

Biobase development and the new eSet

Martin T. Morgan

7 August, 2006

Revised 4 September, 2006 – featureData slot

1 Introduction

These notes help developers who are interested in using and extending the eSet class hierarchy, and using features recently added to Biobase. The information here is not useful to regular users of Biobase.

Recent changes (see the date in the title of this document!) to Biobase introduce new structure and approaches to eSet. The changes make it easy for developers to creatively use and extend the eSet class hierarchy. This document outlines recent changes, and illustrates how developers can exploit the new structure to efficiently use and extend eSet.

The document starts with a brief description of the motivation for change, and a comparison of the old and new eSets and related functionality (e.g., the Versioned class and updateObject methods). We then illustrate how eSet can be extended to handle additional types of data, and how new methods can exploit the eSet class hierarchy. We conclude with a brief summary of lessons learned, useful developer-related side-effects of efforts to revise eSet, and possible directions for future development.

2 Comparing old and new

What is an eSet?

- Coordinate high through-put (e.g., gene expression) and phenotype data.
- Provide common data container for diverse Bioconductor packages.

Motivation for change.

- What was broken? Complex data structure. Inconsistent object validity. No straight-forward way to extend eSet to new data types.
- What forward-looking design goals did we have? Flexible storage model. Class hierarchy to promote code reuse and facilitate extension to new data objects. Methods for updating serialized instances.

Key features in the redesign.

- Simplified data content.
- Structured class hierarchy.
- Alternative storage modes.

*mailto:mtmorgan@fhcrc.org
• More validity checking.
• Conversion of example data in *Biobase*, and many other data sets elsewhere in Bioconductor, to *ExpressionSet*.
• *Versioned* class information (in the development branch).
• `updateObject` methods (in the development branch).

3 **A quick tour**

3.1 **The eSet object: high-throughput experiments**

**Purpose.**

• Coordinate and contain high-throughput genomic data.

**Structure:** virtual base class.

```r
> getClass("eSet")

Virtual Class

Slots:

Name:     assayData  phenoData  featureData
Class:     AssayData  AnnotatedDataFrame  AnnotatedDataFrame

Name:     experimentData  annotation  .__classVersion__
Class:     MIAME  character  Versions
```

**Extends:**

Class "VersionedBiobase", directly
Class "Versioned", by class "VersionedBiobase", distance 2

**Known Subclasses:** "ExpressionSet", "MultiSet", "SnpSet"

• `assayData`: high-throughput data.
• `phenoData`: sample covariates.
• `featureData`: feature covariates.
• `experimentData`: experimental description.
• `annotation`: assay description.
• See below, and ?"eSet-class"

3.1.1 **assayData: high-throughput data**

**Purpose.**

• Efficiently and flexibly contain high-volume data.

**Structure:** *list*, *environment*, or *lockEnvironment* class union.
Each element of list / environment / lockEnvironment is a matrix

- Rows: features, e.g., gene names.
- Columns: samples represented on each chip.
- All matricies must have the same dimensions, row names, and column names.
- Subclasses determine which matricies must be present.
- See ?"AssayData-class"

3.1.2 phenoData: sample covariates

Purpose.
- Contain and document sample covariates.

Structure: AnnotatedDataFrame.

- data: data.frame.
  - Rows: sample identifiers.
  - Columns: measured covariates.
- varMetadata: data.frame.
  - Rows: measured covariate labels.
  - Columns: covariate descriptors.
- See ?"AnnotatedDataFrame-class"

3.1.3 featureData: feature covariates

Purpose.
- Contain and document feature covariates specific to the experiment; use the annotation slot for chip-level descriptions.

Structure: AnnotatedDataFrame.

- data: data.frame.
  - Rows: feature identifiers. These match row names of assayData.
  - Columns: measured covariates.
- varMetadata: data.frame.
  - Rows: measured covariate labels.
  - Columns: covariate descriptors.
- See ?"AnnotatedDataFrame-class"
3.1.4 experimentData: experiment description

Purpose.
- Summarize where and how the experiment was performed.

Structure: MIAME
- title: experiment title.
- name: experimenter name(s).
- preprocessing: list of pre-processing steps.
- Additional slots.
- See ?"MIAME-class".

3.1.5 annotation: assay description

Purpose.
- Link experiment to annotation package.

Structure: character
- Label identifying annotation package.

3.2 Important eSet methods

Initialization.
- eSet is VIRTUAL, initialize via subclass callNextMethod

Accessors (get, set).
- assayData(obj); assayData(obj) <- value: access or assign assayData
- phenoData(obj); phenoData(obj) <- value: access or assign phenoData
- experimentData(obj); experimentData(obj) <- value: access or assign experimentData
- annotation(obj); annotation(obj) <- value: access or assign annotation

Subsetting.
- obj[i, j]: select genes i and samples j.
- obj$name; obj$name <- value: retrieve or assign covariate name in phenoData

3.2.1 Additional eSet methods

- show.
- storageMode: influence how assayData is stored.
- updateObject: update eSet objects to their current version.
- validObject: ensure that eSet is valid.
The `validObject` method is particularly important to `eSet`, ensuring that `eSet` contains consistent structure to data.

```r
> getValidity(getClass("eSet"))
```

```r
function (object)
{
  msg <- NULL
  if (!is(object, "eSet"))
    msg <- validMsg(msg, paste("cannot validate object of class", class(object)))
  msg <- validMsg(msg, isValidVersion(object, "eSet"))
  dims <- dims(object)
  if (!is.na(dims[[1]])) {
    if (any(dims[1,] != dims[1, 1]))
      msg <- validMsg(msg, "row numbers differ for assayData members")
    if (any(dims[2,] != dims[2, 1]))
      msg <- validMsg(msg, "sample numbers differ for assayData members")
    msg <- validMsg(msg, assayDataValidMembers(assayData(object)))
    if (dims[1, 1] != dim(featureData(object))[1])
      msg <- validMsg(msg, "feature numbers differ between assayData and featureData")
    if (!all(featureNames(assayData(object)) == featureNames(featureData(object))))
      msg <- validMsg(msg, "featureNames differ between assayData and featureData")
    if (dims[2, 1] != dim(phenoData(object))[1])
      msg <- validMsg(msg, "sample numbers differ between assayData and phenoData")
    if (!all(sampleNames(assayData(object)) == sampleNames(phenoData(object))))
      msg <- validMsg(msg, "sampleNames differ between assayData and phenoData")
  } else if (dim(phenoData(object))[1] != 0) {
    msg <- validMsg(msg, "sample numbers differ between assayData and phenoData")
  }
  if (is.null(msg))
    TRUE
  else msg
}
```

The validity methods for `eSet` reflect our design goals. All `assayData` members must have identical row and column dimensions and `featureNames`. The names and numbers of samples must be the same in `assayData` and `phenoData` slots. Validity methods are defined for the classes underlying each slot as well. For instance, the validity methods for `AnnotatedDataFrame` check that variables used in `pData` are at least minimally described in `varMetadata`.

### 3.3 Subclasses of eSet

`Biobase` defines three classes that extend `eSet`. `ExpressionSet` (discussed further below) is meant to contain microarray gene expression data. `SnpSet` is a preliminary class to contain SNP data; other classes in development (e.g., in `oligo`) may provide alternative implementations for SNP data. `MultiSet` is an `ExpressionSet`-like class, but without restriction on the names (though not structure) of elements in the `assayData` slot.
3.3.1 ExpressionSet
Purpose:
- Contain gene expression data.

Required `assayData` members.
- `exprs`, a matrix of expression values.

Important methods.
- Initialization (additional details below):
  ```
  > obj <- new("eSet", phenoData = new("AnnotatedDataFrame"), experimentData = new("MIAME"),
  + annotation = character(), exprs = new("matrix"))
  ```
  - `exprs(obj), exprs(obj) <- value`: get or set `exprs`; methods defined for `ExpressionSet`, `AssayData`.

3.3.2 MultiSet and SnpSet

MultiSet.
- Purpose: flexibly contain a collection of expression data matrices.
- Required `assayData` members: none.

SnpSet.
- Purpose: contain genomic SNP calls.
- Required `assayData` members: `call`, `callProbability`.

4 Comments on `assayData`: high-throughput data storage

The `assayData` slot is meant to store high-throughput data. The idea is that the slot contains identically sized matrices containing expression or other data. All matrices in the slot must have the same dimension, and are structured so that rows represent ‘features’ and columns represent ‘samples’. Validity methods enforce that row and column names of slot elements are identical.

For technical reasons, creating instance of `AssayData` is slightly different from the way this is usually done in R. Normally, one creates an instance of a class with an expression like `new("ExpressionSet", ...), with the ... representing additional arguments. `AssayData` objects are created with

```
> assayDataNew("environment", elt)
```

where `elt` might be a matrix of expression values. For the curious, the reason for this setup stems from our desire to have a class that is a list or environment, rather than a class that has a slot that contains a list or environment. The is relationship is desirable to avoid unnecessary function calls to access slots, and requires that a class contain the base type (e.g., `environment`). Until very recently an R object could not contain an `environment`.

The `assayData` slot of `ExpressionSet` objects must contain a matrix named `exprs`. This is different from the structure of an `exprSet`, which was expected to contain a slot for `exprs` and `se.exprs` (containing an measure of uncertain associated with expression measure). The reason for removing `se.exprs` from `ExpressionSet` is pragmatic: many `exprSet` objects did not contain an `se.exprs` object. The frequent absence of a data element means that effective methods cannot be written for the class – a
developer cannot easily write methods for `ExprSet` that anticipate an `se.exprs`, because many objects the method is supposed to work on may not have the necessary data. Nonetheless, the `ExpressionSet` validity method tries to be liberal – it guarantees that the object has an `exprs` element, but allows for other elements too. The prudent developer wanting consistent additional data elements should derive a class from `ExpressionSet` that enforces the presence of their desired elements.

The `AssayData` class allows for data elements to be stored in three different ways (see `?storageMode` and `?"storageMode<-"` for details): as a list, environment, or lockedEnvironment. Developers are probably familiar with list objects; a drawback is that `exprs` elements may be large, and some operations on lists in R may trigger creation of many copies of the `exprs` element. This can be expensive in both space and time. Environments are nearly unique in R, in that they are passed by reference rather than value. This eliminates some copying, but has the unfortunately consequence that side-effects occur – modifications to an environment inside a function influence the value of elements outside the function. For these reasons, environments can be useful as ‘read only’ arguments to functions, but can have unexpected consequences when functions modify their arguments. Locked environments implemented in `Biobase` try to strike a happy medium, allowing pass by reference for most operations but triggering (whole-environment) copying when elements in the environment are modified. The locking mechanism is enforced by only allowing known ‘safe’ operations to occur, usually by channeling user actions through the accessor methods:

```r
> data(sample.ExpressionSet)
> storageMode(sample.ExpressionSet)

[1] "lockedEnvironment"
```

```r
> tryCatch(assayData(sample.ExpressionSet)$exprs <- log(exprs(sample.ExpressionSet)),
+     error = function(err) cat(conditionMessage(err)))
```

cannot change value of locked binding for 'exprs'

```r
> exprs(sample.ExpressionSet) <- log(exprs(sample.ExpressionSet))
```

The `setReplaceMethod` for `exprs` (and `assayData`) succeeds by performing a deep copy of the entire environment. Because this is very inefficient, the recommended paradigm to update an element in a `lockedEnvironment` is to extract it, make many changes, and then reassign it. Developers can study `assayData` methods to learn more about how to lock and unlock environment bindings. `Biobase` allows the experienced user to employ (and run the risks of) environments, but the expectation is that most user objects are constructed with the default `lockedEnvironment` or `list`.

A longer term consideration in designing `AssayData` was to allow more flexible methods of data storage, e.g., through database-hosted arrays. This is facilitated by using generic functions such as `exprs()` for data access, so that classes derived from `AssayData` can provide implementations appropriate for their underlying storage mode.

### 5 Extending eSet

A designer wanting to implement `eSet` for a particular type of data creates a class that ‘contains’ `eSet`. The steps for doing this are described below. One example of such a class is `ExpressionSet`, designed to hold a matrix of gene expression values in the `assayData` slot.

```r
> getClass("ExpressionSet")
```

```r
Slots:
  Name: assayData    phenoData    featureData
```
The data structure of an ExpressionSet is identical to that of eSet, and in fact is inherited (without additional slot creation) from eSet. The main difference is that the validity methods of eSet are augmented by a method to check that the assayData slot contains an entity named "exprs". A valid ExpressionSet object must also satisfy all the validity requirements of eSet, but the developer does not explicitly invoke validity checking of the parts of the data structure inherited from eSet.

Developers often want to create a subclass of eSet, ExpressionSet, or other classes in the eSet hierarchy. We will use the Swirl data set in the sample.eSet data set.

5.1 Implementing a new class: a SwirlSet example

The Swirl data set contains four elements in the assayData slot: R, G, Rb, Gb. To derive a class from eSet for this data, we create a class, and provide initialization and validation methods.

We create a class as follows:

```r
> setClass("SwirlSet", contains = "eSet")
[1] "SwirlSet"
```

Notice that there are no new data elements in SwirlSet compared with eSet. The initialize method is written as

```r
> setMethod("initialize", "SwirlSet", function(.Object, R = new("matrix"),
+     G = new("matrix"), Rb = new("matrix"), Gb = new("matrix"),
+     ...) {
+     callNextMethod(.Object, R = R, G = G, Rb = Rb, Gb = Gb, ...)
+ })

[1] "initialize"
```
The structure of the initialize method is a bit different from those often seen in R. Often, initialize has only .Object as a named arguments, or, if there are other named arguments, they correspond to slot names. Here our initialize method accepts four arguments, named after the assayData elements. Inside the initialize method, the named arguments are passed to the next initialization method in the hierarch (i.e., initialize defined for eSet). The eSet initialize method then uses these arguments to populate the data slots in .Object. In particular, eSet places all arguments other phenoData, experimentData, and annotation into the assayData slot. The eSet initialize method then returns the result to the initialize method of SwirlSet, which returns a SwirlSet object to the user:

```
> new("SwirlSet")

SwirlSet (storageMode: lockedEnvironment)
assayData: 0 features, 0 samples
element names: G, Gb, R, Rb
phenoData
  sampleNames: none
featureData
  featureNames: none
experimentData: use 'experimentData(object)'
Annotation character(0)
```

General programing guidelines emerge from experience with the initialize method of eSet and derived classes. First, an appropriate strategy is to name only those data elements that will be manipulated directly by the initialize method. For instance, the definition above did not name phenoData and other eSet slots by name. To do so is not incorrect, but would require that they be explicitly named (e.g., phenoData=phenoData) in the callNextMethod code. Second, the arguments R, G, Rb, Rg are present in the initialize method to provide defaults consistent with object construction; the ‘full’ form of callNextMethod, replicating the named arguments, is required in the version of R in which this class was developed. Third, named arguments can be manipulated before callNextMethod is invoked. Fourth, the return value of callNextMethod can be captured...

```
> setMethod("initialize", "MySet", function(.Object, ...) {  
+   .Object <- callNextMethod(.Object, ...)
+ })

> .
```

and manipulated before being returned to the user. Finally, it is the responsibility of the developer to ensure that a valid object is created; callNextMethod is a useful way to exploit correctly designed initialize methods for classes that the object extends, but the developer is free to use other techniques to create valid versions of their class.

A validity method might complete our new class. A validity method is essential to ensure that the unique features of SwirlSet – our reason for designing the new class – are indeed present. We define our validity method to ensure that the assayData slot contains our four types of expression elements:

```
> setValidity("SwirlSet", function(object) {
+   assayDataValidMembers(assayData(object), c("R", "G", "Rb",
+     "Gb"))
+ })
```

Slots:
Assigning `fancyAssayData` might invalidate the object, but `justAsFancyPhenoData` restores validity.

### 6 Versioned

One problem encountered in the Bioconductor project is that data objects stored to disk become invalid as the underlying class definition changes. For instance, earlier releases of `Biobase` contain a sample `eSet` object. But under the changes discussed here, `eSet` is virtual and the stored object is no longer valid. The challenge is to easily identify invalid objects, and to provide a mechanism for updating old objects to their new representation.

`Biobase` introduces the `Versioned` and `VersionedBiobase` classes to facilitate this. These classes are incorporated into key `Biobase` class definitions. `Biobase` also defines the `updateObject` generic and methods for conveniently updating old objects to their new representation.

```r
> data(sample.ExpressionSet)
> classVersion(sample.ExpressionSet)

R    Biobase    eSet ExpressionSet
"2.4.0"  "1.11.34"  "1.1.0"   "1.0.0"

> obj <- updateObject(sample.ExpressionSet)

The version information for this object is a named list. The first two elements indicate the version of R and Biobase used to create the object. The latter two elements are contained in the class prototype, and the class prototype is consulted to see if the instance of an object is 'current'. These lists can be subsetted in the usual way, e.g.,

```r
> isCurrent(sample.ExpressionSet)[c("eSet", "ExpressionSet")]

    eSet ExpressionSet
   TRUE        TRUE
```
Versioned classes, `updateObject` and related methods simplify the long-term maintenance of data objects. Take the fictitious `MySet` as an example.

```r
> setClass("MySet", contains = "eSet", prototype = prototype(new("VersionedBiobase", + versions = c(classVersion("eSet"), MySet = "1.0.0"))))
[1] "MySet"
> obj <- new("MySet")
> classVersion(obj)
  R    Biobase   eSet  MySet
 "2.4.0" "1.12.2" "1.1.0" "1.0.0"
```

This is a new class, and might undergo changes in its structure at some point in the future. When these changes are introduced, the developer will change the version number of the class in its prototype (the last line, below):

```r
> setClass("MySet", contains = "eSet", prototype = prototype(new("VersionedBiobase", +    versions = c(classVersion("eSet"), MySet = "1.0.1"))))
[1] "MySet"
> isCurrent(obj)
  R    Biobase   eSet  MySet
 TRUE  TRUE  TRUE  TRUE
```

and add code to update to the new version

```r
> setMethod("updateObject", signature(object = "MySet"), function(object, +    ..., verbose = FALSE) {
  +    if (verbose)
  +      message("updateObject(object = 'MySet')")
  +    object <- callNextMethod()
  +    if (isCurrent(object)["MySet"])
  +      return(object)
  +    if (!isVersioned(object))
  +      new("MySet", assayData = updateObject(assayData(object)),
  +          phenoData = updateObject(phenoData(object)), experimentData = updateObject(experimentData,
  +          annotation = updateObject(annotation(object))))
  +    else {
  +      classVersion(object)["MySet"] <- classVersion("MySet")["MySet"]
  +      object
  +    }
  + })
[1] "updateObject"
```

The code after `if(!isVersioned)` illustrates one way of performing ‘radical surgery, creating a new up-to-date instance by updating all slots. The `else` clause represents more modest changes, using methods to update stale information. `updateObject` then returns a new, enhanced object:

```r
> classVersion(updateObject(obj))
  R    Biobase   eSet  MySet
 "2.4.0" "1.12.2" "1.1.0" "1.0.1"
```

As in the example, versioning helps in choosing which modifications to perform – minor changes for a slightly out-of-date object, radical surgery for something more ancient. Version information might also be used in methods, where changing class representation might facilitate more efficient routines.
6.1 Versioned versus VersionedBiobase

The information on R and Biobase versions is present in eSet derived classes because eSet contains VersionedBiobase. On the other hand, phenoData contains Versioned, and has only information about its own class version.

```r
> classVersion(new("phenoData"))
phenoData
 "1.0.0"
```

The rationale for this is that phenoData is and will likely remain relatively simple, and details about R and Biobase are probably irrelevant to its use. On the other hand, some aspects of eSet and the algorithms that operate on them are more cutting edge and subject to changes in R or Biobase. Knowing the version of R and Biobase used to create an instance might provide valuable debugging information.

6.2 Adding Versioned information to your own classes

The key to versioning your own classes is to define your class to contain Versioned or VersionedBiobase, and to add the version information in the prototype. For instance, to add a class-specific version stamp to SwirlSet we would modify the class definiton to

```r
> setClass("SwirlSet", contains = "eSet", prototype = prototype(new("VersionedBiobase", + versions = c(classVersion("eSet"), SwirlSet = "1.0.0"))))
[1] "SwirlSet"
> classVersion(new("SwirlSet"))
       R    Biobase    eSet  SwirlSet
  "2.4.0"  "1.12.2"  "1.1.0"  "1.0.0"
```

See additional examples in the Versioned help page.

It is also possible to add arbitrary information to particular instances.

```r
> obj <- new("SwirlSet")
> classVersion(obj)["MyID"] <- "0.0.1"
> classVersion(obj)
       R    Biobase    eSet  SwirlSet  MyID
  "2.4.0"  "1.12.2"  "1.1.0"  "1.0.0"  "0.0.1"
> classVersion(updateObject(obj))
       R    Biobase    eSet  SwirlSet  MyID
  "2.4.0"  "1.12.2"  "1.1.0"  "1.0.0"  "0.0.1"
```

There is additional documentation about these classes and methods in Biobase.

7 Summary

This document summarizes recent changes to Biobase, outlining strategies that developers using Biobase may find useful. The main points are to introduce the eSet class hierarchy, to illustrate how developers can effectively extend this class, and to introduce class versions as a way of tracking and easily updating objects. It is anticipated that eSet-derived classes will play an increasingly important role in Biobase development.
8 Session Information

The version number of R and packages loaded for generating the vignette were:

- R version 2.4.0 (2006-10-03), x86_64-unknown-linux-gnu

- Locale: LC_CTYPE=en_US;LC_NUMERIC=C;LC_TIME=en_US;LC_COLLATE=en_US;LC_MONETARY=en_US;LC_MESSAGES=en_US;LC_PAPER=en_US;LC_NAME=C;LC_ADDRESS=C;LC_TELEPHONE=C;LC_MEASUREMENT=en_US;LC_IDENTIFICATION=C

- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils

- Other packages: Biobase 1.12.2