Vending-Machine-Project
RETROSPECTIVE

Nallie, one of my office mates, being reflective.
Really I should say Nallie being reflected. I'm pretty sure his only thought at the time of this photo was where should I swim to next?

Retrospective For The Vending Machine Project

I've been told that I am my own harshest critic and I say, obviously you haven't met my ex-wife. All jokes aside I will strive to be honest about what went well, what could have gone better and lessons learned. I think the best place to start the retrospective is with the goals for the project.

Project Goals

- Create a working example of minimum Continuous Delivery
- Use TDD to develop the project (in Typescript)
- Use ATDD to develop the project
- Implement project as a real-time system

Create a working example of minimum Continuous Delivery

This by far was the primary goal of this project. All the other goals were created to support this primary goal. Was the primary goal met? I'll let you decide. Perhaps the best way to evaluate this is to explain how the project met the minimum activities required for CD as described on the Minimum CD website and grade myself.

1. Use Continuous integration

a. Trunk-based development
    - ◉ All changes integrate into the trunk
    - ◉ If branches from the trunk are used:
        - They originate from the trunk
        - They re-integrate to the trunk
        - They are short-lived and removed after the merge

b. Work integrates to the trunk at a minimum daily

c. Work has automated testing before merge to trunk

d. All feature work stops when the build is red

e. New work does not break delivered work

Trunk-based development
This project does use use short-lived branches and PRs to trigger the CI server to run the commit stage tests. The PR is just used as a triggering device, no approvals are necessary. The tests are the approving mechanism. If there were more developers on the project code reviews would have been accomplished by pair programming. Some would say this in not really CI since branches are used but I think this approach does satisfy the requirement above. This project uses GitHub Actions to implement CD. These short-lived branches always originate from the trunk (main branch in this project.) If you look at branches you will only see one branch listed (main) unless a short-lived branched has been pushed up to be tested and merged. Once all the commit tests are passing locally the developer pushes the branch up to GitHub then submits a PR. This triggers the the CI server (GitHub Actions) to run the commit tests again. Those tests should pass since they were run locally before the push. After they pass the branch is automatically merged by GitHub Actions and the short-lived branch is automatically deleted. See diagrams in the Developer Flow section in the CD-Deployment-Pipelines document. Also see the GitHub Actions Workflow nodejs-ci.yml
Grade: A (96)
I would have scored this A+ if short-lived branches were not used.

Work integrates to the trunk at a minimum daily
This is really a behavioral discipline. Of course, it would be foolish if the technical parts of CI are not in place to just merge to main and those have been addressed. Even though I was really the only developer working on the project I thought it was good to not be lax on this discipline. This meant if that I had to find natural stopping places where I could push and merge at least once per day (when working, I didn't work every day) that would not break anything that was working before. I found this did greatly change my behavior for the better compared to when I've worked on projects where you can have long-lived feature branches.

When I picked up the work again after a break there was no need for me to read a bunch of notes I had left for myself to remind me where things stood in my feature branch. Since I was using TDD all I had to do was look at my tests to see where I left off. If I had any notes at all it was just usually what the next test should be.
Grade: A+ (100)

Work has automated testing before merge to trunk
As I have described above this was done. Something I'm not clear about concerning this requirement is this does not sound like a CI requirement to me. There's a good chance I'm just confused but this requirement sounds like gated CI which is what I implemented for this project. It is my understanding in CI the code will merge regardless of the result of the commit tests passing or failing. If the tests fail then the build is broken and the pipeline is blocked until the issue is either resolved by reverting the change or fixing it. This is a broken build, the build is red.
Grade: A+ (100)

All feature work stops when the build is red
Since this project is using gated CI when the commit tests fail then the merge doesn't happen. This does not block other developers from pushing and merging. So the way I interrupt this requirement for this project that is using gated CI is if your merge is rejected for any reason it must be fixed so that it will merge before work begins on a new feature. I don't think this ever happened while I was working on the project since I always ran the tests locally before pushing. If it had happened it probably would have been a case of it works on my machine but not on the CI server and I would have fixed it immediately. See diagrams in the Developer Flow section in the CD-Deployment-Pipelines document
Grade: A (96)
I would have scored this A+ if short-lived branches were not used and the CI server had a way to enter into a red state due to a broken build.

New work does not break delivered work
The way I interrupt this requirement is all the commit tests must pass, not just tests for the module the developer is working on. Since all the commit stage tests are run by the CI server on every PR I think this requirement was met.
Grade: A+ (100)
Use Continuous integration GPA: A+(98.4)

2. The application pipeline is the only way to deploy to any environment

Honestly I'm not clear on this requirement. I'm assuming the application pipeline is same thing as the CD Pipeline. Deploy, Release and Delivery are terms that are often used but even Dave Farley admits that he isn't completely consistent in how he uses the words. I think deploy in this requirement means *the technical act of copying some new software to a host environment and getting it up-and-running and so ready for use.* In this project the host is your personal computer. It is a simple NodeJS console application. To install the application you simply download a zip file, expand it, then run the application. The host must have NodeJS installed in order to run the application. If this were a commercial product I think it would have to have an application installer and that installer would take care of installing NodeJS if not already available on the computer. It was beyond the scope of this project to create installers.

In this project after the CD Pipeline determines a release candidate is releasable the immutable artifact (a zip file) is renamed to indicated it has been accepted. To deploy the application one runs a separate GitHub Actions Workflow that lists all available accepted release candidates with links for downloading them. Only team members that have contributor access to the project can download these. Releasing an accepted release candidate is performed by running a GitHub Actions Workflow and providing it a valid accepted release candidate name. This workflow uses the secure box.com api to copy the release candidate to a folder on box.com where it can be download by anyone and installed.

So I think the spirt of this requirement has been met but it could have been better..
Grade: B+ (89)
I would have scored this higher if application installers had been created

3. The pipeline decides the releasability of changes, its verdict is definitive

The CD Pipeline runs all the tests and they pass it changes the version number and marks the release candidate as accepted to indicate it is releasable. I think this requirement has been met.
Grade: A (96)
I would have scored this higher if the CD Pipeline had tests beyond functional i.e. security and performance tests.

4. Artifacts created by the pipeline always meet the organization’s definition of deployable

I came up a little short on this one if I’m honest. I suppose I am the organization and my definition of deployable has been as long as a user has NodeJS 20.x or greater installed the application should work correctly. I’ve been using Apple computers for a long time now and didn’t have a Windows PC around. At some point I thought I should make sure this application not only works on Macs but on Windows machines too. I asked a friend that uses a Windows machine to give it a try and it didn’t work. The zip file would not expand correctly. I had to make a minor change to my GitHub Actions deployment workflow. I think I was using tar to make the zip file and Windows didn’t like that format. I changed to zip and that fixed the issue. Since then I bought a cheap Windows machine so I can check for such things in the future.

So, in order to really meet this requirement I think I would need to add tests that uncompress the zip file on both a Windows machine and an Apple machine and perform a smoke test to see if the application runs. This is doable in GitHub Actions since you can specify the operating system for the runner.

Grade: B (86)

I would have scored this higher if there were tests to verify the application ran on both Macs and Windows computers.

5. Immutable artifact (no human changes after commit)

Requirement has been met. The pipeline only builds the application one time. We are deploying the same bits as the pipeline tested.

Grade: B+ (89)

I would have scored this higher but I’m not crazy about this file app-data.ts that is used to store the version number for the app and some configuration data. After this file is “compiled” the javascript version of it is modified by the pipeline to adjust the version number e.g. dropping the -rc after passing all tests. Also, an end user can directly change it to configure the start money once the app is installed.

6. All feature work stops when the pipeline is red

This is really a behavioral discipline and I did follow it.

Grade: A+ (100)

7. Production-like test environment

I think this requirement was met. I was careful to use a lot of dependency injection that allowed me to isolate what I was testing. At the Acceptance test level I was able to test all the code as a system except for three small parts the index file, the input handler (keyboard) and the terminal output.

The index file instantiated all the classes, started the real-time application and the simulator that interacts with it. In our test environment we take the place of this code.

The InputHandler is just a NodeJS event handler that monitors key presses from the keyboard and calls a function to report them. Since we aren't using a keyboard when running automated tests this is replaced with a test double that can be programmatically controlled.

The terminal class is just a simple abstraction for writing output to the console. When running acceptance tests we can find out what is being displayed in the console by injecting our own test double that stores the strings that would have been displayed.

We really do need a test that makes sure these three things are working correctly. Just a simple smoke test to make sure everything is wired up correctly. And we have it! It is named smoke-test.spec This test actually runs the whole application as a NodeJS child process and sends commands via STDIN and checks the output on STDOUT. Perhaps I could have written all the acceptance tests at this level but it wouldn't have been easy. It was my judgment that it was not necessary and would only add unnecessary complexity to the project.
Grade: A+ (100)

8. Rollback on-demand

In spirit, yes. This application is so simple there's no GitHub Action Workflow for rolling back. To roll back one would simply delete a release from box.com. I always have at least 3 versions available at box.com.
Grade: B (86)
Would have scored higher if a GitHub Action Workflow had been created for rolling back. It's always best to automate all processes like this in order to perform the operation in a consistent manner.

9. Application configuration deploys with artifact

I'm going to say yes the requirement has been met. This application is so simple there isn't really a lot of things that you would think of as configuration. The closest thing we have is a file that contains the app's version number and the start money. The version is automatically changed as the release candidate makes it way through the pipeline. The start money (number of quarters, dimes and nickels in the machine at start up) is configurable without recompiling.
Grade: A (96)
I would have scored higher but as mentioned I'm not crazy about how the app-data file handled. It could be better.

Overall Goal GPA: A(93.38)

What went well for this goal ?

- GitHub Actions automation. Once a PR is used to trigger the CI server the automation takes over.
- The approach I used for production like environments
- Behavioral discipline. Although I was the only developer I behaved as though this was a real project with multiple developers.

What could have gone better for this goal ?

- Tests created to verify the application ran on both Macs and Windows computers
- Application installers created to install on Macs and Windows machines
- Tests created beyond functional tests like performance and security
- Automation should have been created for Rollback on-demand

What lessons were learned ?

- I've learned making a project like this viewable by the public is humbling. It's a lot different than sharing with your teammates who also have skin in the game. Here it is, warts and all folks!
- I learned a ton about how to create GitHub Action Workflows
- I learned a ton about linux shell commands I've never used before like `jq` and `sed`
- I learned that just because you write a shell script and it works on your machine perfectly it may not work correctly in a Github Actions Workflow yaml file. I talk about this in some of the log entries in the Developer's Log
- I learned in order to commit to Continuously Delivery when you start a new project you will really increase your odds of success if you follow Bryan Finster's advice. Especially, "The delivery pipeline is feature 0."

Use TDD to develop the project (in Typescript)

This project was an excellent way for me to improve my TDD skills. It's one thing to practice TDD on small code katas but quite another when you apply it to a project even if it is a tiny one. I still suck at TDD but I don't suck as much as I used to. I do think I'm addicted to TDD now for new projects. I really can't imagine not using it for something new.

What went well for this goal ?

- My TDD rhythm really improved
- TDD really helped me with keeping my code modular and testable

What could have gone better for this goal ?

- I may have overused dependency injection via constructor injection. There are other methods for dependency injection.

What lessons were learned ?

- I learned that London style (outside-in) and Chicago style (inside-out) is a much bigger deal when you start working past simple code katas.
- I learned that I don't think strictly following Chicago or London style is the best approach. I think I like a hybrid approach better. I think Dave Farley jokingly calls this hybrid approach Mid Atlantic.
- My TDD skills are improving but I have just scratched the surface. Basically, I've leaned enough to know I have a lot more to learn.
- No matter how loosely coupled you think your components are early architectural decisions play a big role in what you will be able to easily change later. I decided at the start of the project that I would implement the vending machine as a finite state machine. I also subconsciously made the design decision that I would implement that state machine to poll for changes in the outside world. I've done this before in production firmware that shipped in a product so that felt comfortable to me. I thought I would be able to change this implementation easy enough later if I chose to. I was wrong. From my experiments moving to more of an event driven design later proved it was going to be a huge amount of work. A good lesson was learned. I would state it this way, if a design decision is important don't put it off too long even if you are developing using TDD and doing "all the right things" because you can still paint yourself into a corner.
- It's ok to delete some tests if they are no longer useful. I think I have a natural aversion to deleting tests. It just feels wrong. However, I really think it's best if you stop every once in a while to check if you can move some testing up to a higher level of abstraction if it doesn't already exist there. If you find it is already covered then you may just be able to delete the lower level tests. I believe the same way you keep refactoring your project code to make it better should also be applied to your tests.

Use ATDD to develop the project

This went well I think. Since the kata came with good user stories it was pretty easy to come up with tests at this level. I implemented my interpretation of Dave Farley's Four Layer Approach and that seemed to go well. Using this approach I was able to hide the details of how the test was accomplished deeper in the stack allowing the tests to be more clear as to what they were testing.

Example:

```
    it('should accept a quarter and display its value', async () => {
        await driver.insertCoin(Coins.QUARTER);
        const foundDisplay25Cents = await driver.verifyDisplayOutput('0.25');
        expect(foundDisplay25Cents).toBe(true);
    });
```

What went well for this goal ?

- Implementing Dave Farley's Four Layer Approach. I talk about this in Developer's Log 2024.04.01

What could have gone better for this goal ?

- I should have always written the Acceptance Tests first for a new story. Of course they would all fail because the functionality would not have existed. Sometimes I did do this but I wasn't consistent.

What lessons were learned ?

- When I didn't write the Acceptance Tests first I suffered from knowing too much about the implementation. One advantage of having someone else test your code is if they are good at testing they will not assume anything works. It's a blackbox to them. You can have this same advantage IF you write the Acceptance Tests first. If you write them after you've developed the code using TDD you know too much and you may miss an opportunity to catch a bug. This happened to me on this project. I was lucky that I bumped into the bug just fooling around, uh hum, I mean doing some exploratory testing. I talk about the bug in Developer's Log 2024.04.08

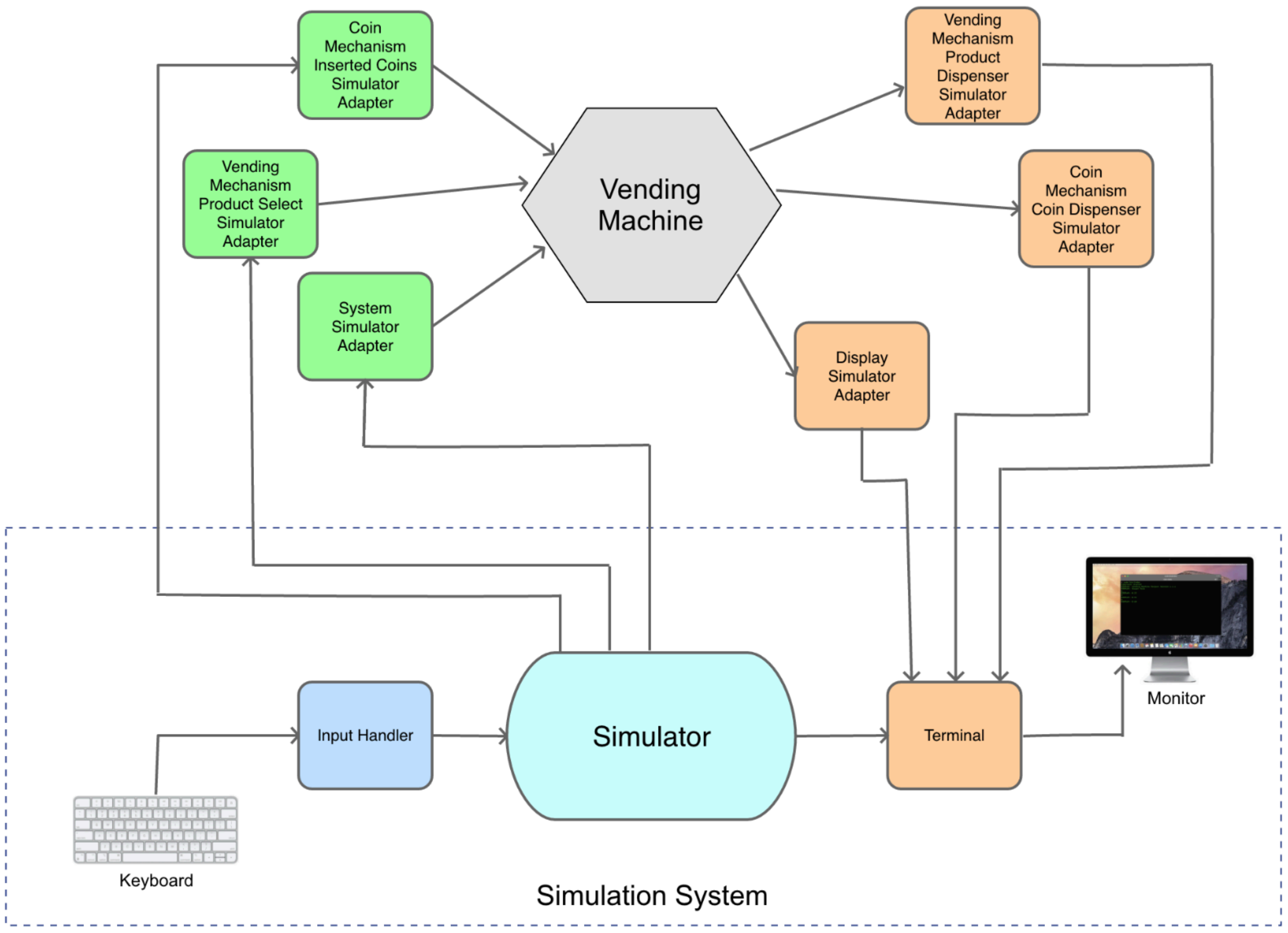
Implement project as a real-time system

The vending machine is a fairly popular code kata and I resisted looking at other implementations until I finished. I have looked at a few of them now and so far I don't think I've found one that implements it as a real-time system. To be fair, I don't think it was the intent of the kata. However, adding the complexity of handling external asynchronous events made this project feel more like a product and less like a coding exercise. The poor man's finite state machine is not glamours but I have good confidence that it works due to how it was developed using TDD and ATDD.

What went well for this goal ?

- When I finally realized that I needed a simulator that communicated with the state machine through its port adapters. I talk about this in Developer's Log 2024.03.07 This way the simulator and vending machine are independent of each other and really they should be. Though it would be economically foolish I think the vending machine code could be used in a real vending machine. Of course new adaptors would be required but the core logic would not. That is if the vending machine only sold the three items at the prices described in the code kata of course. I'm half tempted to prove this to myself by purchasing a real vending machine coin mechanism to connect to my Raspberry Pi.

- Here's a diagram from the design document Harnessing The Power of Hexagonal Architecture in this project. It shows the separation of the simulator from the vending machine.


Coin Mechanism Inserted Coins Simulator Adapter
Vending Mechanism Product Select Simulator Adapter
System Simulator Adapter
Vending Machine
Vending Mechanism Product Dispenser Simulator Adapter
Coin Mechanism Coin Dispenser Simulator Adapter
Display Simulator Adapter
Input Handler
Simulator
Terminal
Monitor
Keyboard
Simulation System

What could have gone better for this goal ?

- Getting rid of the poor man's state machine. I've talked about this above in lessons learned section of the Use TDD goal.

What lessons were learned ?

- I stated it in the lessons learned section of the Use TDD goal. If a design decision is important don't put it off too long even if you are developing using TDD and doing "all the right things" because you can still paint yourself into a corner.