TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

TU/e

Exam 2II45 **Block 2 (Software Architecture, 1.5h)** on Thursday 21 January 2010, 14.00h–17.00h

---

Work clearly. Read the entire exam before you start. **Motivate each answer concisely and to the point.** Maximal scores per question are given between parentheses. The maximum total score is 30 points on 10 questions.

Hint We provide a set of hints for answering the questions. These are not the only correct answers, and in most cases they are not fully written out.

1. (3) Explain the notions of an *Architecture* and an *Architectural Description* according to IEEE Standard 1471, and summarize their differences.

   Hint Every existing (software) system has an *architecture*, in an abstract sense, being 'the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution'. That architecture need not be explicitly described or systematic in nature; it could be implicit in the implementation and it could be chaotic, ad hoc, unmotivated, and even not consciously known to anybody.

   An *architectural description* consists of artifacts that document an architecture. In particular, these artifacts identify Concerns, select Viewpoints, provide Rationale, and consist of Models organized by Views.

   To summarize the differences: an *architecture* exists abstractly and implicitly in an implemented system; an *architectural description* exists explicitly as concrete artifacts separate from any system, describing an architecture of a, possibly non-existent, system. The actual architecture of a system can differ from what is described in the architectural description.

2. (3) How can the verification of software products benefit from architectural design? Present at least three ways.

   Hint An architectural description

   (a) ... can be *reviewed* before code is developed; designing an architecture also provides early feedback on requirements; this reduces the cost of design errors;

   (b) ... defines components, which can be *unit tested* independently before integration, again reducing cost;

   (c) ... facilitates *controlled, stepwise integration* (as opposed to 'big-bang' integration), thereby making it easier to localize integration errors, once again reducing costs.

This takes into account the lessons from the *economy of defects*: the later defects are addressed, the higher the cost (exponentially).

3. (3) Present some modifiability aspects that are and some that are not an architectural concern, and explain why this is the case.

   Hint *Coupling* and *cohesion* are modifiability aspects that are an architectural concern. Use of *standardized architectural styles and patterns* is another. *Coding style* in general is not (though prescribing a uniform style and the use of code templates could be).

   Low coupling and high cohesion improve modifiability, by improving understandability and minimizing the *ripple effect* of changes. Using a good coding style to implement a bad architecture will not improve modifiability of the architecture. (Bad coding style could potentially lead to a bad product (with low modifiability because of poor readability) in spite of a good architecture.)

4. (3) What architectural (sub)views play a role in *Module Architecture Control*, what do these views describe, and what is their role?

   Hint See MAC slides: The module view and code view, as subviews of the development view, play a role. These views describe intended and derived relationships between entities (modules, components). MAC seeks to monitor the consistency of these views as software is developed and evolves under maintenance, by comparing intended and derived module relationships.

5. (3) Present a *general* and a *specific* performance requirements in the form of a *Quality Attribute Scenario*.

   Hint Example of a general performance quality attribute scenario:

   Source internal or external, possibly multiple sources

   Stimulus individual, periodic, sporadic, stochastic events

   Artifact (sub)system

   Environment normal/overlaod mode

   Response handle stimulus, change service level

   Response measure latency, deadling, thoughput, jitter, miss rate, data loss

   A concrete performance quality attribute scenario:

   Source the brake pedal in a car

   Stimulus emergency stop signal (full depression)

   Artifact the anti-lock brake controller

Environment normal operation

Response produce activation signals for brake actuators without locking the wheels

Response measure response latency is always less than 1 ms

A different concrete performance quality attribute scenario:

Source the web interface

Stimulus loan calculation requests at a rate causing a server load of more than 2

Artifact the main computation server

Environment normal operation

Response change into overload mode

Response measure mean response time remains less than 3 seconds, with a standard deviation less than 1 second, taken over 1000 requests

6. (3) Describe the notion of *tactic* to achieve a specified quality, and give two examples of availability tactics.

   Hint See slides 29–35 of Week 11.

   Tactic (slide 29): Design decision that influences control of a quality attribute response. N.B. A tactic is an option worthwhile considering, not a guaranteed solution. So, a tactic is *not* 'a standardized way to reach a specified quality'.

   Availability tactics: See slides 30–31. N.B. Do not confuse the goal of a tactic (e.g., "mitigate system failure") and actual tactic(s) aimed at accomplishing that goal.

7. (3) What is the ATAM and what does it deliver? Give an example of an architectural trade-off point.

   Hint See ATAM slides: ATAM stands for *Architecture Trade-off Analysis Method*. It is a way of organizing and carrying out a *qualitative* evaluation of architural designs for software-intensive systems. It delivers a *utility tree*, *quality attribute scenarios*, *sensitivity* and *trade-off points*, and it identifies *risks* and *non-risks*.

   A trade-off point is a parameter of the architecture that affects multiple quality attributes in opposite directions. For instance, looser coupling (to improve modifiability) or better encryption (to improve security) can negatively affect performance.

8. (3) What is a *Component Model*? How can it be used to make performance predictions?

Hint See CBSE slides: A *component model* specifies the standards and conventions that are needed to enable the composition of independently developed components.

Component developers need to specify a behavior model and a resource model for each component. The application developer selects and connects components, and models appropriate application scenarios. These models are combined, together with information about the infrastructure from the component model, into a complete model (of system and environment), which can be analyzed and/or simulated to predict performance.

9. (3) Describe general steps to extract architectural information from a given source code base, and indicate what information can be obtained in that way. Give at least two reasons why such reverse engineering would be interesting.

   Hint See slides of Week 14 (p. 9, 10): Select desired models, classify nature of given code; extraction steps: Code $\rightarrow$ Data $\rightarrow$ Model $\rightarrow$ Information

   Models for each of Kruchten's 4+1 views can be extracted this way. Typically, logical view and process view, incl. execution scenarios, are addressed. A deployment view may be harder to extract.

   Interesting (a) because architectural documentation may be lacking, and is needed for maintenance; (b) in order to check whether an implementation-under-construction adheres to architectural decisions (cf. question 4); possibly also (c) to obtain quality metrics (especially when the architectural models are informal).

10. (3) What are potential benefits of using an *Architecture Description Language*? What are its drawbacks (current or inherent)?

    Hint See MDE/MDA slides 9 and 10. Pro: formal (more precise), human-and-machine readable, higher level of abstraction, permits analysis and automatic code generation. Con: lack of agreement on semantics, inconvenient formats, limited focus (mostly vertical).