Strategy Blobs -- A Post Mortem

On the 14th of February, 2017, I sat down before my computer with a fairly modest goal: to build a game that could demonstrate my programming skill to prospective employers. I had as guidance little more than the desire to watch objects go down tubes. The following five months were a period of enormous personal and professional growth. In addition to improving my programming abilities, I learned how to properly manage my time, prioritize tasks, and keep myself to a rigorous work schedule. I also developed a set of psychological tools to overcome distraction, anger, apathy, and dispair, that I might get more done and better pursue my passion for games. What follows is a detailed overview of the project: what went right, what went wrong, what its end result was, and what I took away from the experience.

Elevator Pitch

Strategy Blobs is a transportation game somewhere between *Railroad Tycoon* and *Mini Metro*, where players use a small collection of simple but expressive tools to build a complex economy.

Design Goals

- Create open-ended problems, rather than closed puzzles with a single solution, that can be solved expressively through mechanical mastery.
- Let players take a disparate collection of societies and turn them into a complex, interconnected economy.
- Capture the enjoyment of building a simple configuration into a complex system.
- Make it fun to watch items go down tubes.

The Gist of Things

Play occurs on a map consisting of interconnected regions. Each region has a particular terrain type and can support a single construction, either a Society or a piece of infrastructure. To achieve victory, players are tasked with building and maintaining a certain number of Societies of varying types. Each Society produces and consumes certain resources, and can be constructed only on certain terrains. Societies can also complexify and decomplexify by providing them with certain resources above and beyond their usual needs. Societies of higher complexities consume considerably more resources but also provide more valuable products.

In order to develop the map, players can build highways between the various regions, establish Resource Depots to transport resources across larger distances, and construct Highway Managers to increase the efficiency of their network. They can also establish new Societies to produce additional resources.

The bulk of the gameplay revolves around two competing principles: how do I generate the resources I need, and how do I get those resources where they need to go? Players do this by deciding where to place societies on the map, choosing which societies are allowed to complexify, and controlling which resources can go down which highways.

What Went Well or Better Than Expected?

Continual Process Improvement

If the reader has them handy, I'd suggest they take a brief look at the Sprint documents for this project (available online). Take particular note of the differences between Sprint 1 and Sprint 8.

These documents illustrate one of the most important things I gained from working on Strategy Blobs: the ability to work on long-term projects effectively. Before this project, I organized my work under a code-as-you-go methodology, a deeply inefficient way of going about anything beyond student projects. I wanted to do something more intelligent this time around, to figure out how to organize my work to get things done.

Thus I turned to Scrum to try and add some structure to my work, something I had heard about but never really committed myself to. Even though Scrum is focused on development by teams of people, I felt that much of the Scrum methodology would be useful for organizing individual work, as well. It was worth an attempt either way, if only to familiarize myself with one of the industry's standard practices.

However, Scrum is a fairly complex process that takes some getting used to. I realized I wouldn't be able to simply pick it up and immediately become a better programmer, nor would its lessons become apparent until I started using it. Rather, I needed to commit myself to learning Scrum and adapting it to my needs mid-project, updating my workflow as I was working.

And that's exactly what I did. I initiated Sprint 1 from a gestalt of online resources and the suggestions from Jesse Schell's *The Art of Game Design: A Book of Lenses*. As one might expect, the first sprint was rather a mess (you can read about it in the Sprint 1 document). But as time went on, I iterated on that process, improving on it, learning what worked and what didn't, making changes to my workflow along the way. Amongst the changes I made were the following:

1. Removing my tendency to preplan and over-architect.
2. Forbidding myself from re-architecting something in the middle of a sprint.
3. Adding and reasoning about task prioritization to focus on what matters.
4. Cultivating not just a sprint backlog, but a product backlog (of high-level requirements for the project as a whole)
5. Increased focus on ending a sprint with a deployable product
6. Increased focus on iterative development, building things up in small increments rather than constructing grand solutions to problems all at once
7. Better task differentiation
8. More formal retrospection, declaring what went well and what needs to be improved
9. Switching from hourly records to pomodoro time (which are the units of work I organize myself under)
10. Making sure that every task is either player-facing or critical to the design process
11. Enforcing more rigorous work schedules
12. Cursing less in documents I expect future employers to read
13. The importance of optimizing late
14. The proper use of Test-Driven Development and the design of unit tests

You can see this process of evolution by looking at the format and content of the various sprint documents. The first document is messy and hard to read, but as time goes on you can notice a process of continuous improvement: more intelligent sections, better formatting, more efficient use of words, less cursing and emotion, more specific risks and mitigations, clearer writing. By the time I got to Sprint 8, I was writing effective sprints from a clean, concise template with navigable sections and a clear purpose.

And there are even more places where I'm looking to improve my process. Documenting the story, mechanics, aesthetics, and technology of each sprint is one area in particular that I'm looking to improve. Changing the way in which I perform Git commits and use branches is another, less visible change I need to make. And there's always work to be done in specifying tasks and estimating durations.

And all of these changes have paid massive dividends. The first project I attempted when I graduated from college, called Merchant Republics, was a 10-12 month disaster that produced almost nothing of value. I managed in the 5-6 months I spent working on this project (there was some unrecorded preliminary work) to create a far better game than I'd managed in twice that time under less structure. This project is far from perfect, but it's leaps and bounds ahead of what I was doing before.

Even just the act of documenting things helps. There were several times towards the end of a sprint where I was fairly convinced that the sprint had gone poorly, that I hadn't got much done. Yet when I went back to perform my review and retrospection, I became aware of a whole host of changes I'd made that I'd completely forgotten about. Recording the progress of my work has helped me see not just when something's going wrong, but when something's going right. Thus I can use the evidence of everything I've achieved to demonstrate real progress and bolster my morale.

This project has revealed to me not just the importance of having a good process, but of continually improving that process, of working towards better productivity tomorrow while still getting things done today.

Lessions to Take Away

1. Having a good development process can do wonders to improve the outcome of even small projects.
2. There are always ways to work more efficiently, and we should be constantly on the hunt for them.
3. The best way to improve your process is to rigorously document and retrospect on what you've been doing, that you can get a better sense of what actually happened and what actually needs to be changed.

Reasonable Audio Design With No Prior Experience

It was sometime around Sprint 7 that I finally admitted I needed audio. I came to this conclusion with some concern, as I had no prior experience in designing sounds. I'd never made a game that played back any sound at all, let alone had a cohesive soundscape that provided player feedback and UI crunch. The only thing I knew about audio was its difficulty and complexity, so I figured that all but the simplest audio tasks would be beyond me.

Nevertheless, I am aware of the importance of sound to how a game feels, so I committed myself to the task. And while the audio I managed to cobble together was nothing to write home about, it was surprisingly acceptable for a first attempt.

I decided that I needed to set my sights low. My sound design needed to provide response to player actions and reinforce certain concepts. Being pleasant to listen to was a nice thing to have, but beyond my capabilities and thus not a priority. With that in mind, I focused first on audio surrounding the UI: button clicks, toggle clicks, and the opening and closing of menus. I later went into basic commands and important activities for various critical objects: the creation and completion of Construction Zones, the complexification of Societies, and the destruction of various entities.

I spent a considerable chunk of time searching for appropriate samples that I could modify to suit my needs. I wasn't confident in my ability to work an audio processing suite, so I stuck with unmodified samples passed through Unity's built-in Audio Mixer tools. This made finding good samples all the more important.

But I also wanted not to buy anything, so my samples needed to be free, either under attribution or the public domain. This limited my options substantially. After hours of searching I ended up finding a few good samples, but not enough to cover the diversity of sounds I needed. Nor were the samples I found immediately appropriate to my design. Thus I needed to pass them through Unity's Audio Mixer, to modify and differentiate them.

I began studying the various audio effects in my toolbox, imagining how I might use them. While I didn't get a full appreciation for the nuances between, say, wetmix and drymix, I did manage a working understanding of the Echo, Flange, Lowpass, Highpass, Duck, and Pitch Shifter effects. Not enough to implement these effects or theorize about them, but enough to filter my samples.

After a lot of fiddling, I managed to build a reasonable suite of sounds. I modified a barrel-breaking sample into a reasonable object destruction noise. I adapted a screen-sliding noise designed for mobile applications into both a menu entering and a menu leaving sound. And I was able to apply my button-clicking sample not just to buttons and toggles but to the drawing of highways. I also used a positive-sounding UI sample to advertise the complexification of Societies and the completion of Construction Zones, differentiating between the two through pitch and volume

While the audio I created isn't about to win any awards, it does its job reasonably well, all the more so given my prior ignorance. That I was able to build anything at all was a pleasant surprise.

Lessions to Take Away

1. Sound is very important.
2. You can do a lot with four samples, a collection of mixers, and a bit of experimentation.
3. When applying yourself to a task you're not very familiar with, having realistic scope and humble expectations goes a long way. It's better to successfully iterate upon a simple design than to fail at something grandiose.

Targeted and Intelligent Performance Improvements

While I've kept on eye on performance in the past, it's never been a major consideration. That has not stopped me from spending countless hours fiddling senselessly in the name of performance. Performance had, until this point, been a nebulous task that I attended to without much of a plan.

I decided early on to apply a more rigorous structure to performance improvements than I had in the past. I decide on a simple procedure. I would initially solve any problems quickly and comprehensibly, caring not for the performance of the end result. Then, when performance issues arise, I would profile the program, identify performance bottlenecks, and address only those bottlenecks, leaving the rest of the codebase alone.

I also made performance a priority only at the end of the project, figuring that it made no sense to optimize code that might get removed. This meant that for much of the project framerates would drop below 30 on moderately complex maps, despite the triviality of my game's graphics. Yet I held off on optimization, prioritizing other tasks (like making sure there was a game at all) over the framerate.

When it came time to improve performance, I attended to the issue with more care and intelligence than I have in the past. I sat down and profiled my game, using some of Unity's tools to identify exactly which sections of my codebase were slowing things down. And good news that I did.

There were several areas in my program that I thought were going to be performance bottlenecks, most of them revolving around ResourceBlob, the class that manages the resources that flow throughout the map. ResourceBlobs are created via a factory, ResourceBlobFactory, that doesn't implement object pooling. ResourceBlobs also engage in some reasonably complex movement logic, which I thought might be incurring slowdowns as well.

As part of my mantra of comprehensibility, I decided not to implement object pooling in any of my factory classes until I needed it. As performance declined, I assumed this decline emerged from this decision. Every four seconds, each Society in the game produces and consumes ResourceBlobs, and on large maps that can mean the creation and destruction of dozens and management of hundreds of objects.

However, when I profiled my build I noticed a few things. For starters, ResourceBlob wasn't a significant contribution. Neither was ResourceBlobFactory. Bigger than either of them was MapNode, which contained an expensive Update method that was supposed to exist only in the editor. And yet even MapNode was only ~11% of the time spent. By far the biggest contributor to poor performance was a class that I had forgotten about called BlobDistributor. This class performs the logic by which Highways are given blobs to transport. The single instance of that class was taking up the majority of the program's total processing time every frame, even when graphics were taken into account.

Thus I had a clear place to work, thanks to my profiling. And over the course of a few hours, I managed to so thoroughly optimize the way that BlobDistributor worked (caching queries that it needed, removing unnecessary or redundant code, applying Linq queries correctly) that I dropped it to around 10-15% of total processing time and brought the framerate pretty consistently above 60 on complex configurations.

To me, this result is a validation of my optimization plan. It simply doesn't make sense to optimize until you need the performance, and then you should only optimize the places that are suffering the most. Doing that allowed me to avoid hours of work that would've added nothing. I didn't need to concern myself with object pooling, more efficient ResourceBlob movement, inefficiencies in the UI, or any of the other slow code I wrote because it didn't matter in the long run. It's hard for me to say for certain how much time I saved not worrying about performance until it mattered. But given the number of times I repeated queries to guard against errors and save time, it must've been several days at the least.

Lessions to Take Away

1. There is no reason to worry about performance until the build is stable and it becomes a priority.
2. Profiling is critical. Intuition is a nice place to start, but it's a faulty tool and should always be backed up by evidence.
3. You won't know the best way to ensure optimal performance ahead of time. I wouldn't have known to cache MapNode/Highway adjacencies until I checked their performance.

Winning the Psychological Game

A few months after graduating from college, I began work on a fairly ambitious personal project called Merchant Republics. It went poorly. There were a lot of problems with the game I was trying to make, the process I was using, and with my skill at programming. But though I had an impossibly large scope, a code-as-you-go workflow, a complete lack of player feedback, poor prioritization, and a paucity of player-facing features, none of these things destroyed that project. What destroyed Merchant Republics was a collapse in morale that led me to abandon it as a hopelessly broken aspiration.

It was a self-compounding problem. As weeks turned to months and I started gaining the wherewithal to notice the project's issues, I began to lose my passion for it. This made it harder for me to put a solid workday into it, made it more likely that I took days off or cut days short. This slowed progress, exacerbating the project's problems and reinforcing my disdain for it. That formed a positive feedback loop that ultimately led me to kill the project. Merchant Republics was broken due to inexperience, but it died for psychological reasons evident not just within myself, but across a large segment of humanity.

Time passed after I abandoned Merchant Republics. I moved onto another project, a platformer that I called Bouncy Cubes. It was as far a departure as possible from the inward-facing economics simulation that Merchant Republics had been. Another project with problems, but it went considerably better. When it became difficult to reason about that game's level design, I switched over to yet another project, a strategy game that was the pregenitor to this one (which gave it its original name, Strategy Blobs). That, too, went ok, though with problems.

Yet through that whole time, I remembered what happened to Merchant Republics. I remembered how awful it was to watch something I cared about, something I had spent hundreds of hours working on, fall to pieces because I didn't have the heart to save it.

As Merchant Republics was failing (it's hard to say precisely when this started) I began searching for remedies to innoculate myself against this sort of collapse. At first I began correcting my sleep cycle, to add a more consistent structure to my day. Then I found meditation, which empowered me to hold useless emotions at arms' length. Eventually I started looking into ways of improving productivity and managing my life more intelligently. This led me to the discovery of Action Programs and the Pomodoro method, a way of structuring life priorities and work schedules.

And then, on the 14th of February, 2017, at 8:02 AM, I began doing something more intensive. I began keeping a record of my work. Divided into pomodoros, I declared every task that I was performing, the starting and stopping times of each work cycle. For each, I wrote up a brief description of what I had done, what problems I had faced, and what ideas I had to solve them. I recorded the duration of every break, as well. I noted in blue every time I had finished a task, and in red every time a task or a work cycle had gone awry. And then, the following day, at 8:04 AM, I dusted off Strategy Blobs, pulled out *A Book of Lenses*, and began the preparatory work for Sprint 1 of this project.

This collection of activities: a regular sleep schedule, meditation, an Action Program, the Pomodoro method, and recording my work, has for the first time in my life given me actionable tools I can use to fight apathy. I can now combat the loss of spirit that so often grips people when things get difficult and they start to struggle. It has allowed me to check my intuition against reality, that I can know what's actually going on. It has allowed me to see how far I have come and to address the problems I face in my projects with care and rigor. In short, it has empowered me to combat the psychological issues that led to the death of Merchant Republics.

And there are times when this meticulous attention to my own psychological state has been of enormous value. It has kept me from wandering off into useless territory. It has kept me from pushing myself into exhaustion that I could keep working effectively. It has kept me honest about what I have accomplished, about when I have made mistakes. Enforcing regular breaks forced me to step away from problems and return to them with fresh eyes. And when negative emotions arise, threatening to derail a day, I have the tools to let them go and return to my work, to return to what is true and what matters.

Looking back at how faulty my memory is even of previous weeks makes me wonder how bad Merchant Republics really was. Was my disdain for the project even justified? I don't know. I don't have a record of what I accomplished, or what needed to be done. Maybe I was in the process of digging my way out of that hole. Maybe Merchant Republics was coming together, and I just wasn't self-aware enough to realize it. Or maybe it was a catastrophe greater than I had expected. I will never know.

Over the past year, I have come to understand just how critical the psychological game is. So many of the problems I've encountered during this project can be solved with three basic techniques: calming down and regaining focus, stepping back and thinking things through, or walking away and focusing on what matters. I'd imagine this is even more accurate for team activities, where organization is a vastly more complex endeavor and low morale in one person can infect the whole project. For anything can be learned or adapted to, but only if you have the will and the mental wherewithal to do so.

Lessions to Take Away

1. Loss of morale is perhaps the most crippling thing a project can face. Any project can overcome its external obstacles, but no project can survive a collapse of spirit.
2. Managing one's psychological health is critical to being productive in any creative endeavor.
3. Emotion often disrupts our understanding of reality in dangerous ways. The best way to contradict that drift is to keep a careful record of what's actually happened, that we might check our emotional state against reality and know whether our intuition is correct or misleading.

What Went Poorly or Worse Than Expected?

No Playtesting

By far the biggest problem with my development process was a lack of playtesting. Given how critical of a component of player feedback is, this can be considered nothing short of a critical flaw. It was also a problem I was aware about through the entire development process. Most of my sprints list "Did not playtest" as something that needed to be improved, and "Failing to playtest" as a risk that neede to be mitigated. Yet it was not fixed.

So what happened?

There were many excuses. Since this project was intended to prove my skill at programming and not at design, I could always brush off this concern as irrelevant. I'm here to prove I'm a programmer, after all, not a designer. Yet that excuse falls apart when one considers the amount of time I spent attending to design tasks. If I wasn't focused on the design, why did I spend dozens of hours working on it? If I wasn't concerned about the design, why was my product backlog filled with design goals? The excuse isn't valid. I invested too much of myself into the design of this project for that to be true.

Next we might claim the existence of an external barrier of difficulty. I now live in a fairly small town, geographically separated from an already small social network. There's no clear way of gathering the dozens of players it would take to playtest the game properly. A more reasonable excuse, but one that also breaks down upon inspection. I have faced many difficulties during this project. I began it unsure what I was doing, with a whole new set of techniques (Scrum and TDD being the biggest) that I needed to grapple with. It would've been a challenge to collect a robust number of players, but no greater than building the game. I could've solicited online feedback. And I did have options, times where I could've playtested the game in front of somebody and simply didn't.

What really happened here was a psychological battle I didn't win. It boils down to three things: my strong tendency towards self-sufficiency, my desire not to impose myself on others, and (most substantially) general social anxiousness. These three things combine to make asking for playtests a psychologically difficult activity, one rooted in struggles that run deep into my personal history.

These are large problems, to be sure, and problems that I've been working through for years. They are also things that threaten to compromise my development process in more ways than a lack of playtesting. For while I am an excellent communicator and I can grow and maintain social bonds that already exist I struggle to create new relationships. Or, more accurately, I'm decent at making relationships if only I can get past the moments of first contact.

But they are just problems, like any other, and they are things that I've been addressing. For while I've been working on this project, I've been actively trying to extend my social network and hone my social skills through the use of an unorthodox technique: running campaigns of tabletop RPGs online (D&D and the like).

And I say this in all seriousness. For those not familiar with the discipline, being a Game Master (or GM) of a campaign is a very socially demanding task, moreso when done online. In order to be a good GM in this way, I must:

1. Make a pitch for the particular campaign I've run and send it out into the aether (which normalizes expending creative effort in a very public forum)
2. Pick a group of complete strangers who I think would get along well (which tests my social judgment)
3. Gather that group of strangers together and get them all working towards the same creative goal
4. Balance the ideas, needs, and desires of every person at my table (including me) so that every person gets their chance to shine without ruining other peoples' needs and contributions.
5. Engage in extensive improv, presenting an interactive experience filled with characters, settings, actions, and themes in front of said people, all in real-time, with immediate feedback from the other players
6. Resolve any social drama that emerges by mediating conversations and resolving conflicts
7. Simultaneously empower people to act creatively and with agency while guiding them down mutually beneficial paths, all without the use of compulsion (or else they'll just leave)
8. And do all of this to try and create a meaningful experience together that incorporates diverse viewpoints and that everybody at the table enjoys.

That doesn't even mention basic social skills that are in play during this activity: talking to people to learn about them, managing a conversation, listening intently to understand what's going on, communicating clearly and concisely to psychologically diverse individuals. GMing is an intensely social activity, one that I'm using to improve my social skills and address some of the problems I've been facing.

And it is paying dividends. It's already getting easier to manage these sorts of social situations. It's already getting easier to put myself out into the world at large, showing individuals the results of my creative efforts and asking for their collaberation. I'm already addressing the dynamics that made playtesting difficult for me during this project. And there are many other avenues open to me to work on this weakness, which will only become easier as GMing helps me normalize intensive social activity.

Lessions to Take Away

1. I avoided playtesting for social and psychological reasons.
2. I can address these problems by honing my social skills via GMing, a task I'm applying myself to with great enthusiasm.

3. Playtesting is critical and mandatory for any serious design effort. And since I am serious about good design as well as good code, that means it is a mandatory activity for me. I must, from this point on, do the hard thing and get my game in front of people, whoever they might be.
4. I will tolerate the lack of playtesting in this project, considering it as a learning experience. I will not tolerate it in any other.

Inefficient and Confused Testing

This project was my first attempt at performing Test-Driven Development, where I led the programming of modules with the tests that proved their efficacy. This method proved to be more time consuming than I'd anticipated, and through more than just my lack of experience. There were technical concerns that made unit testing slower than it should've been.

Beyond my lack of skill, my difficulties primarily arose from two aspects of Unity's architecture: the MonoBehaviour, and Unity's internal serializer.

Unity is a component-based engine, where objects are assembled by linking together units of behavior. These collections of components are mediated by the GameObject class. Components themselves are managed via a superclass called MonoBehaviour, from which all components must derive. MonoBehaviour is, itself, an unusual class. It straddles the boundary between the scripting front-end that developers have access to (a Mono environment interpreting C#) and Unity's internals (written in C++, ostensibly). Because of this, MonoBehaviour forbids the use of the New keyword to instantiate objects. Instead, one must call UnityEngine.Object.Instantiate() on an existing GameObject or MonoBehaviour or GameObject.AddComponent<T>() on a particular subtype of MonoBehaviour.

This means that substitution and mocking libraries (Like NSubstitute, the one I tried to use) tend to fail whenever you try to substitute in a class deriving from MonoBehaviour.

MonoBehaviours also provide other problems. While Unity contains a unit-testing suite, its baseline activity often acts unusually in the presence of MonoBehaviours. MonoBehaviour provides for the Update pattern by allowing users to declare an Update() method. However, standard tests either don't run the Update method or else run them only one time, and it's unclear to me how much time the system would claim had passed. This means that traditional testing methods have trouble checking time-sensitive behaviors on components.

There are certain tools that allow you to get around this, particularly the UnityTest attribute, but they have their limitations. While UnityTest can run multiple Update loops on your MonoBehaviours, it doesn't help create mocks.

If you wanted to do efficient TDD, then, one solution might be to pull away from the component model and derive fewer classes from MonoBehaviour. But doing that runs us into the other major issue: Unity's internal serialization.

In order for Unity to handle the transfer between various states within its editor (going from editor to play mode, instantiating prefabs, persisting data within scenes across multiple sessions, and many others) Unity serializes all of your scripts and all of your objects. This normally wouldn't be an issue, except that Unity serializes things differently than, say, BinaryFormatter in C#. Of particular note is how Unity's serializer handles references. For MonoBehaviours and other objects deriving from UnityEngine.Object, Unity serializes references just fine. But for every other type of object, Unity discards references, instead instantiating new instances of whatever object existed beforehand. That means that if you need to establish references between two objects (say one has a dependency on the other) you must make them both derive from UnityEngine.Object, which has the same instantiation and mock incompatibility problems as MonoBehaviour.

My design, whether by necessity or incompetence, requires just those sorts of connections in fairly large numbers. A Society must know about the MapNode it's upon, the ComplexityLadder it's climbing, and the ComplexityDefinition upon which it's basing its behavior. MapNode needs to store a reference to its BlobSite. Highway needs to keep track of its MapNode endpoints as well as the BlobTubes it uses to transport things. And all of these things need to persist those connections between runtime environments and work sessions. Thus Society, MapNode, ComplexityLadder, ComplexityDefinition, Highway, BlobTube, and whatnot all must be MonoBehaviours or else similarly derive from UnityEngine.Object, preventing them from being automatically substituted.

This became a problem when it came time to unit test these modules. I needed some way of separating the unit under test from the implementations of its dependencies, but couldn't use a substitution library. My solution was to perform the mocking manually. For each module, I created an abstract base class for it, essentially an interface, that derived from MonoBehaviour. For instance, my Society class derived from SocietyBase, MapNode from MapNodeBase, BlobHighway from BlobHighwayBase, and so on. I then forbade the use of the implementation classes (Society, MapNode, and BlobHighway in these examples) from being referenced outside of their modules. Then, when it came time to mock up an object's dependencies, I simply created a new subclass of the abstract base (MockSociety, MockMapNode, MockBlobHighway) and implemented the mocks as I saw fit.

This led to several problems. For one, it took a lot more time. Instead of using a single line of code to set up the results of calling a method, I had to create a new class and then implement the method myself. Often times I would end up copying over the implementation's code because I didn't want it to act any differently. At others I would have to piece together some acceptable pseudo-logic and configure it within the test. This also separated the logic of the mock from the test using it, and made it a lot harder to provide alternate mocking logic.

And since different tests in different modules needed different mocking behaviour, I ended up with several mock implementations of most of the major interfaces. This meant that every time I changed any of the abstract interfaces, I had to change all of the mocks as well. It also became more tedious to ask certain questions (like how many times a particular method on a dependency got called, or what arguments were passed to it).

This lack of a substitution engine compounded with other problems. When I designed my unit tests, I hadn't yet learned how to create test cases (and Unity makes those harder to build, anyways) or provide data to run the tests multiple times. Thus I ended up hard-coding values that were almost certainly insufficient to prove what I'd set out to demonstrate. I'd imagine that would also have made it harder for others to extend and make more rigorous the tests I'd developed, should it come to that.

Nor was it clear what constituted a unit of my module. Should I test every side-effect of a method in a single test, or test each side-effect in different tests? What constitutes a unit? Is an object and its factory a unit, or are they separate?

My code may also have been too tightly coupled to itself, as many of my tests have considerable setup overhead that makes heavy use of the Copy/Paste Antipattern. I could've done more to collect that code into standardized setup methods, though it remains to be seen if that should've been necessary in the first place.

And then there are issues of test coverage. Most of the user-most edges of the UI are completely uncovered by tests, since that code is simultaneously difficult to test, of minimal value, and subject to frequent changes. It's also not clear if I should've done more large-scale integration testing or whether testing the units was sufficient.

I think there are two general ways in which I could've avoided this problem. One was to devise some way of getting around the serialization issue. For instance, figuring out how to store dependencies as their abstract base classes while revealing them as interfaces (via the Private Class Data pattern perhaps) may have allowed me to circumvent MonoBehaviour's peculiarities and use my substitution libraries properly. Or perhaps I could've injected dependencies at the beginning of each runtime environment, rather than keeping them persistent.

It might also be that my architecture is at fault. Maybe my components shouldn't have this many dependencies. Maybe I should be using the Humble Object pattern more often, separating all the business logic into smaller stateless classes and making the central object simply dispatch method calls. That way I could test, say, just the consumption logic of Society (implemented as a normal C# class) in isolation of any dependencies, then check to make sure Society calls into that logic properly. That might be a smarter way. It would at least make the manually implemented mocks a lot simpler.

Lessions to Take Away

1. Testing is hard. It needs to be done carefully and with intent.
2. Unity makes certain aspects of unit testing more difficult than they ought to be, though there might be ways of working around those challenges.
3. The slowdowns I experienced while doing unit testing may have been a result of poor architectural decisions, or else had architectural solutions to them.
4. The best way of solving dependency mocking is to simply have fewer dependencies.

Ad Hoc UI Development

I take good communication very seriously. Thus I take the design of good UI very seriously, even if I'm not particularly good at the discipline. I've spent a considerable amount of my time trying to make my user interfaces work well and follow good programming standards. Unfortunately, that is not an easy task to manage.

During a previous project, I attempted to create a fairly general and robust system which I could use to program UI intelligently. That attempt led me to a finnicky and brittle system that was unpleasant to work with. Thus I came into this project seeking a less stringent method for designing UI. What I ended up making fell, in my interpretation, dangerously into the realm of ad-hoc programming. I think my methods would've led to considerable problems on a larger problems.

My struggles seem to come down to a few questions:

1. How do I intelligently switch from one screen/view/window/panel to another and capture user input only for the active panels?
2. How do I tie UI elements to their corresponding objects in the codebase without creating lots of dependencies?
3. How do I regression test my UI?

In Merchant Republics, I had addressed the first question by using a hierarchy of nested Finite State Machines that activated or deactivated various UI states and UI panes based on certain transitions. That system was simultaneously too rigid to make even simple UI elements without hassle and too configuration-heavy to easily maintain. For this project I used something far more informal, which essentially amounted to an ad-hoc FSM in some places. I ended up hard-coding every transition, in ways that were often inconsistent with each-other and led to a lot of code that was little more than boilerplate. There was a lot of simple but time-consuming programmatic labor involved in connecting buttons to panel transitions.

I've got to imagine that there are clean, expressive, and rigorous ways of defining the navigational commands for a complex UI, but I've yet to find an example of how to do it. One of these days I'll need to sit down and devise a system that uses Unity's built-in UI system to add sensible menu, screen, and panel transitions. Or I'll need to find a system that already does that for me.

I also used a lot of boilerplate to tie UI elements to particular commands within the game's internals. I ended up dispatching these commands through a set of intermediary classes in the Core module, in order to try and separate UI from game logic in a very severe way. I figured that would allow both the UI and the underlying logic to change independently of each-other, which seemed like a desirable state. But despite those intermediary objects (one for each major module in the game, supporting all of the available user commands) and a corresponding one going in the opposite direction (UIControl and its companions) I still had a lot of repetitive connective code hooking up buttons to commands and displaying the information of a particular object. I can't imagine that any of it is particularly maintainable.

I also know that none of the UI closest to the user is unit tested. Providing tests for those elements seemed like such a waste of time given my struggles with mocking, and yet without code coverage I can't guarantee that my UI works properly.

I'm not sure what would've been better. I'm imagining some expressive and intuitive tool for connecting screens and buttons with methods in the codebase, all without having to build classes with trivial code that pollute namespaces and waste time. Yet I can't imagine what it would look like or how it would function in a concrete manner. Maybe that just means that a less ad-hoc UI is a pipe dream. Or it could mean that I'm not imagining or framing the problem correctly.

Lessions to Take Away

1. UI code is usually not difficult, but it is repetitive and hard to test.
2. There must exist a way of defining the flow between UI elements via data (and ideally a user-friendly tool) rather than frustrating boilerplate code.
3. If I really wanted a robust UI system for Unity, I'd need to design it in isolation as an entire project, but I would also need to specify what exactly I'd need from it beyond "something better and more convenient."

Cross-Platform Deployment

I had originally intended to deploy my project to PC, as that's the platform I'm most comfortable developing in. But at some point I got it in my head to try for a web application, as well. I figured it'd make my project easier to share if people could access it through a browser. It also allows a larger diversity of machines to run it, that a prospective employer with a PC could play, as well.

This turned out to be a poor decision, especially since I made it so late in the development process.

For starters, the amount of time necessary to build a WebGL application was quite substantial. While I could build a standalone executable in around 20 seconds, it took 5-10 minutes to deploy a WebGL build. That overhead made it a lot harder to test changes.

Then I ran into problems with loading maps. For a variety of reasons, I was saving maps not as scenes or assets but as XML files serialized an deserialized via a DataContractSerializer. I was then accessing the file system in order to find, deserialize, and produce lists of available maps. This didn't work for my WebGL application, as browsers do not provide scripts file system access. I spent quite a chunk of time working around that issue by using the built-in WWW class to stream in my maps, which added some extra complexity but was ultimately fine.

Once I actually started playing the maps, I noticed that my camera controls were completely off. It turns out that Unity's Input Module, designed to provide standardized input on all systems, works differently in each of the browsers. Axis values were much higher while running out of Chrome, which made the camera move at an intolerably fast rate. Even worse, the same file system access problems that blocked loading maps also blocked loading and saving particular sessions of play, and the system by which I was keeping track of which maps had been won. These problems presented a greater challenge than map loading because this data needed to be saved on a per-system basis. Evidently there are some ways of doing this (via a system called IndexedDB) but documentation on this subject was sparse.

Then I realized the virtue of deploying my game to a browser was also a serious danger. Rather than worrying about how the game functioned and ran on a PC, I now needed to worry about how it functioned on every web-enabled device: PCs, Macs, Linux systems, smart phones, tablets. I had access to almost none of these devices, which precluded any sort of testing. These systems also have very different input models and performance considerations that I could not test. The realities of the web application Unity builds also meant I couldn't map the Escape key to anything, since that popped the web application out of full-screen mode.

As one would expect, considering this one extra platform opened up a whole can of worms that I was, at that late stage in the development (maybe a week from finishing up the final sprint) unprepared to devote resources to. Thus after a few days I decided to abandon the idea of the WebGL application, instead focusing my time on polishing the PC build.

Lessions to Take Away

1. Cross-platform development adds a whole host of complexity and extra considerations, even in excess of what I was expecting.
2. Such decisions need to be made early on in the development cycle and must be treated with gravity, as they are considerable tasks even in engines (like Unity) that promise easy cross-platform deployment.
3. It's better to deploy a good product to a single platform than a mediocre one to many platforms.

Polygon Detection and Delving Into Graph Theory

Early on in the development cycle, I had decided to represent maps as undirected graphs, with each node representing a location upon which something could be built and each edge a path down which resources could flow. That decision was made for largely mechanical reasons, to make it easier to reason about the flow of resources. It was not, however, made with the aesthetic properties of the game in mind.

When it came time to work on the aesthetics of the map, to turn it into something more than a collection of spheres connected by lines, I was left with something of a problem. I had established terrain as a concept, and thus wanted to display the terrain to players. I also wanted to have the map look like a cohesive entity, like a continuous plane of territory without any holes in it. Fancying myself a mathematically minded individual with some expertise in graph theory, I decided to create an algorithm that would define and color regions on the map for me. that way I could take any reasonable map topology and convert it into a cohesive terrain automatically.

I came to understand the problem as follows. I had before me an undirected planar graph with no overlapping edges, about which no other assumptions could be made. I needed to take that graph and draw non-overlapping polygons around each one, with no vertex in any polygon being greater than a certain distance from the node it was surrounding. And I needed these polygons to leave as little empty space between them as possible. For instance, if two map nodes were sufficiently close (less than twice the maximum vertex distance), then we'd expect both of their terrain polygons to touch each-other, share some number of edges.

I spent some time trying to come up with a reasonable algorithm for solving this problem and made considerable progress towards it. But as I delved deeper and deeper, the problem became more and more complex. At first I thought I could detect adjacencies between nodes by checking their edges, which turned out to be a false assertion. At some point I came up with an algorithm that worked by subdividing the polygons formed in my non-overlapping planar graph. But that required finding all of the polygons in a non-overlapping planar graph. So I found a paper online that presented such an algorithm...only to find it was a composite of several more algorithms. So I found the papers for those, or similar papers besides, to read and understand the mathematics at play.

And that was when I learned that I'm not nearly as good at graph theory as I thought I was. Because suddenly I was confronted with a deluge of unfamiliar technical terms: cycle basis, incidence vector, gaussian elimination, GF(2), Eulerian Path, circuit rank, the Bentley-Ottmann algorithm, the Floyd-Warshall algorithm.

So I sat down and started learning, for hours, digging into this base of knowledge that I had been unaware existed, trying to understand an algorithm that was a simplified version of an algorithm used in a polygon-detection algorithm that I was to use in my terrain-drawing algorithm that I eventually realized wasn't a complete solution because it failed to take into account nodes that weren't completely surrounded by polygons within the graph, that I then had to ask an entirely new set of questions about finding the outer edges in an underdirected planar graph with no overlapping edges so I could use that information to define some points by some policy to make sensible polygons under a certain policy, that would likely need to be controllable by whatever designer was trying to shape the map, and would also have to run quickly so they could iterate on things, which was a problem with the cubic or quartic growth of the base algorithm...

And then I realized that I had gone too deep.

Despite my desire to prove and then to improve my mathematics fundamentals, I ultimately admitted to myself that all of this wasn't worth it. Trying to apply this level of mathematics with my level of expertise had already burned three or four full days of work, 30-40% of a full sprint, and was getting me nowhere. So I reluctantly tossed aside the graph theory and searched for a new solution.

I ultimately came to something much simpler. Instead of working with the graph, I decided to project a hexagonal grid beneath my map's topology (a technology I'd already worked with). I then assigned each hex within that grid to a particular node based on distance: each hex went to the closest node within some maximum distance. Each assigned hex then received the terrain type of its associated node, or "Water" if it had none. Then, in order to differentiate between adjacent regions of the same terrain (which were hard to separate from each-other), I devised an algorithm for drawing a continuous loop of line segments around a group of hexes, a mathy but much simpler task than the pile of algorithms I was trying to grapple with before.

This new technique worked enormously better than the graph-based algorithm would've. Not only did it give me a simple, performant way of defining a chunk of terrain around each node, it also gave me something to do with the regions outside of the reach of any node; I simply filled the excess hexes with water. This turned the map into a cohesive object, one that had islands and continents.

I think my ego kept me married to the fancy graph theory solution longer than I should have been. As soon as realized my solution would require an algorithm that was a composite of other algorithms I'd never heard about, I should've taken a step back and looked for a better solution. But my desire to be fancy kept me on that road far longer than I should've walked it. And while I'm sure it would be useful to understand that discipline of mathematics better, this problem and this project were not the times to do so.

Lessions to Take Away

1. Mathematics has a tendency to get very complex very quickly, especially when you're wandering through a discipline you're not familiar with.
2. Mathematical analysis is a powerful tool, but it needs to be deployed carefully, and never in greater quantities than is needed.
3. The parameters of any complex task, the necessities and requirements, need to be constantly questioned in order to guard against false assumptions that threaten to derail the solution entirely.
4. The ego is not a useful tool for making decisions. Sometimes the right decision is the less impressive one, and sometimes the only way to solve a problem efficiently is to admit that you're out of your depth and find a simpler solution more aligned with your skill set.

Overall, I feel Strategy Blobs has made me a much more effective developer and a much more put-together individual. The skills I've built and the lessons I've learned have proven themselves invaluable, giving me a set of tools for self improvement and self actualization the likes of which I have never had before. And while the project had many problems, I consider it a major success, an experience that has prepared me for the much greater challenges and many better games yet to come.