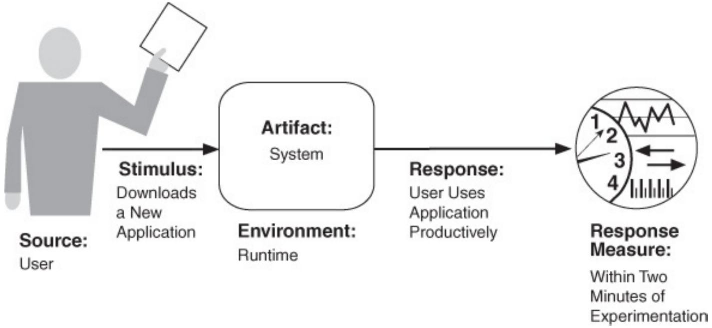
6 Examples of illustrative scenarios of quality attribute requirements VII

- Testability



6 Examples of illustrative scenarios of quality attribute requirements VIII

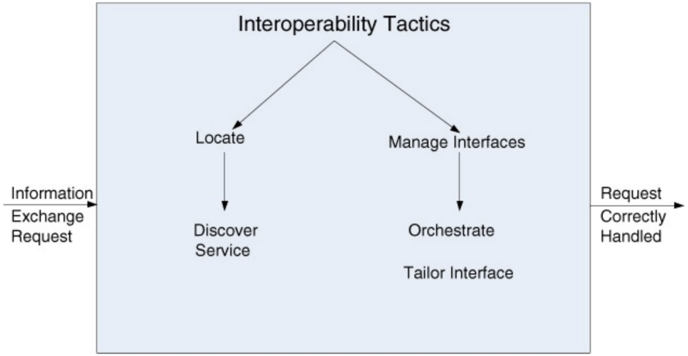
- Usability


Source:
User
Stimulus:
Downloads a New Application
Artifact:
System
Environment:
Runtime
Response:
User Uses Application Productively
Response Measure:
Within Two Minutes of Experimentation
1
2
3
4

7 Example of decisions for some tactics I

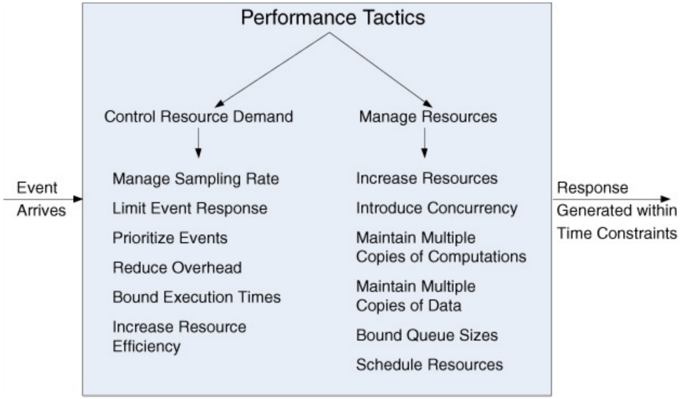
- In [Bass et al., 2012], sets of decisions are graphically displayed. The slides of this section contain some of these graphics.

- Interoperability tactics



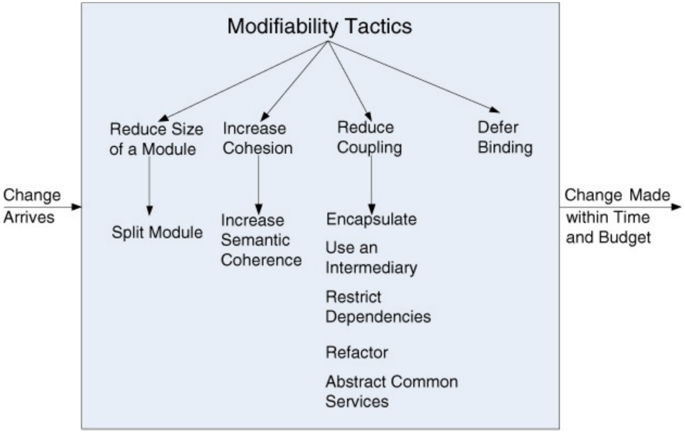
7 Example of decisions for some tactics II

- Performance tactics



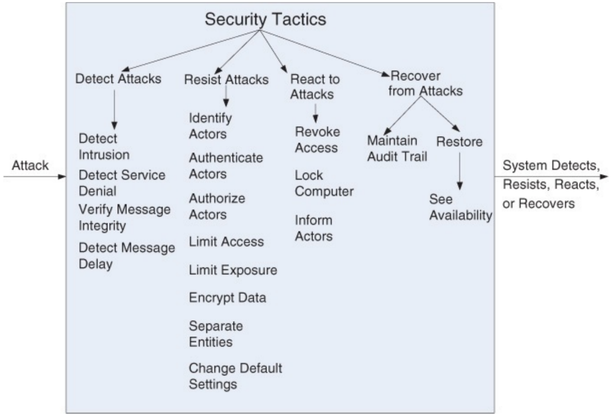
7 Example of decisions for some tactics III

- Modifiability tactics



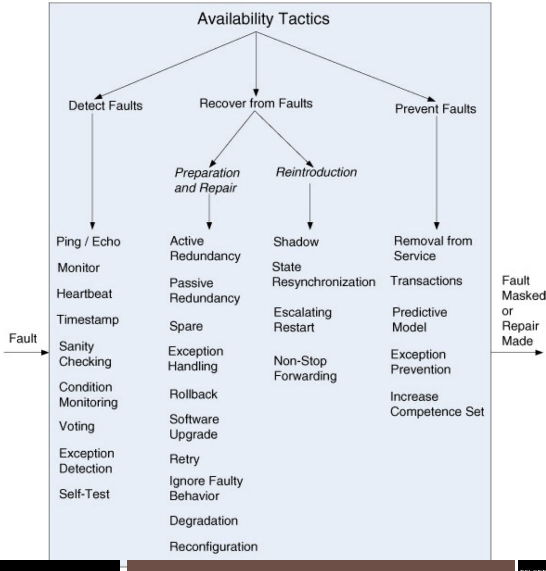
7 Example of decisions for some tactics IV

- Security tactics



7 Example of decisions for some tactics V

- Availability tactics



8 Example of a catalog of questions I

- We can view an architecture as the result of applying a collection of design decisions
- Slides of this section propose some initial categories of decisions
  - During the design of your solution, use these slides as a reminder
- These design decisions are fine-grained; this is why there are inserted in the appendix

8 Example of a catalog of questions II

- Allocation of responsibilities
  - Identifying the important responsibilities, including basic system functions, architecture infrastructure, and satisfaction of quality attributes
  - Determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors)

8 Example of a catalog of questions III

- Coordination model
  - Identifying the elements of the system that must coordinate, or are prohibited from coordinating
  - Determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency
  - Choosing the communication mechanisms (between systems, between our system and external entities, between elements of our system)
    - Stateful versus stateless
    - Synchronous versus asynchronous
    - Guaranteed versus nonguaranteed delivery
    - Performance-related properties such as throughput and latency

8 Example of a catalog of questions IV

- Data model
  - Choosing the major data abstractions, their operations, and their properties
    - How the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed
  - Compiling metadata needed for consistent interpretation of the data
  - Organizing the data
    - In a relational database, a collection of objects, or both
      - If both, then the mapping between the two different locations of the data must be determined

8 Example of a catalog of questions V

- Management of resources
  - Identifying the resources that must be managed and determining the limits for each
  - Determining which system element(s) manage each resource
  - Determining how resources are shared and the arbitration strategies employed when there is contention
  - Determining the impact of saturation on different resources

8 Example of a catalog of questions VI

- Mapping among architecture elements
  - The mapping of modules and runtime elements to each other
    - The runtime elements that are created from each module
    - The modules that contain the code for each runtime element
  - The assignment of runtime elements to processors
  - The assignment of items in the data model to data stores
  - The mapping of modules and runtime elements to units of delivery

8 Example of a catalog of questions VII

- Binding time decisions
  - Binding time decision establishes the scope, the point in the life cycle, and the mechanism for achieving the variation
    - For allocation of responsibilities, you can have build-time selection of modules via a parameterized makefile
    - For choice of coordination model, you can design runtime negotiation of protocols
    - For resource management, you can design a system to accept new peripheral devices plugged in at runtime, after which the system recognizes them and downloads and installs the right drivers automatically
    - For choice of technology, you can build an app store for a smartphone that automatically downloads the version of the app appropriate for the phone of the customer buying the app.

8 Example of a catalog of questions VIII

- Choice of technology
  - Deciding which technologies are available to realize the decisions made in the other categories
  - Determining whether the available tools to support this technology choice (IDEs, simulators, testing tools, etc.) are adequate for development
  - Determining the extent of internal familiarity as well as the degree of external support available for the technology (such as courses, tutorials, examples, and availability of experts) and deciding whether this is adequate
  - Determining the side effects of choosing a technology, such as a required coordination model or constrained resource management opportunities
  - Determining whether a new technology is compatible with the existing technology stack
    - Can the new technology run on top of or alongside the existing technology stack?
    - Can it communicate with the existing technology stack?
    - Can the new technology be monitored and managed?