DARWIN EU(c) R packages: development, maintenance, and software peer review

Edward Burn, Adam Black, Ross Williams

2022-08-18T00:00:00-04:00

Table of contents

Preface

R packages are a central part in undertaking analyses in DARWIN EU(c). This book is written for developers of these R packages, to try and help us align on how we go about building, reviewing, and maintaining them. This source code for the book can be found at this Github repository Please open an issue there if you have a question or suggestion. Pull requests with suggested changes and additions are also most welcome.

1 Introduction

1.1 DARWIN EU(c)

This presentation Peter gave at an OHDSI community call gives a general introduction to the project:

And for even broader context, this webinar organised by the EMA is also worth watching:

1.2 The DARWIN EU(c) development pillar

The DARWIN EU® Coordination Centre involves 5 key pillars: 1) Management and Governance, 2) Development, 3) Technology, 4) Network Operations, and 5) Study Operations. As the name might well suggest, the creation and maintenance of analytic pipelines is a key role of the Development pillar. And at their core these analytic pipelines will be based around R packages.

The Study Operations pillar, responsible for the execution of studies in DARWIN EU(c), will rely in large part on the R packages being built and maintained by the Development pillar. For those of us working on packages this puts us at a tremendous advantage - we have a clear target audience for the tools we are building who can provide help us design what to build and provide feedback on whether what we have made is working as expected. And if making tools that make the lives of your colleagues that bit easier and more productive is not already sufficient motivation, we can keep in mind that we might already, or in the future, be contributing to the Study Operations pillar. So the tools we make today may make our lives easier in the not too distant future if they work well (or they may lead us to curse our past self if they don't!).

While the tools developed will be done so primarily to facilitate research in the DARWIN EU(c) context, they will be made freely available with an Apache 2.0 license. And as we are working with a common data model that is already widely used, our tools may well prove find a broader audience among the many researchers working with health care databases mapped to the OMOP common data model.

2 General policies

2.1 Development

All source code is version controlled on Github. Each R package has a single maintainer who is ultimately responsible for it.

We use issues for tracking bugs, features, code reviews, and anything else related to the code.

The "main" branch is assumed to be the current release. Branch protection should be added to main so that it can only be update through a pull request that has been reviewed. Feature development occurs on separate feature branches. Avoid long-lived feature branches by contributing the features to a release and removing the feature branch once the changes have been merged.

Strive for high unit test coverage of functionality. Use testthat for unit testing.

Use github workflows to add continuous integration testing to your package.

2.2 Lifecycle stages

Lifecycle badges apply at the package level. A version 1.0 release implies that the package is stable. A lifecyle badge on the github repo will be used to communicate the lifecycle stage which can be one of the following:

- Experimental
- Stable
- Superseded (still maintained)
- Deprecated (not maintained)

2.3 R package releases

R packages have releases. These releases are timed according to the needs of users and ultimately determined by the maintainer who is responsible for releasing a package. Package releases may be coordinated with the DARWIN Distribution cycle (once every 2-3 months. A release happens on github when a feature branch is merged into main. Releases should be automatically tagged and then pushed to the DARWIN package manager (forthcoming).

Before releasing a package maintainer must make sure all reverse dependency tests are passing. This involves running the unit tests on any package that depends on the package about to be released. The revdepcheck package can be used to run reverse dependency tests.

2.4 Darwin software distributions

A DARWIN Distribution is a complete set of packages and all their dependencies. It is a frozen set of software in the sense that if two people are using the same DARWIN Distribution then they are using exact same set of software. A Distribution of the complete set of R software needed for DARWIN is represented by a frozen URL from the DARWIN package manager. The frozen URL (example below) can be used to install a specific DARWIN Distribution (snapshot). Offline environments in DARWIN will use the frozen URL to install the entire set of R packages in a distribution that can be used in the future.

Use this URL to receive packages available from snapshots as of Aug 16, 2022 for the Source distribution

```
https://packagemanager.rstudio.com/all/2022-08-16+Y3JhbiwyOjQ1MjYyMTU7QTAyODhGQUE
```

A DARWIN Distribution also specifies the environment that the code is expected to run on including

- R version
- System dependencies (RTools)
- Operating System Characteristics (Linux, available RAM)
- Java version
- Environment variables
- CDM version
- Vocabulary version
- Supported DBMS

2.5 Preparing a distribution

A candidate frozen URL will be generated one or two weeks before the release date. This URL will be used for a set of integration tests. If all tests pass then the URL will be tagged as the next Darwin Distribution.

Integration test will include:

- Fresh install of all of the R packages on a new virtual machine
- Run a set of test studies on all test databases using the new installation

2.6 Distribution/Release cadence

Initially two month intervals. We expect this to reduce to 4 month intervals as the project matures.

October 1st 2022 - all packages alpha, all packages on Github

December 1st 2022 - some stable packages on CRAN?

2.7 OHDSI HADES

DARWIN will adopt some R packages from OHDSI. These packages will be evaluated for dependencies before adoption and must be on CRAN.

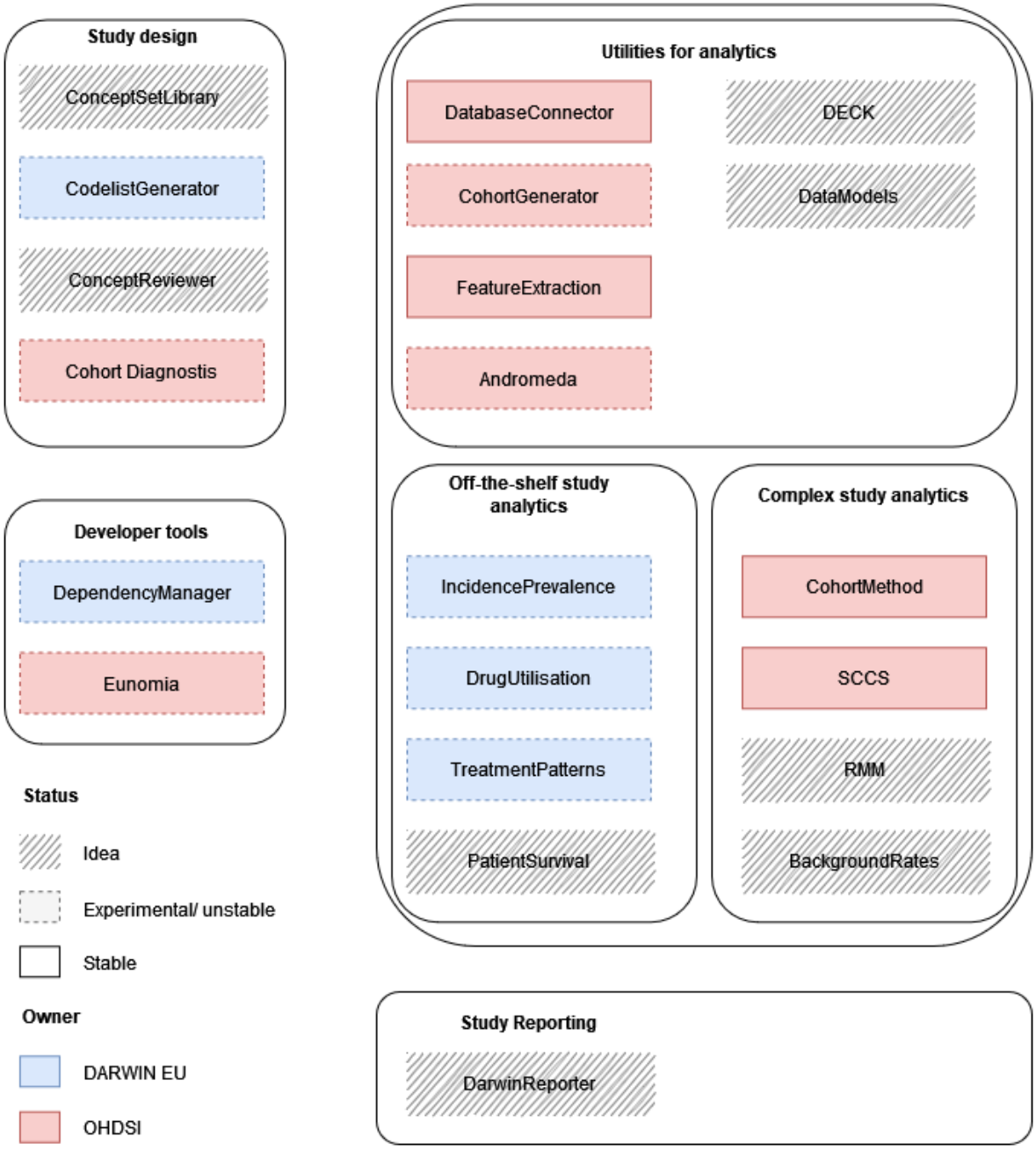
Will we fork these? If so, will we use our fork or the versions in OHDSI?

3 Development Roadmap

3.1 Current status

A general summary of packages envisaged for use in DARWIN EU(c) studies is shown below. You can see that there are many open opportunities to take a package from idea to reality, and to take experimental or unstable to a hopefully a more stable footing.

Please note this diagram is incomplete though. In particular, there are many more utilities underlying the functioning of code in DARWIN EU(c) studies. To see a full list of dependencies for existing packages please see their description files.



3.2 Study design tools

Most of the study design tools are focused on helping define study phenotypes. These are being developed to follow the workflow of phenotype development being used for DARWIN

EU(c). The end goal of the phenotyping process is to generate a cohort definition that will be stored in the DECK and will then be available for use in a study. These tools should though all be general enough to be useful for anyone generating cohort definitions for use with the OMOP common data model.

When designing phenotyping packages (see more details on the process in the next chapter), it is important to keep in mind whether it will be used against patient-level data or only query the vocabulary tables and, if the former, whether they will be run against a database in general (i.e. every time it has a refresh) or in relation to the development of a particular cohort.

3.2.1 ConceptSetLibrary

https://github.com/oxford-pharmacoepi/OmopConceptSetLibrary

https://dpa-pde-oxford.shinyapps.io/OmopConceptSetLibrary/

3.2.2 CodelistGenerator

https://github.com/darwin-eu/CodelistGenerator

3.2.3 ConceptReviewer

Ask Martí nicely for a demo of his prototype

3.2.4 CohortDiagnostics

https://github.com/OHDSI/CohortDiagnostics

3.3 Developer tools

3.3.1 DependencyReviewer

https://github.com/darwin-eu/DependencyReviewer

3.3.2 Eunomia

https://github.com/OHDSI/Eunomia

3.4 Utilities for analytics

3.4.1 DatabaseConnector

https://github.com/OHDSI/DatabaseConnector

3.4.2 CohortGenerator

https://github.com/OHDSI/CohortGenerator

3.4.3 FeatureExtraction

https://github.com/OHDSI/FeatureExtraction

3.4.4 Andromeda

https://github.com/OHDSI/Andromeda

3.4.5 DECK

3.4.6 DataModels

3.5 Off–the-shelf study analytics

3.5.1 IncidencePrevalence

https://github.com/darwin-eu/IncidencePrevalence

3.5.2 DrugUtilisation

3.5.3 TreatmentPatterns

https://github.com/darwin-eu/TreatmentPatterns

3.5.4 PatientSurvival

3.6 Complex study analytics

3.6.1 CohortMethod

https://github.com/OHDSI/CohortMethod

3.6.2 SCCS

https://ohdsi.github.io/SelfControlledCaseSeries/

3.6.3 RMM

3.6.4 BackgroundRates

3.7 Study reporting

3.7.1 DarwinReporter

4 Design considerations

4.0.1 Overview

> "Write programs that do one thing and do it well. Write programs to work together."
>
> Unix philosophy

4.0.2 Tidy tools

Tidyverse design principles

tidyverse design

https://joss.theoj.org/papers/10.21105/joss.01686

Composability - Providing tools...

Consistency - Function names - Argument names - Implementation - Inputs - omop cdm, tibble of patient level data, or aggregated results set - Output of core functions as a tidy tibble or a list of tibbles

4.0.3 Manage dependencies

https://www.tinyverse.org/

4.1

5 Creating a package

5.1 Software specification

So you want to create a new package for DARWIN. The first question to ask is "What is the responsibility of this package and why does it not fit into an existing DARWIN package?" Each DARWIN package represents a set responsibilities and exports the required necessary to meet those responsibilities. Once you have a clear idea of the package responsibility and believe the functionality needs to be separated from other packages the next set is to create the interface. In R the interface will be a collection of functions that work together to solve the problem. R packages can also have data embedded in them so also consider what data your package should export. Aim for a narrow interface that supports deeper functionality instead of a wide interface that supports shallow functionality.

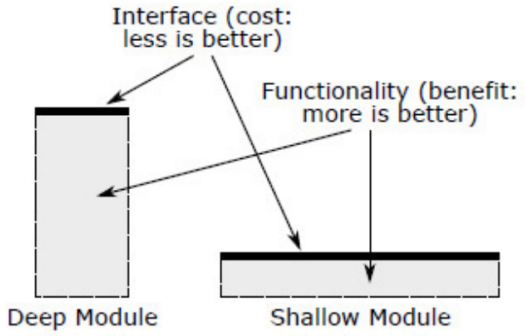


Figure 4.1: Deep and shallow modules. The best modules are deep: they allow a lot of functionality to be accessed through a simple interface. A shallow module is one with a relatively complex interface, but not much functionality: it doesn't hide much complexity.

Figure from "A Philosophy Of Software Design" by John Ousterhout

This does not mean that your package should export one function that does everything. On the contrary it is about what information can be hidden behind the interface that the user never needs to know about. The more information that can be hidden from the user of the interface, the narrower the interface needs to be. Wrapper functions that primarily pass data around tend to be shallow.

Divide what your package needs to do into three categories:

- calculations
- actions
- data

Calculations refer to so called "pure" functions that take some input and return a value. They do not modify the external state at all and have no side effects. Most of your functions should be calculations because they make code safer and easier to reason about. Given the same input a calculation will always return the same output. It does not matter when or how many times a calculation runs.

Actions are functions with side effects. They change the external world in some observable way such as saving a file, creating a new table in a database, updating a database table, printing a value, displaying a plot, changing a global option or environment variable. Your package will need actions but they should be separate from calculations.

Data are inert. Data does not need to do anything which makes it much simpler than calculations or actions. Include inert data that your package needs with `usethis::use_data(iris, internal = FALSE)`. Data can be internal or exported from the package.

Reference "Grokking Simplicity" by Eric Normand

A specification or a blueprint is simply a document that describes what the package does. It should describe the interface and the functionality. It does not describe how the package works. It also does not need to be formal. For R packages we can think of the package manual, function interface documentation, and vignettes as the specification of the software.

5.2 Test plan

As much as possible strive to test all functionality that is added to the code base. This can be difficult because much of DARWIN code is designed to run on remote databases. There will be databases set up to use for testing functionality that requires a remote database. Tests should run quickly if at all possible so that we can all run tests very frequently during development.

5.3 Prototyping

> If a picture is worth 1000 words, a prototype is worth a 1000 meetings
>
> IDEO

"The Pragmatic Programmer" describes a concept called *tracer bullets*. These are working prototypes that mimic the real implementation as much as possible with minimal effort. They are working code examples that could grow into a full solution: a "bare bones" implementation. They give the programmer rapid feedback on an idea.

5.4 An empty package

usethis and devtools

5.5 Adding features with test driven development

5.6 Documentation

> The faintest ink is more powerful than the strongest memory

Documentation is just as important as the working code. As mentioned before the package manual and vignettes are the specification of the software. Vignettes should contain working code examples that should be rebuilt before each release so that they are up to date.

5.7 Seeking feedback

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software"
>
> Agile manifesto

Software is never complete and needs to be designed to change in response to user needs. "Soft" in software is meant to imply "easy to change". Changing software in response to feedback from users and other developers is important to the success of DARWIN software.

User review is the process of requesting input from one or more potential users of your software.

Code review involves having another developer review your code, make comments, and suggest changes.

6 Contributing to a package

6.1

Each package has a maintainer who is ultimately responsible for it. Maintainers can make contributions to their packages as long as they are not breaking other packages (i.e. as long as all reverse dependency tests pass).

To contribute to a package that you are not the maintainer of:

- Open an issue describing the change you want to make
- Once the maintainer agrees they will grant you write access to the repo
- Create a new branch with a descriptive name and make your changes
- Open a pull request
- The maintainer will review and either ask for changes, make their own changes or merge the pull request.

7 Code review

7.1 Process

Before a new piece of software is released for the first time it should go through code review. Code review is tracked using Github issues.

1. Request for review is made by opening a new github issue and assigning a reviewer
2. Reviewer goes though checklist
3. Minor suggestions or comments are discussed on the github issue
4. Each significant change should become a separate github issue where it is discussed
5. The issues may or may not become pull requests depending on the discussion
6. When the code review is completed the issue is closed

7.2 Items/ Checklist for review

1. Read the documentation
    1. Is the purpose of the package/project clear from the documentation?
    2. Are the code examples understandable?
    3. Can you run the code examples yourself?
    4. Can anything be removed from code examples be simplified and still convey the same idea?
2. Read the function signatures
    1. How descriptive is the function name?
    2. Does the name of the function make use of established naming conventions in other DARWIN/OHDSI software?
    3. Do the argument names and allowed argument values make sense?
    4. Do the "data" arguments come before "detail" arguments?
    5. Does the first argument ("the data") allow the function to be used in a pipeline?

6. Do default argument values represent the most common use case? (Describe what the most common use case is.)
7. Is the function called either for it's return value or to cause a side effect? (see side effect soup)
8. If the function has a side effect such as creating a table in a database, is the side effect clearly explained? Can it be easily undone?
9. If the function is called for its side effect, could this same functionality be implemented using a return value?

3. Review function implementations
    1. Is there repeated code that can be factored out into a separate function?
    2. Does the division of responsibility makes sense?
    3. Is there only one "data" argument and does that argument come first?
    4. Can the function be used in a pipeline (i.e. does it return a modified version of it's first argument)
    5. Is the function covered by unit tests that guarantee it produces the correct result?

7.3

7.4

8 Package maintenance

References

A Getting started with R

- Probably out of scope for this guide - would fit better if we make something similar but for analysts….

A.1 Installing R and R Studio

A.2 Installing and loading packages

A.3 Scripts, projects, and packages

A.4 Base R

A.5 Tidyverse

A.6 Wrting a function

Now you've written a function, you're ready to make a package!

B Working with databases from R

Let's start by taking some data and putting it in a database. Here we'll use an in-memory duckdb database, but the same code should work for other databases with only the connection details and the package used to connect to the database changing.

For this example let's use data on Darwin's finches as that seems rather appropriate (link to wiki article on darwin finches)

```
# install packages
# commented out as you might already have them
# but if not then uncomment and run
# install.packages("DBI")
# install.packages("SQLite")
# install.packages("dbplyr")
# install.packages("dplyr")
#
# # load packages
# library(DBI)
# library(SQLite)
# library(dbplyr)
# library(dplyr)

# get data

# move into a database
```

B.1 show_query()

B.2 filter(), select(), mutate()

B.3 right_join(), left_join(), inner_join(), and anti_join()

B.4 summarise()

B.5 collect() and compute()

B.6 working with dates

Here be dragons

B.7 working with strings

B.8 bespoke sql

Alternative approaches

1) Where to do computation

- Database side vs in local memory vs R

2) Scope of a package

3) Scope of analysis code All in one vs one at a time

C Working with Apache Arrow from R

The Apache Arrow project defines two data formats for tabular data. The Arrow Dataset is an in-memory format for columnar data that can be accessed and used from multiple programming languages including R, python, and Java, and more. The feather file format is an on-disk format that can be used to efficently store and manipulate larger than memory dataframes. The arrow R package provides tools for working with both of these from R.

The Andromeda R package provides a way to manipulate larger than memory dataframes in R. There is an open issue to convert Andromeda to arrow and much of that work has been completed but not yet released in OHDSI. Andromeda objects are simply a list references to on-disk feather files that can be manipulated from R using `dplyr`. Andromeda should only be used if data cannot be constrained to available RAM.

D Working collaboratively

D.1 Roles

1. Leads
2.

D.2 Communication

D.2.1 GitHub repositories

D.2.2 Meetings

D.2.3 Other ad hoc

Emails, teams messages etc