CS 488 Task 5: Project Initial Architectural Overview and SRS Assembly

The current contents you've developed almost comprise your SRS. In the evolution from the original description to now, things may have gotten out of sync. The assumption is that the project description, user stories, and requirements are complete and correct at this point. If this is not the case, continue to work on them. We have no more iterations dedicated to that part.

Now you're assembling an architectural overview to complete the SRS as follows.

Each user story gets a separate diagram. The flow through the diagram reflects the execution of the user story.

For example, consider this user story:

As a student I want to post my assignment so it can be graded.

Add components that you already know you need, like a webserver. Others you can add as you encounter them. So in this case, the mental flow could look somewhat like this (recall W5H from CSCD 350: *who*, *what*, *where*, *when*, *why*, *how*). You don't have to write it out, but you do have to be able to articulate it when asked.

What is flowing? The assignment.

What's the form of an assignment? A file.

Where does the flow start? On the webpage in a browser on the client's machine.

Who does the submission? The student.

How do they post an assignment? Lower-level details aren't necessary yet, like the web form. It's enough to say they use a control element to find the file to upload, which is somewhere on their filesystem. This introduces their client filesystem into the diagram. It also introduces a clear boundary between their side (the client filesystem) and the other side (the webserver).

What happens next? The file magically uploads by following an arrow (labeled "file") from the filesystem to the webserver, but where to then? There's probably a server filesystem needed, and an arrow to it.

What happens next? There's probably a record of the submission stored in a database, so introduce the database with another arrow from the webserver. But on this arrow it's the record that moves, so label it so. You don't have to detail what's in the record as long as it's clear in your mind that everything you need to store in the database is available in that unit that's following the arrow. The webserver must produce this output; the database must accept this input.

Keep this up until every aspect of the user story is captured. This is why user stories shouldn't be large. If they are, they probably should be broken into multiple smaller ones.

You can envision there might also be an confirmation email going back to the student through a mail server to a mail client (although this would probably be a separate user story because other features likely use it, too).

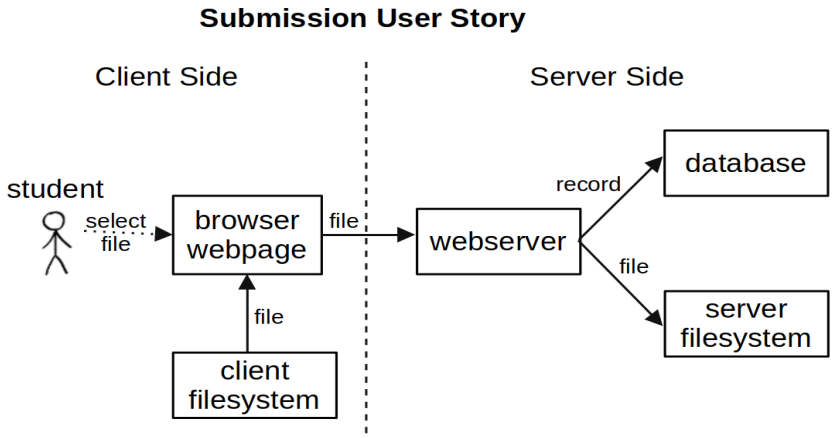
The general form of a flow diagram is an input-processing-output chain or network. Each component is a processing box with input(s) and output(s) (although it's possible to lack either or both). The output(s) become the input(s) to one or more other components until the end of the chain. The processing is magic, as long as it's clear to you what's going on at this point, but you can put some basic description in the box if necessary. At minimum it needs a meaningful label.

Make the diagram readable, generally from left to right, but don't get fancy because this is a draft. Indexing is explained below to make it easier to keep track of things.

Once you're done with the individual diagrams, create a unioned one with all the components and arrows. The data won't be indicated on the arrows because you're not executing a particular user story. This is the architectural overview of the capabilities the system provides. The perimeter is the implied scope of the expected implementation. You should be able to quiz yourself by executing any story on this one without looking at the original one.

If you run into issues, like not knowing how to represent something, it's probably an indication that this is a trouble spot. Do your best to explain what's supposed to happen here, and we'll work through it next week.

Here's a possible diagram for the example above (minus the sequencing):



Note that our user story has the benefit "so it can be graded," but this is not captured here. It would be a different user story with additional components to address the grading part from the professor's perspective. Make sure the stories are complete, correct, and consistent, as well as their corresponding diagrams.

Make sure the architectural overview diagrams capture the sequencing. Put a number on each arc. In some cases more than one order may be possible. It's ok to indicate just one (like the file path then the record path, or vice versa). If there's not a linear flow for some reason, explain what's going on. For example, step 1 is from box A to B, then step 2 is from C to B, and finally step 3 is B to D. Explain how C happens. You might be missing an arc to activate it, even if there's no data going across the arc.

We've avoided any kind of formal indexing for cross-referencing until now. The reason is that a lot of things have appeared, disappeared, moved around, or otherwise changed. Committing to indexing too early just causes trouble because it rarely gets updated properly. There are CASE tools (computer-aided software engineering) to help with this, but that's overkill here. We're adding it now.

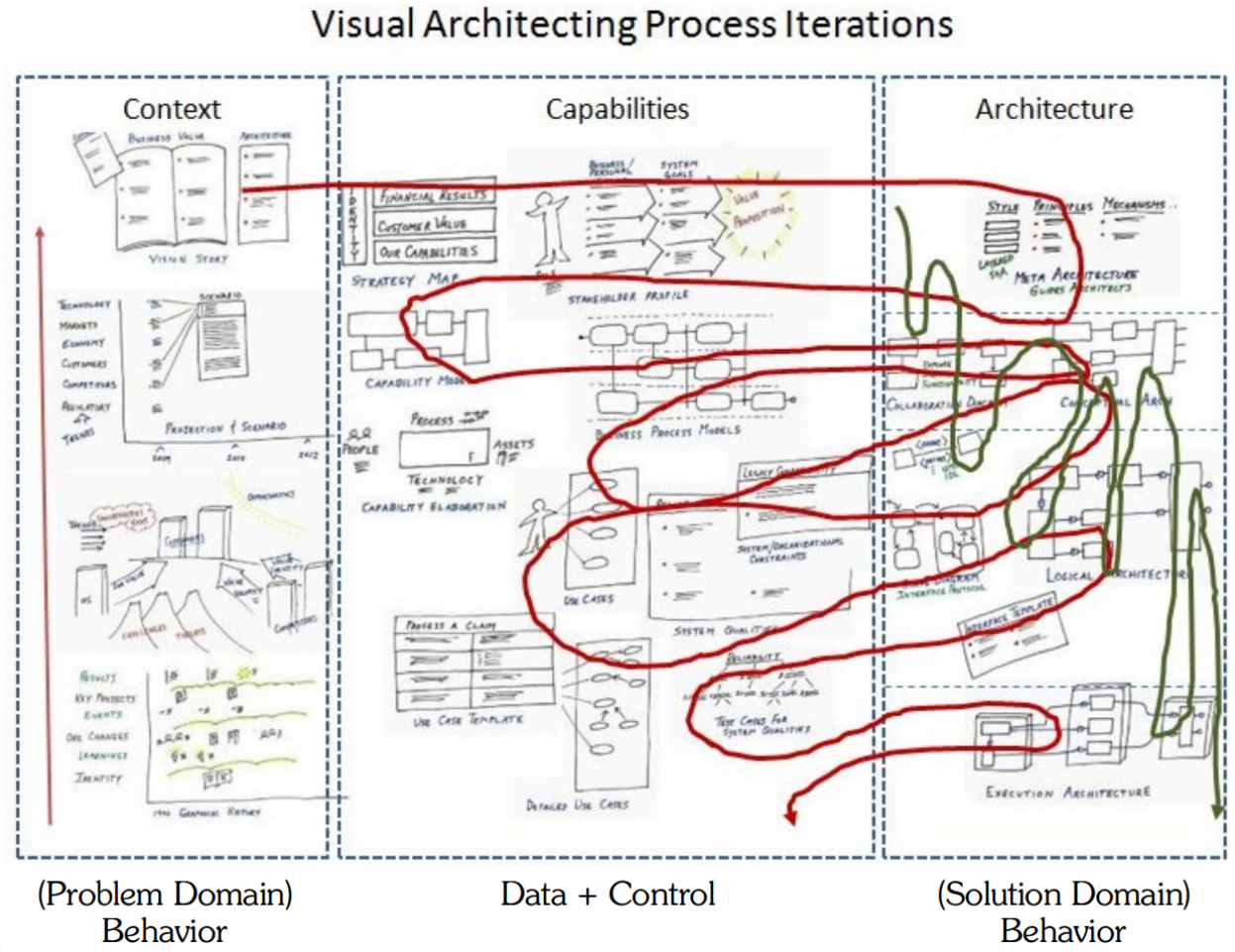
Here's the format:

> Project Description
>
> User stories labeled alphabetically starting with A. If you have more than 26 user stories, your project is probably too large, but you can index them as Z1, Z2, etc.
>
> Under each user story, list the requirements as *letter.number*, restarting with 1 for each user story; e.g., A.1. When you enter these into the reporting system, you'll be issued a different number. The system would ideally record the correspondence between its number and yours, but it doesn't, so you have to here. Whatever it gives you, indicate it here as R# in parentheses; e.g., *A.1 (R4): Requirement title: Requirement description*
>
> Then comes the corresponding flow diagram and and any elaboration.
>
> Finally, the unioned diagram.

Everything should be named for what it is or does, and everything should be or do what it's named. And stay consistent. For example, *location* and *position* are conversationally equivalent, but using two terms for the same thing is confusing. Likewise, don't use the same term for two things. If you have to invent or co-opt a reasonable word, define it in the elaboration.

Unless there are outstanding issues, you can get started on designing and implementing the computational aspects of your solution with OOP constructs and design patterns, etc. Be prepared in briefings to explain what you're doing and why. There should be a plan for how you're chipping away at the requirements. The status reports keep track of what you're doing, but not why you've chosen this order. Be sure to plan carefully and coordinate with your teammates to develop the pieces in an appropriate order. Expect surprises. No plan ever survives the first contact with reality. If things change later in the design or implementation, keep track of them in a copy of your SRS. At the end of next quarter, you'll have to account for these differences as part of the reflection process.

This figure is similar to how we saw it in 350. The Problem Domain is the real world, and its behavior is what happens there. The Solution Domain is the code world, and its behavior is what happens there. These look nothing like each other, but they address the same problem from different perspectives. The Data and Control is the bridge from the Problem Domain to the Solution Domain. These are the capabilities you defined as necessary in the user stories, requirements, and flow diagrams to satisfy the project description.



We also covered this example from 350 in lecture recently. It's very similar to your flow diagrams.