More About Preventing Future Breakage

Preventing future breakage is a bit of a dynamic subject. Probably the most useful techniques here are identifying, isolating, and managing *problem domains* and *failure domains*.

Problem Domains just describe the complexity of a given problem that one is trying to solve. Let's look at an example below:

For example: counting the number of occurrences of a specific word in one of Shakespeare's plays, like Hamlet. This is an indexing problem. And its problem domain is fairly limited in scope. It's a single word, and a single play. A bit of BASH could easily solve this problem. So the problem domain is small, and the technical solution is fairly simple.

However, if the scope is widened slightly to include all of Shakespeare's plays, the problem domain becomes larger. Any software solution used to try and solve this indexing problem has to now handle various logic that it did not have to handle before, like consolidating word occurrences in various plays. I.e. the word 'Brevity' may occur at least once in Hamlet, and N number of times in various other plays. Managing N occurrences of 'Brevity' over M works of Shakespeare is an order of magnitude more complex in terms of describing the problem domain. A bit of BASH could solve this problem, but it might be difficult.

If the problem becomes slightly more complex, such as finding the occurrences of various synonyms to a given word, then the problem domain becomes equally large. Managing original words, their synonyms, and their hit-count across multiple works of Shakespeare is even MORE complex.

So why is any of this useful? Well, if one can easily describe and reason about a problem in a lot of detail, they understand the Problem Domain fairly well. Producing a software solution for a given problem becomes easier when the Software Engineer understands the problem domain fairly well. Of course, building a good understanding of the Problem Domain often requires a lot of experimentation, and iteration. This is why it's good to make a few initial attempts at testing a design before building an entire Production system to solve a problem like indexing Shakespeare.

Failure Domains

Like problem domains, failure domains just describe the complexity of a system. Except, instead of describing the various problems a system tries to solve, failure domains describe various sub-systems which may fail. Using the Shakespeare example again, if one of your systems is responsible for managing the full text of the works of

Shakespeare (a content server), that might be a single failure domain. If another system is responsible for actually searching that content and counting the words (an indexer), that is a separate failure domain. Some failure domains can be within other failure domains. For example, if an indexer fails, the content server may not fail. But if a content server fails, the indexer will probably also fail.

So why do we care about any of this? Well, Problem Domains drive system complexity. Complex systems often have many failure domains. The key to preventing future breakage is to identify, and manage the scope and severity of a failure domain. This may mean redesigning the system in a way that has many smaller failure domains, instead of few large ones.

As another example It's better to have a video streaming service slow down instead of failing entirely. This kind of graceful degradation is can be attributed to isolated failure domains.

This topic can be a bit complex, but there are several community articles on the idea of identifying and managing failure domains. Consolidating and completely eliminating possible failure domains is the key to preventing future breakage. If anything, managing failure domains should keep the scope of a break as small as possible.