CPSC-5200-01 Software Architecture and Design Mid-term Answer Key

The questions

1. We can define a software architecture model as a collection of boxes and lines. But, if we do this we need to give semantics to those boxes and lines. What do they generally represent? (Hint: they do not represent services, or data access layers, or clients).

   The TL;DR here are two of the three Cs: components and connectors. The longer answer is that they represent some concept upon which there's either agreement that it's important (after "the architecture" has been talked about) or concepts that somebody is trying to convey. FWIW, defining software architecture as a collection of boxes and lines is a horrible definition (but highly concise and effective).

2. Technological decisions impact the economics of your software projects. Besides these, what other factors or decisions may have an impact and why do we care?

   Basically everything else. We need to worry about technical decisions, certainly. But, we also need to worry about time-to-market, ability to hire and retain staff, competitive analysis and advantage, skills of existing staff, leadership, feasibility / difficulty to deliver, known vs. unknown, all of the non-functional constraints, phase of the moon, and whether a Kardashian is currently mentioned in People Magazine. Actually, the phase of the moon probably doesn't impact your project economics, but it's probably a good idea to plan risk mitigation anyway.

3. Is software architecture, and by extension the role of the software architect, compatible with Agile processes? In what ways might it be or not be compatible?

   As we discussed in class, my opinion is that Agile, as a software process, does not conflict directly with those things that we mean by software architecture. The main areas where you'll likely run into a conflict will be in the architect's desire to do additional up-front design and where an architect wants to build something "in case we need it in the future" or "to set up for when we need this later".

   Another thing to remember is that "architecture scales agile". What I mean is that without an overall guide (you've heard me call this a "roadmap"), it's very difficult for a team to know how to make a given decision. Having a roadmap does not mean spending a ton of up-front time "doing" design or architecture, it means taking enough time to think about your ultimate destination, and laying out a few guideposts. I think of this as enabling a "go slow to go fast" model.

4. In his 2000 Ph.D thesis, Roy Fielding argues that REST is a logical evolution of existing architectural styles. How does he "create" the REST architectural style from seemingly nothing?

   He starts, in §5.1, explaining the set of constraints on which he'll derived the REST architectural style. The steps are:

   ```
   5.1.1 - start with the Null Style
   5.1.2 - move on to Client-Server
   5.1.3 - add in the Stateless constraint
   5.1.4 - add in a Cache
   5.1.5 - describe the Uniform Interface
   5.1.6 - combine them all into a Layered System
   5.1.7 - add in support for Code-On-Demand
   ```

5. As a software architect you have a number of tools that you can lean on during the design process. However, one of those tools outweighs all of the others. What is that tool and what makes it so powerful?

   The key phrase I'm looking for here is "refined experience". However, any discussion of an architect's direct experience and learning from other architects was accepted.

   What makes this so powerful is that we can, in essence, bet on all of the work done previously to inform our decisions. It's quite rare that there are new or novel approaches necessary to solving basic software architecture problems (keep in mind that this is precisely why we can have architecture styles and design patterns - the problem has already been solved). However, even in the case when we are faced with a new or novel problem we can look at our experience and the experience of others to find patterns that ***would not*** *work and save ourselves effort. We can also look at previous solutions (patterns) and look at ways to adapt those patterns to fit our new needs.*

6. (bonus) An architecture that addresses mainly how connectors and components are put together runs the risk of introducing an N^2 problem. Solving that using a centralized approach introduces a potential single point-of-failure. How might one design a solution that mitigates these concerns?

   The answer to this is heavily based on the non-functional requirements. If the use cases can handle time-independence (i.e. asynchronous requests are fine and a request / response pattern with a waiting end user aren't a concern) then implementing a distributed queue broker (see RabbitMQ as an example) might solve this problem.

   Another alternative might be to use a centralized distribution mechanism with a local connector fallback. However, this approach becomes extremely expensive to maintain if the endpoints require translation in any form. The connector logic becomes more expensive in that the transformation,

connection, location, and call pattern abstractions all need to be handled transparently (asking the caller to change how the call is made based on availability of one of the N nodes simply kicks the problem higher in the stack).