Best Practices for building and using SpaDES modules (a live document)

(Please provide feedback or changes via comments)

These suggestions are derived from 4 years of experience with creating modules that are *extremely* modular and reproducible. Clearly, there are many ways to get to a working module, but these steps are designed by the SpaDES developers to work well to get to a higher standard of re-usability. *The entries are in no particular order of importance.*

Module development

Metadata

- Scan module visually for every place that there is a `sim$...` used in the module code on the ***right hand side*** of an assignment operator (`<-` or `=`). These are *inputs* and should be included as `expectsInputs`
- Scan module visually for every place that there is a `sim$...` used in the module code on the ***left hand side*** of an assignment operator (`<-` or `=`). These are *outputs*.
- Scan module visually for *sneaky* objects that are being pulled from the user's .GlobalEnv. I.e., there is no source inside the function arguments. These should be converted to inputs, i.e., use `sim` and `expectsInputs`.
- Add information on `expectsInputs` (i.e. description) about where would real data come from (i.e. another module, private repository, etc) and if the defaults provided in the module are dummy or real.

Data

- Put any data used as inputs *in the cloud*. To use `prepInputs` to get these, `googledrive` or `ftp` or several options are available currently.
- Have your module download the data, rather than do it manually outside of the module

Inputs

- Don't use objects in the `.GlobalEnv` as they will be lost if `Cache` is used or if given to another person/computer

.inputObjects

- Use this section (it is at the bottom of the module template code when `newModule()` is run) to create a *modular* module, i.e,. Works stand alone and also with other modules.
- In our experience, a module is not particularly useful when data dependencies are left unfilled. See next point.
- The objective of this section is to create a "default" value for *every* `expectsInputs` object
    - This can be of various sorts:
        - "dummy" data, or
        - a specific (small) study area, or
    - This allows a new user to "be up and running" with at least
- Do each `expectsInputs` object, one at a time, using an `if` block as such as this:

```
If (!suppliedElsewhere("theObjectName")) {
  theObjectName <- prepInputs(url = ...)
}
```

- Two key components to this block: `suppliedElsewhere` and `prepInputs`. `suppliedElsewhere` is a way to avoid taking the default if the user has provided a non-default. `prepInputs` is a way to have a reproducible way to get the data. It is built to do many things that are very general.

Saving objects to disk

- Always schedule any saving of objects with `.saveInitialTime` and `.saveInterval`
    - This way module user can opt not to save at the `spades` call
- Do create and use a save event with key objects that are likely to be of interest. But this is not very important as the user can use `outputs` at module initiation

Plotting

- Always schedule any plotting with `.plotInitialTime` and `.plotInterval`. These allow user to turn them off easily with `.plotInitialTime = NA` (in `spades` call or as a module parameter)

Functions

- Avoid using `sim` as an argument in a user defined function → be more precise and specify the actual objects. The exception to this are `doEvent.moduleName` . The module template includes several (e.g., `Init(sim)`), but these are likely to be removed in the future. Any others should avoid using entire `sim`

```
# Instead of:
randomPointsInRaster <- function(sim, numPts) {
  sample(ncell(sim$raster), numPts)
}
# use
randomPointsInRaster <- function(ras, numPts) {
  sample(ncell(ras), numPts)
}
```

- Put functions at the top level of the *moduleName*.R file or in a sub-folder called `R` inside the module. In general, don't define functions inside other functions as there are potentially heavy RAM costs to doing this (for explanation, see `?SpaDES.core::restartR`). Every file that ends in *.r* or *.R* that exists in the R sub-folder will be sourced into the simList object and will be available for the module, and *only the module*, to use.
- If functions are to be "shared" amongst modules, ***make an R package***, make one version of the file and list that package in `pkgDeps` of the module metadata of every package that uses it. The module will then have access to these functions. Note this can be a non-published, non-public package, hosted on *github.com*, and it can be listed as "githubUser/repositoryName@branch" in `pkgDeps`. Try to never copy and paste a function so there is more than one copy -- causes headaches later.

Caching -- omitArgs is important

While using Cache, think of each argument being passed to Cache and determine what happens if it changes: would you like the Cache to rerun or recover a previous Cache call. Some questions: which of the following pairs is likely correct:

```
landscapeLCC <- Cache(postProcess, BigRaster, destinationPath = tempdir(),
                      studyArea = SmallPolygon, useSAcrs = TRUE)
landscapeLCC <- Cache(postProcess, BigRaster, destinationPath = tempdir(),
                      studyArea = SmallPolygon, useSAcrs = TRUE,
                      omitArgs = "destinationPath")
```

Most likely the 2nd one because the destinationPath is just an unimportant place to save a file temporarily. So, if that changes, which it will every R session (because tempdir() is different every time), then this will recalculate in the first version, but not the second which "ignores" destinationPath during assessment of Cache.

Meta-modeling