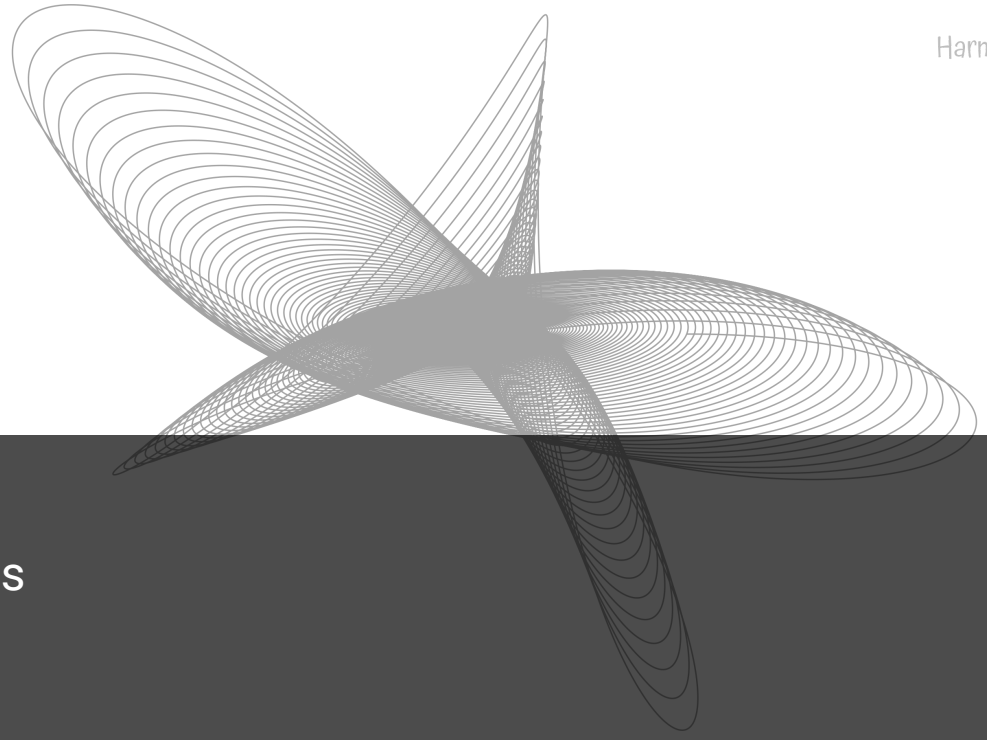
The Art of Design

Software & Experimental Design



Presenter: *Emi Tanaka*

Department of Econometrics and Business Statistics

MONASH University

emi.tanaka@monash.edu @statsgen

Tue 26th Oct 2021 | R-Ladies Melbourne

Now Showing

Act I Software design

Drawing faces under different programming paradigms

Act II Experimental design

Comparative experiments with `edibble`

These slides made using R powered by HTML/CSS/JS can be found at
emitanaka.org/slides-RLadiesMelb2021
All code to reproduce this slide can be found at
github.com/emitanaka/slides-RLadiesMelb2021

Fractal ferns

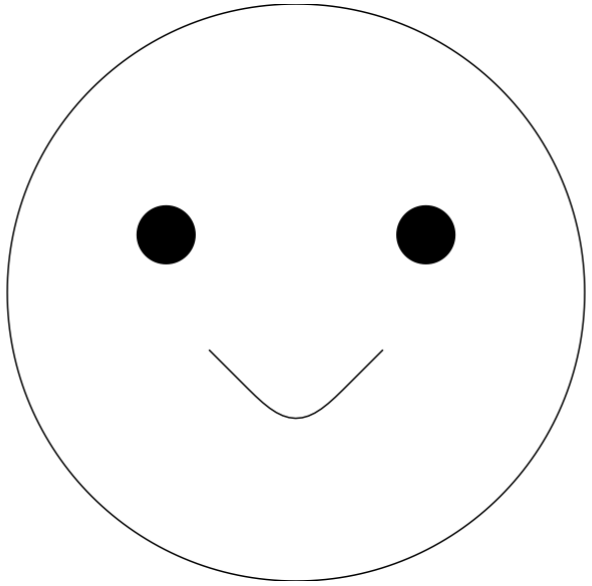
Act I

Software Design

Drawing faces under different programming paradigms

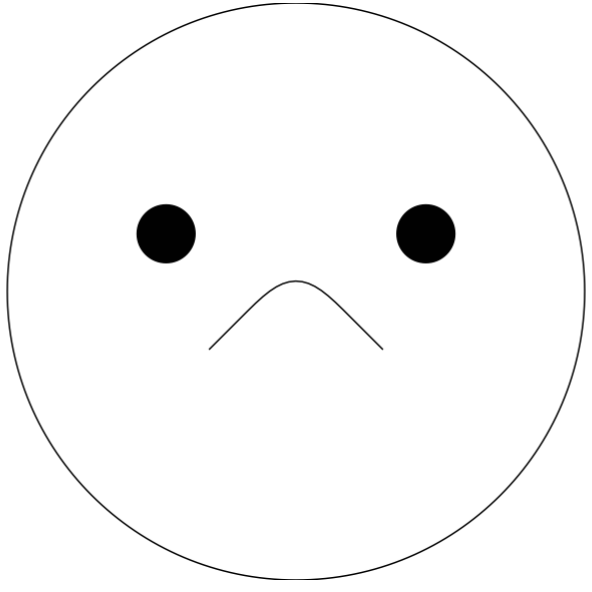
Drawing a happy face 1

```
library(grid)
# face shape
grid.circle(x = 0.5, y = 0.5, r = 0.5)

# eyes
grid.circle(x = c(0.35, 0.65),
            y = c(0.6, 0.6),
            r = 0.05,
            gp = gpar(fill = "black"))

# mouth
grid.curve(x1 = 0.4, y1 = 0.4,
           x2 = 0.6, y2 = 0.4,
           square = FALSE)
```



Drawing a sad face 1

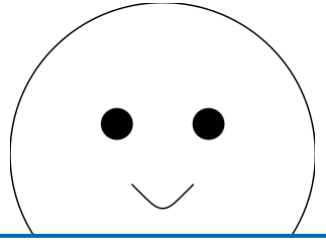
```
library(grid)
# face shape
grid.circle(x = 0.5, y = 0.5, r = 0.5)

# eyes
grid.circle(x = c(0.35, 0.65),
            y = c(0.6, 0.6),
            r = 0.05,
            gp = gpar(fill = "black"))

# mouth
grid.curve(x1 = 0.4, y1 = 0.4,
           x2 = 0.6, y2 = 0.4,
           square = FALSE,
           curvature = -1)
```

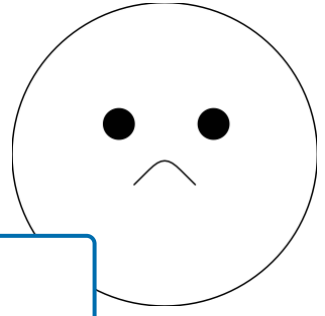


Drawing faces ❶ *Imperative programming* style

```
library(grid)
grid.circle(x = 0.5, y = 0.5, r = 0.5)
grid.circle(x = c(0.35, 0.65),
            y = c(0.6, 0.6),
            r = 0.05,
            gp = gpar(fill = "black"))
grid.curve(x1 = 0.4, y1 = 0.4,
           x2 = 0.6, y2 = 0.4,
           square = FALSE)
```

```
grid.circle(x = 0.5, y = 0.5, r = 0.5)
grid.circle(x = c(0.35, 0.66),
            y = c(0.6, 0.6),
            r = 0.05,
            gp = gpar(fill = "black"))
grid.curve(x1 = 0.4, y1 = 0.4,
           x2 = 0.6, y2 = 0.4,
           square = FALSE,
           curvature = -1)
```





Imperative programming is a programming paradigm that uses statements to change a program's state. —Wikipedia

Think of this as ***"instructions meant for the computer"***

Drawing faces ❷ *Functional programming* style

```
face1 <- function() {
  grid::grid.circle(x = 0.5, y = 0.5, r = 0.5)
  grid::grid.circle(x = c(0.35, 0.65),
                    y = c(0.6, 0.6),
                    r = 0.05,
                    gp = gpar(fill = "black"))
  grid::grid.curve(x1 = 0.4, y1 = 0.4,
                   x2 = 0.6, y2 = 0.4,
                   square = FALSE)
}

face2 <- function() {
  grid::grid.circle(x = 0.5, y = 0.5, r = 0.5)
  grid::grid.circle(x = c(0.35, 0.65),
                    y = c(0.6, 0.6),
                    r = 0.05,
                    gp = gpar(fill = "black"))
  grid::grid.curve(x1 = 0.4, y1 = 0.4,
                   x2 = 0.6, y2 = 0.4,
                   square = FALSE,
                   curvature = -1)
}
```

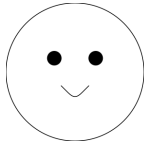
Functional programming is a declarative programming paradigm where programs are constructed by applying and composing functions. —Wikipedia

`face1()`

`face2()`

`face1()`

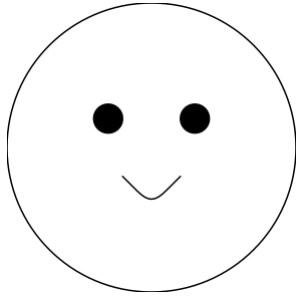
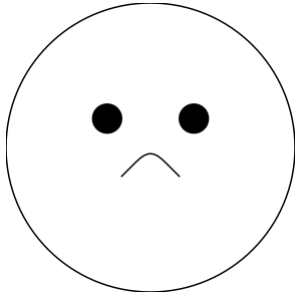
`face1()`



Drawing faces 3 Human-centered design

Computational systems are adapted for human use

> **Syntactic sugar** means using function name or syntax in a programming language that is designed to make things ***easier to read or to express for humans***.

`face1()`	`face2()`	`face3()`
		?

Alternative function names:

`face_happy()`	`face_sad()`	`face_angry()`

What if you want to draw a surprised face? 😮

Or a face with eyebrows? 😲

Or with googly eyes? 😳

You can't without writing a whole new function that describes the entire face from scratch.

https://github.com/emitanaka/portrait

```
remotes::install_github("emitanaka/portrait")
```

library(portrait)

We'll use the above package to draw faces.

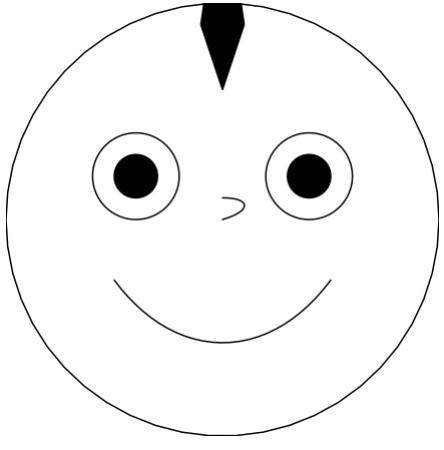
Note: the above package was made for only demonstration purpose for this talk.

Drawing faces 4 Rethinking function arguments as facial parts

- Let's ***reframe*** how we think
- A face is made up of:
  - eyes
  - mouth
  - shape
  - **hair** 👈
  - **nose** 👈

- Adding more arguments:

```
library(portrait)
face(eyes = "googly",
     mouth = "smile",
     shape = "round",
     hair = "mohawk",
     nose = "simple")
```



But what about if I want to add a beard 👱,
an accessory 🤠,
eye brows 🤨, ...?

You are ***reliant on the developer(s)*** to add the argument and/or functionality for you.

Drawing faces 5 *Object-oriented programming* style

Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

Drawing faces 5 *Object-oriented programming* style

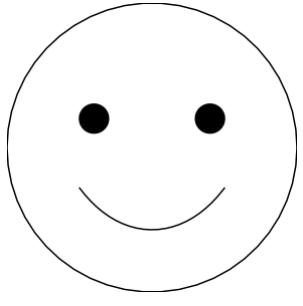
Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

```
library(portrait)
f <- face()

str(f)

## Portrait contains:
## - shape
## - eyes
## - mouth
```

OUTPUT

```
print(f)
```



MAIN POINTS

- Function creates an object that contains a "standard" smiley face
- Only draws the face upon print

Drawing faces 5 *Object-oriented programming* style

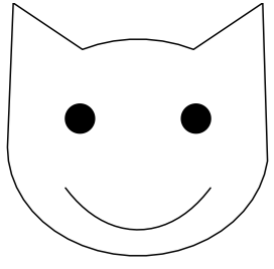
Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

```
library(portrait)
f <- face() +
  cat_shape()

str(f)

## Portrait contains:
## - shape
## - eyes
## - mouth
```

OUTPUT

```
print(f)
```



MAIN POINTS

- `cat_shape` function modifies the shape information within the object

Drawing faces 5 *Object-oriented programming* style

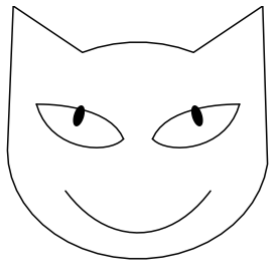
Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

```
library(portrait)
f <- face() +
  cat_shape() +
  cat_eyes()

str(f)

## Portrait contains:
## - shape
## - eyes
## - mouth
```

OUTPUT

```
print(f)
```



MAIN POINTS

- `cat_eyes` function modifies the eye information within the object

Drawing faces 5 *Object-oriented programming* style

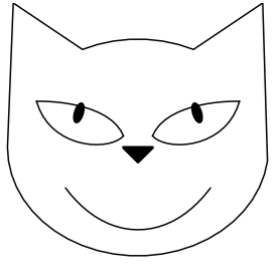
Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

```
library(portrait)
f <- face() +
  cat_shape() +
  cat_eyes() +
  cat_nose()

str(f)

## Portrait contains:
## - shape
## - eyes
## - mouth
## - nose
```

OUTPUT

```
print(f)
```



MAIN POINTS

- `cat_nose` function adds nose information

Drawing faces 5 *Object-oriented programming* style

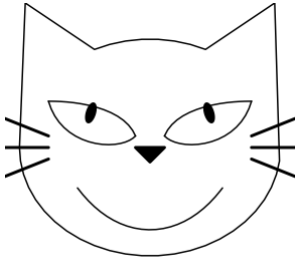
Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

```
library(portrait)
f <- face() +
  cat_shape() +
  cat_eyes() +
  cat_nose() +
  cat_whiskers()

str(f)

## Portrait contains:
## - shape
## - eyes
## - mouth
## - nose
## - whiskers
```

OUTPUT

```
print(f)
```



MAIN POINTS

- `cat_whiskers` function adds whiskers information

Drawing faces 5 *Object-oriented programming* style

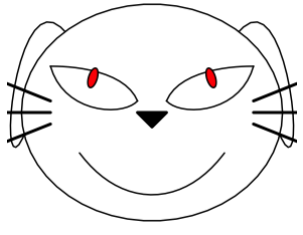
Object-oriented programming (OOP) is a programming paradigm based on objects that have certain attributes and behaviours

- Rethink everything as an ***object***

```
library(portrait)
f <- face() +
  dog_shape() +
  cat_eyes(fill = "red") +
  cat_nose() +
  cat_whiskers()

str(f)

## Portrait contains:
## - shape
## - eyes
## - mouth
## - nose
## - whiskers
```

OUTPUT

```
print(f)
```



MAIN POINTS

- Mix-and-match functions to make other faces
- The functions are modular so can be replaced by user's own functions

Recipe functions →

One function to draw one complete face

A function with multiple arguments →

One function to draw multiple complete faces

Finite number of functions to draw

***infinite* possible *incomplete* and complete faces**

Act II

Experimental Design

Classical "named" experimental designs

A Completely Randomised Design

B Randomised Complete Block Design

C Latin Square Design

D Balanced Incomplete Block Design

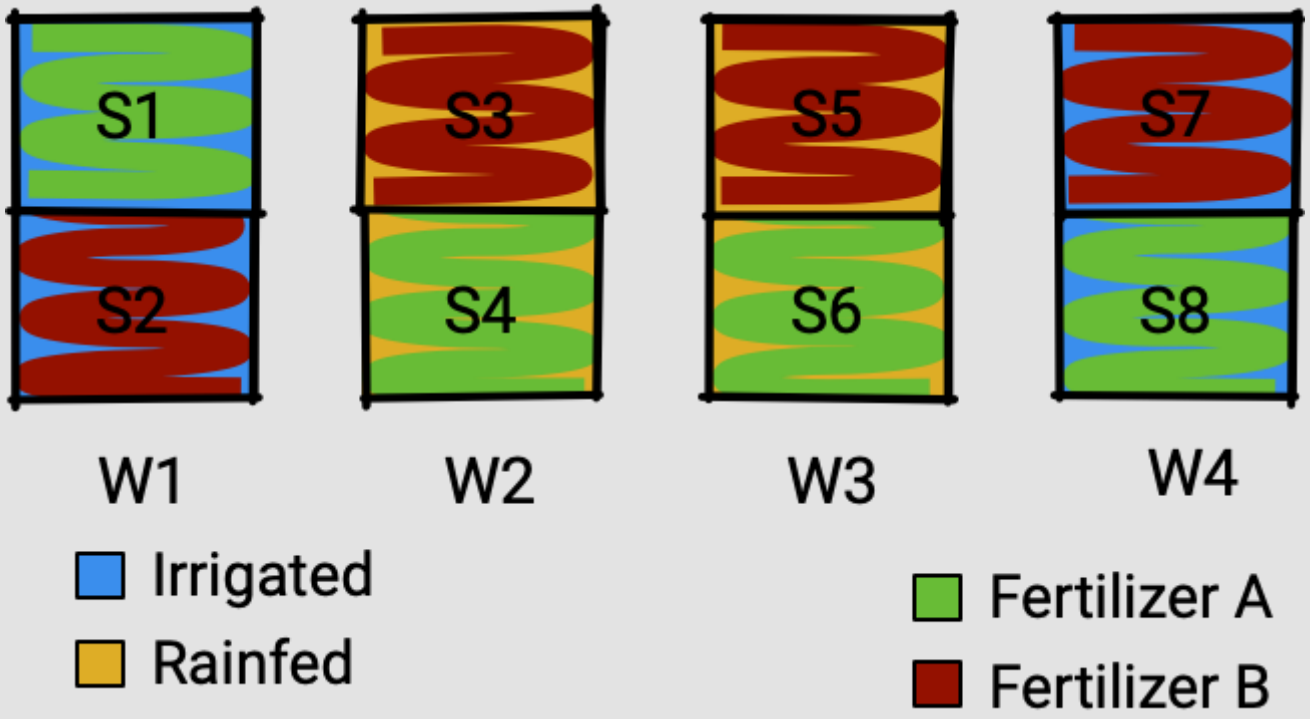
E Factorial Design

F Split-plot Design

Classical split-plot design

CONTEXT

Study of **two irrigation methods** and **two fertilizer brands** on the yields of a crop.



CRAN Task View of Design of Experiments

& Analysis of Experimental Data

contains

109 R-packages

based on the `ctv` package version 0.8.5

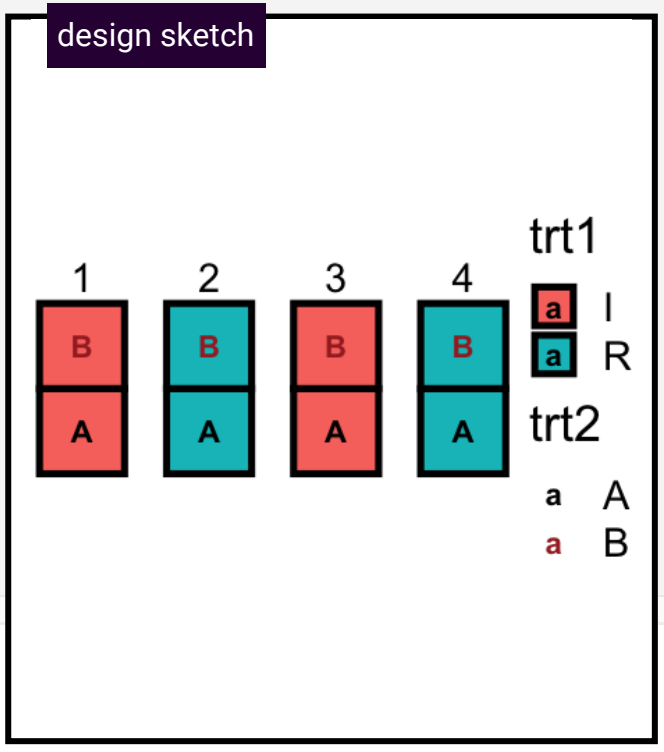
of which `agricolae` package is one of the most popular

agricolae::design.split()

Split-plot design for $t = 2 \times 4$ treatments with 2 replication for each treatment

```
trt1 <- c("I", "R")
trt2 <- c("A", "B")
agricolae::design.split(trt1 = trt1, trt2 = trt2, r = 2, design = "crd")
```

```
## $parameters
## $parameters$design
## [1] "split"
##
## $parameters[[2]]
## [1] TRUE
##
## $parameters$trt1
## [1] "I" "R"
##
## $parameters$applied
## [1] "crd"
##
## $parameters$r
```



Common **software design** for computational systems for constructing experimental design:

- Functions construct the experimental design from ***zero to its entirety***
  - In another words, you can't have an intermediate construct of an experimental design
- Experimental units, observational units, blocks and allocation of treatment to units are often ***implicitly*** **defined**
- The intention of what responses to record are ***not explicitly*** **specified**

Let's *reframe* how we think

but first some terminology so we're on the same page

A basic comparative experiment

- There are three components that are *necessary* to run a comparative experiment:
  - a set of n **experimental units** (Ω)
  - a set of t **treatments** $(\mathcal{T})$
  - **allocation** of treatments to experimental units $(D : \Omega \to \mathcal{T})$ or design matrix $(\mathbf{X}_{n \times t})$
- And decide on **observational units** (Ω_o) which may or may not be the same as Ω
- For the analysis of experimental, you additionally require:
  - the **record of responses** $(\boldsymbol{Y})$ on observational units

Terminology modified versions of Bailey (2008)

Experimental unit (Ω) is the smallest unit that the treatment can be independently applied to.

Observational unit (Ω_o) is the smallest unit in which the response will be measured on. Not to be confused with observations $\boldsymbol{Y}$.

Block, also called **cluster**, is the unit that group some other units (e.g. experimental units) such that the units within the same block (cluster) are more alike (homogeneous).

A **treatment** $(\mathcal{T})$ is the entire description of what can be applied to an experimental unit.

A **design** $(D : \Omega \to \mathcal{T})$ is the allocation of treatments to units.

A **plan** or **layout** is the design translated into actual units; randomisation is usually involved in the process.

Bailey, R. (2008). Design of Comparative Experiments (Cambridge Series in Statistical and Probabilistic Mathematics). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511611483

Experimental Structures as defined by Bailey (2008)

Unit structure means meaningful ways of dividing up Ω and Ω_o.

For example:

- **Unstructured**
- **Blocking**

Treatment Structure means meaningful ways of dividing up $\mathcal{T}$.

For example:

- **Unstructured**: no grouping within $\mathcal{T}$
- **Factorial**: all combinations of at least two factors
- **Factorial + control**

Bailey, R. (2008). Design of Comparative Experiments (Cambridge Series in Statistical and Probabilistic Mathematics). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511611483

The Grammar of Experimental Designs

Lissajous curves

Context is key in experimental design

Domain expert

Statistician

Technician

A successful experiment is a result of collaboration between these "actors".

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
start_design("Wood water resistance")
```

OUTPUT

```
## Wood water resistance
```

Domain expert

"I want to run an experiment to study the water resistance property of wood"

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4))
```

OUTPUT

```
## Wood water resistance
## └─board (10 levels)
##   └─panel (40 levels)
```

Domain expert

"I have 10 boards that I can divide each into 4 small wood panels"

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4)
```

OUTPUT

```
## Wood water resistance
## ├─board (10 levels)
## │ └─panel (40 levels)
## ├─pretreatment (2 levels)
## └─stain (4 levels)
```

Domain expert

"I want to test linseed oil and copper azole preservative as wood pretreatments and four types of stain."

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4) %>%
  allocate_trts(pretreatment:stain ~ panel)
```

OUTPUT

```
## Wood water resistance
## ├─board (10 levels)
## │ └─panel (40 levels)
## ├─pretreatment (2 levels)
## └─stain (4 levels)
```

Domain expert

"The combination of the the treatment factors should be applied on each wood panel."

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4) %>%
  allocate_trts(pretreatment ~ board,
                stain ~ panel)
```

OUTPUT

```
## Wood water resistance
## ├─board (10 levels)
## │ └─panel (40 levels)
## ├─pretreatment (2 levels)
## └─stain (4 levels)
```

Domain expert

"Oh wait, it's hard to apply pretreatment to small wood panels. We can only apply it to the board. The stain can be applied independently to wood panels."

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
set.seed(2021)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4) %>%
  allocate_trts(pretreatment ~ board,
                stain ~ panel) %>%
  randomise_trts() %>%
  serve_table()
```

OUTPUT

```
## # An edibble: 40 x 4
##          board      panel pretreatment    stain
##    <unit(10)> <unit(40)>     <trt(2)> <trt(4)>
##  1     board1    panel1       copper    stain4
##  2     board1    panel2       copper    stain3
##  3     board1    panel3       copper    stain1
##  4     board1    panel4       copper    stain2
##  5     board2    panel5       linseed   stain2
##  6     board2    panel6       linseed   stain3
##  7     board2    panel7       linseed   stain1
##  8     board2    panel8       linseed   stain4
##  9     board3    panel9       copper    stain1
## 10     board3    panel10      copper    stain3
## # … with 30 more rows
```

Domain expert

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
set.seed(2021)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4) %>%
  allocate_trts(pretreatment ~ board,
                stain ~ panel) %>%
  randomise_trts() %>%
  serve_table()
```

OUTPUT

```
## # An edibble: 40 x 4
##          board      panel pretreatment    stain
##    <unit(10)> <unit(40)>     <trt(2)> <trt(4)>
##  1     board1    panel1       copper    stain4
##  2     board1    panel2       copper    stain3
##  3     board1    panel3       copper    stain1
##  4     board1    panel4       copper    stain2
##  5     board2    panel5       linseed   stain2
##  6     board2    panel6       linseed   stain3
##  7     board2    panel7       linseed   stain1
##  8     board2    panel8       linseed   stain4
##  9     board3    panel9       copper    stain1
## 10     board3    panel10      copper    stain3
## # … with 30 more rows
```

Technician

"What am I supposed to measure?"

Constructing experimental designs with `edibble`

Statistician

```
library(edibble)
set.seed(2021)
start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4) %>%
  allocate_trts(pretreatment ~ board,
                        stain ~ panel) %>%
  randomise_trts() %>%
  set_rcrds(panel = swelling,
            board = supplier) %>%
  serve_table()
```

OUTPUT

```
## # An edibble: 40 x 6
##         board      panel pretreatment    stain swelling
##    <unit(10)> <unit(40)>     <trt(2)> <trt(4)>   <rcrd>
##  1     board1    panel1       copper    stain4        ▮
##  2     board1    panel2       copper    stain3        ▮
##  3     board1    panel3       copper    stain1        ▮
##  4     board1    panel4       copper    stain2        ▮
##  5     board2    panel5      linseed    stain2        ▮
##  6     board2    panel6      linseed    stain3        ▮
##  7     board2    panel7      linseed    stain1        ▮
##  8     board2    panel8      linseed    stain4        ▮
##  9     board3    panel9       copper    stain1        ▮
## 10     board3    panel10      copper    stain3        ▮
## # … with 30 more rows
```

Domain expert

"Oh follow Kubojima & Yoshida (2015) and measure diameter swelling. Remember to record wood supplier too."

Constructing experimental designs with edibble

Statistician

```
library(edibble)
set.seed(2021)
des <- start_design("Wood water resistance") %>%
  set_units(board = 10,
            panel = nested_in(board, 4)) %>%
  set_trts(pretreatment = c("copper", "linseed"),
           stain = 4) %>%
  allocate_trts(pretreatment ~ board,
                        stain ~ panel) %>%
  randomise_trts() %>%
  set_rcrds(panel = swelling,
            board = supplier) %>%
  expect_rcrds(swelling = to_be_numeric(with_value(be
  serve_table()
```

OUTPUT

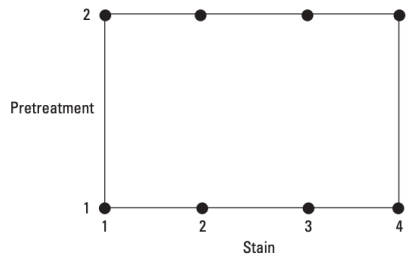
```
export_design(des, "design.xlsx", overwrite = TRUE)

## Loading required package: openxlsx

## √ Wood water resistance has been written to 'design.xls
```

Technician

"Okay, I'll record the swelling. The values should be between 0 and 5 for that."

Wood water resistance experiment

FIGURE 3 Factors That Affect Wood's Water Resistance

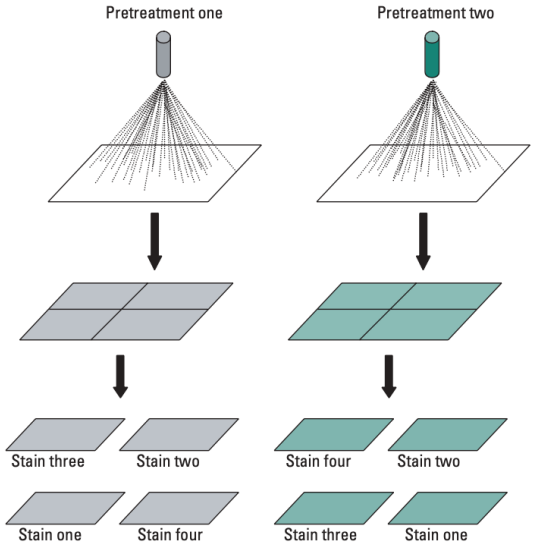


Now let's look at an experiment involving the water resistance property of wood in which you select two types of wood pretreatment (1 and 2) and four types of stain (1, 2, 3 and 4) as variables (see Figure 3).

To conduct this experiment in a randomized fashion, you would need eight wood panels for each full replicate of the design. You would then randomly assign a particular pretreatment and stain combination to each wood panel.

That's when you discover how difficult it is to apply the pretreatment to a small wood panel. The easiest way to do it would be to apply each of the pretreatment types (1 and 2) to an entire board, then cut each board into four pieces and apply the four stain types to the smaller pieces (see Figure 4).

FIGURE 4 Treatment Application



- This experiment is based on the description by Kowalski and Potcner (2003)
- The resulting design is a split-plot design... but we didn't need to know that to construct the experiment!

MAIN POINTS

- The *grammar of experimental design* is a (programming language agnostic) framework that functionally maps the fundamental components of the experiment to an object oriented programming system to build and modify an experimental design, i.e. *reframes construction of experimental design using software*
- The *edibble* R-package is an implementation of the grammar of experimental design in the R language
  </> https://github.com/emitanaka/edibble
- The approach is designed to be *human-friendly* and capture natural order of thinking for specifying experimental structure and encourage steps to be *explicitly specified*
- The approach (IMO) also *promotes higher order thinking about experimental design*
- Finally, the grammar makes each step modular... you can *easily extend* or *mix-and-match methods*

These slides made using R powered by HTML/CSS/JS can be found at emitanaka.org/slides-RLadiesMelb2021
All code to reproduce this slide can be found at github.com/emitanaka/slides-RLadiesMelb2021

Rose curves

Statistical Society of Australia Victoria Branch presents the

Di Cook Award

- Calling for submission for your statistical software product!
- Open only to students (or recent graduates) of Victorian or Tasmanian institutes
- Submissions close at **Fri 26th Nov 2021**
- Winner, announced in March 2022, will win $1,000
- Find more information at

https://statsocaus.github.io/dicook-award/