**Code Stability Recommendations**

These guidelines are a list of recommended best practices. We understand that many of these may be difficult to implement in some contexts and not all of these may be feasible for a given project. That being said, these practices will lead to a more stable interface for adopters.

1. Version your software according to Semantic Versioning: [https://semver.org/](https://semver.org/)
2. Carefully develop the separation of public and private interfaces in your code. These define the changes which will mean a new version and how others will interact with it.
   a. Private instance methods can be defined via the ‘private’ keyword.
   c. Private classes: [https://blog.arkency.com/2016/02/private-classes-in-ruby/](https://blog.arkency.com/2016/02/private-classes-in-ruby/)
   d. Private method documentation: [http://www.rubydoc.info/gems/yard/file/docs/Tags.md#private](http://www.rubydoc.info/gems/yard/file/docs/Tags.md#private)
3. When changing public interfaces you must give deprecation warnings in a release prior to making the change in the next major version.
   a. Ensure deprecation warnings are specific about what’s changed and try to provide information about where the triggering code is and how to resolve it. Don’t be afraid to have long deprecation messages. Consider looking at ActiveSupport::Deprecation or the Deprecation gem.
   b. Example of a deprecation that was handled well: When RSpec went from 2 to 3 there was a 2.99 version which was full of deprecation warnings, which when corrected let you get to RSpec 3 seamlessly.
      i. [http://rspec.info/upgrading-from-rspec-2/](http://rspec.info/upgrading-from-rspec-2/)
   c. Tips for identifying breaking changes:
      i. Editing tests might be a symptom that you are making a breaking change
      ii. Changing any public method signature or behavior is a breaking change.
         1. If you want to change a signature you must call the new version from the old version and provide an appropriate deprecation warning.
4. Provide automated or well-documented paths to upgrading between major versions.
   a. If your changes require a data migration, provide tools to perform it. If you want data changes in a minor version, the code needs to be able to handle both kinds of data.
      i. Example migration tool and dashboard: Avalon’s data migration for Avalon 5 to Avalon 6 (Fedora 3 to Fedora 4 and model changes)
         1. [https://wiki.dlib.indiana.edu/display/VarVideo/Migration+Tool](https://wiki.dlib.indiana.edu/display/VarVideo/Migration+Tool)
         2. [https://wiki.dlib.indiana.edu/display/VarVideo/Migration+Report](https://wiki.dlib.indiana.edu/display/VarVideo/Migration+Report)
   b. A scheme change which requires any sort of script or altering existing data requires a new major version.
   c. Provide a changelog with each release to summarize downstream expectations. This may be best accomplished by building a CHANGELOG.md on a per-commit basis.
5. Perform quality assurance of your code via release candidates and manual testing prior to cutting a final major version.
   a. Devote enough resources to this release process to ensure a release window that is as short as possible while ensuring a stable release.
   b. Don’t add features to the master branch during this window.
   c. Example of Hyrax release QA process: [https://samvera.github.io/release_testing.html](https://samvera.github.io/release_testing.html)
6. Do whatever you can to avoid long-running branches or large pull requests. They are hard to review and put significant strain on adopters.
   a. If you have a large feature, before any code is written, get together with other developers to talk about how work could be done in phases or in smaller non-destructive chunks. The Samvera-Tech call may be a good place for this conversation.
   b. Potential Options:
      i. Create a feature flipper or some sort of internal logic branch which will let the code sit in the repository’s master branch without being run until it’s ready to go.
      ii. Write the large feature as an external plugin of some sort with a dependency on the library.
   c. Keep the scope of changes or features as tight as possible.
7. Document your classes and public methods with YARD.
   a. [https://yardoc.org](https://yardoc.org)
   b. [http://www.rubydoc.info/gems/yard/file/docs/GettingStarted.md](http://www.rubydoc.info/gems/yard/file/docs/GettingStarted.md)
   c. Devote ongoing resources to this, whether it’s during every sprint or on a continuous basis. New developers have a hard time getting oriented without it - ask new developers how they feel getting started.
8. Commit messages and pull request summaries are important documentation for future developers. Make sure they’re relevant and informative.
   a. If your pull request is part of a larger scope, explain and link to the scope.
   b. Have a pull request template as part of your Github repository to encourage the content you’d like in your pull request messages.

**Potential Reading**

The group hasn’t read all of the below resources, but we’ve come across them during our research and they seem useful to the community. Any personal experience with these resources would be helpful to share.

1. Maintaining Open Source Projects
2. Working With Legacy Code