Ten Simple Rules for taking advantage of GitHub: Supplementary Note

Section 1: Git Large File System (LFS)

GitHub supports all kind of files independently of their extension, type or content. If the file in your repository is bigger than 50 Mb, the file should be commit using the Git LSF. A minimal space quote is provided for personal/organization repositories without charge (1GB). If you exceed this quota, you can still clone repositories with large assets, but you will only retrieve the pointer files. In order to use the git LFD service for big files, the user should download the git plugin from LSF Page (https://git-lfs.github.com/). Then, the next steps should be followed:

```
git lfs track "*.psd"
git add file.psd
git commit -m "Add design file"
git push origin master
```

Section 2: Testing Levels and Continues integration

Software testing refers to the practice of testing certain functions and areas of our software code. It can be slitted in different categories or levels of complexity from unit tests to system testing. We would explain in details Unit tests and integration tests which are the start point to provide a functional software to the community.

Unit testing is piece of code in your project which test individual units of source code, sets of one or more modules together with associated control data, usage procedures, and operating procedures. Ultimately, this helps us to identify failures in our algorithms and/or logic to help improve the quality of the code. These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

```
public class TestPTM {

    // can it add the positive numbers 1 and 1?
    public void testPTMIdentifier() {
        PTM ptm = new PSIModPTM();
        assert(ptm.parent.add(1, 1) == 2);
    }
}
```

Unit testing finds problems early in the development cycle. This includes both bugs in the programmer's implementation and flaws or missing parts of the specification for the unit. The process of writing a thorough set of tests forces the author to think through inputs, outputs, and error conditions, and thus more crisply define the unit's desired behavior. Some argue that code that is impossible or difficult to test is poorly written, thus unit testing can force developers to structure functions and objects in better ways. The unit tests can be more complex to explore the *logic* of your code by testing the exptected results of an specific algorithm or mnathematical model.

```
@Test
public void TestGetPTms() {
     List<PTM> ptms = modReader.getPTMListByPatternDescription("Phospho");
     assertTrue("Number of PTMs with Term 'Phospho' in name:", ptms.size() == 106);
}
```

However, when your software is run, all those units and modules have to work together, and the whole is more complex and subtle than the sum of its independently-tested parts. Proving that components X and Y both work independently doesn't prove that they're compatible with one another or configured correctly. Problems during this interaction
may bear no relationship to the symptoms an end user would experience and report. If you automate this sort of 'component interaction' testing in order to detect breakages when they happen in the future, it's called integration testing and typically uses different techniques and technologies than unit testing.

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

With Travis-CI () the user can test their software and the following dependencies, see the following example from pIR package (https://github.com/ypriverol/pIR/):

```
sudo: required
language: c

before_install:
  - curl -OL http://raw.github.com/craigcitro/r-travis/master/scripts/travis-tool.sh
  - chmod 755 ./travis-tool.sh
  - ./travis-tool.sh bootstrap

install:
  - ./travis-tool.sh install_bioc BiocInstaller
  - ./travis-tool.sh install_deps
  - ./travis-tool.sh r_install knitr

script: ./travis-tool.sh run_tests

on_failure:
  - ./travis-tool.sh dump_logs

notifications:
  email:
    on_success: ypriverol@gmail.com
    on_failure: ypriverol@gmail.com

after_failure:
  - ./travis-tool.sh dump_logs
```

The provided example installed all the dependencies of the library (BiocInstaller) and run all the corresponding tests in the R package. More documentation about the Travis-CI integration can be found here https://travis-ci.org/ .

Section 3: Source code documentation

Documenting the code is a complex topic in software development. Some developers argue that it s bad practice and make the code more complicate and less readable, other developers encourage the use of documentation of the code using good practices. As other fields, both sides has their strong points and their weakness. From our point of view source code documentation make sense when it provides useful information around the classes, algorithms, and functions. Documentation should introduce details about the method, the rationale behind the algorithm and the possible flows of the algorithm. They should be self-explanatory and not include more noise than insides about the code. For example, the following examples are bad practice of source code documentation:

```
for( int i = 0 ; i < list.size(); i++)  // "loop across all the elements of the list.
```

The previous example represents the same information for the source code and the comment (Comments are not subtitles). Let says that the text contains two different languages (programming language and english) to explain the same content. Some developer argue that if you have a 1-1 or even a 5-1 ratio of LOC (lines of code) to comments, you are probably overdoing it. The need for excessive comments is a good indicator that your code needs refactoring.

Some developers use the comment section to explain decisions in the code and who took the corresponding decision (Comments are not source control):

```
// Revisions: Sue (2/19/2014) - Lengthened monkey's arms
//            Bob (2/20/2015) - Solved drooling issue

    void pityTheFoo() {
```

In contrast the previous example, the following code is an example of good practices:

```
/**
     * Returns <tt>true</tt> if this map maps one or more keys to the
     * specified value.  More formally, returns <tt>true</tt> if and only if
     * this map contains at least one mapping to a value <tt>v</tt> such that
     * <tt>(value==null ? v==null : value.equals(v))</tt>.  This operation
     * will probably require time linear in the map size for most
     * implementations of the <tt>Map</tt> interface.
     *
     * @param value value whose presence in this map is to be tested
     * @return <tt>true</tt> if this map maps one or more keys to the
     *         specified value
     * @throws ClassCastException if the value is of an inappropriate type for
     *         this map
     * (<a href="Collection.html#optional-restrictions">optional</a>)
     * @throws NullPointerException if the specified value is null and this
     *         map does not permit null values
     * (<a href="Collection.html#optional-restrictions">optional</a>)
     */
    boolean containsValue(Object value);
```

The example (Java API) provides information about the method, what it should return and possible exceptions related with the *Map* collection behaviour. In Table 1 we provide some useful links to documentation best practices and styles. If the source code is well documented different tools has been integrated to GitHub to

generate the final documentation for the project such as Sphinix (http://www.sphinx-doc.org/) and "Read the Docs" (https://readthedocs.org/). However, we strongly think the documentation of the source code should be maintained as the source code itself. When a piece of code is changed the corresponding documentation and comments should be reviewed by the team. Below, we provide some useful documentation about source code comment and documentation.

- 13 Tips to Comment Your Code: http://www.devtopics.com/13-tips-to-comment-your-code/
- How to Write Doc Comments for the Javadoc Tool: http://www.oracle.com/technetwork/articles/java/index-137868.html
- Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes: http://dl.acm.org/citation.cfm?id=1339530
- CodeAsDocumentation: http://martinfowler.com/bliki/CodeAsDocumentation.html