Hugbúnaðarverkefni 2 / Software Project 2

12. Software Architecture

HBV601G – Spring 2019

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI- OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz 10 Prep

- Please prepare a small scrap of paper with the following format:

ID: _____@hi.is Date: _____

a) _____ e) _____
b) _____ f) _____
c) _____ g) _____
d) _____

- During class, I'll show you questions that you can answer with a letter
- Hand in your scrap at end of class
- All questions in a quiz have same weight
- All quizzes (8-10 throughout semester) have the same weight
  - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam

Margaret Burnett & Anita Sarma: Gender-Inclusivity Software Engineering

Webcast and Q&A on Thu 25 Apr, 16:00-17:00 GMT

- Research shows that different people often work differently with software, and that some of these differences statistically cluster by gender. This talk presents GenderMag, a method that can be used to find and fix gender biases in software. At its core are five facets of cognitive style differences that are also statistically gender differences, drawn from foundational work in computer science, psychology, education, communications, and women's studies. The talk presents results of using GenderMag on commercial and open-source products, and provides guidelines for applying the method in practice.

- **Margaret Burnett** is a Distinguished Professor at Oregon State University, known for pioneering work in visual programming languages, end-user software engineering, and gender-inclusive software. She is an ACM Fellow, a member of the ACM CHI Academy, and a member of the Academic Alliance Advisory Board of the National Center for Women & Information Technology (NCWIT).
- **Anita Sarma** is an Associate Professor at Oregon State University. She is an NSF CAREER award winner with a research focus on how socio-technical dependencies affect team work, how onboarding barriers in open source can be alleviated, and how software can be made gender-inclusive.

Register for live or on-demand access: *https://event.on24.com/wcc/r/1954235/BD30AEC62459D7274BE57AC255421962*

Recap: Architectural Analysis

- How do we come up with a software architecture?

- **The essence of architectural analysis** is to
  - **identify** **factors** that should influence the architecture,
  - **understand** their **variability** and **priority**, and
  - **resolve** them by making architectural **decisions**.

- Challenge: It's difficult to…
  - Know what questions to ask
  - Weigh the trade-offs
  - Know the many ways to resolve an architecturally significant factor
  - Decide on the best way under the given circumstances

Recap: Architectural Structures

- An **architectural structure** is a set of software **elements** and the **relations** between them *(a "perspective" for reasoning about certain aspects of a system)*
  - **Module structures** show how a system is to be statically structured as a set of code or data units that have to be constructed or procured.
    - Allocation of responsibilities
    - Data model
  - **Component-and-connector structures** show how a system is to be dynamically structured as a set of modules having runtime behavior (components) and interactions (connectors).
    - Coordination model
    - Variations and binding times
  - **Allocation structures** show how software structures are to be mapped to the system's organizational, developmental, installation and execution environments
    - Mapping between architectural elements
    - Management of resources
  - **Crosscutting all structures:** Technology choices

In-Class Quiz 9 Solution: Architectural Design Decisions

a) Decisions about the **allocation of responsibilities**
   - determine where functional requirements will manifest themselves (4)

b) Decisions about the **coordination model**
   - determine how modules interact through designed mechanisms (5)

c) Decisions about the **data model**
   - determine the representation of entities whose processing is the main purpose of the system (2)

d) Decisions about **management of resources**
   - determine the arbitration of use of shared resources, e.g. CPU, storage, peripherals etc. (1)

e) Decisions about the **mapping between architectural elements**
   - determine the association of elements of development and execution; and software and hardware elements (3)

f) Decisions about **technology choices**
   - determine the selection of suitable technologies to support all these architectural decisions (7)

g) Decisions about **variations** and **binding times**
   - determine allowable range, time and mechanism of variations of a system (6)

Architectural Tactics

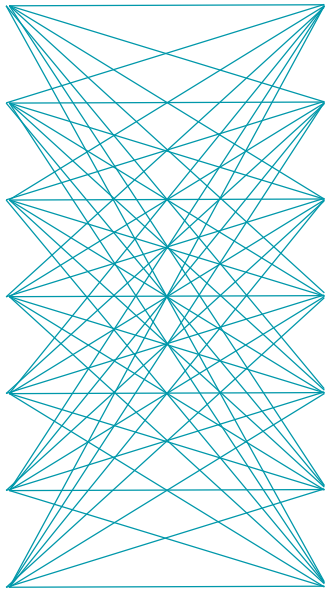
see also:

- Bass et al.: Software Architecture in Practice, Part II

Architectural Tactics

Design Decisions

- Allocation of responsibilities
- Coordination model
- Data model
- Management of resources
- Mapping of arch. elements
- Variations and binding times
- Technology choices



Quality Attributes

- Availability
- Interoperability
- Modifiability
- Performance
- Security
- Testability
- Usability
- Variability
- Portability
- Scalability
- Mobility
- Safety
- Deployability
- Monitorability
- Shorability

- An **architectural tactic** is a design decision that affects a quality attribute.

Availability Tactics

- **Prevent faults**
  - Removal from service
  - Transactions
  - Predictive model
  - Exception prevention
  - Increase competence set
- **Detect faults**
  - Ping/echo
  - Monitor
  - Heartbeat
  - Timestamp
  - Sanity checking
  - Condition monitoring
  - Voting
  - Exception detection
  - Self-test
- **Recover from faults**
  - **Preparation and repair**
    - Active redundancy
    - Passive redundancy
    - Spare
    - Exception handling
    - Rollback
    - Software upgrade
    - Retry
    - Ignore faulty behavior
    - Degradation
    - Reconfiguration
  - **Reintroduction**
    - Shadow
    - State resynchronization
    - Escalating restart
    - Non-stop forwarding

Modifiability Tactics

- **Reduce module size**
  - Split module
- **Increase cohesion**
  - Increase semantic coherence
- **Reduce coupling**
  - Encapsulate
  - Use an intermediary
  - Restrict dependencies
  - Refactor
  - Abstract common services
- **Defer binding**

Security Tactics

- **Detect attacks**
  - Detect intrusion
  - Detect service denial
  - Verify message integrity
  - Detect message delay
- **React to attacks**
  - Revoke access
  - Lock computer
  - Inform actors
- **Resist attacks**
  - Identify actors
  - Authenticate actors
  - Authorize actors
  - Limit access
  - Limit exposure
  - Encrypt data
  - Separate entities
  - Change default settings
- **Recover from attacks**
  - Maintain audit trail
  - Restore *(see Availability)*

Testability Tactics

- **Control and observe system state**
  - Specialized interfaces
  - Record / playback
  - Localize state storage
  - Abstract data sources
  - Sandbox
  - Executable assertions
- **Limit complexity**
  - Limit structural complexity
  - Limit nondeterminism

Usability and Interoperability Tactics

Usability Tactics

- **Support user initiative**
  - Cancel
  - Undo
  - Pause / resume
  - Aggregate
- **Support system initiative**
  - Maintain task model
  - Maintain user model
  - Maintain system model

Interoperability Tactics

- **Locate**
  - Discover service
- **Manage interfaces**
  - Orchestrate
  - Tailor interface

Performance Tactics

- **Control resource demand**
  - Manage sampling rate
  - Limit event response
  - Prioritize events
  - Reduce overhead
  - Bound execution times
  - Increase resource efficiency
- **Manage resources**
  - Increase resources
  - Introduce concurrency
  - Maintain multiple copies of computations
  - Maintain multiple copies of data
  - Bound queue sizes
  - Schedule resources

Example: Design Checklist for Performance

Category	Checklist
Allocation of Responsibilities	Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those responsibilities, identify the processing requirements of each responsibility, and determine whether they may cause bottlenecks. Also, identify additional responsibilities to recognize and process requests appropriately, including ▪ Responsibilities that result from a thread of control crossing process or processor boundaries ▪ Responsibilities to manage the threads of control—allocation and deallocation of threads, maintaining thread pools, and so forth ▪ Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Performance

Coordination Model	Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that do the following: ▪ Support any introduced concurrency (for example, is it thread safe?), event prioritization, or scheduling strategy ▪ Ensure that the required performance response can be delivered ▪ Can capture periodic, stochastic, or sporadic event arrivals, as needed ▪ Have the appropriate properties of the communication mechanisms; for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency
Data Model	Determine those portions of the data model that will be heavily loaded, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those data abstractions, determine the following: ▪ Whether maintaining multiple copies of key data would benefit performance ▪ Whether partitioning data would benefit performance ▪ Whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible ▪ Whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Performance

Category	Checklist
Mapping among Architectural Elements	Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency. Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity. Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance. Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.
Resource Management	Determine which resources in your system are critical for performance. For these resources, ensure that they will be monitored and managed under normal and overloaded system operation. For example: ▪ System elements that need to be aware of, and manage, time and other performance-critical resources ▪ Process/thread models ▪ Prioritization of resources and access to resources ▪ Scheduling and locking strategies ▪ Deploying additional resources on demand to meet increased loads

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Performance

Binding Time	For each element that will be bound after compile time, determine the following: ▪ Time necessary to complete the binding ▪ Additional overhead introduced by using the late binding mechanism Ensure that these values do not pose unacceptable performance penalties on the system.
Choice of Technology	Will your choice of technology let you set and meet hard, real-time deadlines? Do you know its characteristics under load and its limits? Does your choice of technology give you the ability to set the following: ▪ Scheduling policy ▪ Priorities ▪ Policies for reducing demand ▪ Allocation of portions of the technology to processors ▪ Other performance-related parameters Does your choice of technology introduce excessive overhead for heavily used operations?

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Modifiability

Category	Checklist
Allocation of Responsibilities	Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes: ▪ Determine the responsibilities that would need to be added, modified, or deleted to make the change. ▪ Determine what responsibilities are impacted by the change. ▪ Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules.
Coordination Model	Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at runtime, or will the communication protocol change at runtime? If so, ensure that such changes affect a small number set of modules. Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules. For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish-subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast.

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Modifiability

Data Model	Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction. For each change or category of change, determine if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties. For each potential change or category of change: ▪ Determine which data abstractions would need to be added, modified, or deleted to make the change. ▪ Determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions. ▪ Determine which other data abstractions are impacted by the change. For these additional data abstractions, determine whether the impact would be on the operations, their properties, their creation, initialization, persistence, manipulation, translation, or destruction. ▪ Ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes. Design your data model so that items allocated to each element of the data model are likely to change together.

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Modifiability

Category	Checklist
Mapping among Architectural Elements	Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors) at runtime, compile time, design time, or build time. Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of the following, for example: ▪ Execution dependencies ▪ Assignment of data to databases ▪ Assignment of runtime elements to processes, threads, or processors Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.
Resource Management	Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example: ▪ Determining what changes might introduce new resources or remove old ones or affect existing resource usage ▪ Determining what resource limits will change and how Ensure that the resources after the modification are sufficient to meet the system requirements. Encapsulate all resource managers and ensure that the policies implemented by those resource managers are themselves encapsulated and bindings are deferred to the extent possible.

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Example: Design Checklist for Modifiability

Binding Time	For each change or category of change: ▪ Determine the latest time at which the change will need to be made. ▪ Choose a defer-binding mechanism (see Section 7.2) that delivers the appropriate capability at the time chosen. ▪ Determine the cost of introducing the mechanism and the cost of making changes using the chosen mechanism. Use the equation on page 118 to assess your choice of mechanism. ▪ Do not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of Technology	Determine what modifications are made easier or harder by your technology choices. ▪ Will your technology choices help to make, test, and deploy modifications? ▪ How easy is it to modify your choice of technologies (in case some of these technologies change or become obsolete)? Choose your technologies to support the most likely modifications. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.

Bass et al.: Software Architecture in Practice, 3rd Ed. Addison-Wesley 2013

Software Architecture in Practice

see also:

- Bass et al.: Software Architecture in Practice, Ch. 15, 17

Structural Recommendations for Good Architectures

- Have well-defined, well-encapsulated modules.
- Use well-known architectural patterns and tactics to achieve quality attributes.
- Don’t depend on a particular version of a particular product or tool, but structure the system so that swapping it out is straightforward and inexpensive.
- Separate modules that produce data from modules that consume data.
- Don’t expect a one-to-one correspondence between design-time modules and their run-time instances.
- Ensure that any process can be (re)assigned to any processor, even at run time.
- Use the same small set of module interaction paradigms throughout the system.
- Ensure that areas of resource contention are small,
  and that clear conflict resolution mechanisms are specified for them.

Process Recommendations Towards Good Architectures

- The architecture should be the responsibility of a single lead architect who has a strong connection to the development team, to ensure conceptual integrity.
- The architect should base the architecture on a prioritized list of well-specified quality attributes that inform the inevitable trade-offs.
- The architecture documentation should address the concerns (analysis, construction, maintenance, training) of the most important stakeholders.
- The architecture should be evaluated early and repeatedly for its ability to deliver the system's important quality attributes.
- The architecture should lend itself to incremental implementation
  - e.g. by building a skeletal system first in which just the communication paths are in place, and to which the functional components can be added incrementally.

Iterative-Incremental Development of Architecture

- **Inception**
  - Candidate architectures are considered.
  - If architectural feasibility of key requirements is questionable, a light proof-of-concept architecture may be implemented to determine feasibility.
- **Elaboration**
  - Architectural drivers (i.e. major architectural risks) are identified and resolved.
  - An executable architecture that can serve as the backbone of further implementation is built.
- **Construction**
  - Business components are built and integrated with the core architecture.
  - No major changes of the architecture should be necessary.
- **Transition**
  - The final architecture is documented as a basis for future maintenance and evolution.

Up-Front Architecture Planning vs. Agility

- Pressure to deliver a working system with demonstrable features early can lead to a disregard of architectural concerns in an agile project
  - Each feature would be independently designed and implemented
  - Concerns cutting across more than one feature may be easily missed, resolved in redundant or incompatible ways
- In an architecture-centric project, up-front analysis of functional and quality requirements, and making suitable design decisions, likely leads to a stable architecture, but takes time in which the customer does not get tangible results
  - Ideally, agile projects will want to evolve the architecture as they go along, while traditional projects invest more time in up-front analysis and planning
    - ➢ Balance modifiability tactics with the agile YAGNI principle (“don’t do what you don’t have to do”)
- The whole point of choosing how much time to budget for architecture is to reduce financial / political / operational / reputational risk.

Up-Front Architecture Planning vs. Agility

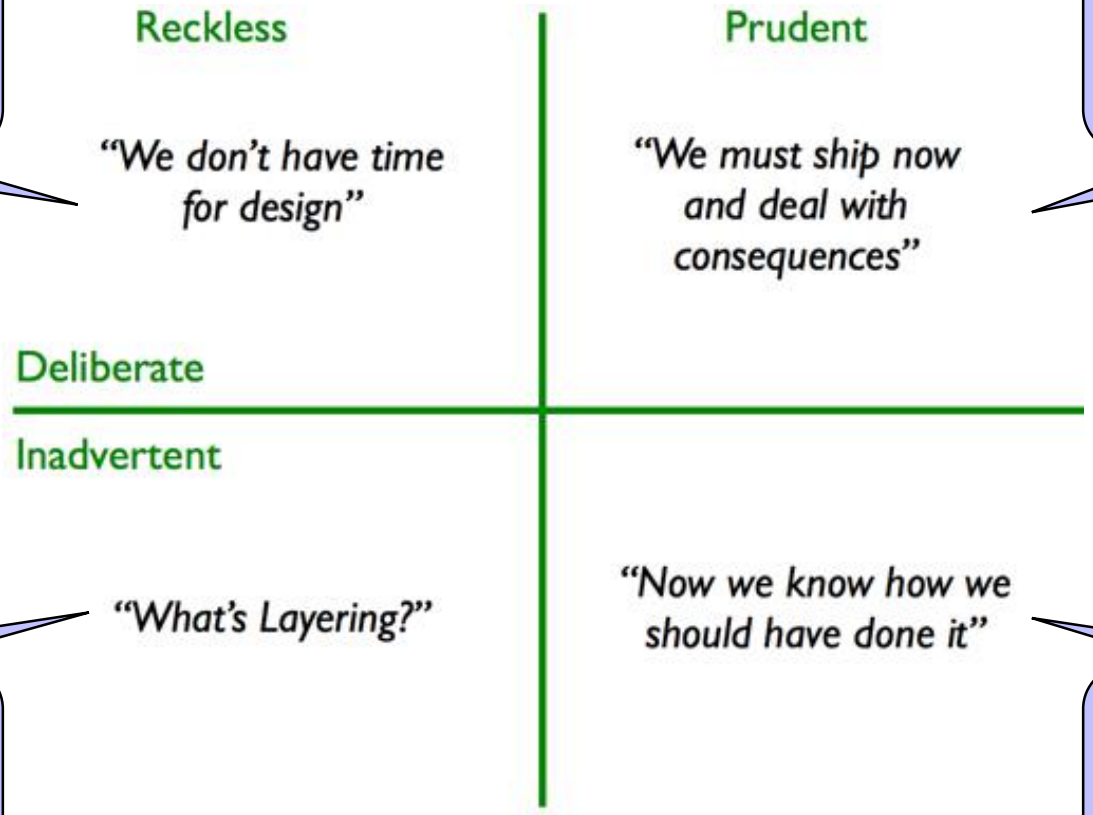
- Project realities often force architects to deal with conflicting goals:
  - Pleasing the customer by enabling the team to deliver many requested features fast, vs.
  - Ensuring the system's longevity by designing for modifiability, extensibility, portability up front
  - Many requirements trade off against each other
    - e.g. security comes at expense of performance, modifiability comes at expense of time to market, availability and performance come at expense of modifiability and cost, etc.
- Strive to work from two perspectives simultaneously:
  - **Top-down view:** Design and analyze architectural structures that meet quality requirements and make appropriate trade-offs
  - **Bottom-up view:** Analyze wide array of implementation- and environment-specific constraints and find solutions to them
- The latter activities address questions (= project risks) that can better be answered experimentally than analytically
  - Experiments / prototypes to eliminate these risks are the focus of "spikes" in agile planning

Technical Debt

- Technical debt is most commonly incurred by quick-and-dirty implementations that incur penalties in the future, in terms of increased maintenance costs.
- The longer technical debt remains in the system, the higher the necessary maintenances costs due to the amount of code that needs to be refactored.
  - “As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.” (Meir M. Lehman, 1980)
  - “Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.” (Ward Cunningham, 1992)
  - “Developers who evolve such systems suffer [due to] never being able to write quality code because they are always trying to catch up. Users suffer the consequences of capricious complexity, delayed improvements, and insufficient incremental change.” (Grady Booch, 2014)

Causes of Technical Debt

Incurring debt knowingly due to underestimation of associated cost



Incurring debt after making an informed cost/benefit analysis

Incurring debt due to lack of competence, leading to inappropriate design decisions

Incurring debt despite trying best to avoid it, as more knowledge surfaces while project progresses

Martin Fowler: Technical Debt Quadrant, 2014
http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Suggested Architecture Focus in Different Project Types

- Large and complex systems with relatively stable, well-understood requirements
  - Perform a large amount of architecture work up front
  - Low risk of changes, high pay-off in quality
- Big projects with vague or unstable requirements
  - Start by designing a complete candidate architecture (even if sketchy and omitting details)
  - Make sure it's enough to demonstrate end-to-end functionality and link major functionality
  - Be prepared to elaborate and change this as necessary
  - Necessity for changes should be determined through spikes and prototypes
  - Architecture will not be perfect, but help guide understanding, development
- Smaller, less critical projects with uncertain requirements
  - Get agreement on central patterns to be employed
  - But don't spend too much time on up-front architecture analysis and documentation

The Pivotal Role of Software Architecture

- Having a good software architecture is a key to a project's success because:
  - An architecture will inhibit or enable a system's driving quality attributes.
  - Architecture analysis enables early prediction of a system's qualities.
  - Architecture decisions allow you to reason about and manage system change and evolution.
  - A documented architecture enhances communication among stakeholders.
  - An architecture carries the earliest, most fundamental, hardest-to-change design decisions.
  - An architecture defines a set of constraints on subsequent implementation.
  - An architecture dictates the structure of an organization, or vice versa.
  - An architecture can provide the basis for evolutionary prototyping.
  - An architecture is the key artifact enabling reasoning about cost and schedule.
  - An architecture can be a transferable, reusable model at the heart of a product line.
  - An architecture focuses attention on assembly rather than just creation of components.
  - An architecture reduces design and system complexity by restricting design alternatives.
  - An architecture can be the starting point for introducing new team members to a project.

In-Class Quiz #10: Software Architecture Summary

Fill in the blanks using the following:

- **(A)llocation Structure, (C)omponent-and-Connector Structure, (D)esign Decisions, (M)odule Structure, (Q)uality Attributes, (S)ystem Architecture, Conte(x)ts**

- A (a) _____ can be considered from several perspectives:
  - (b) _____**:** Design-time view of divisions and relationships of constructed modules
  - (c) _____**:** Run-time view of interactions of module instances
  - (d) _____**:** Design- and run-time view of mapping arch. elements to each other
- A system's architecture is **established through a series of** (e) _____**.**
  - e.g. allocation of responsibilities, coordination model, data model, resource management, mapping of architectural elements, variations and binding times, technology choices
- Design decisions **influence** (f) _____ of the system.
  - e.g. availability, interoperability, modifiability, performance, security, testability, usability
- Decisions affect & depend on technical, project, business, professional (g) _____**.**