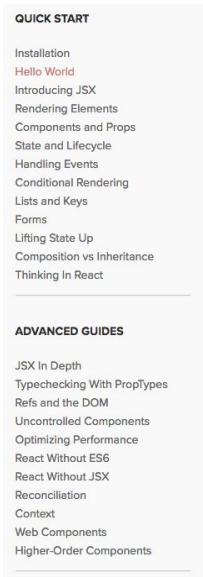
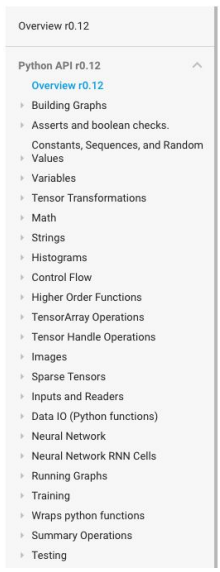
Ishan Saksena
ishans@uw.edu
CSE490: Software Quality
Winter 2017

Software Development Difficulties

Myopic Documentation

The vast majority of libraries, frameworks and packages have documentation that covers the content and abilities of the code itself. For example, the various algorithms implemented in a library, what counts as valid input or what output to expect with which functions. What most documentation doesn't cover is use cases and purpose of the code base. This often results in people using entire codebases for unintended purposes. This is especially common with web development javascript frameworks. React.js for example, starts out with instructions for installation, and, using the library. It completely skips explaining why it exists and what it's used for. A Virtual DOM for quick and dynamic re-rendering is not mentioned on its own. The documentation never details use cases and never explains what the framework is for. This information is usually picked up from other sources like blogs, co-workers and news sources. Most documentation is also myopic in that it doesn't comment on how well it integrates with other frameworks. This is left to stack overflow pages and dev forums.





React Documentation Index TensorFlow API Documentation Index

Current documentation could have introductions which cover purpose, use cases and comparisons to other similar frameworks.

Changing Environments and Obsolescence

This is something I've struggled with quite a bit. Every once in awhile, environments change completely, and libraries and frameworks take way too long to keep up with those updates. The apple development environment is a good example of this. During the last WWDC, Apple announced Swift 3. It is not backwards compatible with previous versions of Swift. Apple made certain new frameworks only available in Swift 3. But, third party frameworks were far from being updated any time soon. I was building an iOS app from scratch and was unable to reuse a lot of code due to obsolescence. Using multiple versions of the same language in the same project is difficult, so I had to rewrite a lot of code.

Services exist to automatically translate code from one version to another, but they produce a lot of errors as they go. This is mostly because they replace all old syntax with new syntax, but this doesn't always suffice. For example, if the new method requires extra parameters there is no way the service can come up with it.

Testing Hinders Maintainability

Testing tends to be a huge hindrance to extending or maintaining code.

Firstly, in larger codebases, it is really difficult to locate each unit test for every piece of code that is being updated. Testing code resides in completely different parts of the codebase. Tests are often unhelpfully named names like "Test1", "Main_Tests". There is sometimes also a lack of documentation in the code base as to which tests correspond to which pieces of code.

Secondly the user has to revisit and make any necessary updates to the unit tests. Quite often, tests are not factored in a manner susceptible to being refactored. In extreme cases, the tests cannot be updated to keep up with the code or specification changes, in which case they have to be re-written.

Thirdly, most code happens to be interdependent. Classes instantiate other classes and use other modules. Changing any part of the code necessitates revisiting tests.

While actual code is held to high standards of internal quality, testing code is not, especially for:

1. Readability/Understandability
2. Extensibility
3. Modifiability/Maintainability
4. Reusability

The only solution I can think of is writing test code with the above listed factors in mind.