

# Lessons for Designing Scalable and Maintainable Shiny Apps

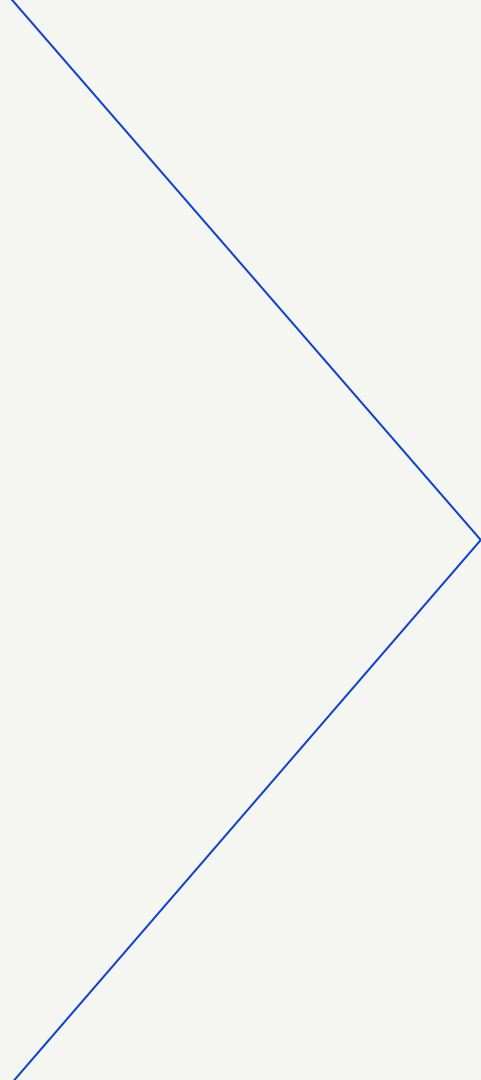
Pawel Rucki  
Daniel Sabanes Bove  
Adrian Waddell

# Tutorial Objectives

Learn how to productionize your Shiny app!

- How to design your solution
- How to use Shiny modules to make configurable Shiny apps
- How to simplify the reactivity graph
- How to implement R6 classes for advanced use cases
- How to organize the code in packages for testing and deployment

# Table of contents

- 
- A large blue graphic on the left side of the slide, consisting of a vertical line and two diagonal lines that meet at a point, forming a large right-pointing arrow shape.
1. Introduction
  2. Designing the Solution
  3. Mastering the Implementation
  4. Packaging the Product
  5. Summary

# Introduction

# Motivation

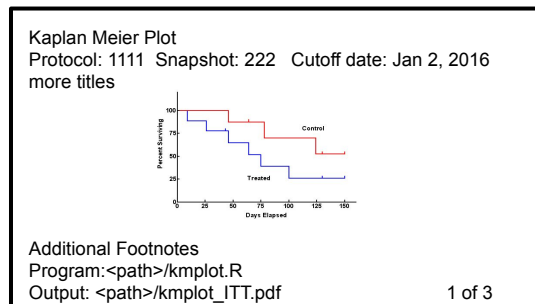
- We want to build a **production grade Shiny app** that is scalable and maintainable.
- One example is **teal**
  - which has useful design considerations
  - which is now open sourced ([github.com/insightsengineering/teal](https://github.com/insightsengineering/teal))
- Today we focus on **design & implementation** & **testing** considerations.
- We illustrate these with examples and provide respective code for you to try out after the presentation.

# Product Overview: teal

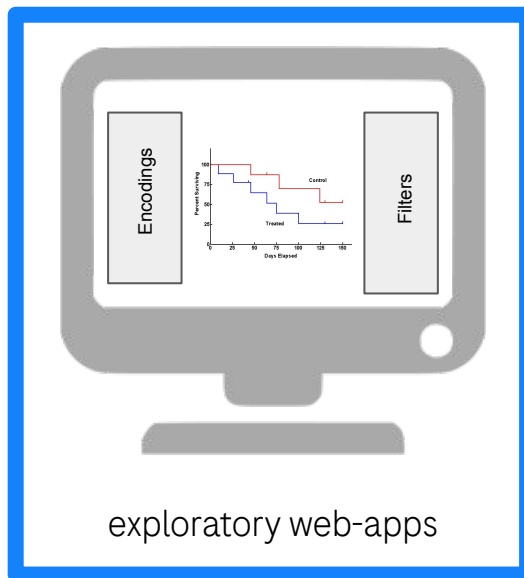
- At useR 2021 Adrian Waddell gave an introduction to some of the R tools in development at Roche to analyze clinical trials data
  - see the video recording here: <https://youtu.be/8kB6jrErgrk?t=3376>

# Product Overview: teal

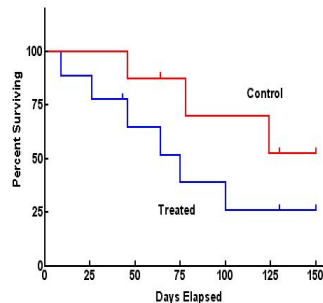
- At useR 2021 Adrian Waddell gave an introduction to some of the R tools in development at Roche to analyze clinical trials data
  - see the video recording here: <https://youtu.be/8kB6jrErgrk?t=3376>
- Recap



reviewing static outputs



```
> plot(survfit(Surv(AVAL ~ !CNSR) ~ ARM, data = ADTE))
```



CLI ad-hoc analysis  
rmarkdown

# Example: teal product

teal is a R shiny based framework that allows you to:

- to streamline the creation of interactive data analysis web applications
- dynamically filter/subset the data
- change the encoding of the output (tables, model summaries, data tables, graphs)
- reproduce analyses via show-R-code functionality
- create reports
- access data from various sources/databases requiring authentication
- work with various data models
  - e.g. general relational data, biomarker data



demo

[https://genentech.shinyapps.io/teal\\_efficacy/](https://genentech.shinyapps.io/teal_efficacy/)

# Business Context & Impact

then

- analyses performed with proprietary data analysis software
- talents came with R skills from academia, little R work available
- few central shiny apps that required analyst large upfront configuration
- one RStudio instance and one shiny server

now

- analyses performed with proprietary software & R (incl. teal)
- teal apps (R) are recommended as part of the analysis plan
- lots of study specific teal apps that allow analysts directly to jump into analyzing study data
- evolving compute and data infrastructure, validated and non-validated systems

# What's next

- We looked at the example (teal) for a production grade Shiny app
- Now we look at various aspects of Shiny app development to create a scalable and maintainable shiny based framework:
  - Designing the Solution
  - Mastering the Implementation
  - Packaging the Product

# Designing the Solution

# Problem statement

User Story

As a **data scientist ...**

... in order to **perform interactive clinical data analysis ...**

... I need an **app that will access my dataset(s) and create tabulations and visualizations.**

# Problem statement

Who? What? When? Where? Why?

- Who?** Data Scientists / Statisticians / Subject Matter Experts
- What?** An application visualizing the data
- Why?** Interactive Exploratory Data Analysis
- Where?** Web application
- When?** ASAP!

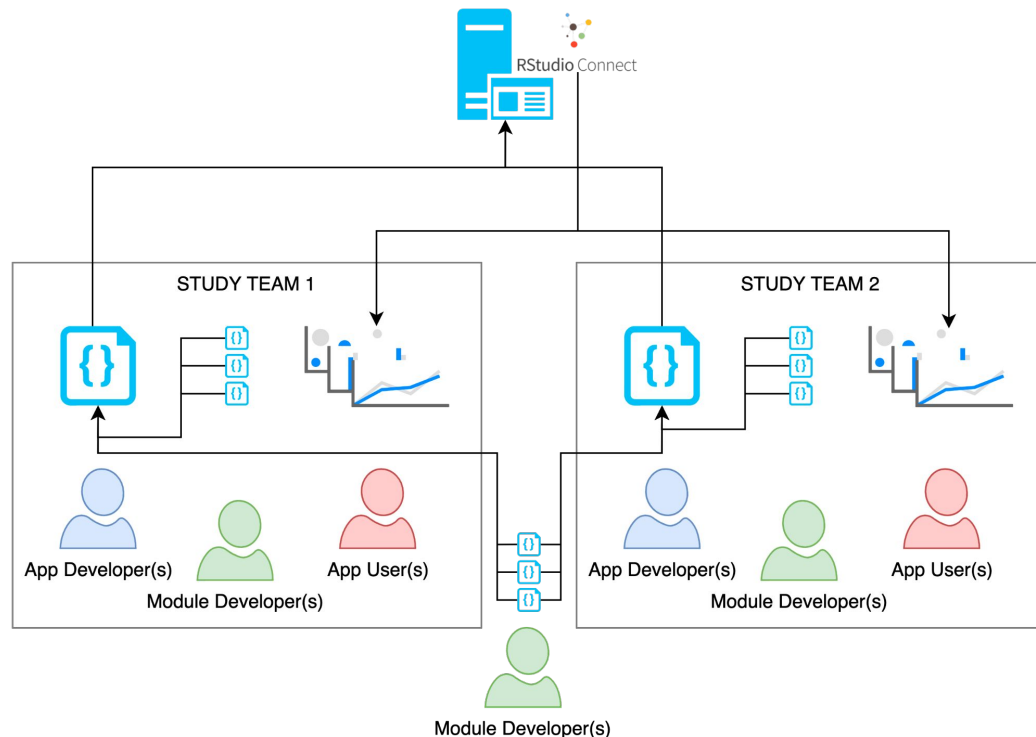
# Proposed solution

Set of **R packages** enabling users to create **fit-for-purpose apps** that meet specific requirements.

In particular: *not* a single application that has many various outputs.

This approach has number of **advantages**:

- flexibility
- scalability
- ease of customization and configuration
- access rights
- R package release cycle
- ...



# Proposed solution - stakeholders definition

App Developer, Module Developer and App User



## App Developer

R programmer

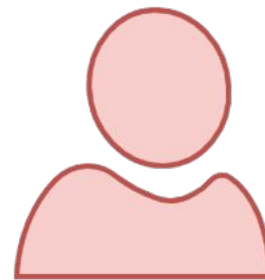
Creates an app and pushes it to the application server.



## Module Developer

R programmer

Creates reusable modules / functions to be used by App Developer.



## App User

Data Analyst

User of the app. Interactively creates visualisations, interpret it and draw conclusions.



# Proposed solution - stakeholders definition

App Developer and Module Developer - Who does what?

```
module_a <- function(...){...}
module_b <- function(...){...}
module_c <- function(...){...}
```



Module Developer

```
app <- create_app(
  data = get_my_data(...),
  modules = list(
    module_a(...),
    module_b(...),
    module_c(...)
  ),
  ...
)
```

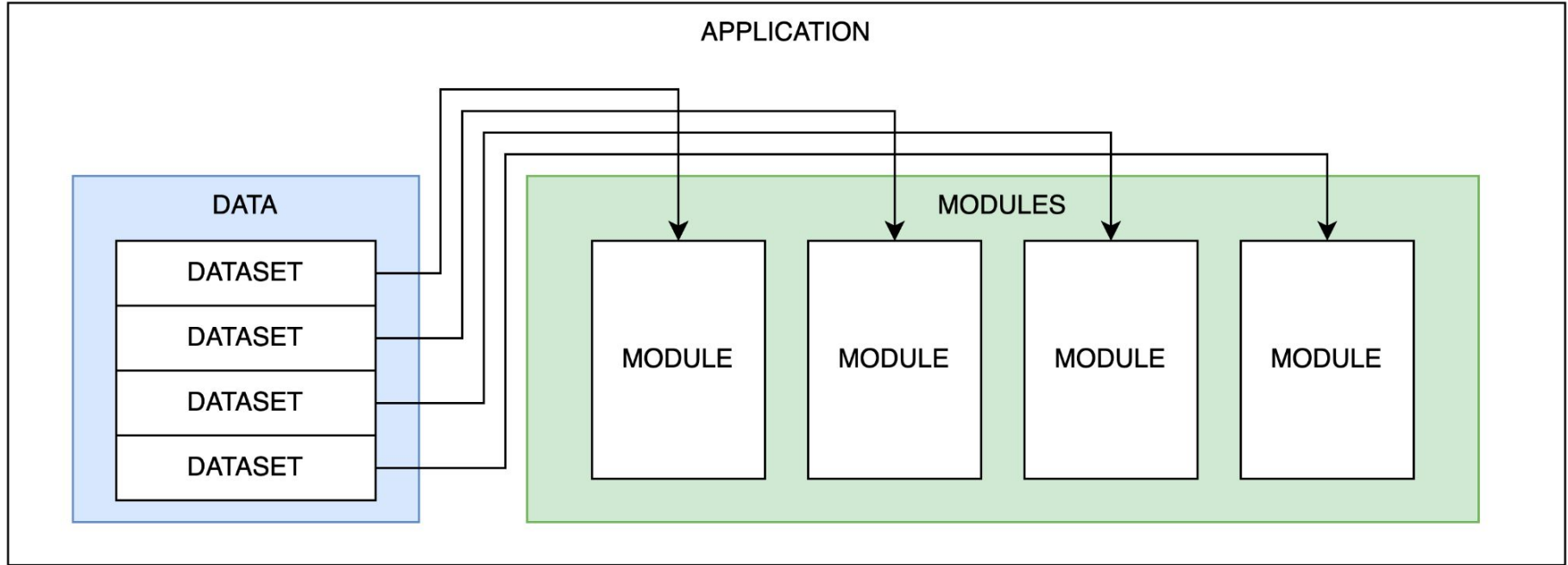


App Developer

```
runApp(app)
```

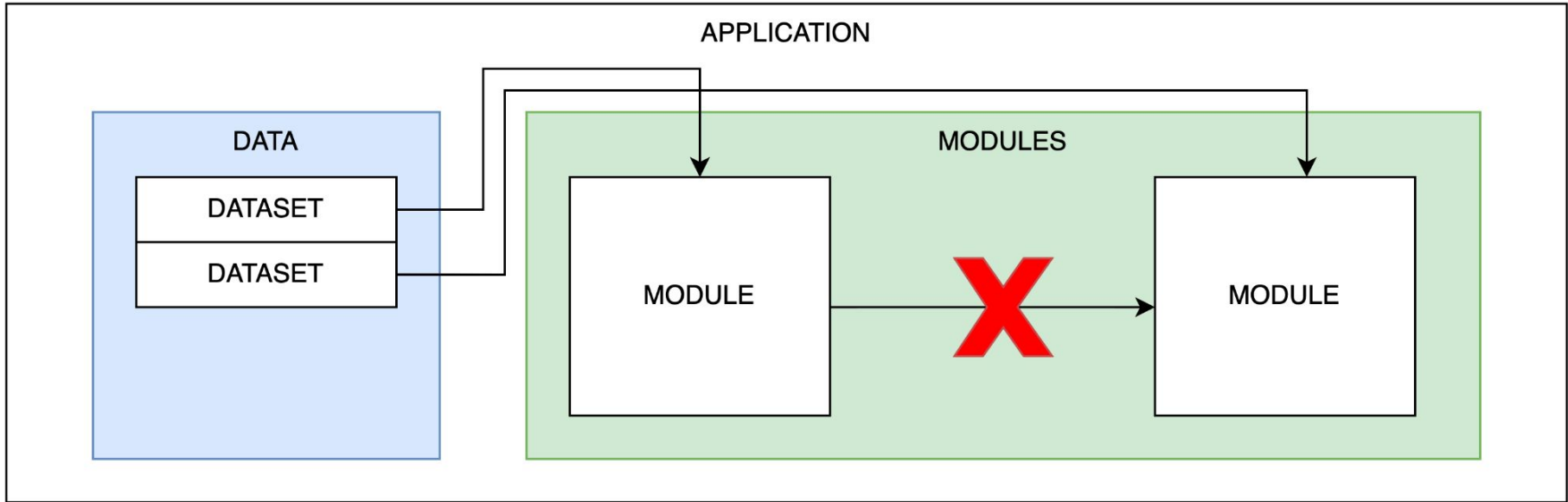
# Design - data flow

Data and Modules



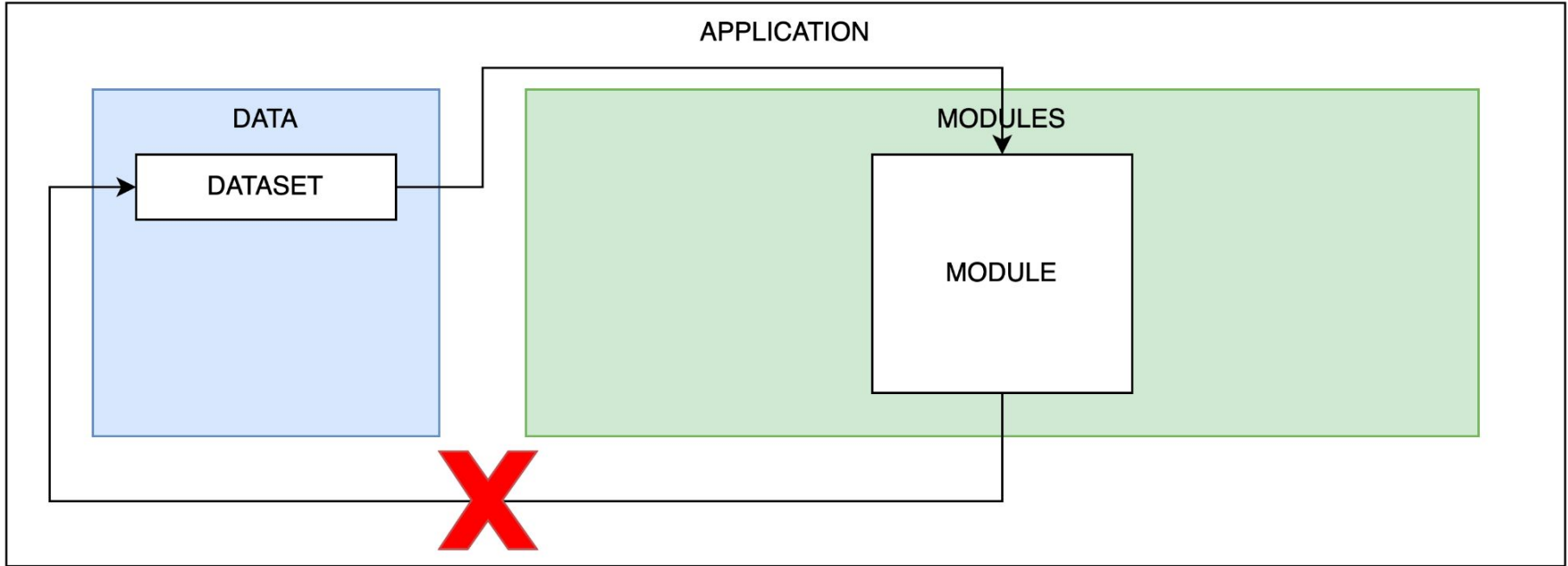
# Design - data flow

Data and Modules - no interaction between modules

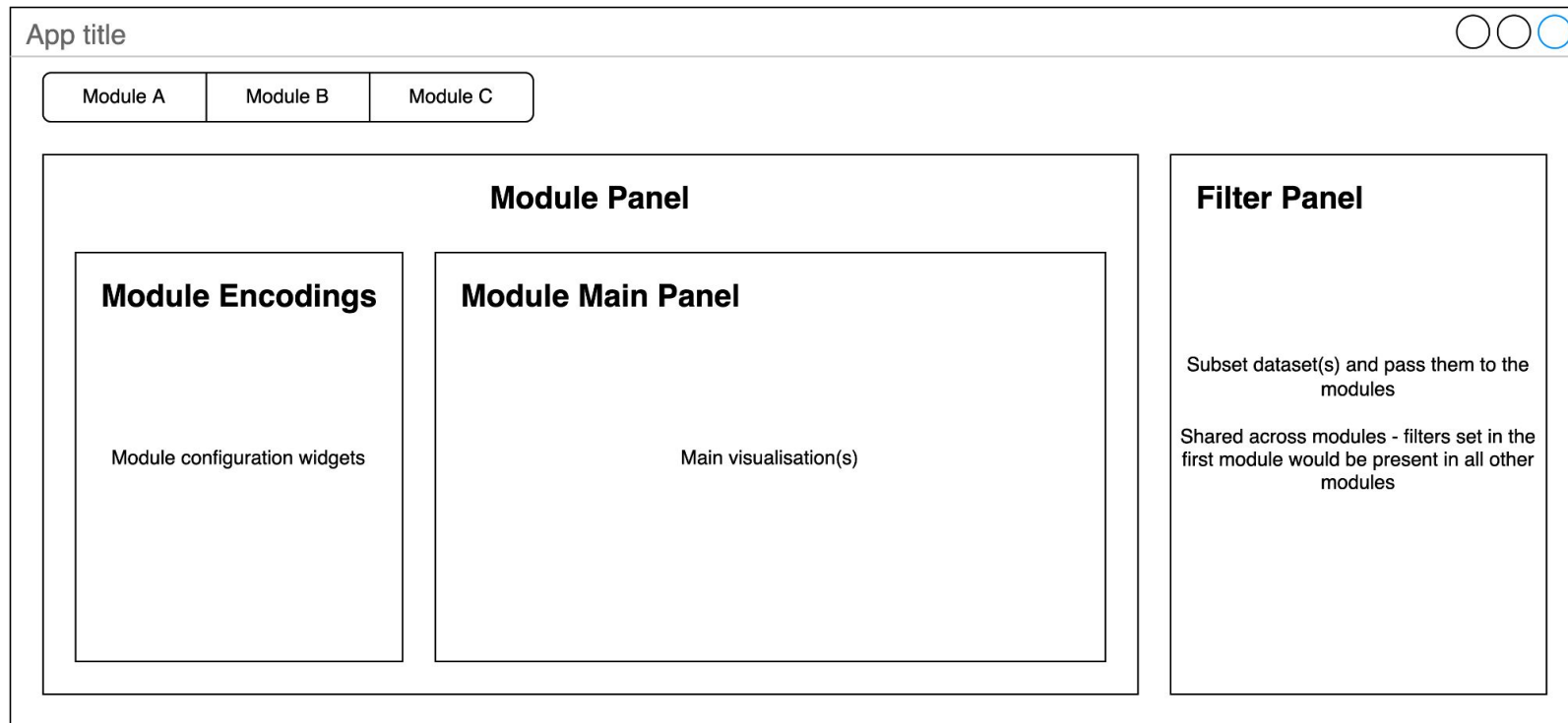


# Design - data flow

Data and Modules - immutable data

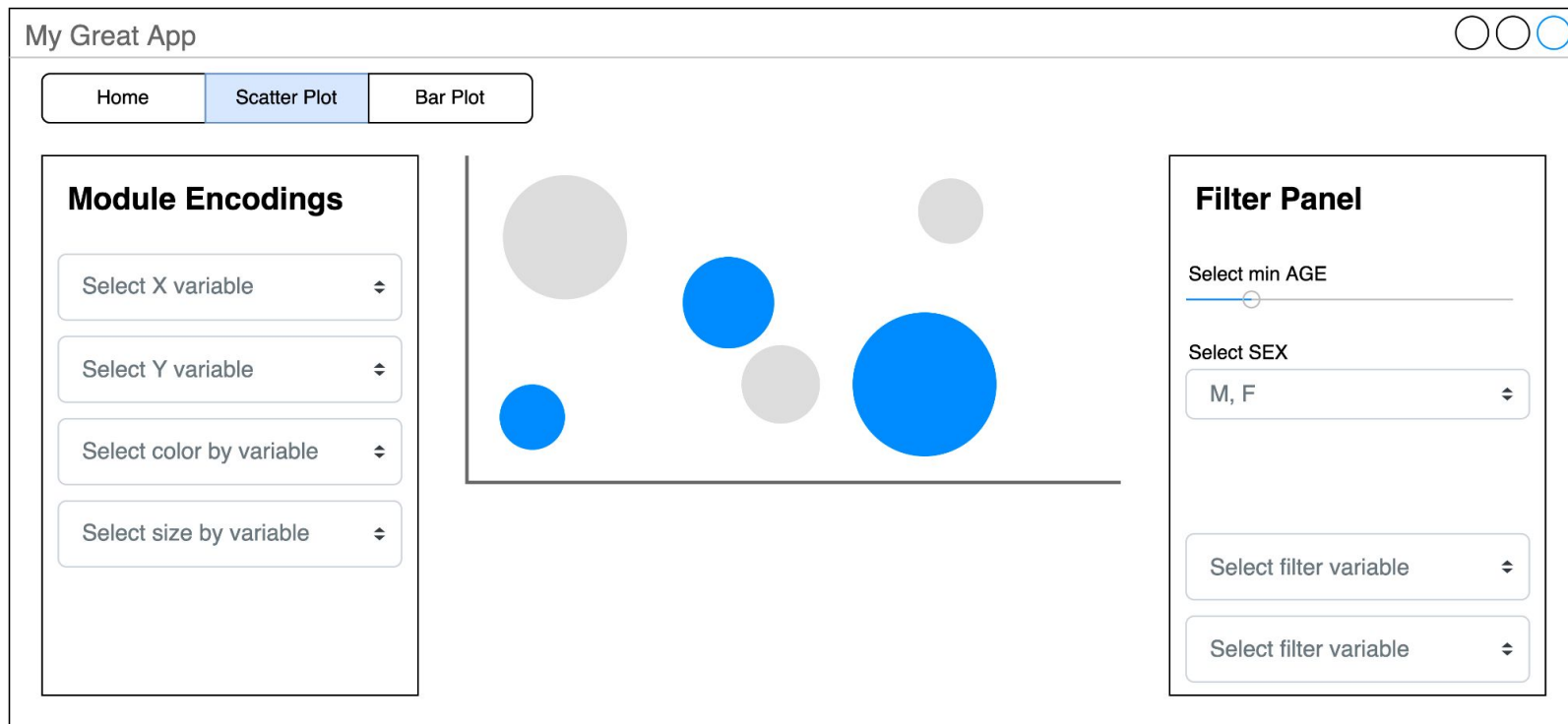


# Design - application layout



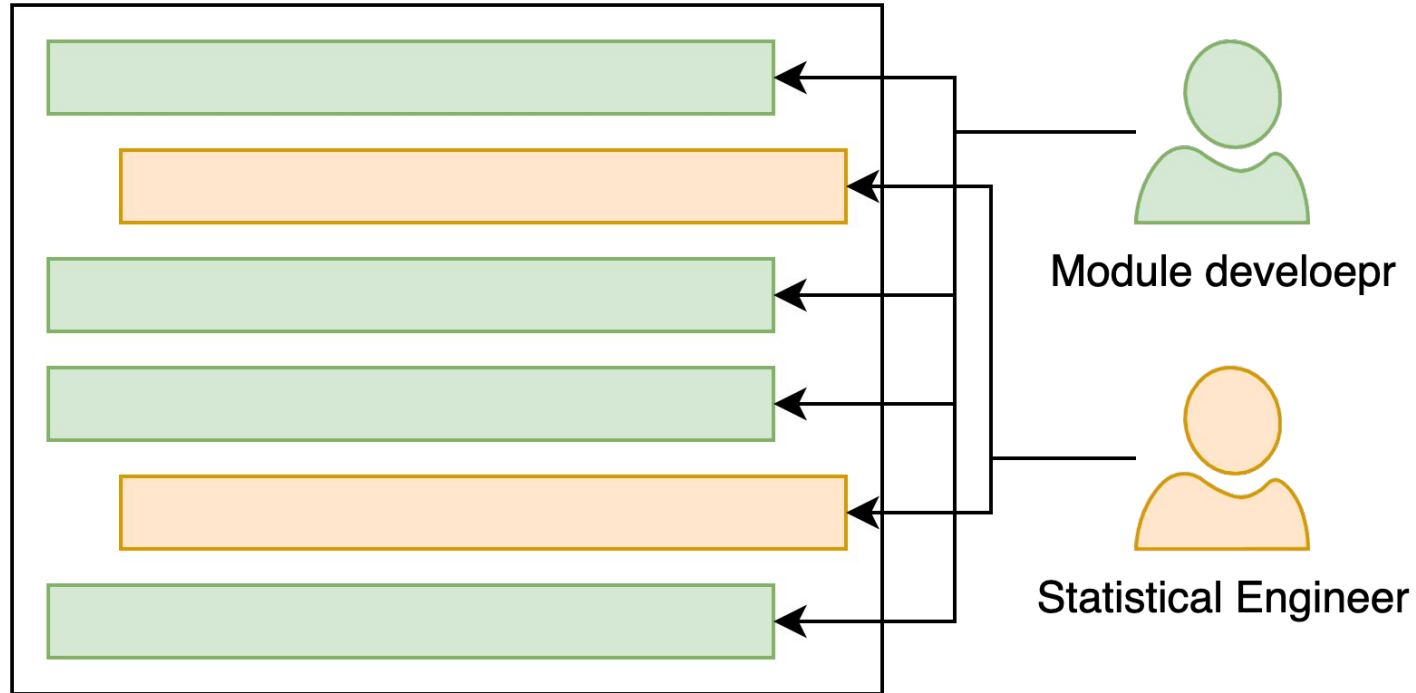
# Design - application layout

Example



# Module design

Decouple shiny reactive programming from business logic - **design**



# Module design

Decouple shiny reactive programming from business logic - **functions**

```
foo_calc <- function(x, y, z, ...) {  
  (...)  
}  
  
foo_plot <- function(x, ...) {  
  (...)  
}  
  
my_module <- function(...) {  
  return(list(  
    server = function(id, ...) {  
      shiny::moduleServer(id, function(input, output, session) {  
        (...)  
        results <- reactive({  
          foo_calc(input$x, r_y(), r_z(), ...)  
        })  
        output$plot <- reactive({  
          foo_plot(results(), ...)  
        })  
        (...)  
      })  
    },  
    ui = ...,  
    ...  
  ))  
}
```



# Module design

Decouple shiny reactive programming from business logic - **package objects**

```
my_module <- function(...) {  
  return(list(  
    server = function(id, ...) {  
      shiny::moduleServer(id, function(input, output, session) {  
        (...)  
        results <- reactive({  
          pkg::foo_calc(input$x1, r_y(), r_z(), ...)  
        })  
        output$plot <- reactive({  
          pkg::foo_plot(results(), ...)  
        })  
        (...)  
      })  
    },  
    ui = function(id, ...) {  
      ns <- shiny::NS(id)  
      shiny::div(...)  
    },  
    ...  
  ))  
}
```

# Module design

Decouple shiny reactive programming from business logic - **pros and cons**

- **Single responsibility** principle
- Easier to **trace and debug** (avoid long functions that does too much)
- **Testability**
  - Often business logic depends on the data - separating it enables testing using various test data sets
  - More accurate test results
- **Reusability**
  - calculations could be used in multiple modules
  - calculations could be used outside of the modules

## Module design (cont.)

Decouple shiny reactive programming from business logic - pros and cons

- Better **file structure** management
  - e.g. you know where to look when introducing changes due to bugs / feature requests
- If separated into the packages:
  - Better **package dependencies** management - module *requires* {shiny} et al. whereas business-logic-package *requires* statistical packages
  - Better **release cycle** - you could release either part as opposed to everything together
- Requires **integration tests** :(
  - (more on it in testing part later on)

# Module design


The art of function definition

- What should be configurable?
- What arguments should be exposed to the app developer?
- How to organize arguments?
- How to write good and maintainable code?

# Module design

The art of function definition

use: arguments in the right order



```
foo <- function(  
  x,  
  y,  
  color_by,  
  title,  
  subtitle,  
  legend,  
  h_line,  
  v_line,  
  ...  
) {...}
```

# Module design

The art of function definition

avoid: dependencies between arguments

use: parameter object pattern

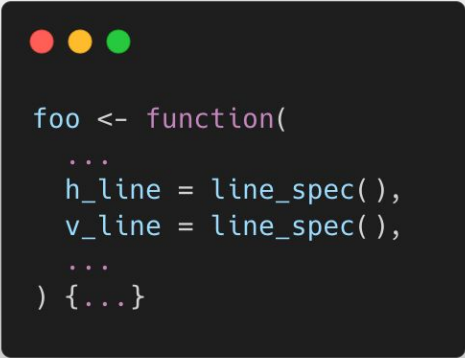
```
foo <- function(  
  ...  
  h_line_val,  
  h_line_color,  
  h_line_label,  
  h_line_xyz,  
  ...  
) {...}
```

```
foo <- function(  
  ...  
  h_line = line_spec(),  
  ...  
) {...}
```

# Module design

The art of function definition

use: consistent naming convention



```
foo <- function(  
  ...  
  h_line = line_spec(),  
  v_line = line_spec(),  
  ...  
) {...}
```

# Module design

The art of function definition

avoid: ambiguous class of arguments values

```
foo <- function(  
  ...,  
  title = FALSE,  
  ...  
)
```

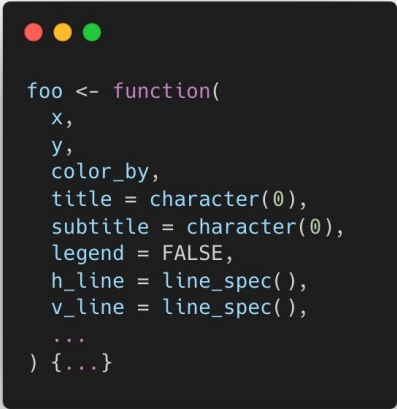
```
foo <- function(  
  ...,  
  title = character(0),  
  ...  
)
```



# Module design

The art of function definition

use: required arguments shouldn't have defaults; enumerate possible values



```
foo <- function(  
  x,  
  y,  
  color_by,  
  title = character(0),  
  subtitle = character(0),  
  legend = FALSE,  
  h_line = line_spec(),  
  v_line = line_spec(),  
  ...  
) {...}
```

# Module design

The art of function definition

use: defensive programming via assertions on arguments value

```
foo <- function(  
  ...,  
  title = character(0),  
  legend = FALSE,  
  h_line = line_spec(),  
  v_line = line_spec(),  
  ...  
) {  
  assertthat::assert_that(  
    is.character(title) && length(title) == 1  
  )  
  assertthat::assert_that(  
    assertthat::is.flag(legend)  
  )  
  assertthat::assert_that(  
    inherits(h_line, "line_spec")  
    inherits(v_line, "line_spec")  
  )  
}
```

# Module design

The art of function definition

Know the use case: your module to be called together with other modules

- Keep consistency across the module functions (within a package)
- Session settable arguments of multiple functions
  - use: defaults from R option

```
foo <- function(  
  ...,  
  color = getOption("pkg.color")  
  ...  
) {...}  
  
bar <- function(  
  ...,  
  color = getOption("pkg.color")  
  ...  
) {...}
```

# Module design

Make your code easy to maintain by avoiding code smells

- **avoid: Code Smells**

- **Bloaters**

- Code that have increased to such size that they are difficult to work with.*

- Examples: large functions, long parameter list

- **Object-Orientation Abusers**

- Incomplete or incorrect application of object-oriented programming principles.*

- Examples: switch statements, alternative functionalities with different interfaces

- **Change Preventers**

- Introducing change in one place require many changes in other places.*

- **Dispensables**

- Something unneeded whose absence would make the code cleaner, more efficient and easier to understand.*

- Examples: comments, duplicate code, dead code

- **Couplers**

- Excessive coupling between functionalities*

# Module design

Make your code easy to maintain by using principles

- **use: SOLID**

- details → <https://en.wikipedia.org/wiki/SOLID>
- especially: open-closed principle:  
"Software entities ... should be open for extension, but closed for modification."

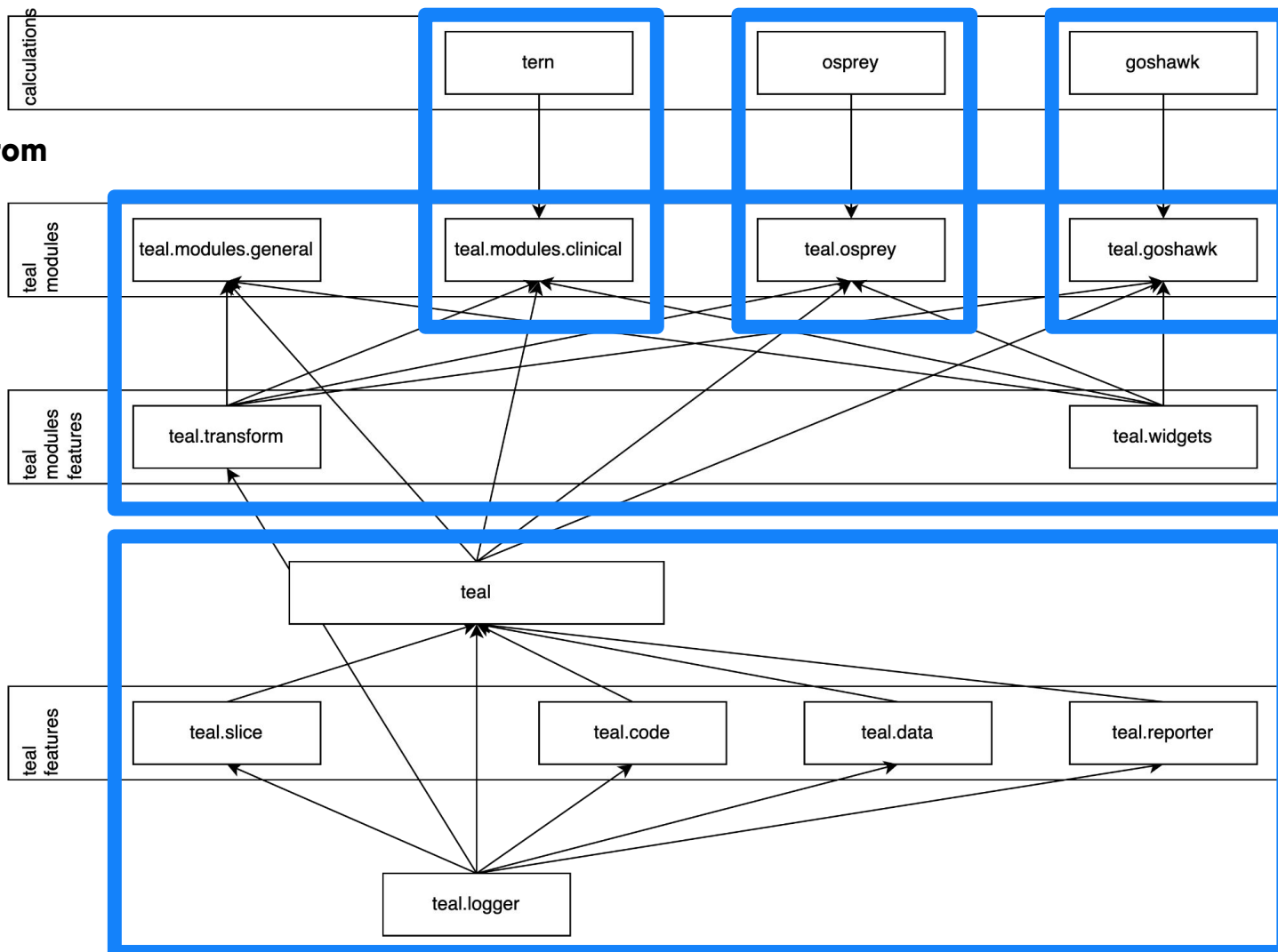
- **use: YAGNI**

- "You aren't gonna need it"
- a programmer should not add functionality until deemed necessary

**Decoupling  
business logic from  
shiny layer**

**Object pattern**

**Decoupling  
functionals**



# Designing the Solution: Summary

- Business context: switch towards open-source technologies such as R.
- **Know Your Customer**: identify the roles and responsibilities. Make reasonable assumptions (especially regarding technical skills) so as not to require too much.
- A decision to create **suite of tools** (R packages) used to create fit for purpose applications. Our users are analysts who are capable to write R codes on their own.
- **Scalability**: Horizontal scaling via modules and vertical scaling via module customization.
- **Maintainability**: R packages providing good code management, user documentation and more.
- It's relevant to split business logic from application codes.
- Write a good and **clean** code! Especially for the public interface (i.e. exported functions).

## Further materials

- [tidyverse design guide](#)
- [Engineering Production-Grade Shiny Apps](#)
- [Design Patterns](#)



# Mastering the Implementation

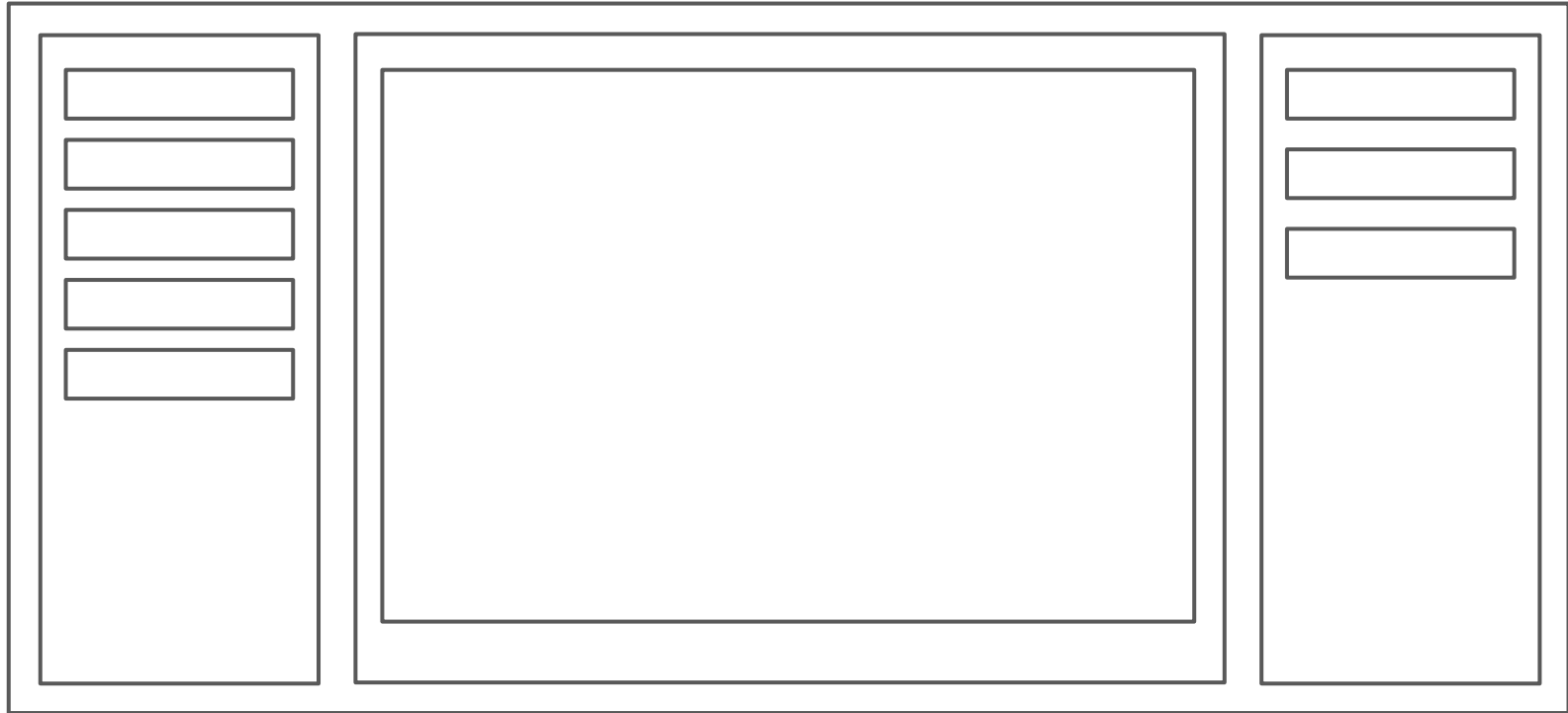
**Mastering the Implementation → Reactivity in {shiny}**

# Reactivity - Overview

- When building production shiny apps you won't get around learning reactivity
- Some guidance:
  - keep user interface simple → often keeps reactivity simple → often less confusing to user
  - resolve reactive inputs early on & validate generously
  - design towards "stringy" reactivity graphs
  - don't interrupt reactivity
  - preferably reactive over reactiveValues & observeEvent over observe

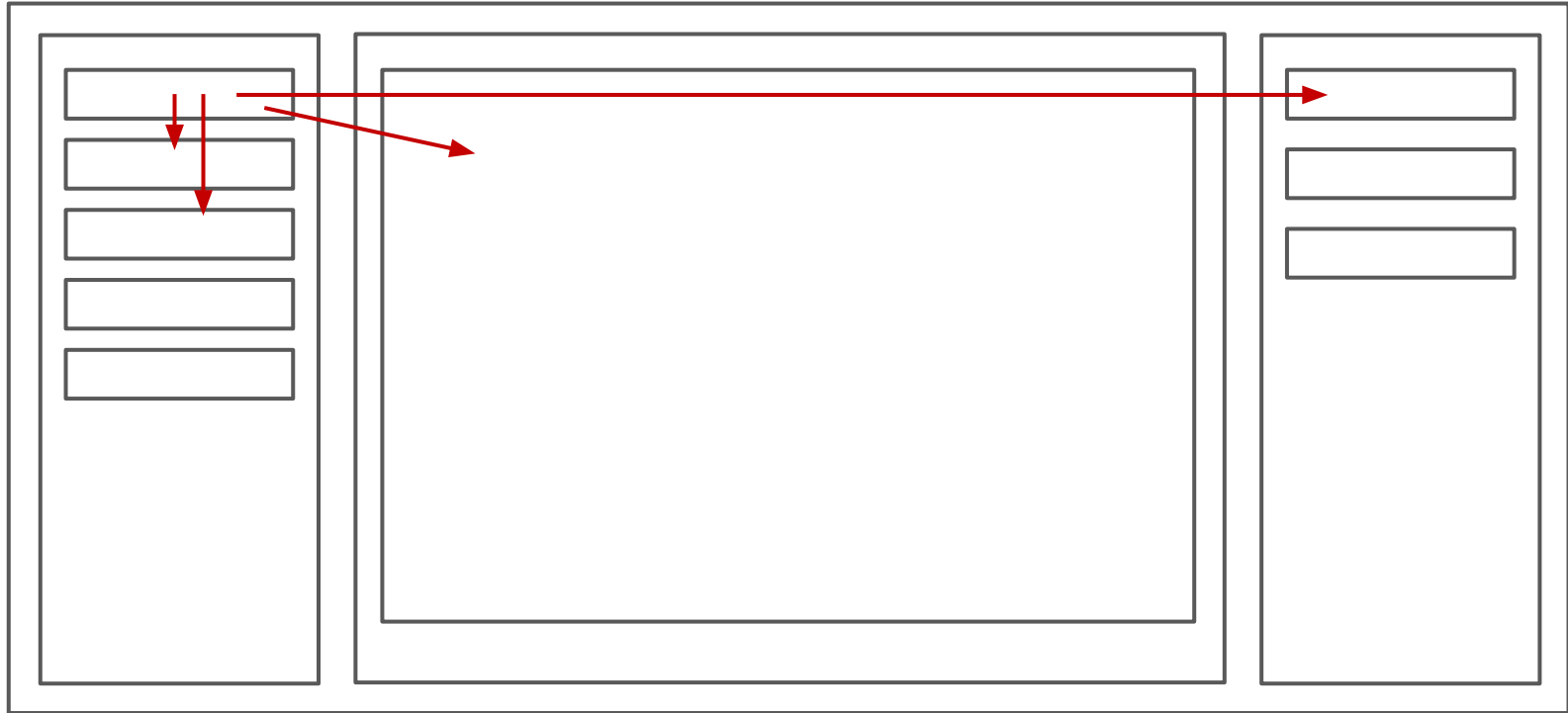
# Keep the user interface simple

what changes what?



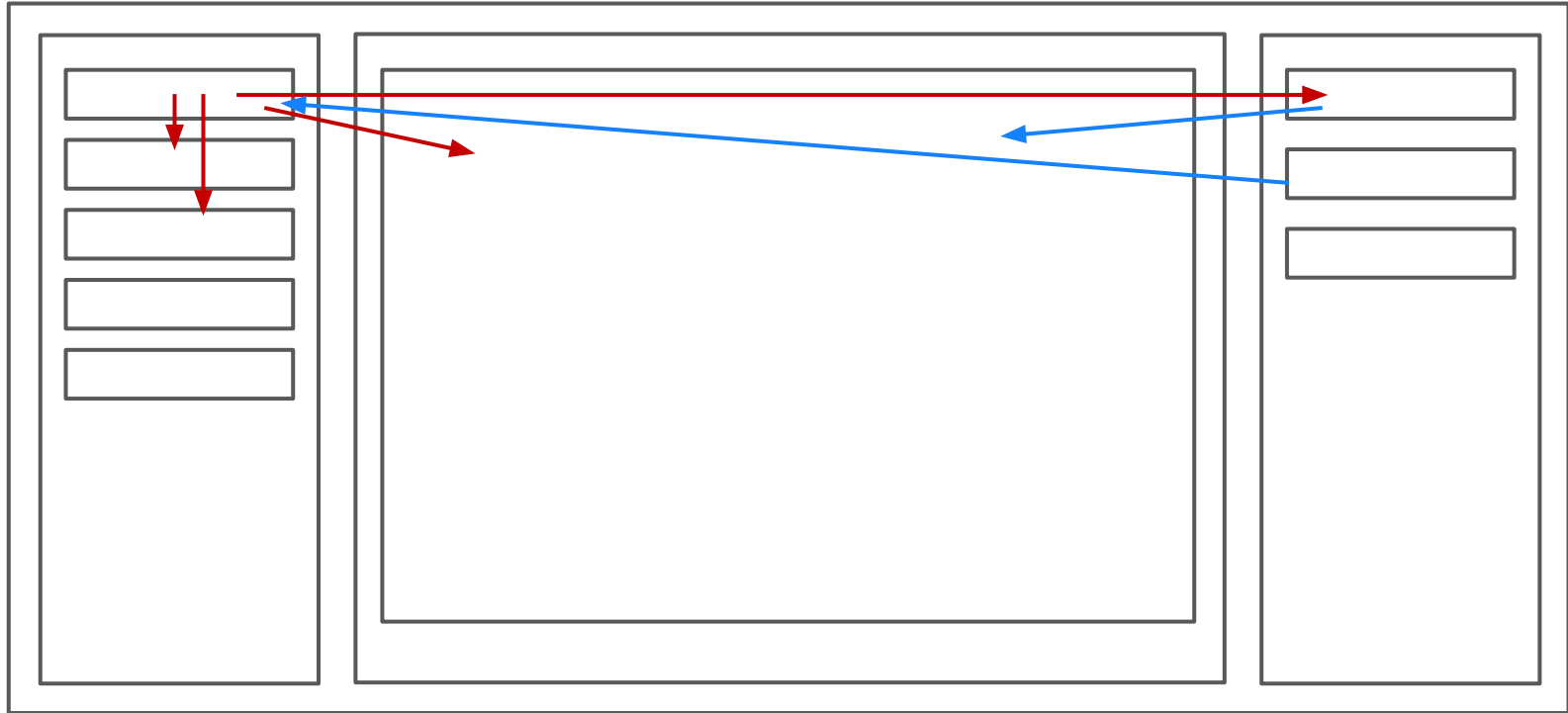
# Keep the user interface simple

what changes what?



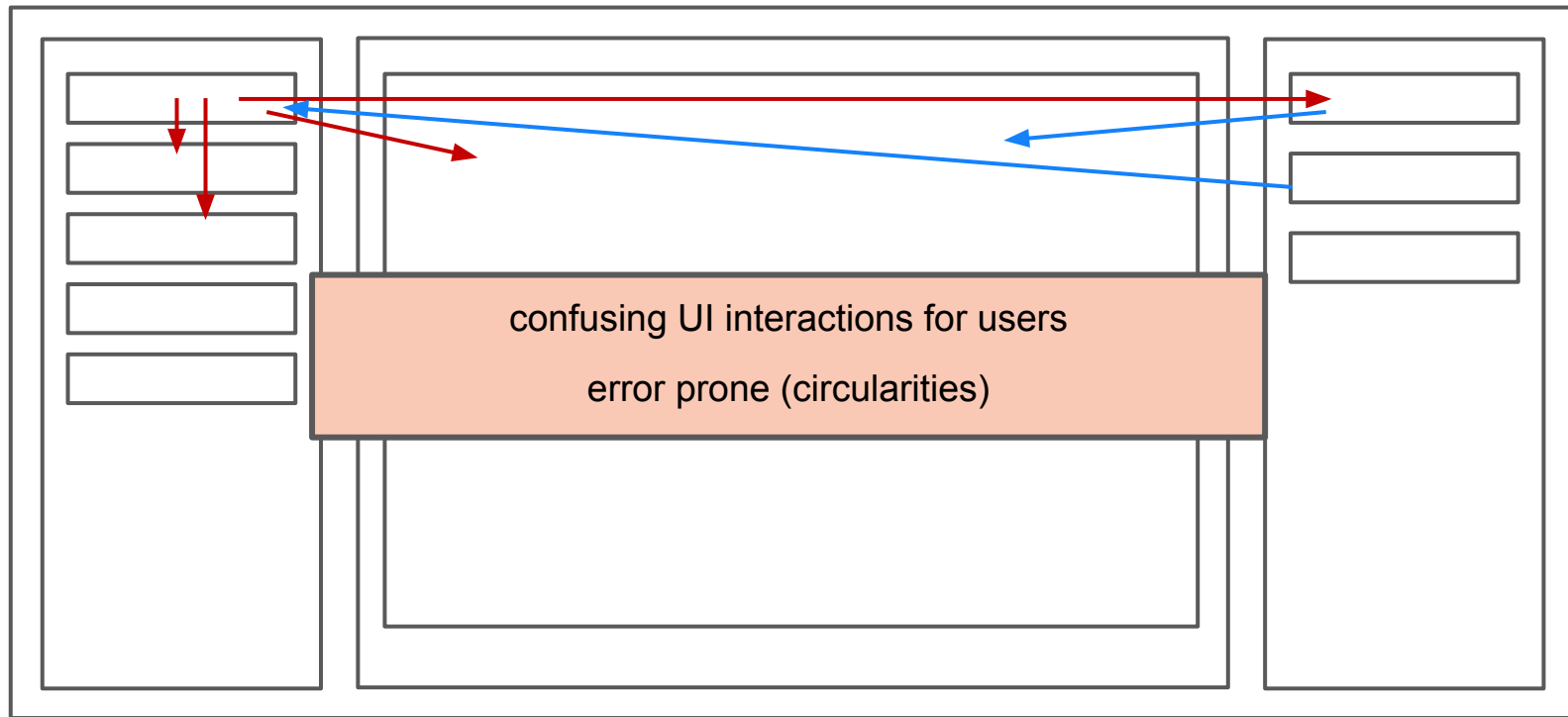
# Keep the user interface simple

what changes what?



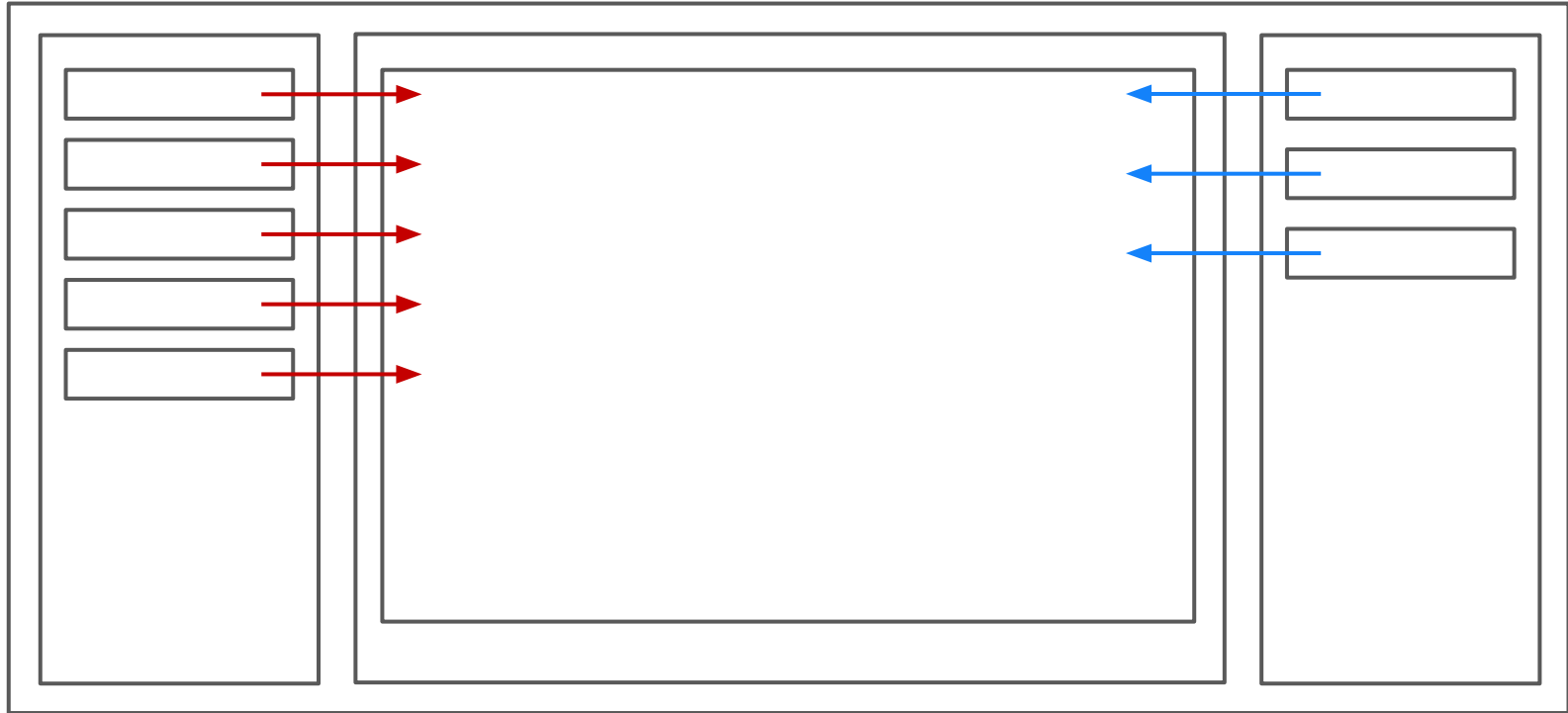
# Keep the user interface simple

what changes what?



# Keep the user interface simple

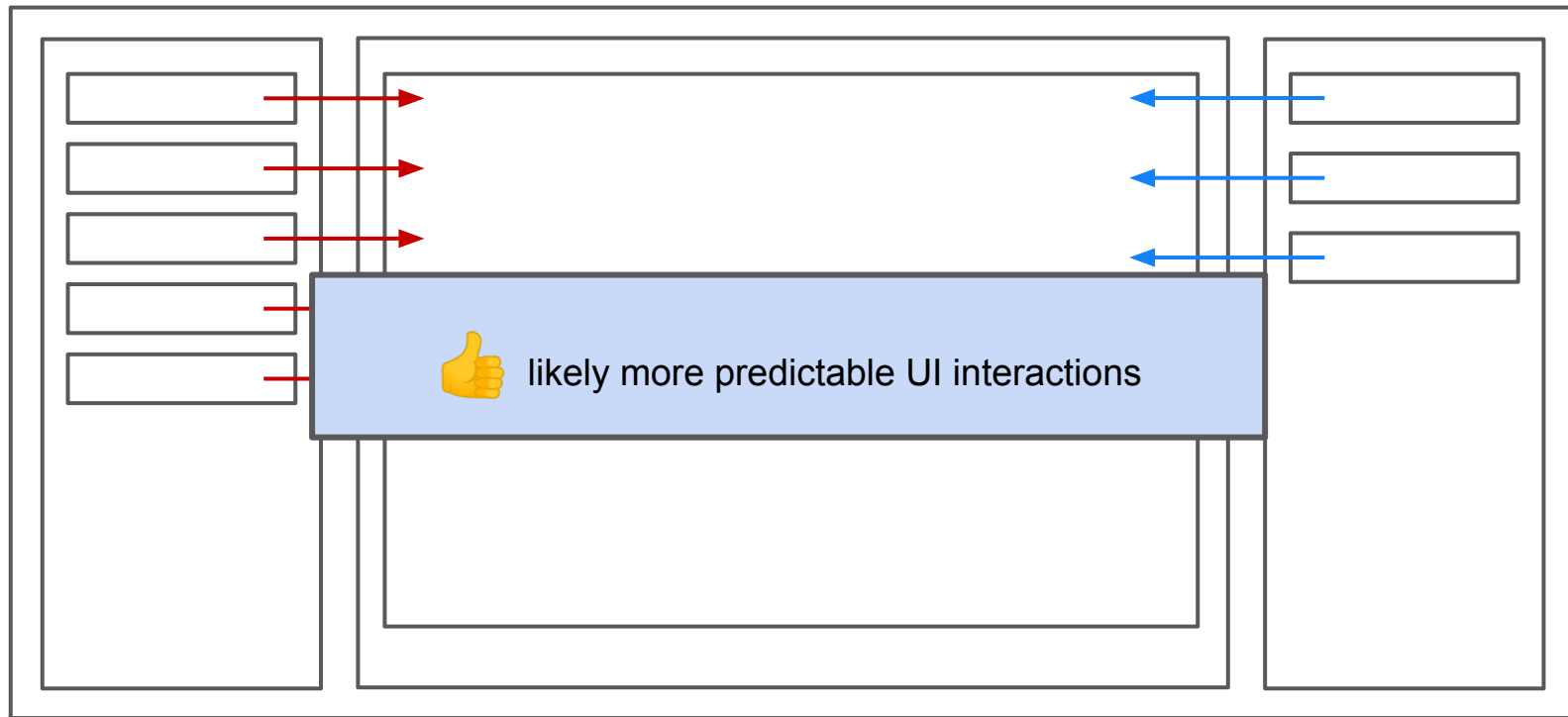
what changes what?





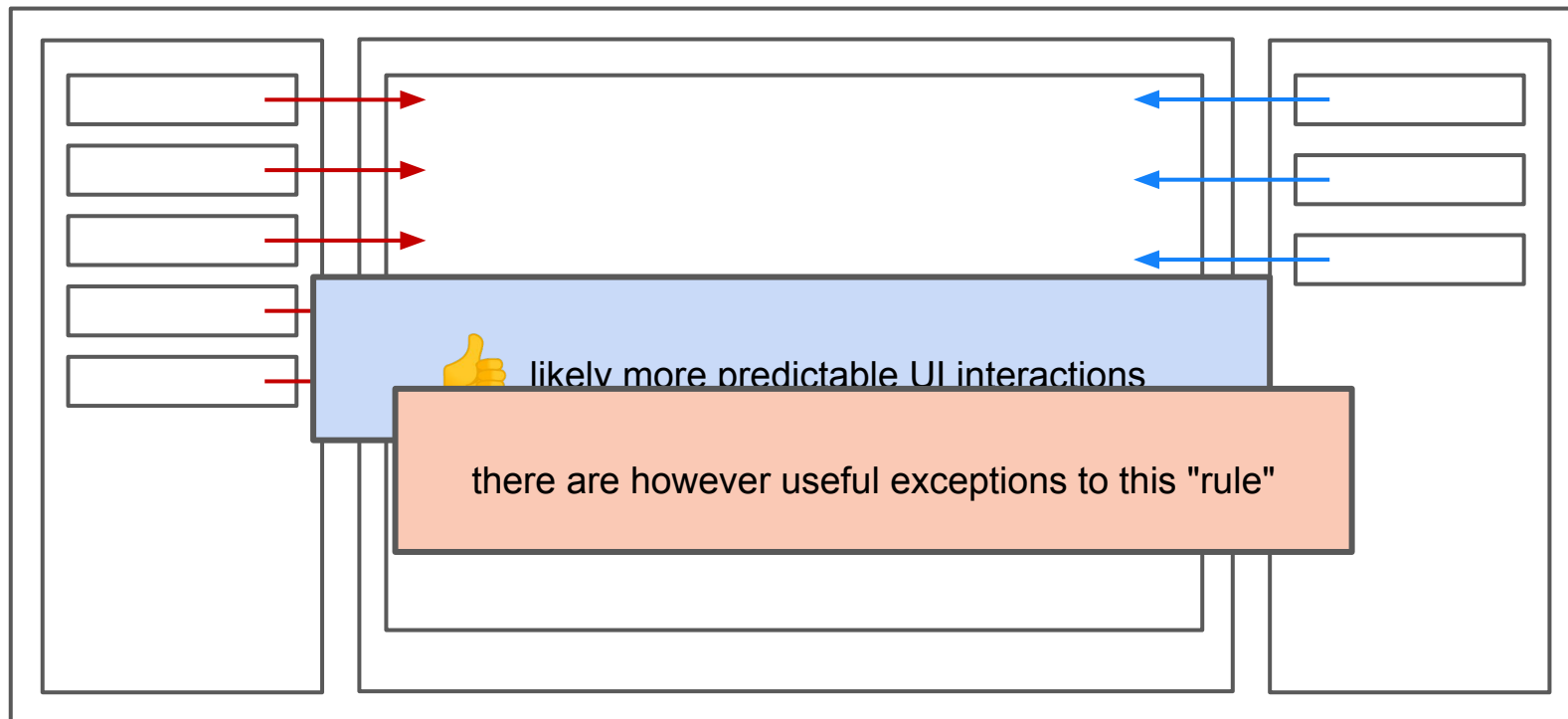
# Keep the user interface simple

what changes what?



# Keep the user interface simple

what changes what?



# Resolve reactive inputs early on

and validate generously

```
server <- function(input, output, session) {  
  output$plot <- renderPlot({  
    # resolve reactive values  
    xvar <- input$xvar  
    yvar <- input$yvar  
    req(xvar, yvar)  
  
    # validate - is it likely that the plot will be meaningful?  
    validate(  
      need(xvar %in% names(df), glue("xvar \"{xvar}\" does not exist in data")),  
      need(yvar %in% names(df), glue("yvar \"{yvar}\" does not exist in data")),  
      need(nrow(df) > 3, "too few data points for meaningful plot")  
    )  
  
    # write "business logic" in a unit testable function  
    my_special_plot(df[[xvar]], df[[yvar]])  
  })  
}
```

# Resolve reactive inputs early on

and validate generously

```
server <- function(input, output, session) {  
  output$plot <- renderPlot({  
    # resolve reactive inputs early  
    xvar <- input$xvar  
    yvar <- input$yvar  
    req(xvar, yvar)  
  
    # validate - is it likely that the plot will be meaningful?  
    validate(  
      need(xvar %in% names(df), glue("xvar \"{xvar}\" does not exist in data")),  
      need(yvar %in% names(df), glue("yvar \"{yvar}\" does not exist in data")),  
      need(nrow(df) > 3, "too few data points for meaningful plot")  
    )  
  
    # write "business logic" in a unit testable function  
    my_special_plot(df[[xvar]], df[[yvar]])  
  })  
}
```

users should not see red error messages in the app,  
grey messages from validate are clearer

invalidation also propagate in reactivity graphs

# Resolve reactive inputs early on

and validate generously

```
server <- function(input, output, session) {
```

```
  output$plot <- renderPlot({
```

```
    # resolve reactive
```

```
    xvar <- input
```

```
    yvar <- input
```

```
    req(xvar, yvar)
```

users should not see red error messages in the app,  
grey messages from validate are clearer

invalidation also propagate in reactivity graphs

```
    # validate -
```

```
    validate(
```

```
      need(xvar
```

```
      need(yvar
```

```
      need(nrow
```

if your app keeps crashing at a certain part, it may help  
to write the relevant values to the global environment to  
debug your code

```
      st in data")),
```

```
      st in data")),
```

```
    )
```

```
  # write "business logic" in a unit testable function
```

```
  my_special_plot(df[[xvar]], df[[yvar]])
```

```
})
```

```
}
```

# Resolve reactive inputs early on

and validate generously

```
server <- function(input, output, session) {  
  output$plot <- renderPlot({  
    # resolve reactive values  
    xvar <- input$xvar  
    yvar <- input$yvar  
    req(xvar, yvar)
```

```
# validate -
```

```
validate(  
  need(xvar  
  need(yvar  
  need(nrow  
)
```

if your app keeps crashing at a certain part, it may help  
to write the relevant values to the global environment to  
debug your code

```
st in data")),  
st in data")),  
)
```

```
# write "business logic" in a unit testable function
```

```
.GlobalEnv$xvar <- xvar; .GlobalEnv$yvar <- yvar; stop()
```

```
my_special_plot(df[[xvar]], df[[yvar]])
```

```
})
```

# Design toward "stringy" reactivity graphs

Reactive source



Reactive conductor



Reactive endpoint



<https://shiny.rstudio.com/articles/reactivity-overview.html>

# Design toward "stringy" reactivity graphs

Reactive source



Reactive conductor



Reactive endpoint



<https://shiny.rstudio.com/articles/reactivity-overview.html>

You can visualize your reactivity graph with [reactlog](#).



# Design toward "stringy" reactivity graphs

Reactive source



Reactive conductor



Reactive endpoint



<https://shiny.rstudio.com/articles/reactivity-overview.html>

You can visualize your reactivity graph with [reactlog](#).

Plan your reactivity graphs to be "stringy" rather than "highly connected graphs".

# Design toward "stringy" reactivity graphs

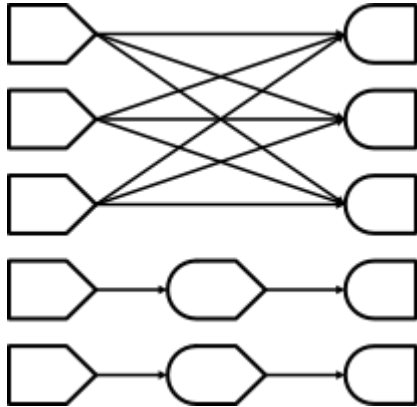
Reactive source



Reactive conductor



Reactive endpoint



Many edges make it difficult for developers to anticipate reactivity behaviour

# Design toward "stringy" reactivity graphs

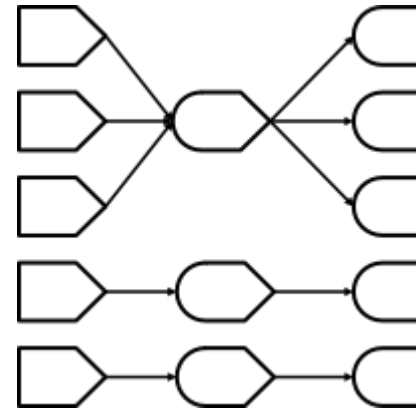
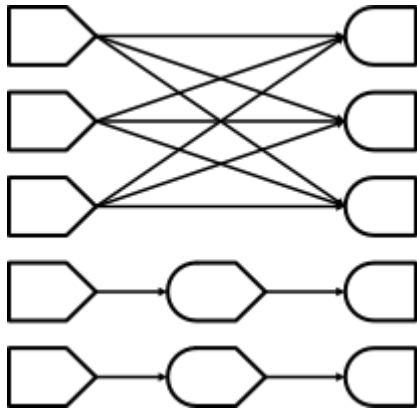
Reactive source



Reactive conductor

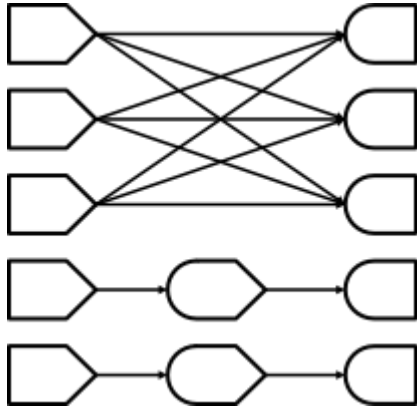
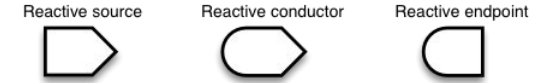


Reactive endpoint

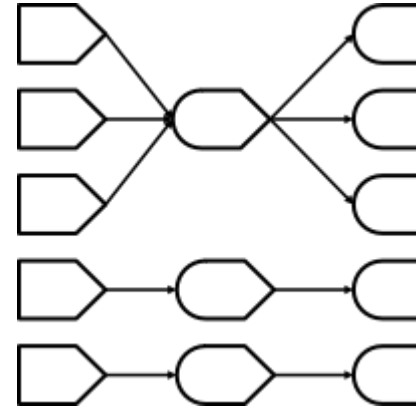


Many edges make it difficult for developers to anticipate reactivity behaviour

# Design toward "stringy" reactivity graphs

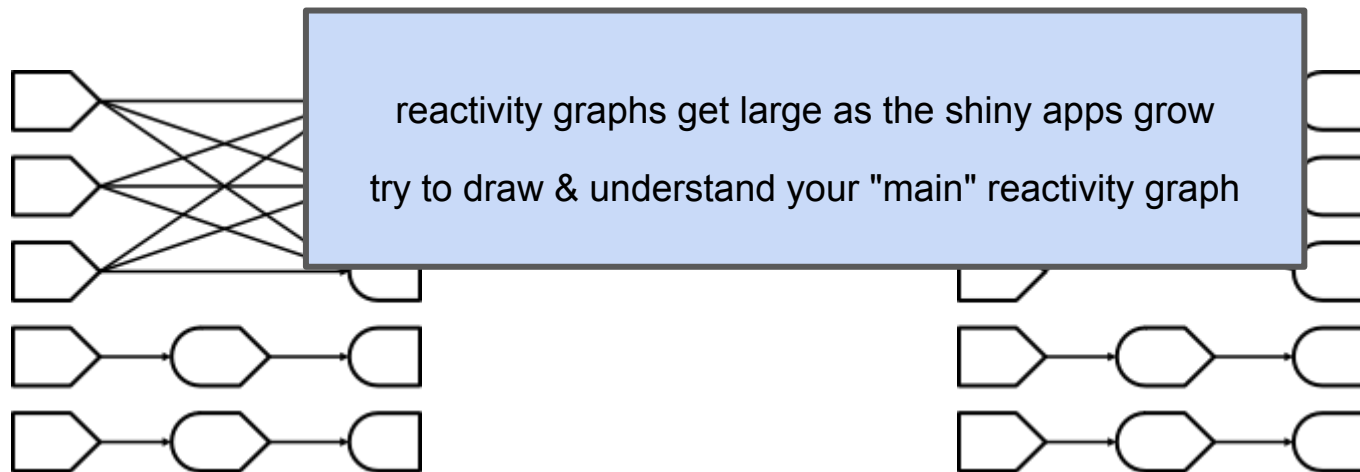
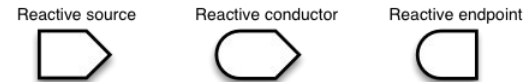


Many edges make it difficult for developers to anticipate reactivity behaviour



This reactivity graph is easier to understand and debug

# Design toward "stringy" reactivity graphs



Many edges make it difficult for developers to anticipate reactivity behaviour

This reactivity graph is easier to understand and debug

# Don't interrupt reactivity



Main disadvantages

- lazy evaluation (non-visible endpoints)
- can lose default shiny bookmarking feature

# Don't interrupt reactivity, simple example

```
library(shiny)

ui <- fluidPage(
  titlePanel("Interrupt Reactivity"),

  sidebarLayout(
    sidebarPanel(sliderInput("x", "select x:",
min = 1, max = 50, value = 30)),
    mainPanel(verbatimTextOutput("txt"))
  )
)

double_last <- function(x, name) {
  append(x, setNames(tail(x, 1) * 2, name))
}
```

```
server <- function(input, output) {

  a <- reactive({
    xval <- input$x
    double_last(c(x = xval), "a")
  })

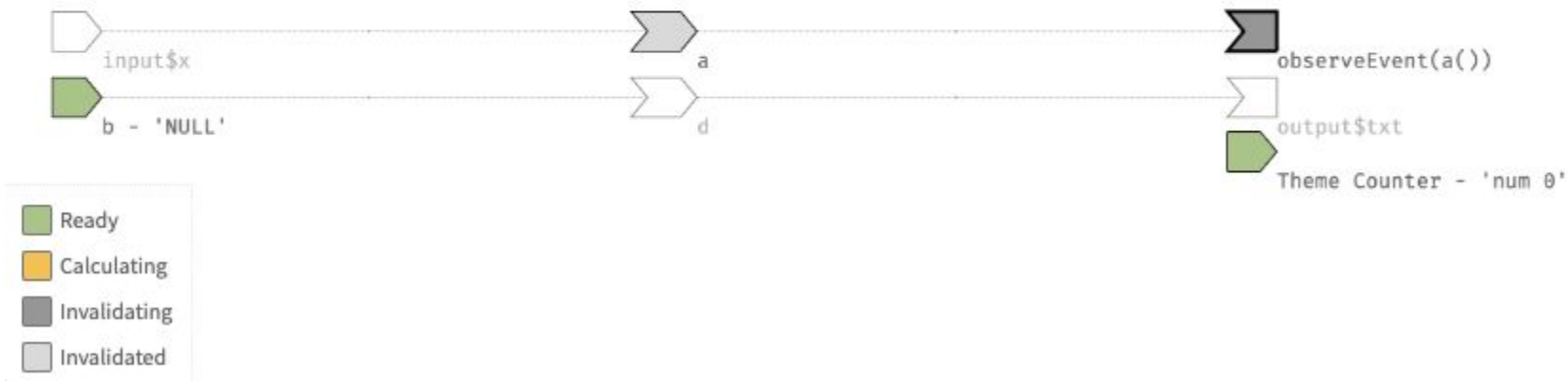
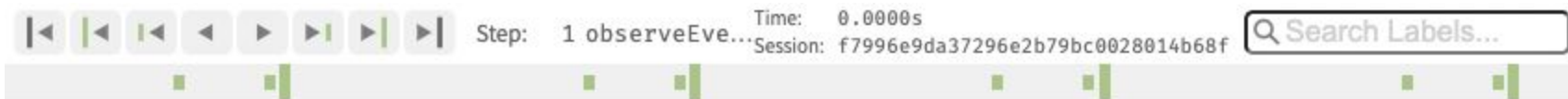
  b <- reactiveVal(NULL)
  observeEvent(a(), {
    aval <- a()
    b(double_last(aval, "b"))
  })

  d <- reactive({
    bval <- b()
    double_last(bval, "d")
  })

  output$txt <- renderText({
    dval <- d()
    req(dval)
    paste(paste(names(dval), "=", dval), collapse = "\n")
  })
}

shinyApp(ui = ui, server = server)
```

# Don't interrupt reactivity





## reactive vs. reactiveValues & observeEvent vs. observe

We have found that when there are scenarios where either

- `reactive` or `reactiveValues`
- `observeEvent` or `observe` with `isolate`

can solve the problem → using `reactive` and `observeEvent`, respectively, was preferable.

## Further Material

<https://shiny.rstudio.com/articles/understanding-reactivity.html>

<https://shiny.rstudio.com/articles/reactivity-overview.html>

<https://shiny.rstudio.com/articles/scoping.html>

<https://rstudio.github.io/reactlog/>



**Mastering the Implementation → OOP in R**

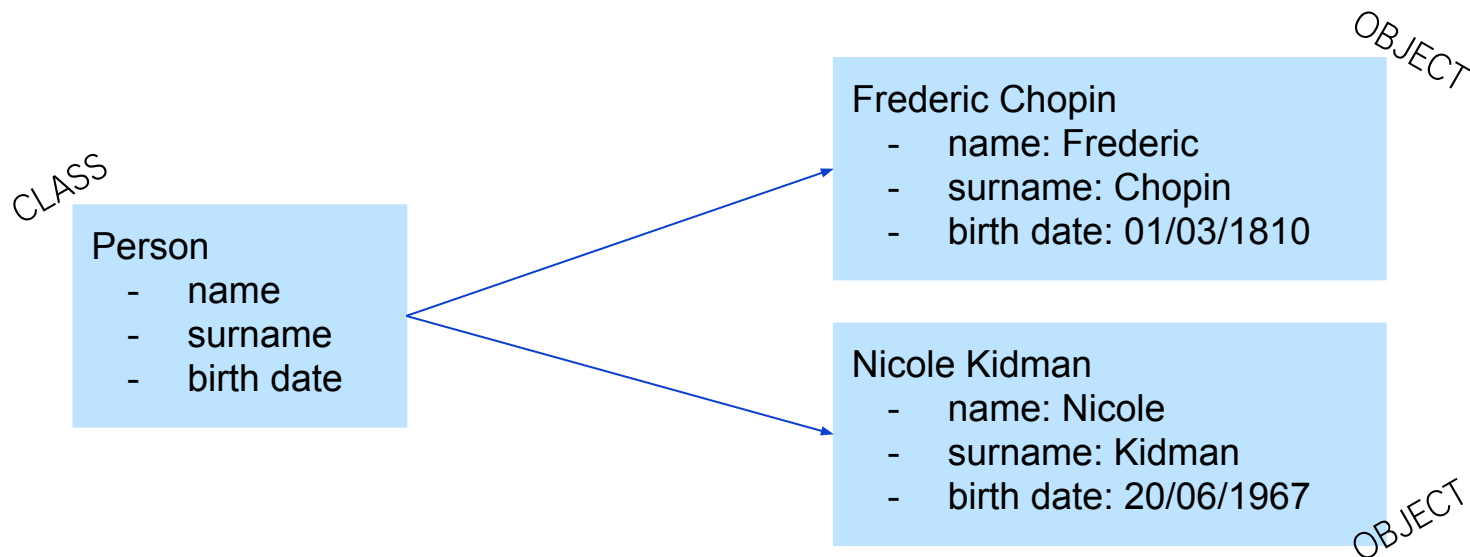
# Object Oriented Programming

## Quick recap

Uses the concept of *objects* to manage the complexity of the program.

Each *object* is an instance of a *class*.

Each *object* contain data (*fields*) and code (*methods*) attached to them.



# Object Oriented Programming

Quick recap - key assumptions:

- **abstraction**

Ability to define *abstract* actors that can perform a basic action therefore separating some (complicated and unnecessary) logic from the clients.

- **encapsulation**

Bundling the data (along with some methods) into the objects. Hiding some data and preventing direct access in a way that could expose implementation details.

- **polymorphism**

Single interface to objects of different types.

- **inheritance**

Basing one class upon another one. A *child-class* acquires (almost) all the properties of its *parent-class*.

# Object Oriented Programming in R

S3, S4 and R6 (and more)

- **S3** is the first OOP system adapted in R. It's being used throughout base and stats packages and it's commonly used in many CRAN packages. This is probably the easiest system to learn.
- **S4** is much more strict and rigorous approach compared to S3. It's very popular in Bioconductor packages. S4 supports multiple inheritance and multiple dispatch. It relies on the concept of slots which are accessible using the @ operator.
- **R6** requires the external {R6} package. R6 object instances are mutable which is different to R's copy-on-modify semantics. This is relevant for programming in {shiny} as it does not trigger reactivity chain.
- There are other OOP system implemented in R such as {R.oo}, {proto} and others.

See: <https://adv-r.hadley.nz/oo.html>

# OOP - exercise

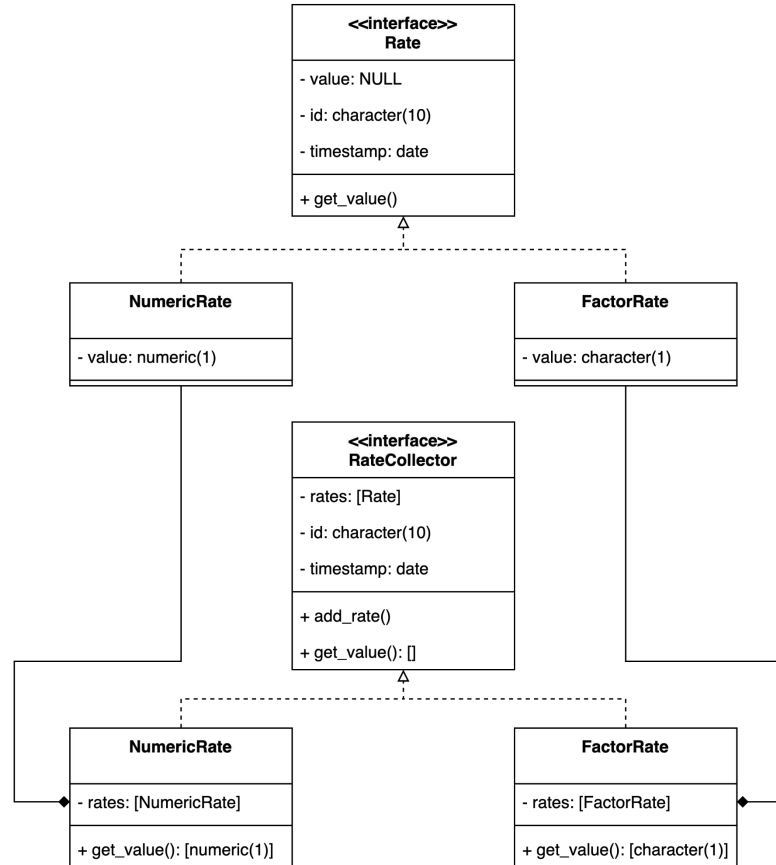
**Task:** design an application to submit numeric and factor type of rates / votes, store it and present basic aggregative summary.

Do it using S3 / S4 / R6 OOP systems.

[GH repo with \*\*solution\*\*](#)

# OOP - exercise

class diagram





# OOP - exercise

Instance creation

- S3  
`numeric_rate_s3(x)`
- S4  
`.NumericRate(x)`
- R6  
`numeric_rate_r6$new(x)`

# OOP - exercise

## Object class inheritance

- S3
 

```
r$> class(numeric_rate_s3(1))
[1] "NumericRate" "RateS3"      "Rate"
```
- S4
 

```
r$> is(.NumericRate(1))
[1] "NumericRate" "RateS4"      "Rate"
```
- R6
 

```
r$> class(numeric_rate_r6$new(1))
[1] "NumericRate" "RateR6"      "Rate"      "R6"
```

# OOP - exercise

Slot accessors - getters

- S3  
r\$> x\$value  
[1] 1
- S4  
r\$> x@value  
[1] 1
- R6  
r\$> x\$value  
[1] 1

# OOP - exercise

## Slot accessors - setters

- S3  
`r$> x$value <- 1`
- S4(\*)  
`r$> x@value <- 1`
- R6(\*\*)  
`r$> x$value <- 1`

(\*) - setter method might be overwritten

(\*\*) - note that slots might be set to private and then you won't be able to set it

# OOP - example

factory pattern

**Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

[\[source\]](#)

See: `rate_factory.R`, `rate_collector_factory.R`

# OOP - example

template method

**Template Method** is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

[\[source\]](#)

See: `validate()` method in `rate_r6.R`

# OOP - example

composite

**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

[\[source\]](#)

See: `get_value()` passed through `RateCollector` into the collection of `Rate` objects

# OOP - example

facade

**Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

[[source](#)]

See: simple interfaces via S3 dispatch to various classes in `rate_utils.R`, `rate_collector_utils.R`



# OOP

Few takeaways from our experience

- **S3** is the easiest to learn and it's very useful especially as a facade. You should master it as soon as possible. I highly recommend S4 + S3 and R6 + S3 combinations. I do not recommend S3-only approach.
- **S4** is the only one which supports multiple class inheritance as well as multiple arguments dispatch. Formal inheritance enables transparent extensions in user code or add-on packages.
- **R6** is the best one to maintain, e.g. decent {roxygen2} support for class documentation, all of the method definitions lives inside class definition.
- **R6** blocks {shiny} reactivity because of its mutate attribute. If modifications to instances should trigger reactivity chain - use other systems.
  - It is still possible to use R6 in {shiny} but data slots needs to be a `reactiveVal`. This tightly couples the class to {shiny}.
  - Be very careful with the scoping rules in {shiny}. It would be the best to create R6 instances within an user session. In extreme case, you might find one session modifying the data for the other one!

# OOP

Few takeaways from our experience

- OOP together with a good implementation of design patterns would help you build maintainable and scalable products.
- **Good design matters!** It's take some time and effort to achieve that.  
*Good design matters!* (myself)
- There is **no single and universal answer** to your question. There might be a few good solutions.
- Please consider and anticipate (if possible) **changing requirements** or development assumptions.  
*To expect the unexpected shows a thoroughly modern intellect* (Oscar Wilde)
- Do not go blindly for a given pattern.
  - Each of them has its pros and cons and it needs to be used with caution.
  - Please find the right balance and make appropriate assessment before implementing.
- **Do not overcomplicate!**  
*Simple is better than complex* (The Zen of Python)

# Mastering the Implementation: Summary

## Takeaways

- keep the UI simple, understand reactivity behaviour, try to keep the reactivity graphs simple and stringy
- write good, clean and maintainable codes
- apply design patterns if possible (and if it makes sense) - even if you don't use OOP

However, this is not sufficient - we need to make our Shiny product:

- **maintainable** ( $\Rightarrow$  tests)
- **understandable** ( $\Rightarrow$  documentation)
- **deployable** ( $\Rightarrow$  dependency management)

The next section provides our experience on these critical topics.

# Packaging the Product

# Packaging the Product

Tests, Documentation and Deployment are needed for sustainable apps

- **Tests** ensure that the app works as intended
  - When you change code (new fixes, features, etc.)
  - Without manually and randomly clicking around in the app
- **Documentation** ensures that you and other developers understand the code
  - Also in the future, not just right now
  - So that the code can be maintained sustainably
- **Deployment** delivers the app to the users
  - Requires diligent dependency management
  - Needs server infrastructure in place so that users can access app in the browser remotely

**Packaging the Product → Tests**

# Motivation

More fun for developers and users - less firefighting / bug fixing

## Validation step



# Overview

Layered approach supported by continuous integration infrastructure

## Unit Tests

... ensure that a section of an application (known as the “unit”) meets its design and behaves as intended (Wikipedia).

Here each (static) R function (or class) is tested separately.

## Separate Shiny UI, Server Tests

... ensure that no accidental changes to the UI occur,  
and (more importantly) that the server logic works as expected.

## Shiny App Tests

... ensure that UI and server work well together and result in the intended app behavior.

## Continuous Integration (CI)

... runs the tests automatically (e.g. on GitHub, Gitlab, etc.) when we create a pull request (i.e. code modification)

- Note: We follow a flow similar as in <https://mastering-shiny.org/scaling-testing.html>
- Practical tips & tricks with few live examples, mainly to show that it is easy to start writing shiny tests



# Why should I write unit tests?

Because it saves you a lot of hassle later!

- **Faster Debugging:** Only need to search narrow (unit) scope for the root cause.
- **Faster Development:** Have confidence that no side-bugs from new code.
- **Better Design:** Encourages aggressive refactoring into small maintainable units.
- **Better Documentation:** Developers can look at the unit tests to understand a function's usage and behavior.
- **Excellent Regression Tool:** Every time we fix a bug, we add a corresponding unit tests, so that it does not bite us again.
- **Reduce Future Cost:** Writing unit tests is an investment that pays off long-term because bugs can be orders more expensive than testing (time for users and developers)

# What are unit tests in R packages?

Small tests are easy with `testthat`

- Lots of examples and details: <https://r-pkgs.org/tests.html>
- Run `usethis::use_testthat()` in your package to get started
- Structure:

```
test_that("my_fun can do xyz as expected", {  
  input <- ... # prepare input  
  result <- my_fun(input, ...) # do xyz  
  expected <- ... # hardcode expectation  
  expect_identical(result, expected) # compare  
})
```

# When should I write unit tests?

Consider unit tests as being of same importance as the package code

- **Before coding:**
  - “Test-Driven Development”
  - Not realistic when you start the package, but can be an option for new small features later
- **During coding:**
  - When developing new functions, you anyway need to do some (interactive) testing.
  - Directly starting with the unit test for doing that saves development time.
- **For code review** (pull requests):
  - Allows the reviewer to check the behavior without running code themselves.
- **When you find a bug** (regression test):
  - Reproduce the bug with a new unit test.
  - Fix the problem.
  - Confirm that the corresponding unit test (and all others) passes now.

# How do static Shiny UI Tests work?

Snapshotting the HTML code

- **Shiny UI functions** return shiny tag objects, i.e. HTML code
- Can use **snapshot tests** to make sure this HTML does not accidentally change
  - See <https://testthat.r-lib.org/articles/snapshotting.html> for more details
  - Value is limited
  - Can create issues because sometimes non-reproducible hashes are part of the HTML

```
test_that("myInput UI module creates expected HTML", {  
  input <- "foo"  
  set.seed(123)  
  datasets <- mock_datasets()  
  expect_snapshot(myInput(  
    "my_test",  
    datasets = datasets,  
    input = input  
  ))  
})
```

# How do Shiny Server Tests work?

Testing the inner workings of the server function

```
test_that("server works", {  
  testServer(server, {  
    session$setInputs(...)  
  
    print(reactive1())  
    print(output$output1)  
    # etc  
  
    # To interactively play:  
    # browser()  
  
    expect_equal(...)  
    # etc.  
  })  
})
```

- Code provided is run inside the `server` function
- Therefore can access everything inside the `server` function (`reactives`, `outputs`)
- `session` object is used to simulate user actions
- Can then also perform comparisons vs. `expected` values
- Note: cannot step through the code interactively line by line, but need to insert `browser()`
- This also works for module server functions, see details: <https://mastering-shiny.org/scaling-testing.html#modules>

# How do Shiny Server Tests work?

Example

[github.com/danielinteractive/useR2022](https://github.com/danielinteractive/useR2022) → testing/ex1-testServer.R

# What are the Limitations of Shiny Server Tests?

It is not a full application

- **Time** does not advance automatically
  - Workaround: manually if needed with `session$elapse()`
- **No UI defaults** in the input values
  - Workaround: can set manually with `session$setInputs()`
- **No UI** is available
  - Therefore e.g. **cannot test any UI updates** coming from server
- **No JavaScript** will be run
  - E.g. showing notifications or modals (dialog boxes) e.g. does not work

# How do Shiny App Tests work?

Simulating a real user interaction with the app

```
test_that("my app works", {
  library(shinytest)
  app <- ShinyDriver$new(
    "myAppDir/",
    loadTimeout = 1e5,
    phantomTimeout = 1e5,
    debug = "all"
  ) # app$getDebugLog()
  app$takeScreenshot()
  app$setInputs(name = "Hadley")
  app$getValue("greeting")
  #> [1] "Hi Hadley"
  app$click("reset")
  app$getValue("greeting")
  #> [1] ""
})
```

- Note: {shinytest2} is recent alternative, better integrated with {testthat}, migration is possible.
- Usually the app is defined in **myAppDir/app.R**
- Note that the ShinyDriver is an R6 class and therefore methods are called with **\$fun()** syntax
- A new app instance is started with the constructor **ShinyDriver\$new**
- Full Shiny app is run in a headless browser including UI, updates, JavaScript, time, etc.
- There is no browser window, but we can look into the whole app/elements via **takeScreenshot()**
- We can **set inputs**, **get values** of inputs/outputs, **click** buttons etc.
- **Value is high** because we can really store in code what a user would do during testing in a real browser



# How do Shiny App Tests work?

Example

[github.com/danielinteractive/useR2022](https://github.com/danielinteractive/useR2022) → testing/ex2-ShinyDriver.R

# Tips for and Limitations of Shiny App Tests

Lesson learned the hard way: Be patient with the app :-)

- Important: **wait long enough** for everything.
  - Startup: increase timeout arguments for the constructor
  - Inputs and outputs:
    - use `app$waitForValue()`
    - ignore the right values (i.e. the old/initial value)
    - use `Sys.sleep(1)` when this strategy is not sufficient
- How to **quickly get started**:
  - use `recordTest()`
  - See available values (input/output) with `app$getAllValues()`
- **Use NS() function** to make code better readable
  - e.g. first get namespace for specific module and then use the `ns()` function to construct name
- **Limitations**:
  - Can only test the outside (like a user), so Server Tests can complement this
  - Relatively slow (see above)
- **Additional tools**: [helper functions for testthat](#), [extensions for ShinyDriver](#)

# How do Shiny App Tests work?

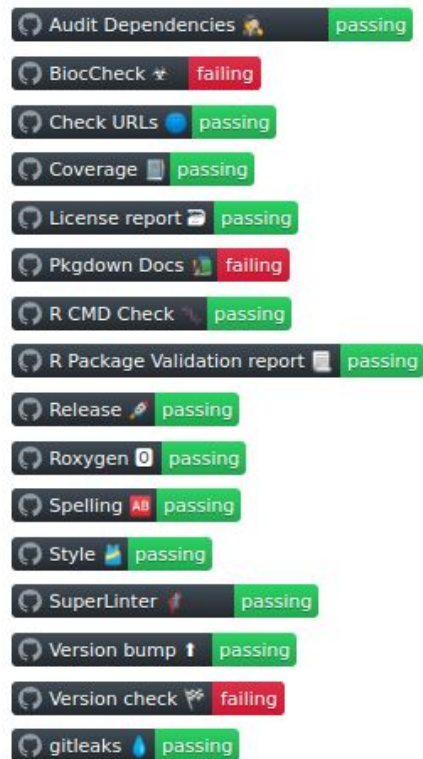
One more example

[github.com/danielinteractive/useR2022](https://github.com/danielinteractive/useR2022) → testing/ex3-test\_module.R

# How to set up the continuous integration?

Template is available

- **Template** is available at <https://github.com/insightsengineering/r.pkg.template>
  - specific for Github Action workflows
- **Tips:**
  - Minimize detailed formatting checks on the server side by **running pre-commit checks** (lintr, spelling, etc.)  
→ [{precommit}](#)
  - Private repositories on Github will require payment for CI minutes, whereas open source repositories don't  
→ another reason for **open source!**
- **[{staged.dependencies}](#)** is very helpful when working with **orchestrated multiple packages**
  - e.g. one package contains static business logic
  - the other package contains the Shiny code
  - **allows updating both in parallel**



# Details on staged.dependencies

Emulating a mono repository across multiple git repositories

When the functionals are splitted into multiple packages you might find adding new changes more difficult!

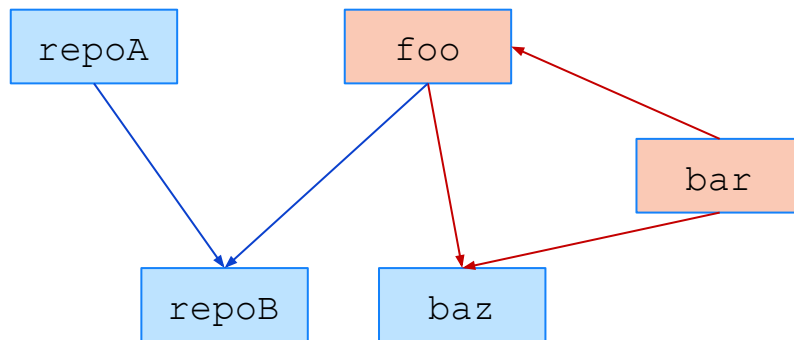
## Use case:

- want to add a new feature (say xyz) in the package {foo}
- but this requires upstream dependency update in the package {bar}
- and we must not break tests in downstream package {baz}

**Solution: staged.dependencies + strict branch naming convention**

⇒ ***monolithic repository behaviour***

```
repoA:  main
foo:    feature/xyz
bar:    feature/xyz
baz:    main
repoB:  main
```



**Packaging the Product → Documentation**

# Documentation - Best practices

Almost as important as tests for the maintenance of the Shiny app package

- **README:** quick entry point on the Github page
  - [example](#)
  - `usethis::use_readme_rmd()`
- **NEWS:** Inform app developers (not frontend users though) of important changes
  - [example](#)
  - `usethis::use_news_md()`
- **Vignette:** Show how the modules and app can be configured
  - [example](#) (non-Shiny though)
  - `usethis::use_vignette()`
- **Roxygen** documentation of functions, classes and data in the package:
  - Most important for future developers; Try to document everything, including internals
  - Use consistent argument description, consider including expected class → [example](#)
  - Can be instructive and efficient to use central documentation pages with common arguments (and then use with `@inheritParams`) → [example](#)
  - `usethis::use_roxygen_md()` to start using Markdown in Roxygen

# Documentation - Best practices (cont.)

Almost as important as tests for the maintenance of the Shiny app package

- **Lifecycle** management: Important to be able to phase out functions as deprecated later
  - `usethis::use_lifecycle()`
- **pkgdown** documentation: Can be a nice user and developer resource in the browser, in particular the package API reference page
  - `usethis::use_pkgdown_github_pages()`
- **Developer** guide: How could others contribute to the package?
  - example [Markdown file](#)
  - `usethis::use_tidy_contributing`
- Before releasing the package it also is useful to take the user perspective and **really read through** the documentation pages. → If you polish this, your users will be grateful!
- *Not covered here:* (Interactive) **documentation for the non-R only-app users**
  - We believe recorded video explanations are more helpful than popup windows in the app
  - Also validation checks that point the app user to inputs they need to change and avoid red errors from underlying static package are important
  - Consider adding feedback / question buttons to your app to understand their problems



**Packaging the Product → Deployment**

# Dependency management

Less is more here

- **R package versioning:**

- The x.y.z versions of packages correspond to
  - major version x - breaking features
  - minor version y - major improvements
  - patch version z - bug fixes and minor improvements
- But note that many packages are not following this semantic strictly.

- **Minimum versions:**

- If you depend on a package and you only have a too old version then likely your app will fail
- Therefore it is important in the DESCRIPTION of your Shiny app package to mention which minimum version of the packages you need, e.g.
  - `dplyr (>= 1.0)`

- **Less is more:**

- Try to reduce dependencies. Rather than using `fancy::short_function()` try to do it yourself.
- See <https://www.tinyverse.org/> for more ideas.
  - “Every dependency you add to your project is an invitation to break your Shiny App project.”<sup>106</sup>

# Deployment

External vs. Internal

- <https://www.shinyapps.io/>
  - Allows for **quick sharing** of Shiny apps on the cloud
  - **Free version** is good for quick tests or experiments
  - Serious use needs **subscription** for the cloud service
- **Internal RStudio workbench** server is internal alternative
  - More features than above cloud service
    - One detail example: can use environment variables
  - See <https://www.rstudio.com/products/shiny/shiny-server/> for the differences.

# Packaging the Product: Summary

- **Tests** ensure that the app works as intended.
  - Use **unit tests** for static functions - as much as possible
  - Use **server tests** to make sure server functions work
  - Use **shiny tests** to make sure UI and server work together correctly
- **Documentation** ensures that you and other developers understand the code.
  - Many different facets to this - from NEWS file to vignettes
  - Make sure to **tick all the boxes** here
- **Dependency management** should be ...
  - ... **minimizing dependencies** as much as possible.
  - ... using **staged.dependencies** for integration tests.

# Summary

# Summary of this Tutorial

Lessons for Designing Scalable and Maintainable Shiny Apps

## Designing the Solution

- **Know Your Customer** and align with business context to succeed.
- **Layer-based architecture** where the base needs to be simple and solid.
- **Scalability** by (horizontal) extendability and (vertical) customization of your app.

## Mastering the Implementation

- **Keep it Simple:** Simple UI interaction logic.
- **Plan your reactivity graph** and keep it "stringy" if possible.
- Make good and clean codes that would be easy to maintain.

## Packaging the Product

- **Tests** are as important as the features themselves. They make the real difference for maintenance.
- **Documentation** is essential such that you will have developers that can maintain the package code.
- **Deployment** requires diligent dependency management.

**Doing now what patients need next**