TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

TU/e

Exam 2II45 **Block 2 (Software Architecture, 1.5h)** on Tuesday 13 April 2010, 09.00h–10.30h

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 a *Viewpoint* and a *View* according to IEEE Standard 1471, and explain their relationship to Kruchten's 4+1 views.

   Hint Each viewpoint defines a set of (usually tightly related) stakeholder concerns. A single view offers a partial description of an architecture as seen from a particular viewpoint. Such a view consists of one or more (formal) models. N.B. Views need not be disjoint, but will typically overlap each other.

   Kruchten's 'views' can be considered to represent four common (library) viewpoints: logical, process, development, and physical. These are viewpoints that will be relevant for most software-intensive systems, in particular representing four common groups of stakeholders: end-users, integrators, programmers, and system engineers respectively. (N.B. For many systems, it is important to include other views as well, e.g., addressing a security concern.) The '+1' stands for *scenarios* that can be traced through the architecture to help illustrate and understand the four views and their relationship (e.g., consistency).

2. (3) Describe three ways in which delivering an architectural design is also beneficial from a project management point of view.

   Hint Having a well documented architectural design also benefits project management, because

   - it defines separate modules that can serve as (the basis of) work packages; when modules are well defined, it is also easier to estimate their development cost;
   - it documents module dependencies and thereby helps decide in what order modules can be produced, tested and integrated;
   - it can serve as a framework for tracking progress of implementation work.

3. (3) Present three testability aspects that are an architectural concern, and explain why this is the case. Describe a testability aspect that is *not* an architectural concern, *or* explain why testability is inherently architectural.

   Hint Modularization is an architectural concern that directly affects testability. Well-chosen modules allow for effective unit testing.

   Coupling between modules is an architectural concern, and low coupling simplifies integration testing.

   System-wide logging facilities are also an architectural concern that can benefit testing, because such a facility increases observability of behavior.

   Code complexity (within functions) is not architectural, but affects testability.

4. (3) *Module Architecture Control* (MAC) involves the *Module View* and *Code View*. Explain these views and their main ingredients, explain how they are related to each other, and what their role in MAC is.

   Hint See MAC slides: The module view and code view are subviews of the development view. The module view describes intended relationships between design entities (systems, subsystems, modules, components), in particular, the 'is part of' and 'uses' relationships, which can express layering and orthogonality.

   The code view describes the organization of the implementation code, in particular, how code is distributed over various files and directories, how files depend on each other (e.g., through 'includes'), and how design entities are expressed in the code. From the code view one can derive the actually implemented relationships between design entities.

   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 requirement 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 examples of usability tactics, covering both design time and runtime 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'.

   Usability tactics: See slide 35. 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 are its primary purposes? Give an example of a *sensitivity 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. For purposes, see Slide 11; for sensitivity point, see Slide 30.

8. (3) What is a *Component Model* in CBSE? Describe Bondarev's four-step strategy to make performance predictions using CBSE.

   Hint See CBSE slides and slides on *Performance analysis of component-based architectures*: 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 three kinds of 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) Is it possible to extract behavioral architectural models from a software system for which (a) the complete working source code is available, *and* (b) the source code cannot be compiled and executed, *and* (c) no architectural documentation is available? If no, explain why. If yes, which techniques play a role?

   Hint See slides of Week 14:

   Yes, 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.

   To extract behavioral models from source code one needs a parser for the programming language and custom scripts to extract dynamic relationships (e.g. which objects send what messages to which other objects in what order). The extracted relationships are then incorporated in a model (e.g. a Sequence Diagram).

10. (3) What is the motivation behind *Model Driven (Software) Engineering*, and how does this affect (the activity of designing a) software architecture?

    Hint See MDE/MDA slides. Software products have become very complex, and manual production of large (software) systems is too costly (too much work, too much risk). Software can (partly) be generated from higher-level descriptions, typically in the form of formal models. The main idea is to use models as formal carriers of design decisions throughout the product life cycle.

    Architectural Design is one phase in the life cycle, and, in MDE, architectural models must be defined as machine readable artifacts that drive maintenance and evolution of the end product.