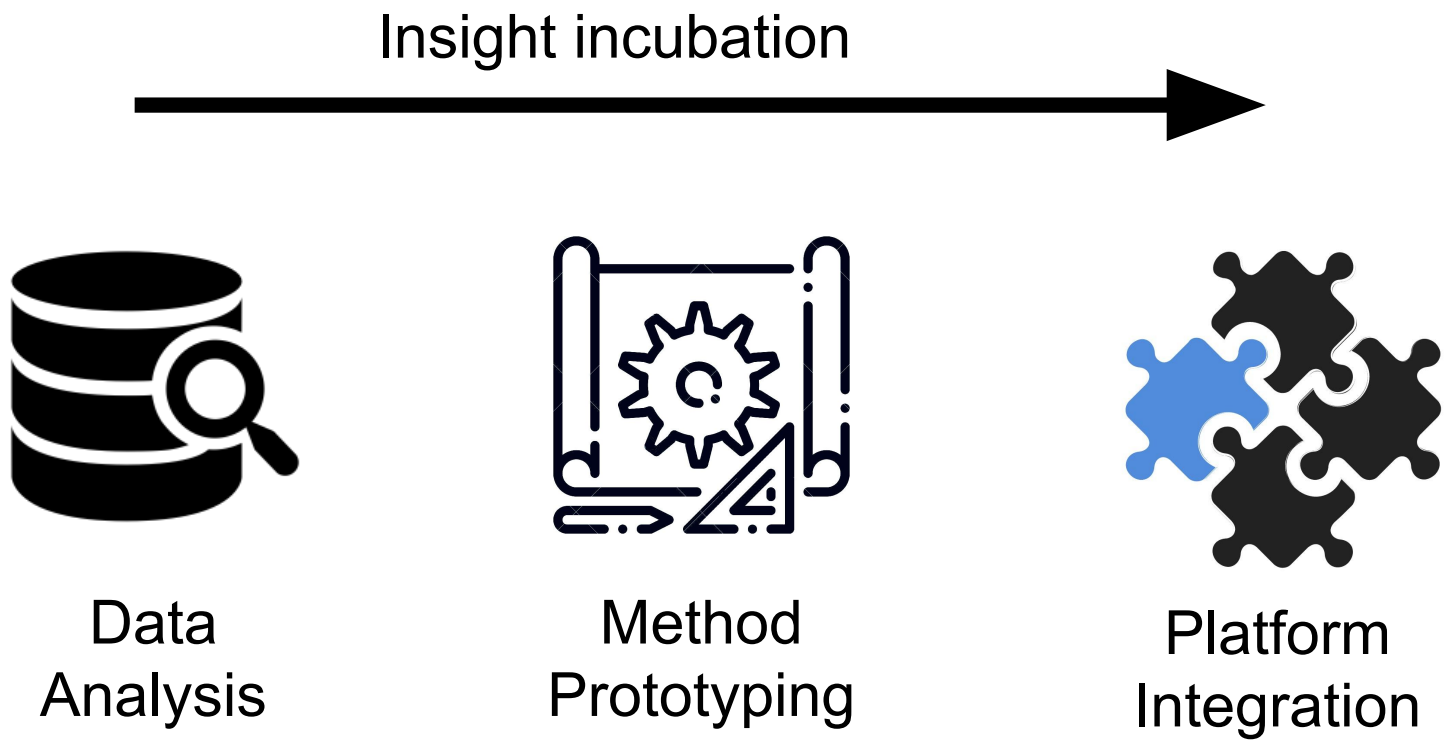
Usability of the Bioconductor Infrastructure

Michael Lawrence

July 28, 2017

Outline

The Ranges infrastructure is an incubator



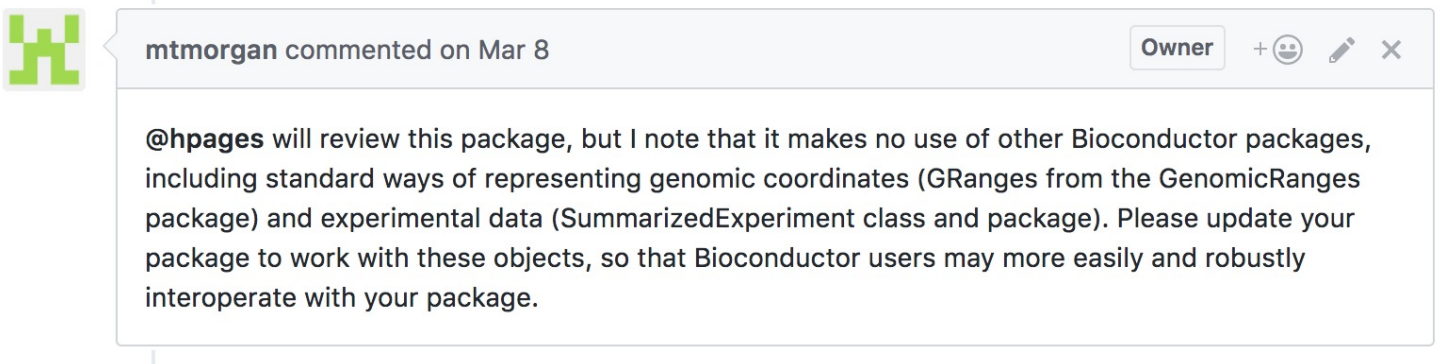
- Should be accessible to the average Bioconductor user

Is the transition happening?

- From a typical package submission:

  `Imports: checkmate, dplyr, ggplot2, tidyr`

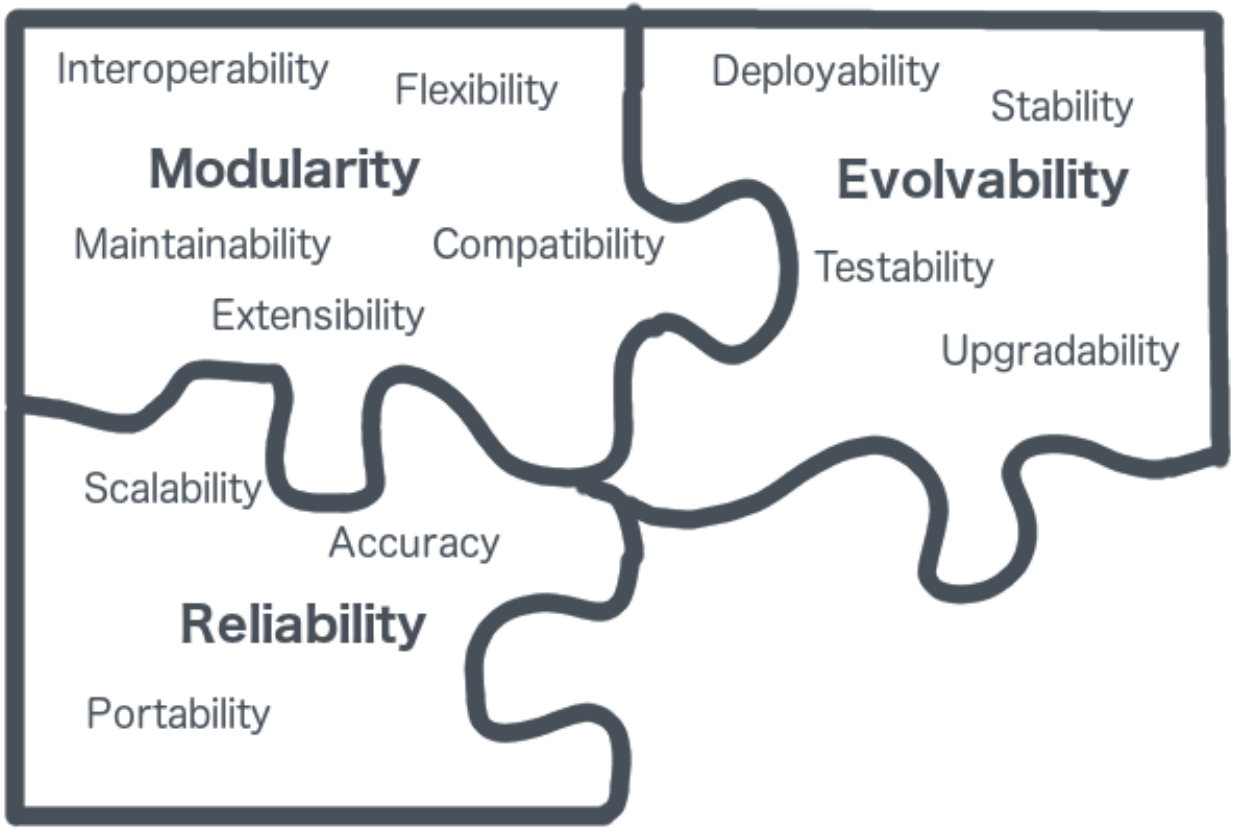
- A typical initial response:



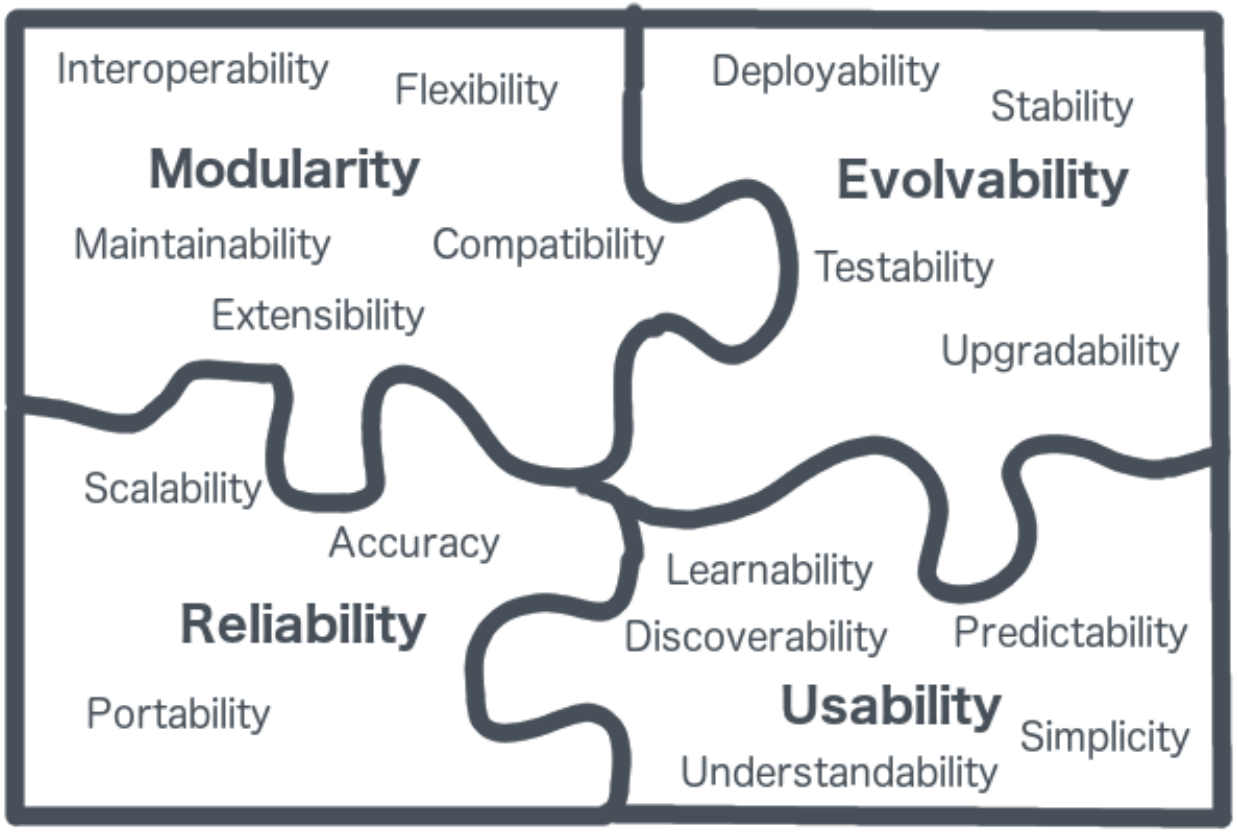
Why not?

- Education?
- Documentation?
- The software?
  - It all starts here

Aspects of software quality: the ilities



Aspects of software quality: the ilities



Cognitive Dimensions of Notations

- Thomas Green and Marian Petre (1996) proposed 14 dimensions of usability in the context of visual programming
- Many are interrelated and in balance with each other
- Guide for evaluating usability and as a framework for discussing interface design trade-offs

Green's cognitive dimensions

- Abstraction gradient
- Closeness of mapping
- Consistency
- Diffuseness
- Error-proneness
- Hard mental operations
- Hidden dependencies
- Provisionality
- Premature commitment
- Progressive evaluation
- Role-expressiveness
- Secondary notation
- Viscosity (robustness)
- Visibility

Abstraction

Procedural abstraction

A compound operation that enables the user tell the computer what to do without telling it how to do it.

Data abstraction

> *"A methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects"*

Structure and Interpretation of Computer Programs (1979)

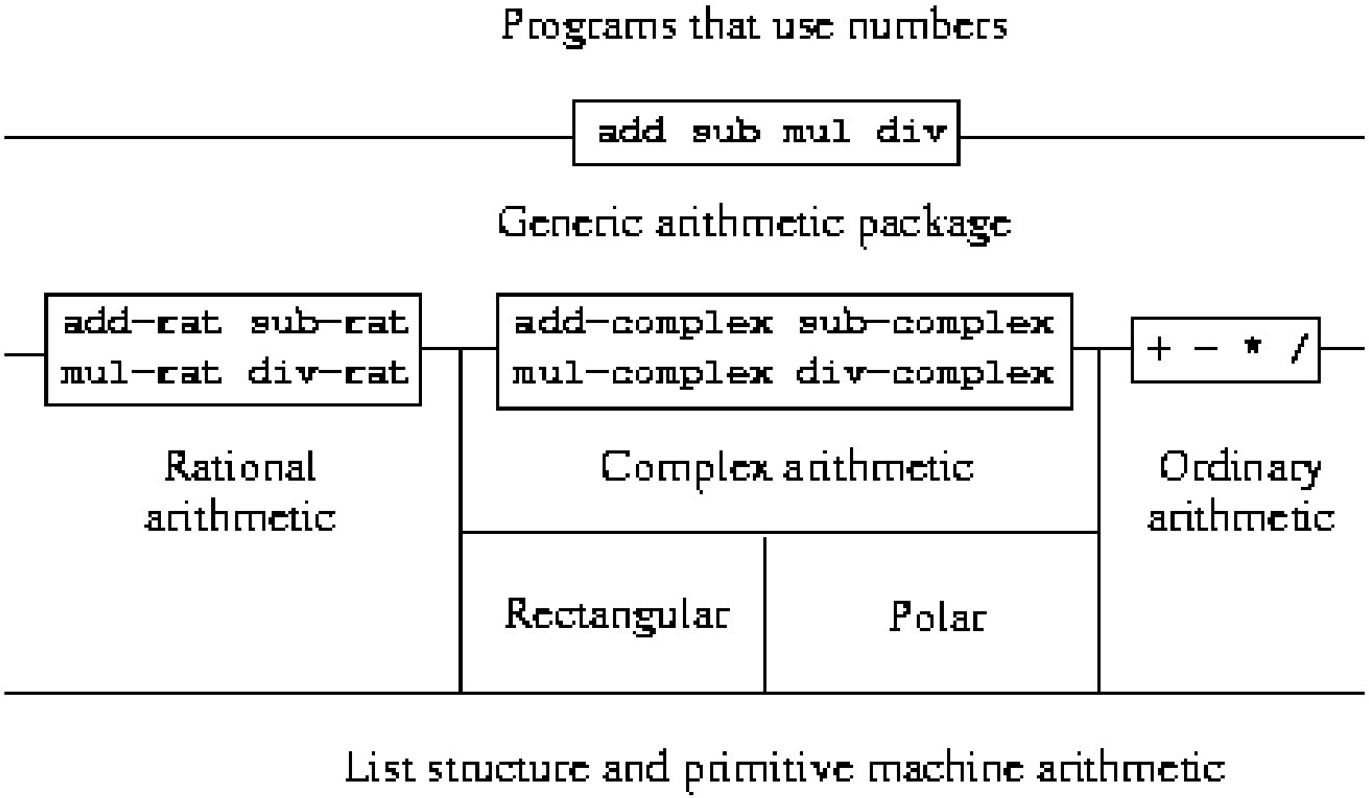
- We can implement data abstractions using a set of procedures (*constructors* and *selectors*) that satisfy some contract.

Abstraction gradient

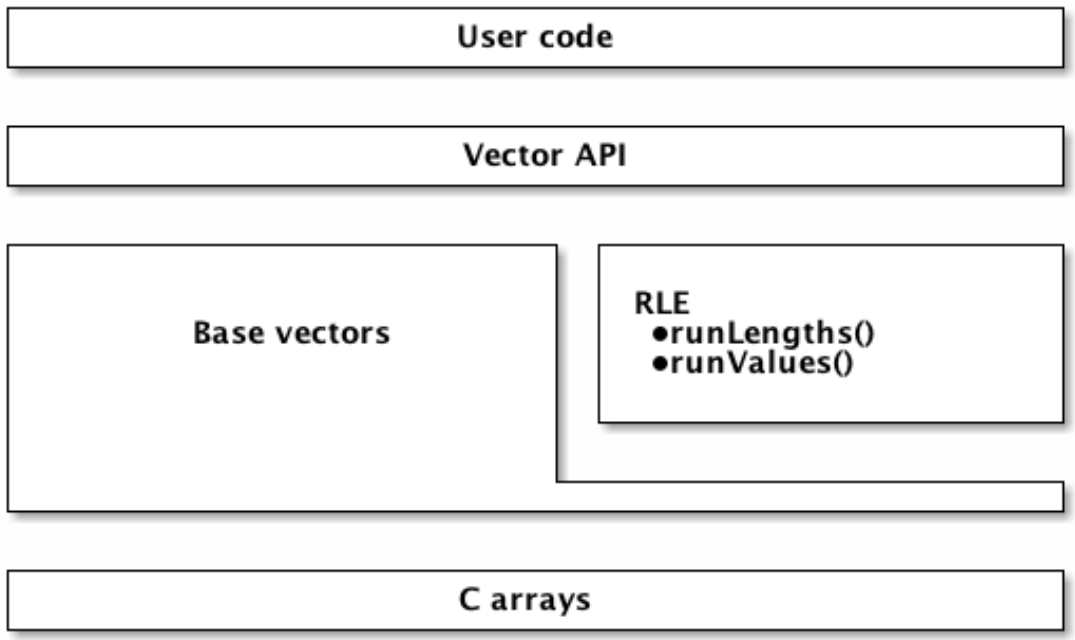
Stratified design

"The notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages"

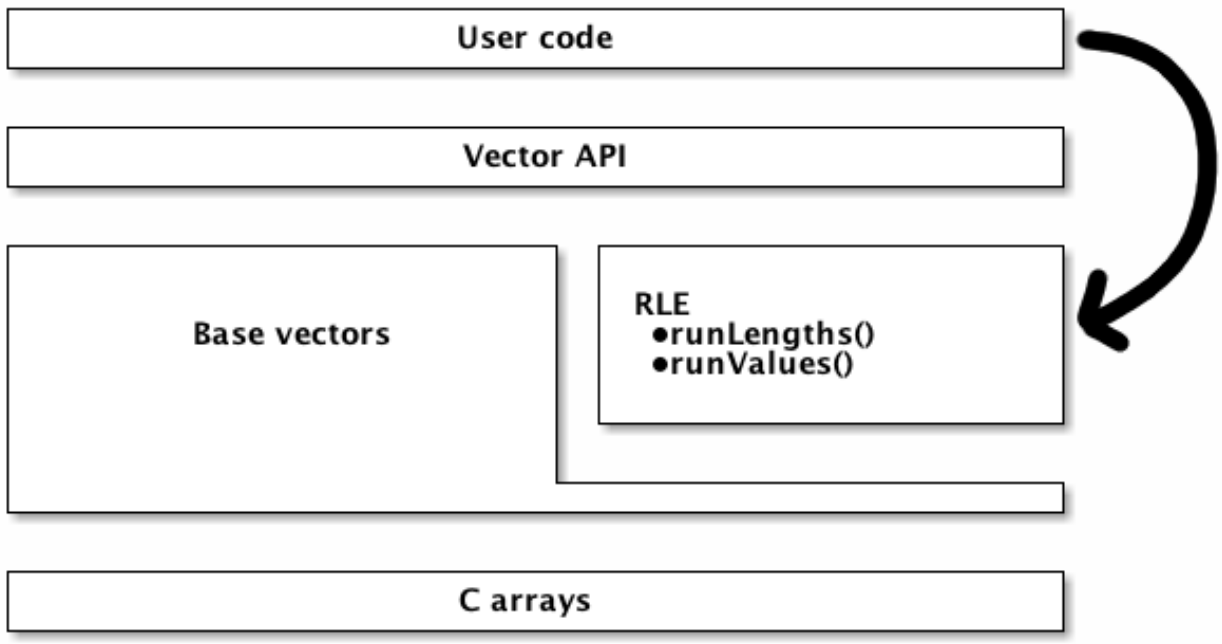
- Structure and Interpretation of Computer Programs (1979)



Rle: Run-Length Encoded vector



Subverting abstractions



Example: finding peak summits

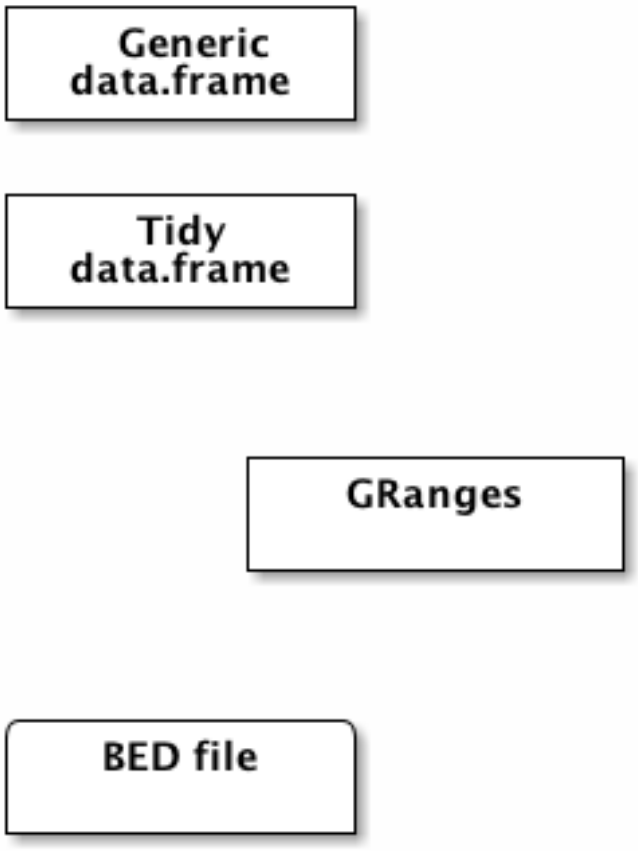
Find maximum of tallest peak

```
max_pos <- which.max(cov_rle)
```

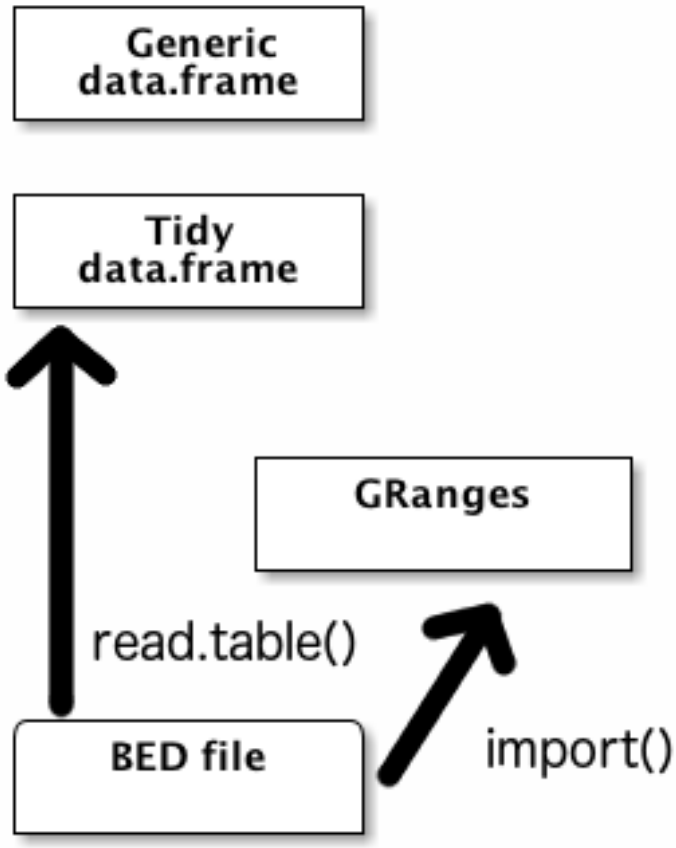
Find the summit range of the peak

```
max_run <- findRun(cov_rle, max_pos)
summit <- ranges(cov_rle)[max_run]
```

The GRanges gradient



The GRanges gradient

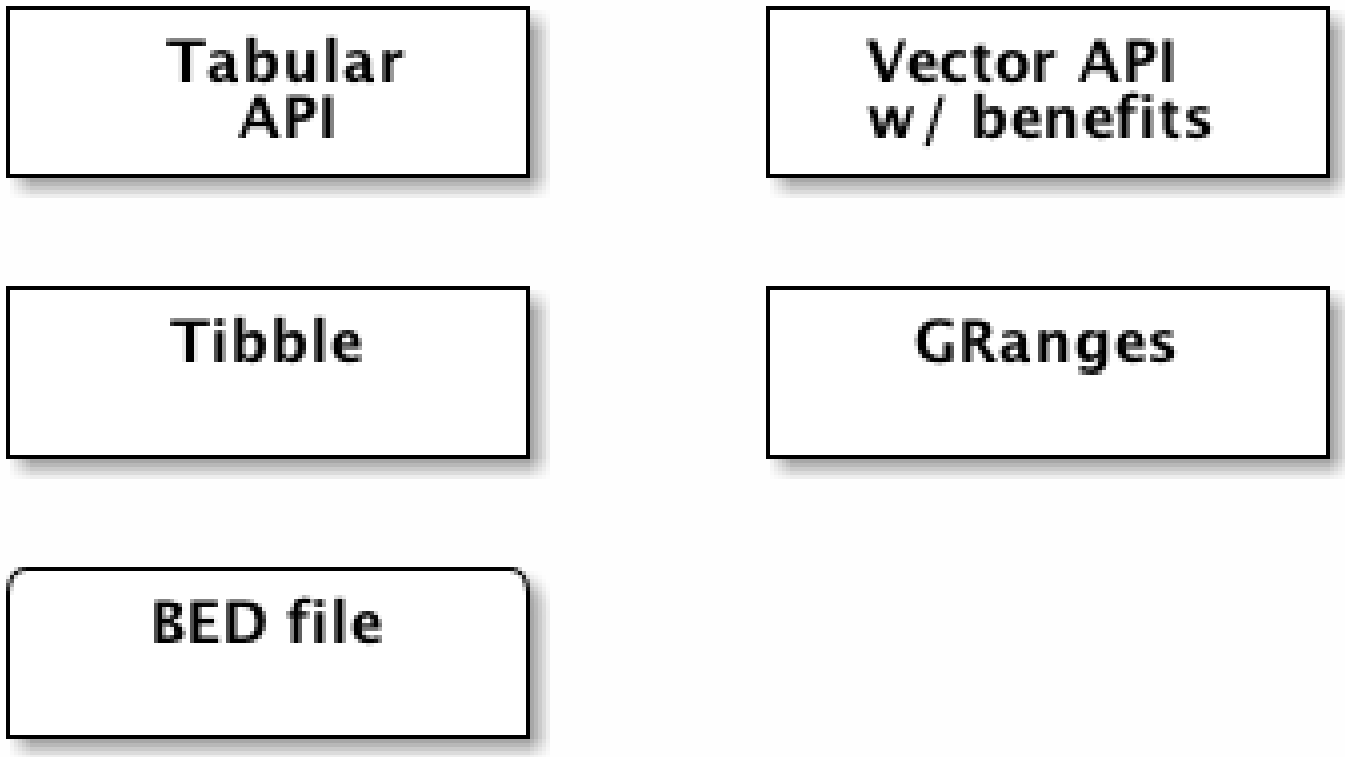


Closeness of mapping

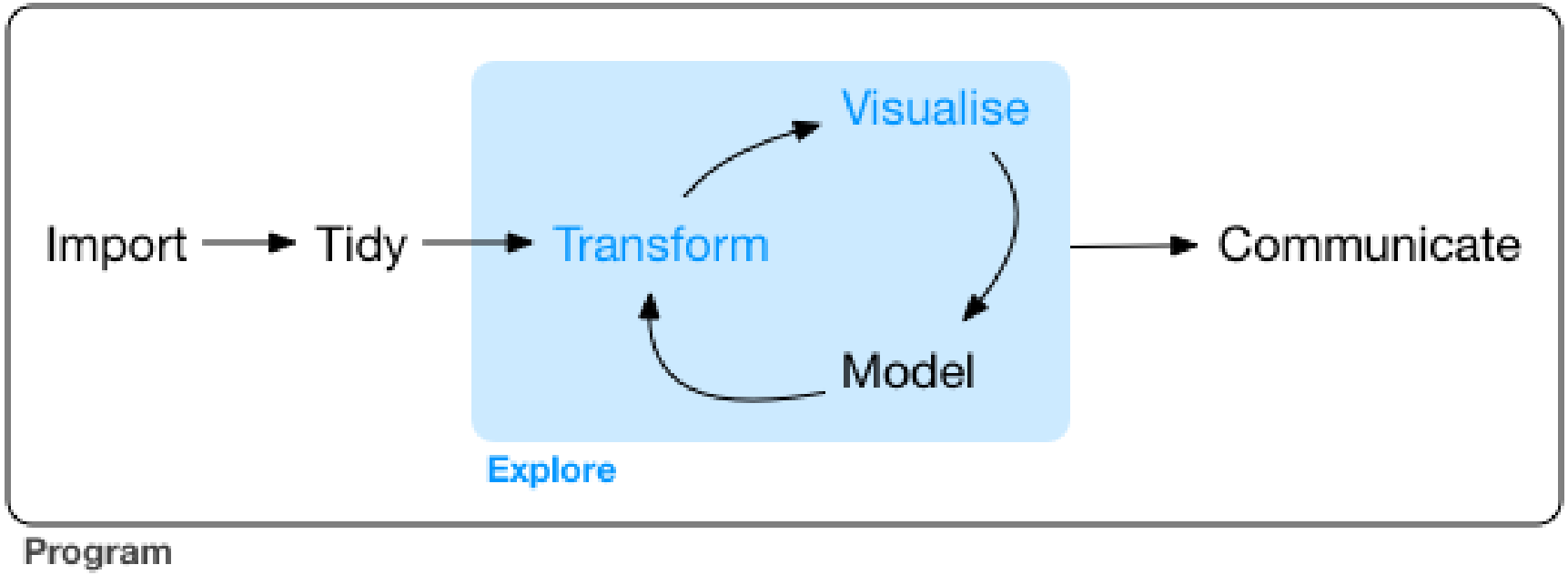
Relates to the mapping from the user's mental model of the data and processes to their computational representations.

- For genomic data, the user is thinking in biological terms, so our data structures should embed biological semantics

GRanges is out of alignment with the user

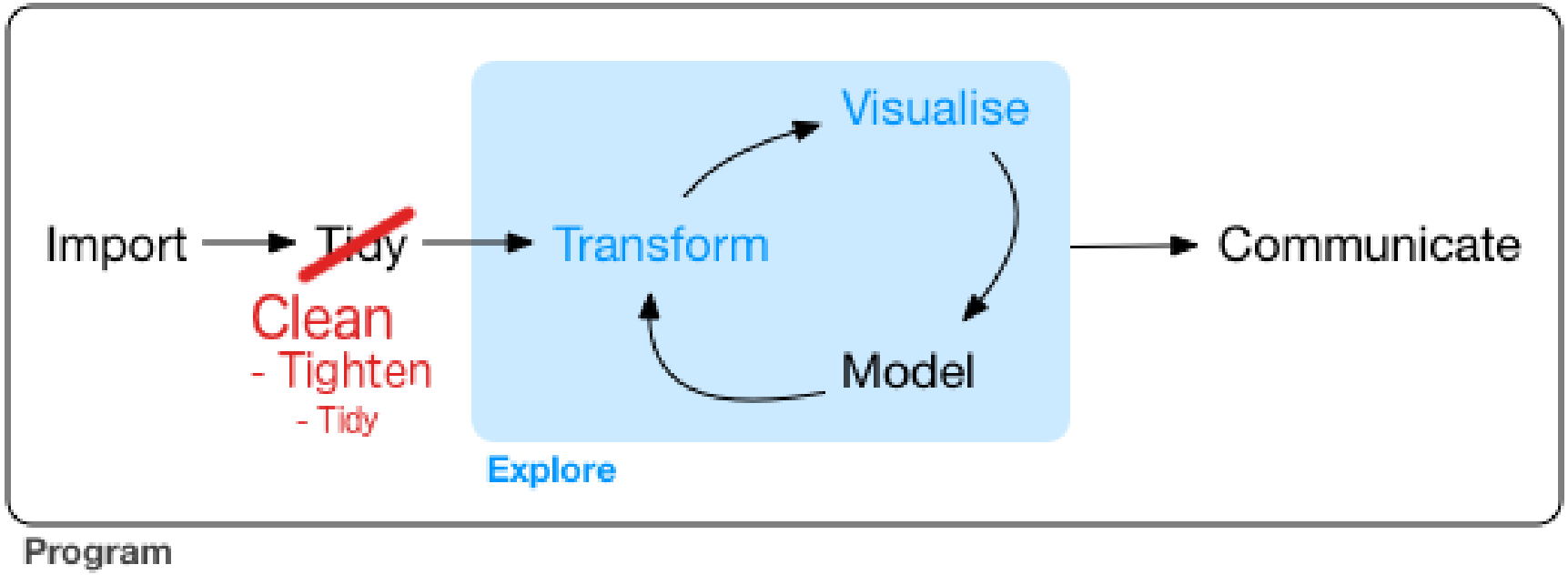


Tightening the cognitive link is the job of the user



- Explicitly declaring semantics:
  - Helps the software do the right thing
  - Helps the user be more *expressive*
- Developers should provide the necessary tools

Tightening the cognitive link is the job of the user



- Explicitly declaring semantics:
  - Helps the software do the right thing
  - Helps the user be more *expressive*
- Developers should provide the necessary tools

Consistency

Refers to consistency of the mappings between the mental and computational models.

- Bioconductor is:
  - Internally consistent
  - Consistent with base R
  - Inconsistent with fluent, tidyverse-style APIs

Diffuseness (vs expressiveness)

- Relates to the information density of the code and how well it communicates the *intent* of the programmer
- Enable the user to convey more meaning with less code
- Terseness for its own sake makes code obscure, difficult to unpack
- For genomic data, we want the user to express computations in terms of the biology

findOverlaps() API

Joining by overlaps is currently a *diffuse* operation:

```
hits <- findOverlaps(query, subject)
query$variable <- extractList(subject$variable,
                              as(hits, "List"))
joined <- expand(query, "variable")
```

Why not abstract that to something like:

```
joined <- overlap_join(query, subject, "variable")
```

import() API

Importing a file with rtracklayer is overly *terse* and abstract:

```
obj <- import(file) # what sort of data is this?
```

A bit better:

```
obj <- import.bed(file)
```

> *"Expert programmers know how to choose the level of abstraction appropriate to their task".*

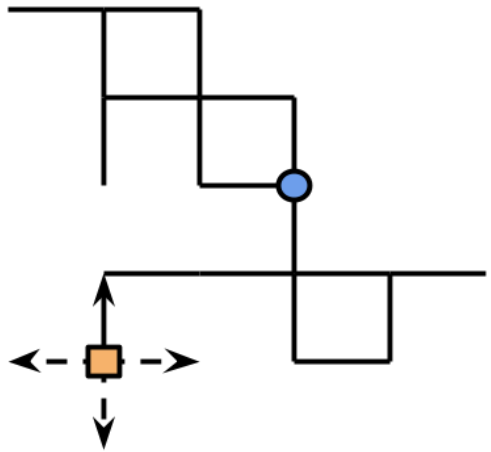
Hard mental operations

How hard the user has to think about things other than the motivating task

Classic example

Optimizing using R vector operations when an algorithm is more intuitively expressed with loops

A Random Walk



- Simulate n steps from a 2-dimensional random walk
- How would you write the code?

Naive Code

```
1  rw2d = function(n = 100) {
2    x = numeric(n)
3    y = numeric(n)
4
5    for(i in 2:n) {
6      # Forward or backward?
7      if (runif(1) > .5)
8        delta = 1
9      else
10       delta = -1
11
12     # Horizontal or vertical?
13     if (runif(1) > .5) {
14       x[i] = x[i-1] + delta
15       y[i] = y[i-1]
16     } else {
17       x[i] = x[i-1]
18       y[i] = y[i-1] + delta
19     }
20   }
21
22   list(x = x, y = y)
23 }
```

Ross Ihaka's Vectorized Code

```
1  rw2d_vectorized = function(n = 100) {
2    # Sample from four separate directions
3    xsteps = c(-1, 1, 0, 0)
4    ysteps = c(0, 0, -1, 1)
5    dir = sample(1:4, n - 1, replace = TRUE)
6    x = c(0, cumsum(xsteps[dir]))
7    y = c(0, cumsum(ysteps[dir]))
8
9    list(x = x, y = y)
10 }
```

- Fast
- Not intuitive
- Doesn't generalize

AtomicLists for data aggregation

- IRanges defines List subclasses specific to atomic vector types
  - *IntegerList*, *CharacterList*, etc
- Useful for modeling ragged data
- Implement summary functions for aggregating their elements.
  - `sum()`, `mean()`, etc

Example: comparing overlapping regions

Compute the total difference between the endpoints of overlapping pairs of ranges and find the closest for each query range.

```
hits <- as(findOverlaps(x, y), "List")
yl <- extractList(y, hits)
dev <- abs(start(x) - start(yl)) + abs(end(x) - end(yl))
unlist(yl[phead(order(dev), 1L)])
```

Visibility (discoverability)

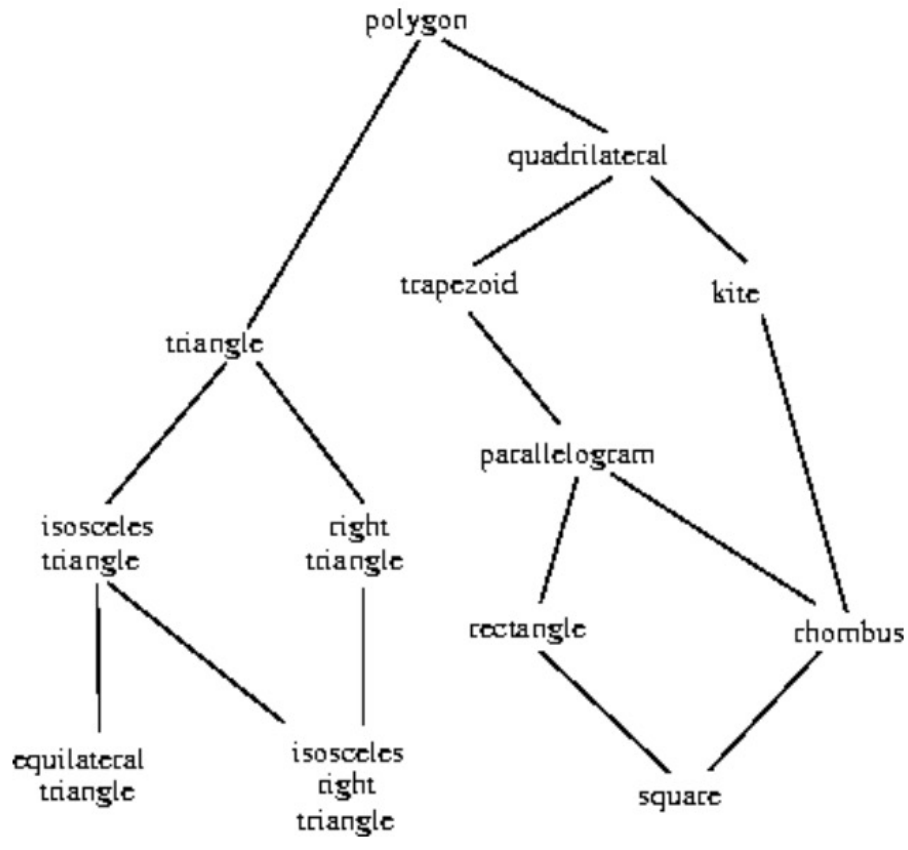
The ability of a user to find:

- Procedures, and what they do,
- Data structures, and what operations they support.

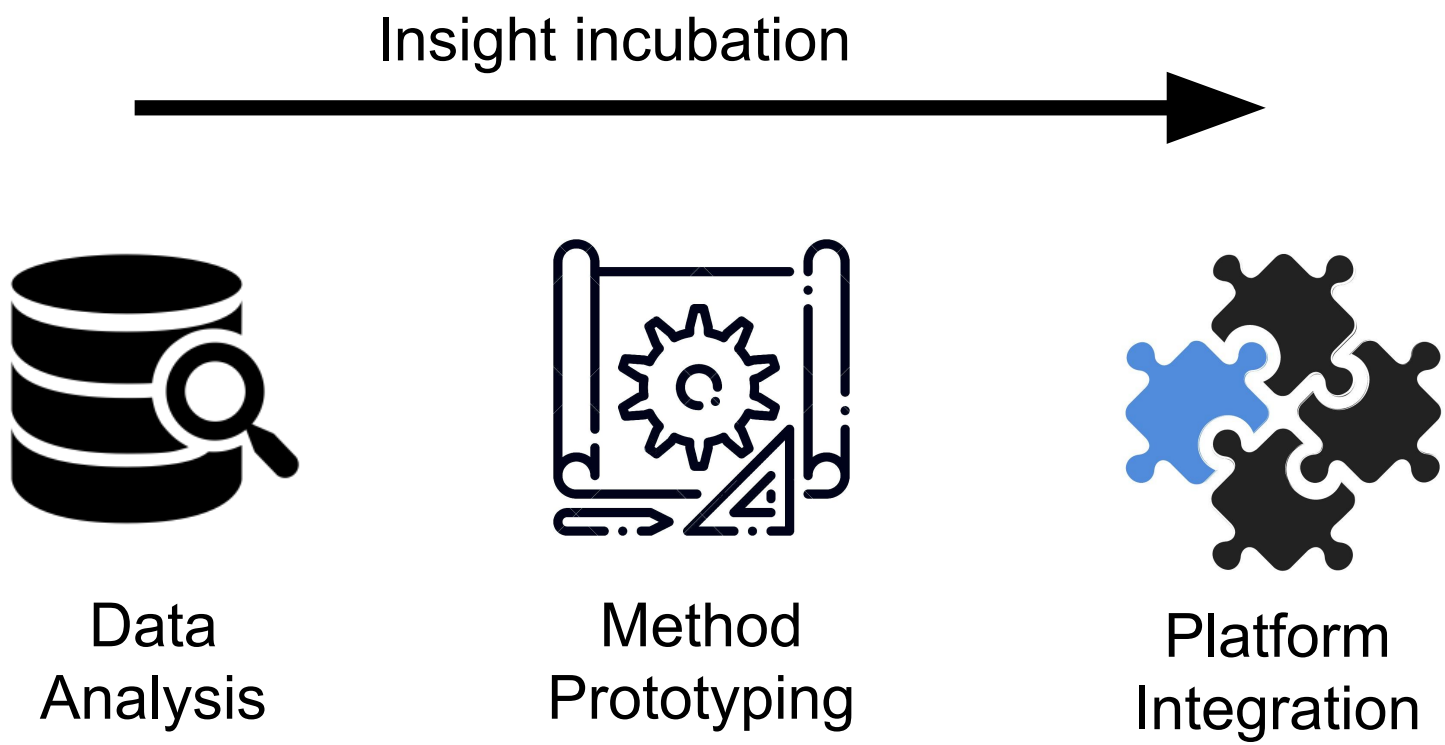
S4 benefits developers

- Automatically validates object integrity upon modification/construction
- Implements polymorphism through dispatch (*data-directed programming*) and inheritance
- These benefits trickle-down to users, but indirect

Operations	Types: Polar	Types: Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular



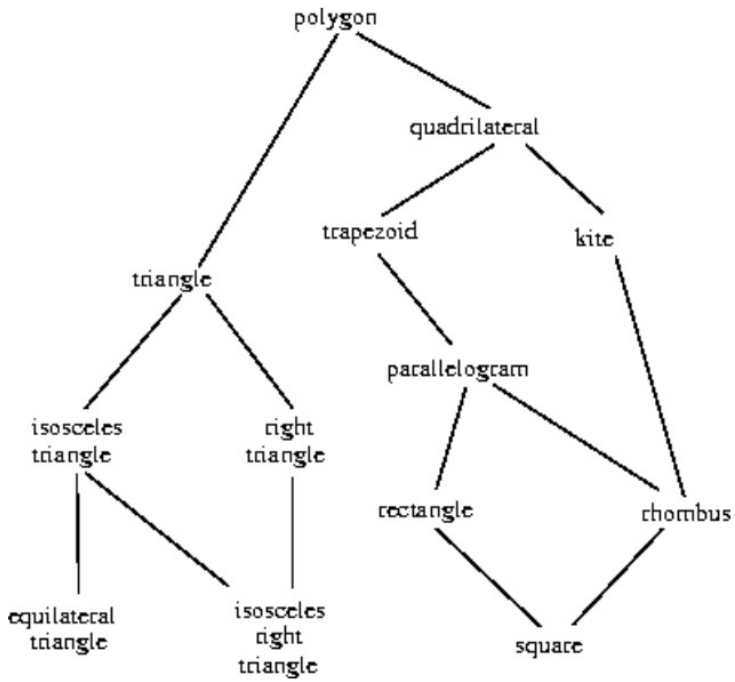
Users are (becoming) developers



S3/S4 dispatch challenges users

- Users want *visibility* into:
  - What happens when calling a function on a specific type of object
    - Bridge: `selectMethod(generic, sig)` vs. `generic.sig`
  - Which operations are possible on a specific type of object
    - Bridge: `methods(class)`
- Bridges are available, but still a layer of indirection

Operations	Types: Polar	Types: Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular



S4 challenges developers, and thus users

Challenges

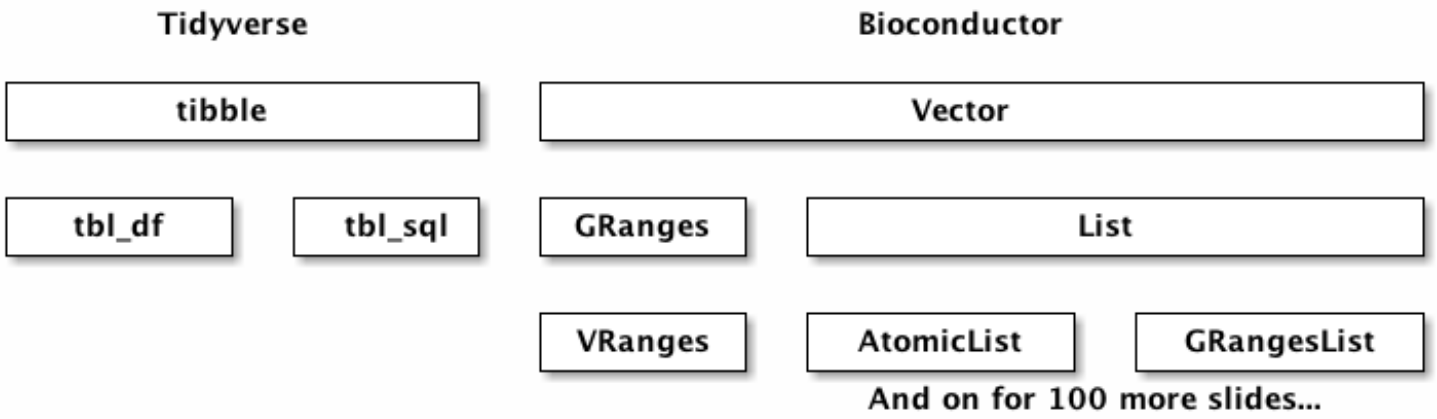
- Debugging:
  - Dispatch layers obscure stack traces
  - Selecting a method to debug is complex
    - `debug(generic, sig = "Foo")` still harder than `debug(generic.Foo)`
- Poor reliablity due to some implementation choices
  - JMC says it is time to rewrite
- Syntax for defining classes, generics and methods rubs people the wrong way

Impact

Lack of developer adoption leads to lack of emphasis on or even discouragement against S4 in R education

Challenges are more acute for Bioconductor users

- Heavy reliance on S4 features
  - Multiple dispatch
  - Complex class hierarchies
- Multiple abstractions at different semantic levels
  - Tidyverse has a single abstraction, just with multiple representations, mostly transparent to the user



Expose generics judiciously

- In principle, only accessors *need* to be generic
- Additional methods only needed if:
  - Data structures have different constraints relating accessor behavior
    - `ncol(x) == length(x)` for data.frame but not matrix
  - A method is making assumptions about how the accessors (and the data structure) are implemented (optimization)
- Those situations should be avoided
- And fixed through better abstractions
  - E.g., `NROWS()`, `extractROWS()` unify data.frame, matrix and vectors

Expose classes judiciously

- Abstract away classes that are only instantiated as intermediates
  - Hide *Hits* with high-level joins
  - Hide *RleList* with high-level coverage operation
- Still want biologically meaningful data structures
- In most cases, user only needs *GRanges(List)* and *SummarizedExperiment*

Fluent, endomorphic APIs for Bioconductor objects

- The fluent and endomorphic APIs of the tidyverse adhere to good UI practices and empirically proven to be useful
- Provide alternatives to "gets" (`accessor<-()`) syntax and maybe accessors altogether
- This syntax:

```
mcols(x)$foo <- bar
```

- Would gain a more functional alternative:

```
mutate_mcols(x, foo=bar)
```

- Reservations: non-standard evaluation

A "tidy" API for Bioconductor objects

- Genomic data fits neatly into a table
- Could map tidy API onto Bioconductor data structures
- Endomorphic constraints would only apply at the "table" level
  - Stripping range information would drop to a tibble
- Mixed metaphor problem:
  - Tidy APIs are table-oriented
  - Bioconductor objects are rarely tabular
  - Even when we can wrap Bioc objects in tabular APIs, the interface will present a mash up of models

The "Query" project

- Tidy-mapping for Bioconductor objects with deferred evaluation
- Research project with Stuart Lee @ Monash w/ Di Cook
- Will be the backend for ggbio2
- Constructs an optimized execution plan
  - Pushes down operations like restriction

Example: compute coverage and find islands

Bioconductor API

```
gr_a <- import("exons.bed")
cov <- coverage(gr_a)
ans <- GRanges(cov)
subset(ans, score > 0)
```

New Query API

```
BEDFileQuery("exons.bed") %>% import() %>%
    compute_coverage() %>% filter(score > 0)
```

- `compute_coverage()` skips the intermediate RleList
- Calling `subsetByOverlaps()` would only compute coverage for the specified regions