1 Introduction

The goal of GenomicRanges is to provide general containers for genomic data. The central class, at least from the user perspective, is GRanges, which formalizes the notion of ranges, while allowing for arbitrary “metadata columns” to be attached to it. These columns offer the same flexibility as the venerable data.frame and permit users to adapt GRanges to a wide variety of adhoc use-cases.

The more we encounter a particular problem, the better we understand it. We eventually develop a systematic approach for solving the most frequently encountered problems, and every systematic approach deserves a systematic implementation. For example, we might want to formally store genetic variants, with information on alleles and read depths. The metadata columns, which were so useful during prototyping, are inappropriate for extending the formal semantics of our data structure: for the sake of data integrity, we need to ensure that the columns are always present and that they meet certain constraints.

We might also find that our prototype does not scale well to the increased data volume that often occurs when we advance past the prototype stage. GRanges is meant mostly for prototyping and stores its data in memory as simple R data structures. We may require something more specialized when the data are large; for example, we might store the data as a Tabix-indexed file, or in a database.

The GenomicRanges package does not directly solve either of these problems, because there are no general solutions. However, it is adaptable to specialized use cases.

2 The GenomicRanges abstraction

Unbeknownst to many, most of the GRanges implementation is provided by methods on the GenomicRanges class, the virtual parent class of GRanges. GenomicRanges methods provide everything except for the actual data storage and retrieval, which GRanges implements directly using slots. For example, the ranges are retrieved like this:

```r
> library(GenomicRanges)
> selectMethod(ranges, "GRanges")
```

Method Definition:
Extending *GenomicRanges*

```r
definition (x, ...) 
{
  .local <- function (x, use.names = TRUE, use.mcols = FALSE) 
  {
    if (!isTRUEorFALSE(use.names))
      stop("'use.names' must be TRUE or FALSE")
    if (!isTRUEorFALSE(use.mcols))
      stop("'use.mcols' must be TRUE or FALSE")
    ans <- x@ranges
    if (!use.names)
      names(ans) <- NULL
    if (use.mcols)
      mcols(ans) <- mcols(x)
    ans
  }
  .local(x, ...)
}
```

Signatures:
- `x`
  - target "GRanges"
  - defined "GRanges"

An alternative implementation is *DelegatingGenomicRanges*, which stores all of its data in a delegate *GenomicRanges* object:
```
> selectMethod(ranges, "DelegatingGenomicRanges")
```

Method Definition:

```r
function (x, ...)
ranges(x@delegate, ...)
```

3 Formalizing `mcols`: Extra column slots

An orthogonal problem to data storage is adding semantics by the formalization of metadata columns, and we solve it using the "extra column slot" mechanism. Whenever *GenomicRanges* needs to operate on its metadata columns, it also delegates to the internal `extraColumnSlotNames` generic, methods of which should return a character vector, naming...
the slots in the GenomicRanges subclass that correspond to columns (i.e., they have one value per range). It extracts the slot values and manipulates them as it would a metadata column – except they are now formal slots, with formal types.

An example is the VRanges class in VariantAnnotation. It stores information on the variants by adding these column slots:

```r
> GenomicRanges:::extraColumnSlotNames(VariantAnnotation:::VRanges())
```

```
[1] "ref"          "alt"          "totalDepth"
[4] "refDepth"    "altDepth"    "sampleNames"
[7] "softFilterMatrix"
```

Mostly for historical reasons, VRanges extends GRanges. However, since the data storage mechanism and the set of extra column slots are orthogonal, it is probably best practice to take a composition approach by extending DelegatingGenomicRanges.