Principles of Software Engineering, Part 1

TUESDAY, APRIL 2, 2013

This is the first in a series of posts on the principles of software engineering. There's far more to software engineering than just "making computers do stuff" – while that phrase is accurate, it does not come close to describing what's involved in making robust, reliable software. I will use my experience building large scale systems to inform a first principles approach to defining what it is we do – or should be doing – as software engineers. I'm not interested in tired debates like dynamic vs. static languages – instead, I intend to explore the really core aspects of software engineering.

The first order of business is to define what software engineering even is in the first place. Software engineering is the construction of software that produces some desired output for some range of inputs. The inputs to software are more than just method parameters: they include the hardware on which it's running, the rate at which it receives data, and anything else that influences the operation of the software. Likewise, the output of software is more than just the data it emits and includes performance metrics like latency.

I think there's a distinction between programming a computer and software engineering. Programming is a deterministic undertaking: I give a computer a set of instructions and it executes those instructions. Software engineering is different. One of the most important realizations I've had is that while software is deterministic, you can't treat it as deterministic in any sort of practical sense if you want to build robust software.

Here's an anectode that, while simple, hits on a lot of what software engineering is really about. At Twitter my team operated a Storm cluster used by many teams throughout the company for production workloads. Storm depends on Zookeeper to store various pieces of state relating to Storm's operation. One of the pieces of state stored is information about recent errors in application workers. This information feeds a Storm UI which users look at to see if their applications have any errors in them (the UI also showed other things such as statistics of running applications). Whenever an error bubbles out of application code in a worker, Storm automatically reports that error into Zookeeper. If a user is suppressing the error in their application code, they can call a "reportError" method to manually add that error information into Zookeeper.

There was a serious omission in this design: we did not properly consider how that reportError method might be abused. One day we suddenly received a

flood of alerts for the Storm cluster. The cluster was having serious problems and no one's application was running properly. Workers were constantly crashing and restarting.

All the errors were Zookeeper related. I looked at the metrics for Zookeeper and saw it was completely overloaded with traffic. It was very strange and I didn't know what could possibly be overloading it like that. I took a look at which Zookeeper nodes were receiving the most API calls, and it turned out almost all the traffic was coming to the set of nodes used to store errors for one particular application running on the cluster. I shut that application down and the cluster immediately went back to normal.

The question now was why that application was reporting so many errors. I took a closer look at the application and discovered that all the errors being reported were null pointer exceptions – a user had submitted an application with a bug in it causing it to throw that exception for every input tuple. In addition, the application was catching every exception, suppressing it, and manually calling reportError. This was causing reportError to be called at the same rate at which tuples were being received – which was a lot.

An unfortunate interaction between two mistakes led to a major failure of a production system. First, a user deployed buggy, sloppy code to the cluster. Second, the reportError method had an assumption in it that errors were rare and thereby the amount of traffic to that method would be inconsequential. The user's buggy code broke that assumption, overloading Zookeeper and causing a cascading failure that took down every other application on the cluster. We fixed the problem by throttling the rate at which errors could be reported to Zookeeper: errors reported beyond that rate would be logged locally but not written to Zookeeper. This made reportError robust to high traffic and eliminated the possibility for cascading failure due to abuse of that functionality.

As this story illustrates, there's a lot of uncertainty in software engineering. You think your code is correct – yet it still has bugs in it. How your software is actually used differs from the model in your head when you wrote the code. You made all sorts of assumptions while writing the software, some of which are broken in practice. Your dependencies, which you use as a black box, randomly fail due to a misunderstanding of their functional input range. The most salient feature of software engineering is the degree to which uncertainty permeates every aspect of the construction of software, from designing it to implementing it to operating it in a production environment.

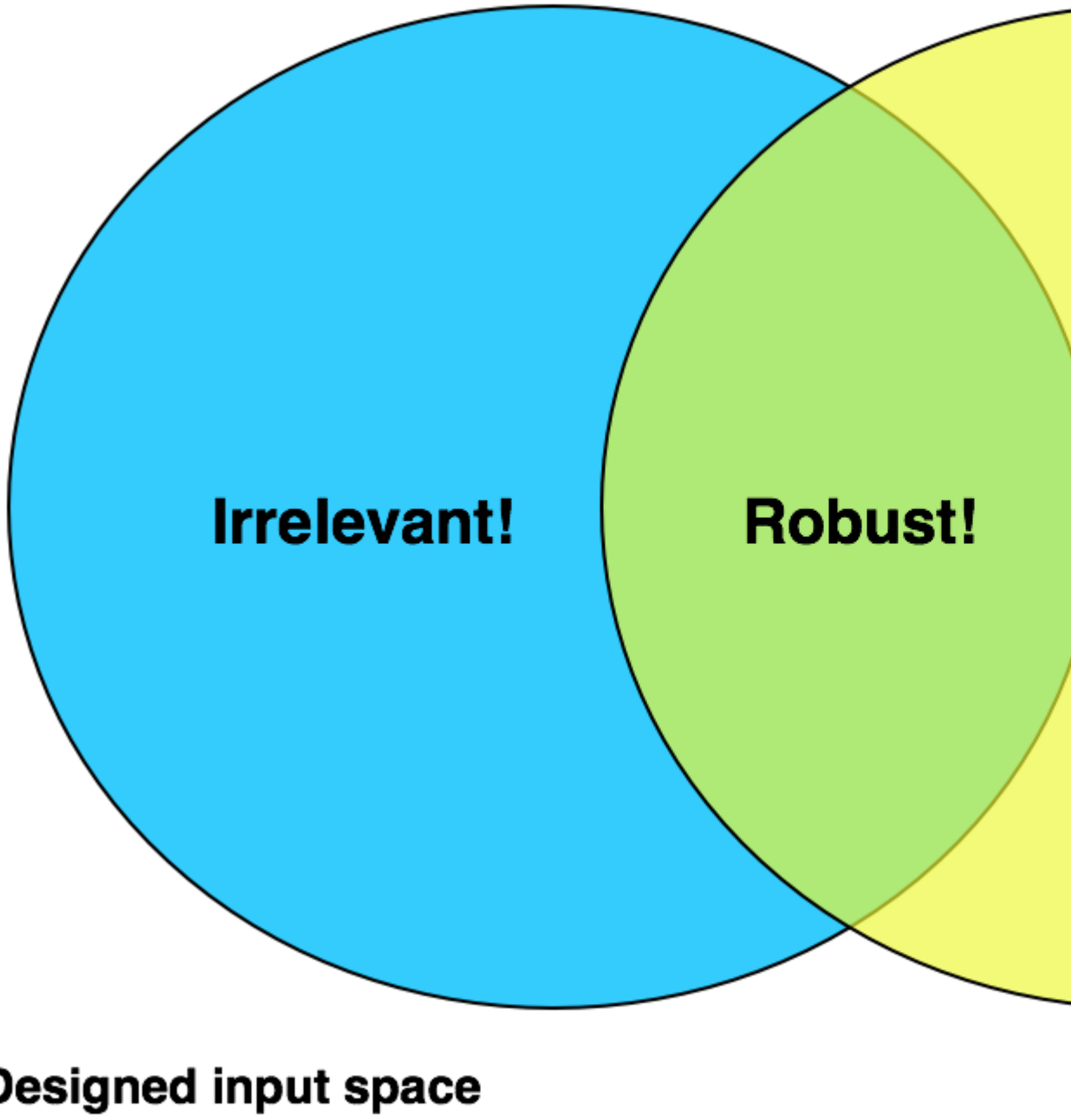
Learning from other fields of engineering

It's useful to look at other forms of engineering to learn more about software engineering. Take bridge engineering, for example. The output of a bridge is a stable platform for crossing a chasm. Even though a bridge is a static structure, there are many inputs: the weight of the vehicles crossing, wind, rain, snow, the occasional earthquake, and so on. A bridge is engineered to operate correctly under certain ranges of those inputs. There's always some magnitude of an earthquake for which a bridge will not survive, and that's deemed okay because that's such a low probability event. Likewise, most bridges won't survive being hit by a missile.

Software is similar. Software operates correctly only within a certain range of inputs. Outside those inputs, it won't operate correctly, whether it's failure, security holes, or just poor performance. In my Zookeeper example, the Zookeeper cluster was hit with more traffic than it could handle, leading to application failure. Similarly, a distributed database can only handle so many hardware failures in a short amount of time before failing in some respect, like losing data. That's fine though, because you tune the replication factor until the probability of such an event is low enough.

Another useful field to look at is rocket engineering. It takes a lot of failure and iteration to build a rocket that works. SpaceX, for example, had three failed rocket launch attempts before they finally reached orbit. The cause of failure was always something unexpected, some series of inputs that the engineers didn't account for. Rockets are filled to the brim with telemetry so that failures can be understood, learned from, and fixed. Each failure lets the engineers understand the input ranges to the rocket a little better and better engineer the rocket to handle a greater and greater part of the input space. A rocket is never finished – you never know when there will be some low probability series of inputs you haven't experienced yet that will lead to failure. STS-107 was the 113th launch of the Space Shuttle, yet it ended in disaster.

Software is very similar. Making software robust is an iterative process: you build and test it as best you can, but inevitably in production you'll discover new areas of the input space that lead to failure. Like rockets, it's crucial to have excellent monitoring in place so that these issues can be diagnosed. Over time, the uncertainty in the input space goes down, and software gets "hardened". SQL injection attacks and viruses are great examples of things that take advantage of software that operates incorrectly for part of its input space.



There's always going to be some part of the input space for which software fails – as an engineer you have to balance the probabilities and cost tradeoffs to determine where to draw that line. For all of your dependencies, you better understand the input ranges for which the dependencies operate within spec and design your software accordingly.

Sources of uncertainty in software

There are many sources of uncertainty in software. The biggest is that we just don't know how to make perfect software: bugs can and will be deployed to production. No matter how much testing you do, bugs will slip through. Because of this fact of software development, all software must be viewed as probabilistic. The code you write only has some probability of being correct for all inputs. Sometimes seemingly minor failures will interact in ways that lead to much greater failures like in my Zookeeper example.

Another source of uncertainty is the fact that humans are involved in running software in production. Humans make mistakes – almost every software engineer has accidentally deleted data from a database at some point. I've also experienced many episodes where an engineer accidentally launched a program that overloaded a critical internal service, leading to cascading failures.

Another source of uncertainty is what functionality your software should even have – very rarely are the specs fully understood and fleshed out from the get go. Instead you have to learn as you go, iterate, and refine. This has huge implications on how software should be constructed and creates tension between the desire to create reusable components and the desire to avoid wasted work.

There's uncertainty in all the dependencies you're using. Your dependencies will have bugs in them or will have unexpected behavior for certain inputs. The first time I hit a file handle limit error on Linux is an example of not understanding the limits of a dependency.

Finally, another big source of uncertainty is not understanding the range of inputs your software will see in production. This leads to anything from incorrect functionality to poor performance to security holes like injection or denial of service attacks.

This is by no means an exhaustive overview of sources of uncertainty in software, but it's clear that uncertainty permeates all of the software engineering process.

Engineering for uncertainty

You can do a much better job building robust software by being cognizant of the uncertain nature of software. I've learned many techniques over the years on how to design software better given the inherent uncertainties. I think these techniques should be part of the bread and butter skills for any software engineer, but I think too many engineers fall under the "software is deterministic" reasoning trap and fail to account for the implications of unexpected events happening in production.

Minimize dependencies

One technique for making software more robust is to minimize what your software depends on – the less that can go wrong, the less that will go wrong. Minimizing dependencies is more nuanced than just not depending on System X or System Y, but also includes minimizing dependencies on features of systems you are using.

Storm's usage of Zookeeper is a good example of this. The location of all workers throughout the cluster is stored in Zookeeper. When a worker gets reassigned, other workers must discover the new location as quickly as possible so that they can send messages to the correct place. There are two ways for workers to do this discovery, either via the pull method or the push method. In the pull method, workers periodically poll Zookeeper to get the updated worker locations. In the push method, a Zookeeper feature called "watches" is used for Zookeeper to send the information to all workers whenever the locations change. The push method immediately propogates the information, making it faster than the pull method, but it introduces a dependency on another feature of Zookeeper.

Storm uses **both** methods to propogate the worker location information. Every few seconds, Storm polls for updated worker information. In addition to this, Storm uses Zookeeper watches as an **optimization** to try to get the location information as fast as possible. This design ensures that even if the Zookeeper watch feature fails to work, a worker will still get the correct location information (albeit a bit slower in that particular instance). So Storm is able to take advantage of the watch feature without being fundamentally dependent on it. Most of the time the watch feature will work correctly and information will propogate quickly, but in the case that watches fail Storm will still work. This design turned out to be farsighted, as there was a serious bug in watches that would have affected Storm.

There's always a tradeoff between minimizing dependencies and minimizing the amount of code you need to produce to implement your application. In this case, doing the dual approach to location propogation was a good approach because it was a very small amount of code to achieve independence from that feature. On the other hand, removing Zookeeper as a dependency completely would not have been a good idea, as replicating that functionality would have been a huge amount of work and less reliable than just using a widely-used open-source project.

Lessen probability of cascading failures

A cascading failure is one of the worst things that can happen in production – when it happens it feels like the world is falling apart. One of the most common causes of cascading failures in my experience are accidental denial of service attacks like in my reportError example. The ultimate cause in these

cases is a failure to respect the functional input range for components in your system. You can greatly reduce cascading failures by making interactions between components in your system explicitly respect those input ranges by using self-throttling to avoid accidental DOS'ng. This is the approach I used in my reportError example.

Another great technique for avoiding cascading failures is to isolate your components as much as possible and take away the ability for different components to affect each other. This is often easier said than done, but when possible it is a very useful technique.

Measure and monitor

When something unexpected happens in production, it's critical to have thorough monitoring in place so that you can figure out what happened. As software hardens more and more, unexpected events will get more and more infrequent and reproducing those events will become harder and harder. So when one of those unexpected events happens, you want as much data about the event as possible.

Software should be designed from the start to be monitored. I consider the monitoring aspects of software just as important as the functionality of the software itself. And everything should be measured – latencies, throughput stats, buffer sizes, and anything else relevant to the application. Monitoring is the most important defense against software's inherent uncertainty.

In the same vein, it's important to do measurements of all your components to gain an understanding of their functional input ranges. What throughputs can each component handle? How is latency affected by more traffic? How can you break those components? Doing this measurement work isn't glamorous but is essential to solid engineering.

Conclusion

Software engineering is a constant battle against uncertainty – uncertainty about your specs, uncertainty about your implementation, uncertainty about your dependencies, and uncertainty about your inputs. Recognizing and planning for these uncertainties will make your software more reliable – and make you a better engineer.