Managing a Student Project with Enterprise Architect – Part 3

Doug Rosenberg -- ICONIX and USC

www.iconixsw.com

Introduction

In Part 1 and Part 2 of this article series we introduced a student project that I’m managing at the University of Southern California Center for Systems and Software Engineering (USC CSSE). USC’s location in downtown Los Angeles is at the epicenter of a lot of bad driving, so we’re attempting a “crowdsourced bad driver reporting system” this semester, and because we need to be really productive, we’re using Enterprise Architect to model the project.

![Figure 1 - the traffic map was littered with accidents as I was driving home from teaching CS590 last week](image)

The statistics on the costs of bad driving to society and to insurance companies are staggering. Among the high points from the article linked above:

- A trillion dollars a year in costs due to vehicle accidents.
- Over 5 million reported accidents a year.
- An estimated 10 million additional unreported accidents a year

With these numbers there’s a strong argument to be made for a crowdsourced approach – it’s the only viable way to get enough eyes on the road. So this semester I’m working with a group of 15 Masters students to build a “proof of concept” system, following the Resilient Agile process and leveraging parallelism, with each student assigned to a different use case and all students working in parallel.
Here’s a reminder of the process we’re following. We discussed the Build the Right System assignment in Part 2, this article will focus on Building the System Right.

1. Plan for Parallelism (identify dependencies and architect for parallelism)
2. Build the Right System (discover requirements, prototype areas of technical risk, and agree on conceptual designs)
3. Build the System Right (carefully review detailed designs)
4. Integrate as often as necessary

Building the System Right

After completing the “build the right system” assignment, we had a few dozen pages of storyboards, requirements, and use case descriptions. In the “build the system right” assignment students produced sequence diagrams, state machines, and activity diagrams as appropriate, and began to move their use cases towards code. In Part 2, we looked at Ankita’s use case for an insurance agent querying the bad driver database. We’ll follow that use case again here.

Conceptual designs morph to detailed designs

Ankita disambiguated her first draft use case by doing a conceptual MVC decomposition, and verified traceability by linking requirements to the controllers. Once the use case had been defined, she proceeded to show the details of how her code will work on a sequence diagram.

On the sequence diagram, we’re showing how the Angular JS pages will interact with the collections that we’ve defined and code generated for Mongo DB. The Angular pages call our REST API to access the database. Our test team will verify that each of the controllers is functioning properly and that all of the requirements have been met.
Storyboards morph into Angular JS pages

It’s a lot easier to write clean code if you’re not also trying to figure out the behavior requirements and the user interface while you’re coding, especially if you’re also trying to learn Angular JS at the same time, which was the case for several of our web app developers. Dealing with multiple learning curves at the same time makes things go slowly and increases the error rate.

Figure 3 - it’s a short leap from a storyboard to an Angular page

So it’s helpful to start from a storyboard. After the visual elements on the screen have been defined, just follow the logic on the sequence diagram, and the rest is “simply a matter of programming”. Being able to review the sequence diagram before a lot of energy has been invested in coding and unit testing helps me as a manager to be able to keep the project on track with no wasted effort. Because our Mongo database and REST API was code-generated, we can just connect the Angular pages to the database using Node JS calls.

Controller testing maximizes QA “bang for the buck”

Using the ICONIX/DDT add-in to Generate Tests from Robustness diagrams allows our test team, Parth and Seema, to quickly identify where the should be focusing their testing effort. For Ankita’s use case, this means they need to make sure the query logic is working, and that reports select and display correctly.

Figure 4 - The ICONIX/DDT add-in generates test cases for all controllers

Further testing is obviously possible and desirable, but controller testing is a great way to rapidly make sure the essential functionality of a system is working.
As I'm writing this article the semester is winding down and we're scrambling to get everything tested with live data.

Here’s Ankita’s query page working with a bad driver report filed under a fake license plate number:

![Query Page Screenshot]

**Figure 5 - the semester is too short for us to do much more than initial testing**

And here’s the resulting report display page:

![Report Display Page Screenshot]

**Figure 6 – but so far our Angular JS pages are working to spec**
Model the results of prototyping

As mentioned in Part 2, Zhongpeng (Android) and Qinfeng (iOS) undertook technical prototyping for continuous video capture to a circular buffer. Because their classmates need to understand how the video buffer code works, and because the requirements for iOS and Android are the same, these two students collaborated on a common state machine that explains how video capture works and how it's integrated with the app's voice commands.

**Figure 7** - we need to understand states and modes for video capture

For similar reasons, we documented how the voice command prototyping works using a sequence diagram.

**Figure 8** - Our mobile app is voice-activated so people don't fiddle with their phones
Wrapping up Part 3

Part 1 of our article series introduced you to the crowdsourced bad driver reporting system project, and in Part 2 we covered Planning for Parallelism and Building the Right System. Our next article will show the final results of the class and hopefully will include a link to a demonstration video on baddriverreports.com.

To recap what we covered in this article:

**Building the Right System** involves building on the conceptual design, transforming storyboards to screens, then connecting the screens to the database by calling the system's auto-generated API that realizes the cloud database. Conceptual MVC decompositions are reworked as solid Model-View-Controller designs using the database collections and Angular pages on sequence diagrams. Testing is initially performed at the conceptual design level, keeping the testing focused on the behavior of the use case rather than at the unit test level.

Our student team is currently in the middle of finishing up their use case implementations, and we've begun posting them to the (under construction) baddriverreports.com website. We've started testing both the Android and iOS apps this week, and are starting to run some actual bad driver video through the various posting and review steps we've developed.

My next article will show the final results of this semester's class.