Implementing S4 objects in your package: Exercises

Hervé Pagès[*]

17-18 February, 2011

Contents

1 Introduction

Throughout this lab you will implement a class, named *GWASdata*, which purpose is to bind together the experimental data and metadata from a genome-wide association study. We will assume that the GWAS experimental data is a matrix (with 1 row per subject and 1 col per SNP) stored in a NetCDF file and that the metadata associated with the subjects and SNPs are stored in an SQLite db file.

Implementing an S4 class typically consists in the following steps:

---

[*] Fred Hutchinson Cancer Research Center, Seattle, WA 98008

1. A class definition where the name and type of each slot is specified. Unlike with other OO programming languages, the methods that will operate on this class are not part of the class definition.

2. A constructor so we can create *GWASdata* instances. A common practise is to define an ordinary function named like the class itself for this. Note that this is not enforced by by the S4 class system, just a consensual practise among the Bioconductor core developpers. This `GWASdata` function will take care of doing some basic argument checking and to populate the slots of the instance to be returned.

3. Some accessor methods to get values from (or set values to) the slots of a *GWASdata* object. Note that direct slot manipulation by the end user via the `@` operator is generally not recommended. Providing our own set of accessors will hopefully discourage the user of our objects from doing this. It's also a way for us to formally specify which slots are ok to be accessed and how they should be accessed (read-only slot or read-write slot).

4. Other accessor-like methods that are not *slot* accessors (i.e. they are not getting or setting the content of a slot, strictly speaking) but are returning some combination of slot values (e.g. `dim`) or some data that is retrieved from disk (e.g. `getCols`).

5. A `show` method, so our objects display nicely with some useful information.

6. A *validity method* that will take care of checking that our *GWASdata* objects are valid i.e. that their slots contain values that make sense individually and *as a whole*. Note that this certainly requires some extra effort whose benefits maybe aren't immediatly obvious, but it is considered good practise since it makes your class implementation more robust and it pays off in the long term maintenance of your package. In this lab, because of time constraints, we will implement an incomplete *validity method* for our *GWASdata* objects.

7. Some *coercion methods* to turn our *GWASdata* objects into other types of objects, with or without loss of information. In this lab, we will implement a *coercion method* for turning a *GWASdata* object into a *raw* matrix.

8. Other high-level methods that don't fall into any of the previous categories (i.e. not accessor, show, validity or coercion methods). Depending on the kind of object that is being implemented, those can be methods for subsetting, plotting, normalizing, generating an HTML report, etc...

This is really what we mean when we say *implementing S4 objects*.
The lab is divided in 2 parts:

- Part I: Implement the *GWASdata* class in a standalone file.

- Part II: Integrate this work to the *StudentGWAS* package. This means: add the file with the code produced in Part I to the package, modify the `Collate` field, import the *methods* package (if not already done), modify the `NAMESPACE` file, and add a man page documenting the class. Once everything is in place, we will be able to build and check our package with `R CMD build` and `R CMD check`.

2 Part I: Implementing the *GWASdata* class

The process of designing and implementing a new class requires that the developper spends some time thinking about:

- what s/he wants to achieve exactly with the class,
- how is the class going to be used, by who, for doing what,
- what are the typical use cases,
- what is the typical size of the data that will be manipulated, small (< 1 Mb), big (> 100 Mb), very big (> 10 Gb),
- how the class will interact with other packages and classes in CRAN/Bioconductor,
- how the facilities provided by the class will fit within the tools and file formats commonly used inside or outside Bioconductor,
- etc...

It's generally considered good design to avoid storing redundant information (although some exceptions can be made for performance considerations) and to keep things as simple as possible.

2.1 Class definition

For our *GWASdata* class, we want the following slots:

- `datapath`: the path to a NetCDF file containing the GWAS experimental data (a matrix). This will be of type *character*.
- `dataconn`: the connection to the NetCDF file pointed by `datapath`. A connection to a NetCDF file is an object of class *ncdf*. Note that there is a complication here due to the fact that the *ncdf* class is defined in the *ncdf* package and that this package has no NAMESPACE. This makes it impossible to import the *ncdf* package in our *StudentGWAS* package, and, in particular, we can't use `"ncdf"` to specify the type of our slot. So we will use the type `"list"` for this slot. This works because *ncdf* is an S3 class (aka *old style* class) and in the S3 class system, every object is a list with a class attribute attached to it.

- `metadatapath`: the path to an SQLite db file containing the GWAS metadata. This will be of type *character*.
- `metadataconn`: the connection to the SQLite db pointed by `metadatapath`. This will be of type *SQLiteConnection*.
- `nrow`: the nb of rows in the GWAS matrix (this is also the nb or rows in the `subjects` table contained in the SQLite db). This will be of type *integer*.
- `ncol`: the nb of cols in the GWAS matrix (this is also the nb or rows in the `snps` table contained in the SQLite db). Also of class *integer*.

Exercise 1
Start a new file (let's name it `GWASdata-class.R`) and write the `setClass` statement for the GWASdata class.

```
setClass("GWASdata",
    representation(
        datapath="character",
        ...
        ...
    )
)
```

2.2 Constructor

For the *GWASdata* constructor, we are going to write a function that takes 2 arguments: `datapath` and `metadatapath`. Those 2 arguments will contain the user-supplied paths to the NetCDF file and SQLite db file, respectively. The constructor must perform the following tasks:

1. Open the NetCDF and SQLite connections.
2. Drop the class attribute of the NetCDF connection (with `class(dataconn) <- NULL`), otherwise it won't be possible to assign it to the `dataconn` slot.
3. Obtain the values to assign to the `nrow` and `ncol` slots by counting the nb of rows in the `subjects` and `snps` SQL tables, respectively.
4. Finally call `new("GWASdata", ...)` with named arguments. The names of the arguments must correspond to slots in the class definition. Their values must correspond to the values to assign to the slots.

Exercise 2
a. *Add the GWASdata constructor to the `GWASdata-class.R` file.*

b. *Start R, manually load the RSQLite and ncdf packages, source the `GWASdata-class.R` file (or copy/paste its content into your session), do `showClass("GWASdata")`, and finally, try to use the GWASdata constructor on the `small_snpData.nc` and `small_metadata.sqlite` files that are included in the StudentGWAS package. Note: if the StudentGWAS package is installed, the paths to those files can be obtained with:*

   ```
   datapath <- system.file("extdata", "small_snpData.nc",
                           package="StudentGWAS")
   metadatapath <- system.file("extdata", "small_metadata.sqlite",
                               package="StudentGWAS")
   ```

Now we are able to create *GWASdata* objects! Keep your *R* session live for further testing on the *GWASdata* object you just created (let's call this object `gwas`).

It's pretty clear that we will need to implement a `show` method. However it's better to start by implementing a few accessor methods so we can use them later in the `show` method and in our code in general.

2.3 Accessors

In the next exercise we will implement the following accessors:

- the `dataconn`, `nrow` and `ncol` slot accessors;
- the `dim` accessor (which is not a *slot* accessor, strictly speaking).

Because of time constraints, we won't implement the full set of slot accessors (for completeness, the `datapath`, `metadatapath` and `metadataconn` slot accessors should also be provided).

We want to implement those accessors as *methods* for *GWASdata* objects, not as ordinary functions. This is *the* recommended way to implement accessors.

Let's distinguish between 2 situations:

- For accessors with a name that doesn't correspond to any existing function (e.g. `dataconn`), we need to define a *generic* function before we can write a method for it. This is done with a `setGeneric` statement. The simplest form of the `setGeneric` statement is the following (for a generic function `foo` with a single argument):

  ```
  setGeneric("foo", function(x) standardGeneric("foo"))
  ```

- For accessors with a name that corresponds to an existing function (e.g. `nrow`, `ncol` and `dim`), we generally don't need a `setGeneric` statement. (If the existing function is not already a generic function, then it will be automatically turned into an *implicit* generic function.) In that case the programmer must check the signature of the existing function and make sure that s/he uses exactly the same signature in his/her method definition.

The definition of the method itself is done with a `setMethod` statement. For example, in the case of a generic function dispatching on 1 argument only (the most common situation), the `setMethod` statement looks like:

```
setMethod("foo", "GWASdata",
    function(x)
    {
        ...
    }
)
```

Exercise 3

a. *Implement the* `dataconn` *accessor. Note that, for this accessor, we need to make the following small modification to the value of the* `dataconn` *slot (*`x@dataconn`*) before we return it:*

   ```
   dataconn <- x@dataconn
   class(dataconn) <- "ncdf"
   ```

   This is in order to restore the class attribute that was dropped by the constructor. Copy/paste the new code into your current R session and test the `dataconn` *accessor on your GWASdata object.*

b. *Implement the* `nrow`*,* `ncol` *and* `dim` *accessors. Test them.*

In the next exercise we implement the `getCols` method for extracting a set of adjacent columns (specified by the user) from the matrix of data stored in the NetCDF file. Of course we want to re-use the `getGWAScols` utility function implemented in a previous lab.

Exercise 4

a. *Define the* `getCols` *generic with the following arguments:* `(x, first, last = first)`*. Note that we want to dispatch on the first argument only so we need to specify this with*

   ```
   setGeneric("getCols", signature = "x", ... ... ...)
   ```

   otherwise the generic function will dispatch on all its arguments (multiple dispatch).

b. *Implement the* `getCols` *method. Note that when writing code in a standalone file like* `GWASdata-class.R`*, your code doesn't have access to the* `getGWAScols` *function that is defined in the StudentGWAS package and not exported. However, you can still put a call to* `getGWAScols` *in your code, and your code will work just fine later when we integrate it to the StudentGWAS package (because then it will have access to* everything *that is defined in the package).*

c. *Copy/paste the new code into your current R session and test the* `getCols` *accessor on your GWASdata object. Tip: for this to work, you will first need to load the StudentGWAS package and do:*

```
getGWAScols <- StudentGWAS:::getGWAScols
```

Note that it would be a bad idea to put a call like `StudentGWAS:::getGWAScols()` *inside the* `getCols` *method.*

2.4 The `show` method

`show` is a generic function defined in the *methods* package (which is also the home of the `setClass`, `setGeneric` and `setMethod` functions and the S4 class system in general). Do `?show` in your *R* session. The important bit here is that the name of the argument is `object` so that's what you need to use in your method definition.

Exercise 5

a. *Write a* `show` *method that displays something like:*

```
GWASdata instance with 50 subjects and 25 SNPs
```

Internally, use the `cat` *function to print the information, and use the* `nrow` *and* `ncol` *accessors (instead of doing direct slot access with* `object@nrow` *and* `object@ncol`*). Also, even if you think you know the class of the object being displayed, it's better to use* `class(object)` *than to hardcode* `"GWASdata"`*. You never know, maybe one day someone decides to extend your GWASdata class. When this happens, your* `show` *method will work out-of-the-box on instances of the derived class (thanks to inheritance), and, because you used* `class(object)`*, it will correctly display their class.*

b. *Copy/paste the definition of the* `show` *method into your current R session and try to display your GWASdata object again (by just typing the name of the object followed by <Enter>).*

2.5 The validity method

One limitation of the `setClass` statement is that the `representation` component only allows us to specify the types of the slots, but not their lengths or any other restriction that we'd want to impose.

For example, the `setClass` statement for our *GWASdata* class just requires the `datapath` slot to be a *character* vector, without imposing any restriction on its length or content. But what we really want is a single string i.e. a *character* vector of length 1 that is not an `NA`. A *GWASdata* object with a *character* vector of length 0 or an `NA` in its `datapath` slot could fairly be considered broken. Of course, we could put some sanity checkings in the *GWASdata* constructor in order to avoid this, but, a better approach is to define a *validity method* that will be in charge of those checkings.

Any S4 object can be validated at any time with a call to `validObject`. By default (i.e. if no *validity method* is defined), the validation only consists in checking that the types of the slot values are *compatible* with the expected types i.e. with the types that are specified in the class definition (*compatible* here means that the slot value belongs to the specified class or to one of its subclasses). This validation is automatically performed by the low-level constructor `new` (and this is why trying to create an object with an incompatible slot value generates an error).

By defining a *validity method* for his/her objects, the developper can be much more specific about what values can go into each slot. Furthermore, it allows him/her to validate an object *as a whole* by checking that the values in the different slots are compatible with each other.

Defining a *validity method* is done with a `setValidity` statement:

```
setValidity("GWASdata",
    function(object)
    {
        ...
        ...
    }
)
```

The method should return `TRUE` if the object is valid, and one or more descriptive strings if any problems are found. It should never generate an error.

In the next exercise, we implement a simple (incomplete) *validity method* for *GWASdata* objects.

Exercise 6

a. *Implement a* validity method *for* GWASdata *objects that will be in charge of checking that:*
    - *the* `datapath` *slot is a single string (i.e. a* character *vector of length 1 that is not an* `NA`*);*
    - *the* `nrow` *slot is a single non-negative* integer.
b. *Copy/paste the definition of the* validity method *into your current* R *session and call* `validObject` *on your* GWASdata *object. Break the object by setting its* `nrow` *slot to -2 (note that doing* `gwas@nrow <- -2` *won't work because* `-2` *is not of type* integer *in* R*, but* `-2L` *is). Call* `validObject` *again on the object.*

2.6 Coercion methods

It's often convenient for the user to be able to turn an object of a given class (the *original* class) into an object of another class (the *target* class). This transformation is called *coercion* in *R* jargon (*explicit type-casting* or *type conversion* in other programming languages). Depending on the classes that are involved, the coercion can be with or without loss of information.

When implementing an S4 class, it's good to think about potentially useful coercions that the user might need. In the case of our *GWASdata* class for example, we'd like the user to be able to turn a *GWASdata* object into a *raw* matrix.

R supports 2 syntaxes for performing a coercion: (1) the `as.`*targetclass*`(x)` syntax, and (2) the `as(x, "targetclass")` syntax.

The former syntax only supports a limited set of *target* classes thru some predefined generic functions such as `as.logical`, `as.integer`, `as.double`, `as.numeric`, `as.complex`, `as.character`, `as.raw`, `as.vector`, `as.list`, `as.factor`, `as.matrix`, `as.array`, `as.data.frame`, etc...

The latter syntax makes use of a single generic function, the `as` generic. This is the preferred syntax when working with S4 objects: it offers greater flexibility and better integration to the S4 class system itself.

So we want our user to be able to turn a *GWASdata* object `x` into a *raw* matrix with `as(x, "matrix")`. For this to work, we need to implement a *coercion method*. This is done with a `setAs` statement:

```
setAs("GWASdata", "matrix",
    function(from)
    {
        ...
        ...
    }
)
```

The `from` argument contains the object to coerce. The method should return the coerced object.

Exercise 7

a. *Implement the* coercion method *from* GWASdata *to* matrix*. The method must extract the entire matrix of data stored in the NetCDF file and return it into a* raw *matrix.*

b. *Copy/paste the definition of this* coercion method *into your current R session and test it by doing* `as(gwas, "matrix")`*.*

3 Part II: Integrating the *GWASdata* class to the package

We will now integrate the code produced in Part I to the *StudentGWAS* package. This is done in 4 steps.

3.1 Step 1: Add the `GWASdata-class.R` file to the package

Exercise 8

a. *Put the `GWASdata-class.R` file under the `R/` folder of your package. In case you are using a* revision control system *like Subversion to develop your code, don't forget to add the file to the system with e.g.* `svn add`*.*

b. *Add the name of the new `.R` file to the `Collate` field of the `DESCRIPTION` file. The new file should be listed after any other file that contains material used in the new file. In the case of the StudentGWAS package, it should go after the `utils.R` file which contains low-level stuff used in the new file.*

3.2 Step 2: Import the *methods* package and modify the `NAMESPACE` file

Exercise 9

a. *Make sure the methods package is in the `Imports` field of your package. Add it if needed.*

b. *Then make the following modifications to the `NAMESPACE` file of your package:*

   - *Make sure the file contains the following directive:*

     ```
     import(methods)
     ```

     If not, add it before any other imports.

   - *Export the* GWASdata *class by adding the name of the class inside the `exportClasses` directive. Syntax:*

     ```
     exportClasses(
         Class1,
         Class2,
         ...
         ...
     )
     ```

   - *In the `export` directive: Add the* functions *(non-generic and generic) defined in your package that you want to export. Note that what you export will need to be documented in a man page. The stuff that is not intended to be used directly by the user of your package should not be exported (and not documented, of course, but that doesn't mean it doesn't deserve some brief documentation in the form of a short comment in your source code).*

   - *In the `exportMethods` directive: Add the* methods *you want to export (usually all the methods defined in your package, except the* validity method*). Note that the names you need to put in the directive are those of the corresponding generics with no specification* *of the classes for which the methods are defined. This means that if you implemented more than one method for the generic `foo`, then `foo` only needs to be listed once in the `exportMethods` directive:*

```
exportMethods(
    ...
    foo,  # exports all the methods attached to this generic
    ...
)
```

Dont't forget to export the coercion methods. This is done by adding `coerce` to the `exportMethods` directive.

3.3 Step 3: Add a man page for the *GWASdata* class

Documenting the new class and its basic functionalities might not be the most exciting part of the story but, unfortunately, it's an indispensable one! To help get us motivated, let's remember that undocumented functionalities are probably not going to be used, or, in the best case, they'll make our most adventurous users feel frustrated.

An easy approach would be to use `promptClass("GWASdata")` which automatically generates a minimalist man page for our class. However, in our opinion, this automatic man page does not provide useful information to the user. It's also a little bit misleading since it encourages the user to create objects with direct calls to (new) (instead of using our higher-level constructor) and to manipulate slots directly (instead of using our accessors).

In our experience, using the following template for documenting our classes leads to more valuable documentation than the `promptClass` solution:

```
\name{GWASdata-class}
\docType{class}

\alias{GWASdata-class}
\alias{GWASdata}
\alias{...}
\alias{...}
\alias{...}

\title{GWASdata objects}

\description{
  ~~ A concise (1-5 lines) description of what the class is. ~~
}

\section{Constructor}{
  \describe{
    \item{}{
      \code{GWASdata(...)}:
      ~~ A description of the constructor and its arguments. ~~
    }
  }
}

\section{Accessors}{
  In the code snippets below, \code{x} is a GWASdata object.

  \describe{
    \item{}{
      \code{accessor1(x)}:
      ~~ A description of accessor 1. ~~
    }
    \item{}{
      \code{accessor2(x)}:
      ~~ A description of accessor 2. ~~
    }

    ... etc ...

  }
}

\section{Coercion}{
  In the code snippets below, \code{x} is a GWASdata object.

  \describe{
    \item{}{
      \code{as(x, "class1")}:
      ~~ A description of what this coercion does. ~~
    }
    \item{}{
      \code{as(x, "class2")}:
      ~~ A description of what this coercion does. ~~
    }

    ... etc ...

  }
}

\references{
  ~~ Put references to the literature/web site here. ~~
}
```

```
\author{Student Name}

\seealso{
  ~~ Put links of the form \code{\link{FUNCTIONNAME}} here ~~
  ~~ to link to other functions. ~~
  ~~ Put links of the form \code{\linkS4class{CLASSNAME}} here ~~
  ~~ to link to other classes. ~~
}

\examples{
  ~~ Put code here that illustrates at least the use of the ~~
  ~~ constructor, accessors and coercion methods (if any). ~~
}

\keyword{classes}
```

Exercise 10
Use the above template to produce the `GWASdata-class.Rd` file. This file needs to be located under the `man/` folder of your package. In case you are using a revision control system, don't forget to add the file to it. Note that:

- *There must be an alias of the form*

  ```
  \alias{foo}
  ```

  for each exported function (ordinary or generic). Also there must be an alias for each exported method. The form of this alias depends on the number of arguments involved in the dispatch. It's

  ```
  \alias{foo,Class1-method}
  ```

  for dispatch on 1 argument (e.g. for the accessor methods), and

  ```
  \alias{bar,Class1,Class2-method}
  ```

  for dispatch on 2 arguments (e.g. for the coercion methods), and so on...

- *There is no alias for the validity methods (they are not exported and they don't need to be documented). What needs to be documented with great details however is what the arguments of our high-level constructor are expected to be. In the case of paths to on-disk files like for our (GWAS-data) constructor, it's also a good idea to describe what the content of those files is expected to be.*

- *There must be an alias for the `show` method just to avoid an `R CMD check` warning (see below) even though it's ok to not document the method.*

- *The* `examples` *section is probably the most important part of any man page since most users tend to go directly there without taking the time to read the whole story (either because they already know it or because they are in a hurry).*

3.4 Step 4: Check the package

Exercise 11

a. *Run* `R CMD build` *on the package source tree. This produces a* source tarball. *Then run* `R CMD check` *on this source tarball and pay attention to any NOTE or WARNING that shows up. Fix them if necessary.*

b. *Install the source tarball by running* `R CMD INSTALL` *on it. Start a fresh R session, load the package, and try to use the new code. In particular, go to the new man page (*`?GWASdata`*) so you can see what it looks like from an end-user point of view.*

If you are using a *revision control system* and are satisfied with your work so far, then it's a good time to commit it.