Constructing and visualising experimental designs with the `edibble` R-package

Presenter: *Emi Tanaka*

Department of Econometrics and Business Statistics,
Monash University, Melbourne, Australia

emi.tanaka@monash.edu

@statsgen

20 May 2021 @ Qld DAF Biometry workshop

today's menu

starters

Overview of comparative experiments 8.30

fundamental experimental components

mains

Constructing experimental designs with `edibble` 8.40

`start_design`, `set_context`, `set_units`, `set_trts`, `set_rcrds`, `expect_rcrds`, `nested_in`, `allocate_trts`, `randomise_trts`, `serve_table`, `export_design`

dessert

Visualising experimental designs with `deggust` 9.50

`autoplot`

The R-packages: `edibble`, `deggust` and `edibbleGUI` are all in active development so many more additions (and possibly breaking changes) are envisioned!

All code are available are online:

- https://github.com/emitanaka/edibble
- https://github.com/emitanaka/deggust
- https://github.com/emitanaka/edibbleGUI

Not an R user? It's never too late to learn! You can join the community of R learners, e.g. R for data science.

Overview of

comparative experiments

A minimal comparative experiment

There are **three components** that are *necessary* to run a *comparative* experiment:

- a set of experimental units (Ω),
- a set of treatments $(\mathcal{T})$, and
- allocation of treatments to experimental units $(D : \Omega \rightarrow \mathcal{T})$

and for the analysis of experiment, you additionally require the:

- response measure on observational units (Y).

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

Observational unit is the unit in which the response is measured on.

Blocking units

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

- Essentially **blocks are a unit factor** which nest another unit factor within it.
- An experimental unit, observational unit and blocking unit are all just simply referred to as "unit" in edibble.

`edibble` implementing the "grammar of experimental design"

The **grammar of experimental design** is a framework that functionally maps the fundamental components of an experiment to an object oriented system to build and modify the experimental design.

- In `edibble`, the design is built step-by-step with each step modularised to an individual function.
- The three main components of `edibble` are:
  1. units,
  2. treatments, and
  3. allocation of treatments to units
- An optional component is records that capture any measurement taken on units (e.g. responses).

Constructing experimental designs with edibble

⚠ This package is still experimental!

```r
remotes::install_github("emitanaka/edibble")

library(edibble)
```

! Rapid development phase — use with caution !

edibble::start_design()

- Begin with `start_design()`

```r
library(edibble)
start_design()
```

```
An edibble design
```

- This doesn't do much except create a new **edibble design** object.
- An edibble design contains an **edibble graph** (and later **edibble table**)
- You can give it a name to your design — think of it as the title of your experiment

```r
start_design("My diet experiment")
```

```
My diet experiment
```

edibble::set_context()

- Set small notes that remind you the context of the experiment

```r
start_design("My diet experiment") %>%
  set_context(aim = "Understand relation between diet and weight gain",
              experimenter = "Taylor Alwyn {.email tswift@fakemail.com}",
              "More details in {.file details.txt}")
```

```
── Context of the experiment ──────────────────────────────────────

• aim: Understand relation between diet and weight gain
• experimenter: Taylor Alwyn 'tswift@fakemail.com'
• More details in 'details.txt'

──────────────────────────────────────────────

My diet experiment
```

Persistent reminder of experimental context so information is not lost in your email or elsewhere!

- Context data is preserved in the object, displayed when object is printed and can be exported

`edibble::set_units()` Part 1

- A "**unit**" in edibble is any entity, physical or otherwise, that pertain to the experiment.
- A *single integer is a shorthand for the number of levels*.

```r
start_design("My diet experiment") %>%
  set_units(subject = 20)
```

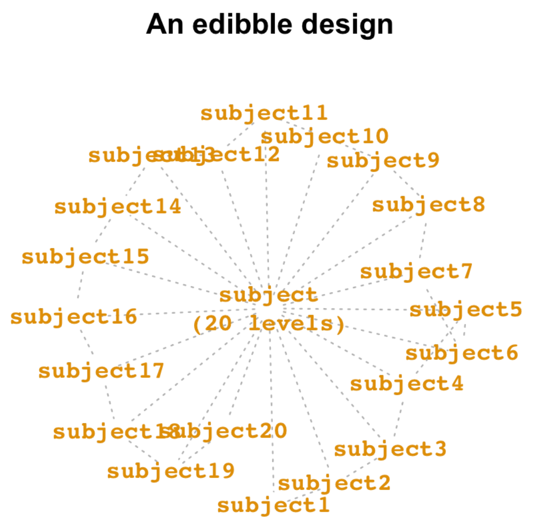
```
My diet experiment
└─subject (20 levels)
```

- Above specify there are **20 subjects** for the experiment.
  - Under the hood, there is an **edibble graph** that contains:
    - a node corresponding to the variable **subject** and
    - 20 other nodes corresponding to the levels associated with *subject*.

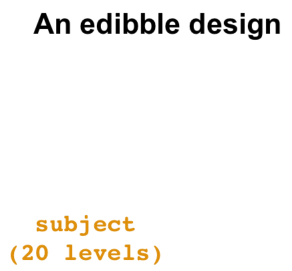
An edibble graph

The full graph:

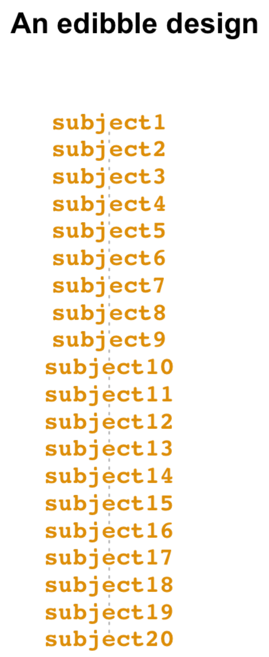
```r
set.seed(1)
library(edibble)
g <- start_design() %>%
  set_units(subject = 20)
plot(g, view = "all")
```



```r
# default
plot(g, view = "high")
```



```r
plot(g, view = "low",
     layout = igraph::layout
     asp = 10)
```



An edibble table

```
start_design("My diet experiment") %>%
  set_units(subject = 20) %>%
  serve_table()
```

edibble::set_units() Part 2

- A *string vector* is a short hand for the names of the levels.

```r
start_design("My diet experiment") %>%
  set_units(subject = c("Bettie", "Javid", "Yohan", "Marco", "Joani",
                        "Tynika", "Lakendrick", "Stephanos", "Lavonda", "Benny",
                        "Daniell", "Juanito", "Kele", "Delance", "Shekelia",
                        "Meghan", "Lynzie", "Viraaj", "Jeffrey", "Sunni"))
```

```
My diet experiment
└─subject (20 levels)
```

Seeing the level names in edibble table

```
start_design("My diet experiment") %>%
  set_units(subject = c("Bettie", "Javid", "Yohan", "Marco", "Joani",
                        "Tynika", "Lakendrick", "Stephanos", "Lavonda", "Benny",
                        "Daniell", "Juanito", "Kele", "Delance", "Shekelia",
                        "Meghan", "Lynzie", "Viraaj", "Jeffrey", "Sunni")) %>%
  serve_table()
```

edibble::set_units() Part 3

- *The argument name can be anything!*
- So user may use names that match the experimental context.

```r
start_design("My diet experiment") %>%
  set_units(fly = 20)
```

```
My diet experiment
└─fly (20 levels)
```

- Above reads that there are **20 flies** in this diet experiment.

```r
start_design("My diet experiment") %>%
  set_units(pig = 20)
```

- Now it reads that there are **20 pigs** in this diet experiment.

edibble::set_units() Part 4

- You can add "block" units, or any other types of "units".

```r
start_design("My diet experiment") %>%
  set_units(pen = 10,
            pig = 50)
```

```
My diet experiment
├─pen (10 levels)
└─pig (50 levels)
```

- Above reads that there are **10 pens** and **50 pigs**.
- What do you think happens if we `serve_table()` on this? What's your expectation of the output?

Relationship between variables

We say that an edibble graph is **reconcilable** to an edibble table if for every variable, each level of the variable has a single linkage with level of another variable.

- Below edibble graph cannot be *reconciled* to an edibble table

```
start_design("My diet experiment") %>%
  set_units(pen = 10,
            pig = 50) %>%
  serve_table()
```

```
# An edibble: 0 x 2
# … with 2 variables: pen <unit(10)>, pig <unit(50)>
```

edibble::nested_in() Part 1

- But the units must be related in someway.

```
start_design("My diet experiment") %>%
  set_units(pen = 10,
            pig = nested_in(pen, 5))
```

```
My diet experiment
└─pen (10 levels)
  └─pig (50 levels)
```

- Above reads that there are **10 pens** and **5 pigs** in each pen.

The edibble table with nesting structure

```r
start_design("My diet experiment") %>%
  set_units(pen = 10,
            pig = nested_in(pen, 5)) %>%
  serve_table()
```

`edibble::nested_in()` Part 2

- More than one level of nesting:

```
start_design("My diet experiment") %>%
  set_units(pen = 10,
            pig = nested_in(pen, 5),
            time = nested_in(pig, 3)) %>%
  serve_table()
```

edibble::nested_in() Part 3

- Syntactic sugar for unbalanced unit structure. Again, the units don't need to be physical objects

```r
start_design("My diet experiment") %>%
  set_units(pen = 10,
            pig = nested_in(pen, 1:2 ~ 5,
                                 3 ~ 4,
                                 . ~ 2),
            time = nested_in(pig, 3))
```

```
My diet experiment
└─pen (10 levels)
  └─pig (28 levels)
    └─time (84 levels)
```

- Above reads that there are **10 pens**, **5 pigs** in pens 1 & 2, **4 pigs** in pen 3, and **2 pigs** in the remaining jars (so a total of $2 \times 5 + 4 + 2 \times 7 = 28$ pigs), and **3 time points** observed for each pig.

`edibble::nested_in()` Part 4

- You can refer units by its label instead of level:

```r
start_design("My diet experiment") %>%
  set_units(pen = c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J"),
            pig = nested_in(pen, c("A", "B") ~ 5,
                                        "C" ~ 4,
                                          . ~ 2),
            time = nested_in(pig, 3))
```

```
My diet experiment
└─pen (10 levels)
  └─pig (28 levels)
    └─time (84 levels)
```

edibble::set_trts() Part 1

- Treatment factors are like "units" but some distinguishable attributes added to the object if you use `set_trts`.

```r
start_design("My diet experiment") %>%
  set_trts(diet = c("NF", "HCD", "HFD", "HPD"))
```

```
My diet experiment
└─diet (4 levels)
```

- Below looks the same as above but also encodes another (long) label as well. Note: long label not used yet downstream.

```r
start_design("My diet experiment") %>%
  set_trts(diet = c(  "normal food" = "NF",
                   "high-carbon diet" = "HCD",
                      "high fat diet" = "HFD",
                  "high protein diet" = "HPD"))
```

edibble::set_trts() Part 2

- You can set the treatment first then units or vice-versa.

```
start_design("My diet experiment") %>%
  set_units(pen = 10) %>%
  set_trts(diet = c("NF", "HCD", "HFD", "HPD"))
```

```
My diet experiment
├─pen (10 levels)
└─diet (4 levels)
```

```
start_design("My diet experiment") %>%
  set_trts(diet = c("NF", "HCD", "HFD", "HPD")) %>%
  set_units(pen = 10)
```

```
My diet experiment
├─diet (4 levels)
└─pen (10 levels)
```

edibble::set_trts() Part 3

- Factorial treatments

```
start_design("My diet experiment") %>%
  set_trts(type = c("carb", "fat", "protein"),
           level = c("high", "low"))
```

```
My diet experiment
├─type (3 levels)
└─level (2 levels)
```

Setting variables later

- You can break the unit or treatment factors to another function later in the pipeline:

```
start_design("My diet experiment") %>%
  set_trts(type = c("carb", "fat", "protein")) %>%
  set_units(pen = 10) %>%
  set_trts(level = c("high", "low")) %>%
  set_units(pig = nested_in(pen, 5)) %>%
  set_units(time = nested_in(pig, 3))
```

```
My diet experiment
├─type (3 levels)
├─pen (10 levels)
│ └─pig (50 levels)
│   └─time (150 levels)
└─level (2 levels)
```

Revisiting the pig study

```r
library(edibble)
des <- start_design("My diet experiment") %>%
  set_trts(diet = c("carb", "protein", "fat"),
           breed = c("standard", "new")) %>%
  set_units(pen = 10,
            pig = nested_in(pen, 5))
```

`edibble::allocate_trts()` Part 1

```
des %>%
  allocate_trts( ~ pen)
```

Here, the experimental unit is easy to identify.

- Above is the same as below.

```
des %>%
  allocate_trts(diet:breed ~ pen)
```

- If treatment is not specified, factorial combination is assumed.
- But you can assign a treatment factor to another unit factor instead.

```
des %>%
  allocate_trts(diet ~ pen,
                breed ~ pig)
```

- Above is like the classic split-plot design where `diet` is allocated to `pen` and `breed` is allocated to `pig`.

`edibble::allocate_trts()` Part 2

```
des %>%
  allocate_trts(diet ~ pen,
                breed ~ pig) %>%
  serve_table()
```

Do you notice anything about the treatment order?

Yup, it's systematically ordered!

edibble::randomise_trts()

```r
set.seed(1)
des %>%
  allocate_trts(diet ~ pen,
                breed ~ pig) %>%
  randomise_trts() %>%
  serve_table()
```

Mix and match with other tools

- A lot of recent research efforts in experimental design are in developing algorithms to find the optimal design.
- `edibble` doesn't aim to be the best at randomisation and never will.
- Rather, the hope is that this step is replaced by others' great work!
- Data: 400 varieties in two blocks of 20 columns and 40 rows

```
rc <- start_design("od") %>%
  set_units(Rep = 2,
            Row = nested_in(Rep, 40),
            Plot = nested_in(Row, 20)) %>%
  set_trts(Variety = 400) %>%
  allocate_trts(Variety ~ Plot) %>%
  serve_table() %>%
  dplyr::mutate(across(Rep:Variety, as.factor))
```

```
od::od(fixed = ~ 1,
       random = ~ Variety + Rep:Row,
       permute = ~ Variety, swap = ~ Rep,
       search = 'random',
       data = rc, maxit = 1000)
```

edibble::make_classical

- Still want the "named" experimental design?

```
make_classical("crd", t = 5, n = 20)
```

edibble::set_rcrds()

- ⚠ Please note this API will likely change (🏳)

```
des %>%
  allocate_trts(diet ~ pen,
                breed ~ pig) %>%
  randomise_trts() %>%
  set_rcrds(pen = avg_temp,
            pig = c(diseased, inital_weight, final_weight)) %>%
  serve_table()
```

- The ***intention*** of ***which metric*** to capture ***on what*** is clearly specified.
- What is an observational unit is more obvious.

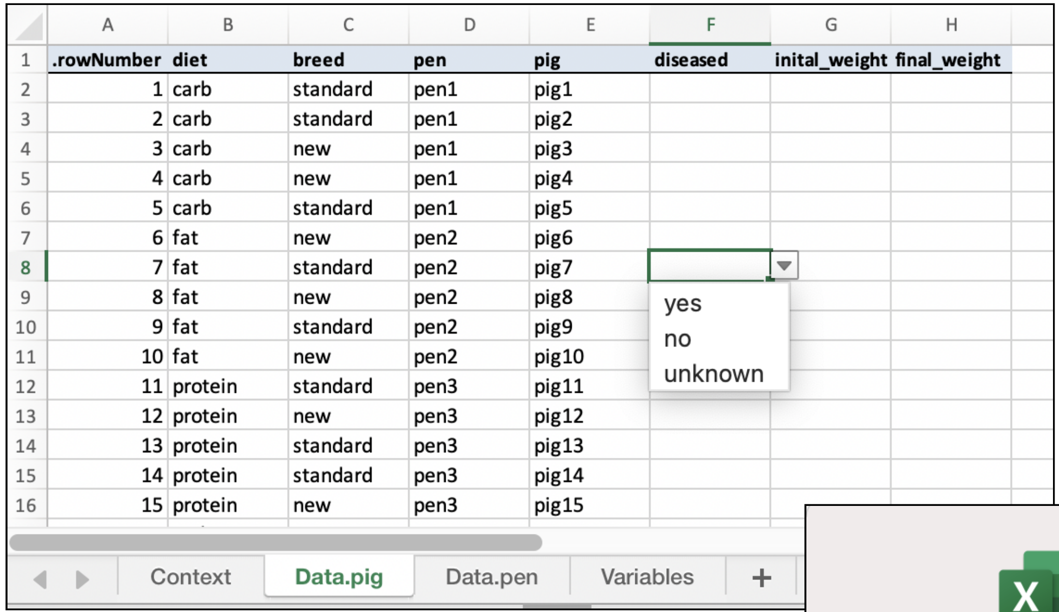
edibble::expect_rcrds()

```r
df <- des %>%
  allocate_trts(diet ~ pen,
                breed ~ pig) %>%
  randomise_trts() %>%
  set_rcrds(pig = c(diseased, inital_weight, final_weight),
            pen = avg_temp) %>%
  expect_rcrds(diseased = to_be_factor(levels = c("yes", "no", "unknown")),
               inital_weight = to_be_numeric(with_value(">=", 0)),
               final_weight = to_be_numeric(with_value(">=", 0))) %>%
  serve_table()
```

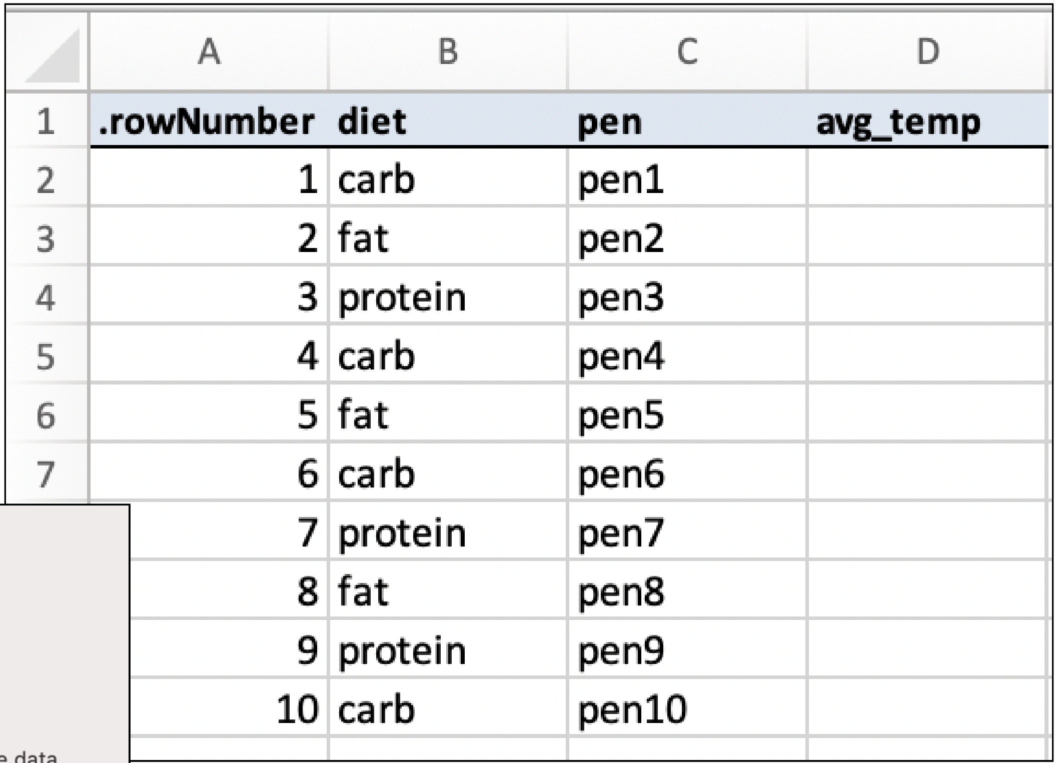
- This encodes data validation rules in the edibble object.
- But why bother do this?

edibble::export_design()

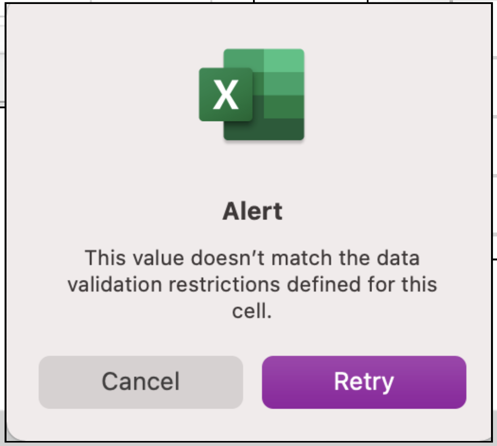
```r
export_design(df, file = "design.xlsx")
```



	A	B	C	D	E	F	G	H
1	.rowNumber	diet	breed	pen	pig	diseased	inital_weight	final_weight
2	1	carb	standard	pen1	pig1			
3	2	carb	standard	pen1	pig2			
4	3	carb	new	pen1	pig3			
5	4	carb	new	pen1	pig4			
6	5	carb	standard	pen1	pig5			
7	6	fat	new	pen2	pig6			
8	7	fat	standard	pen2	pig7			
9	8	fat	new	pen2	pig8			
10	9	fat	standard	pen2	pig9			
11	10	fat	new	pen2	pig10			
12	11	protein	standard	pen3	pig11			
13	12	protein	new	pen3	pig12			
14	13	protein	standard	pen3	pig13			
15	14	protein	standard	pen3	pig14			
16	15	protein	new	pen3	pig15			



	A	B	C	D
1	.rowNumber	diet	pen	avg_temp
2	1	carb	pen1	
3	2	fat	pen2	
4	3	protein	pen3	
5	4	carb	pen4	
6	5	fat	pen5	
7	6	carb	pen6	
	7	protein	pen7	
	8	fat	pen8	
	9	protein	pen9	
	10	carb	pen10	



Alert

This value doesn't match the data validation restrictions defined for this cell.

`edibble::fill_rcrds` or `edibble::simulate_rcrds`

- This is in my plans to implement 🚩
- `fill_rcrds` is a simple, quick way to simulate dummy data
- `simulate_rcrds` is a more flexible, proper simulation of data

Other plans

- Diagnosis of the design (e.g. skeleton ANOVA 🚩, design anatomy and efficiency calculation 🚩, Hasse diagram...)
- Suggest model for analysis 🚩
- Warning to user for unreplicated experiments 🚩

Visualising experimental designs with deggust

☢ This package is still nuclear-level experimental!

```r
remotes::install_github("emitanaka/deggust")

library(deggust)
```

❗ Rapid development phase — use with caution ❗

The **origin of the name**:

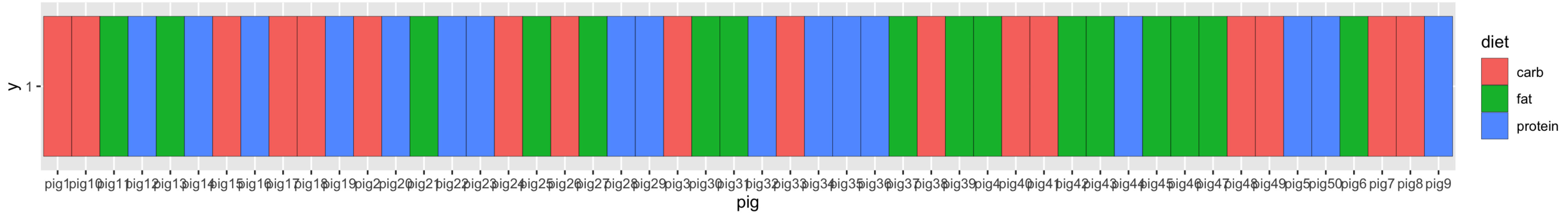
- **deggust** as in degustation
- **de** = design of experiments
- **gg** = ggplot object

Back to the pig design

```r
set.seed(1)
df1 <- start_design("My diet experiment") %>%
  set_trts(diet = c("carb", "protein", "fat")) %>%
  set_units(pig = 50) %>%
  allocate_trts(~ pig) %>%
  randomise_trts() %>%
  serve_table()
```

Visualising designs with ggplot2

```r
library(ggplot2)
df1 %>%
  # make it normal data frame
  as_data_frame() %>%
  # plot using ggplot
  ggplot(aes(pig, "1", fill = diet)) +
    geom_tile(color = "black")
```

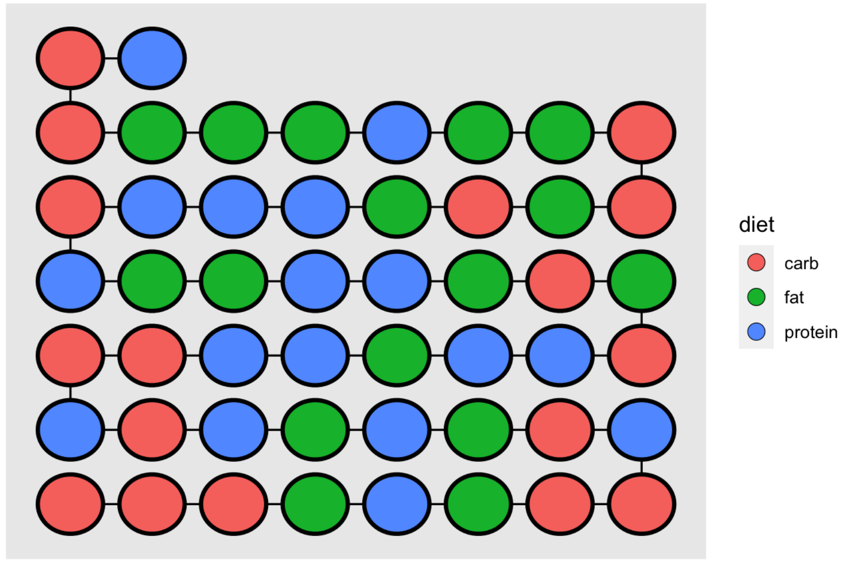


- Slightly painful if you want to *quickly* visualise your design.

deggust::autoplot() Part 1

- Just autoplot it!

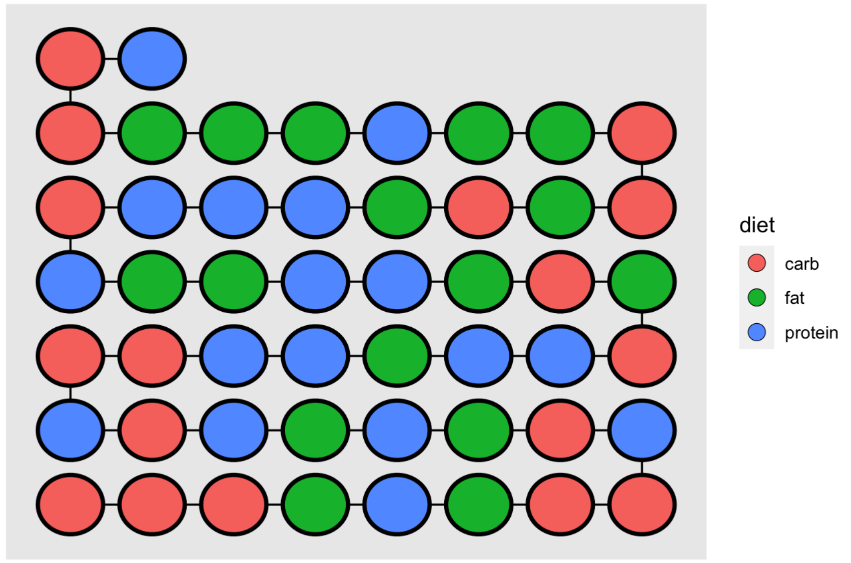
```r
library(deggust)

autoplot(df1)
```



deggust::autoplot() Part 2

- It makes some decision for you of how to plot which can be customised in two ways:
  1. modified scale and theme like any ggplot objects!
  2. as arguments in the `autoplot` function

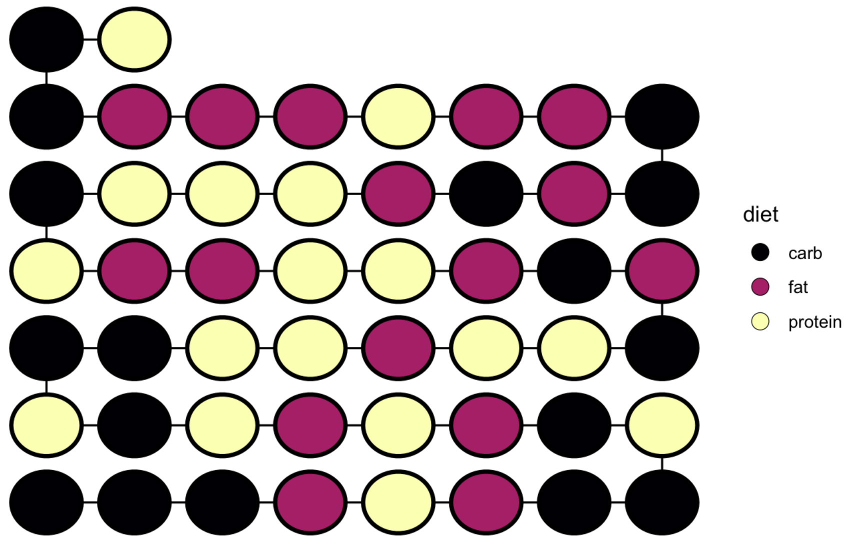
```
autoplot(df1)
```



deggust::autoplot() Part 3

- It makes some decision for you of how to plot which can be customised in two ways:
  1. **modified scale and theme like any ggplot objects!**
  2. as arguments in the `autoplot` function

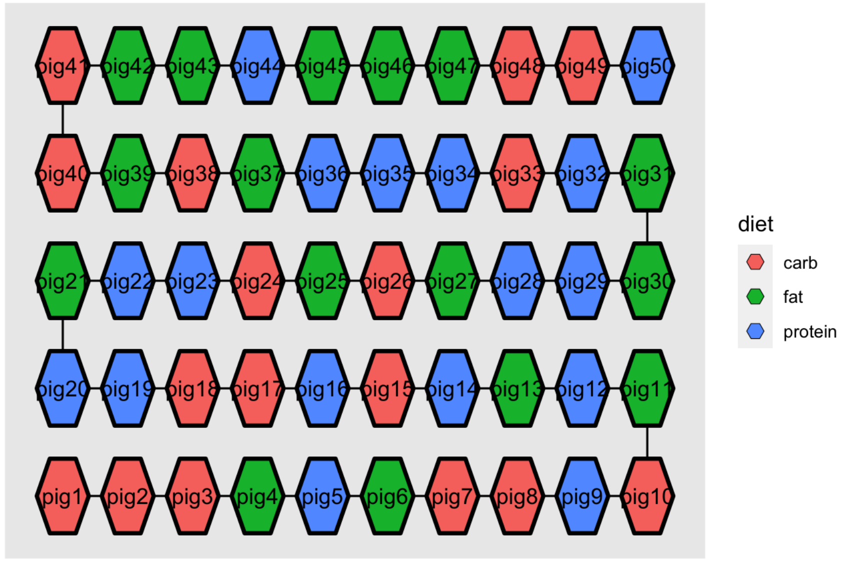
```
autoplot(df1) +
  # ggplot2 functions below
  theme_void() +
  scale_fill_viridis_d(option = "A")
```



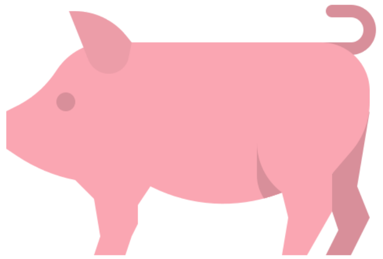
`deggust::autoplot()` Part 4

- It makes some decision for you of how to plot which can be customised in two ways:
  1. modified scale and theme like any ggplot objects!
  2. as arguments in the `autoplot` function

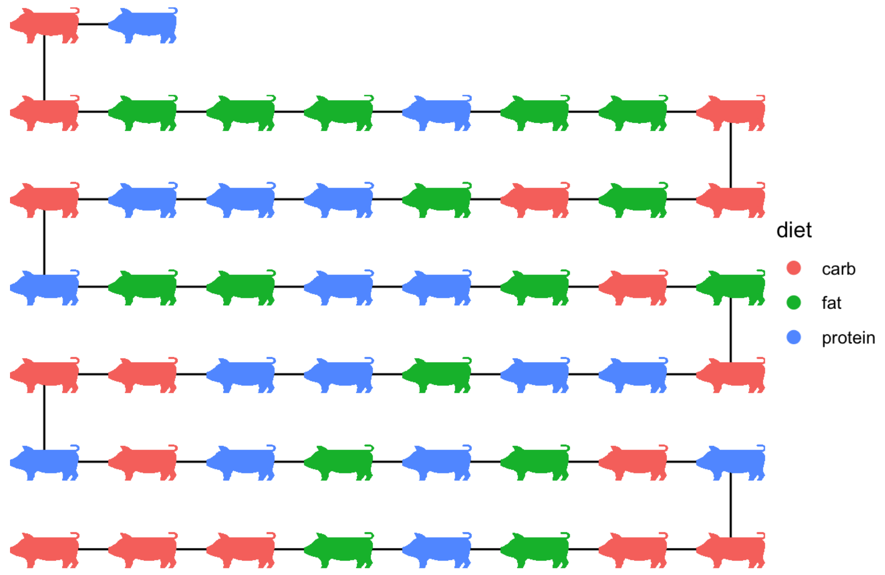
```
autoplot(df1,
         shape = "hexagon",
         text = TRUE,
         aspect_ratio = 0.5)
```



deggust::autoplot() Part 5



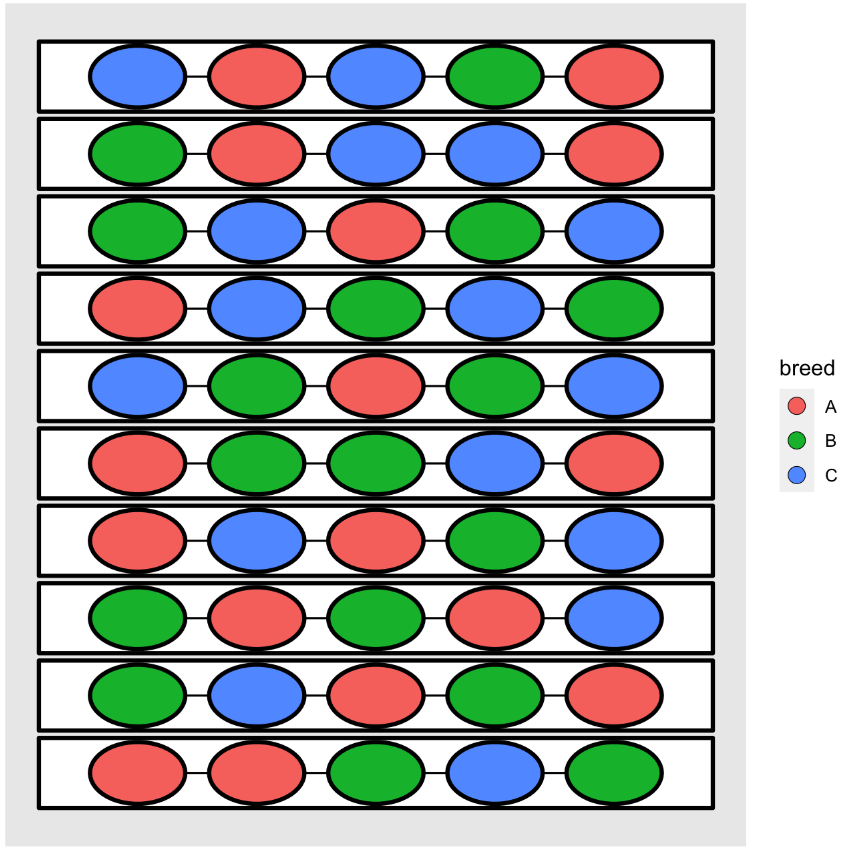
```
autoplot(df1,
         image = "images/pig.png") +
  theme_void()
```



deggust::autoplot() Part 6

- Nested design

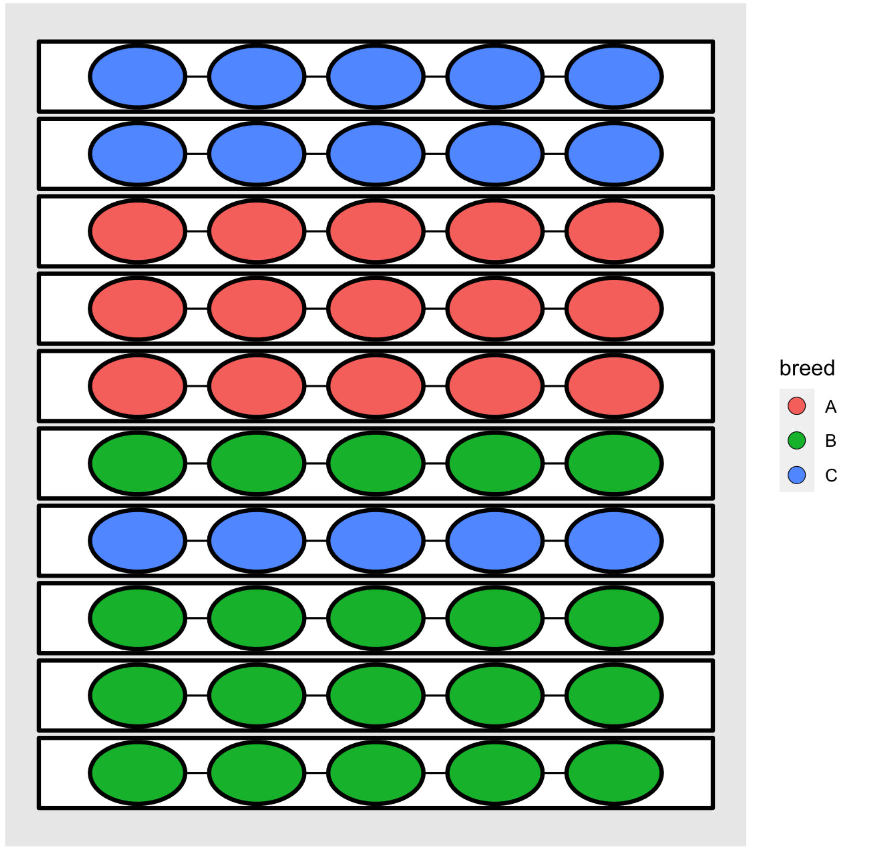
```
set.seed(2021)
start_design() %>%
  set_units(pen = 10,
            pig = nested_in(pen, 5)) %>%
  set_trts(breed = c("A", "B", "C")) %>%
  allocate_trts(breed ~ pig) %>%
  randomise_trts() %>%
  serve_table() %>%
  autoplot()
```



deggust::autoplot() Part 7

- What changed here?

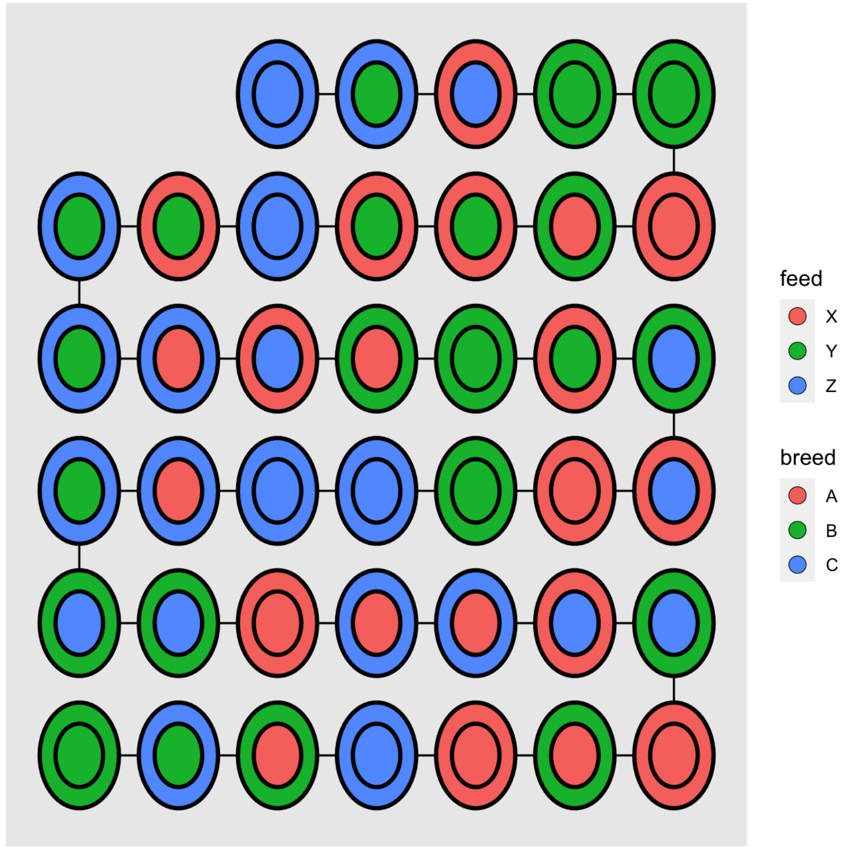
```
set.seed(2021)
start_design() %>%
  set_units(pen = 10,
            pig = nested_in(pen, 5)) %>%
  set_trts(breed = c("A", "B", "C")) %>%
  allocate_trts(breed ~ pen) %>%
  randomise_trts() %>%
  serve_table() %>%
  autoplot()
```



deggust::autoplot() Part 8

- Factorial experiment

```
set.seed(2021)
start_design() %>%
  set_units(pig = 40) %>%
  set_trts(breed = c("A", "B", "C"),
           feed = c("X", "Y", "Z")) %>%
  allocate_trts(breed:feed ~ pig) %>%
  randomise_trts() %>%
  serve_table() %>%
  autoplot()
```

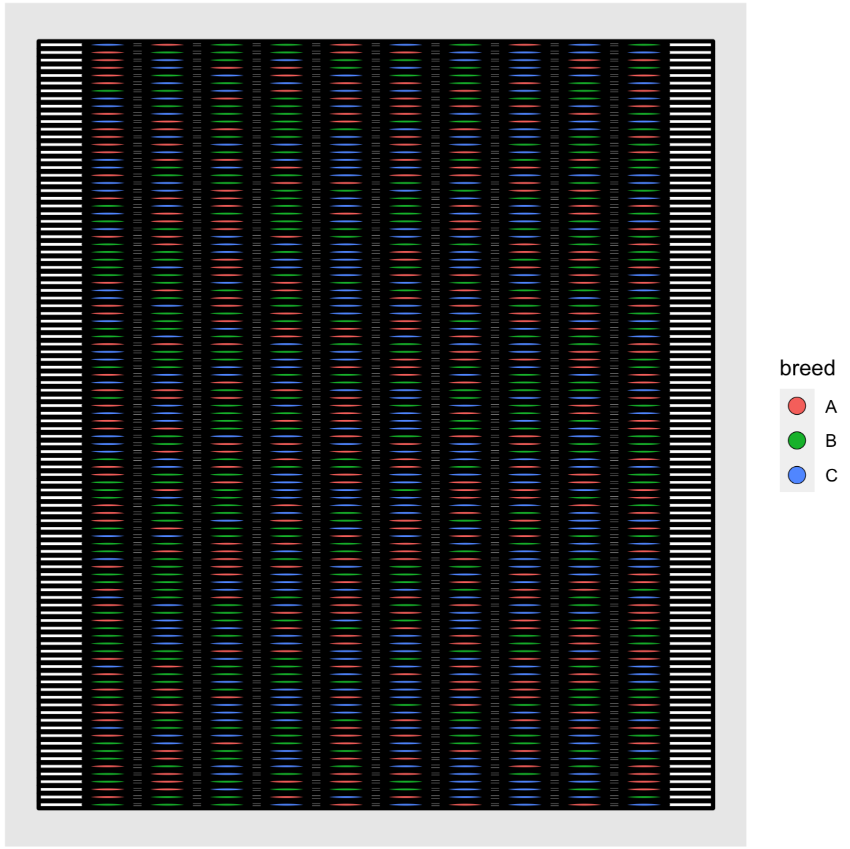


MONASH University

deggust::autoplot() Part 10

- Is your design too big to fit in the plot?

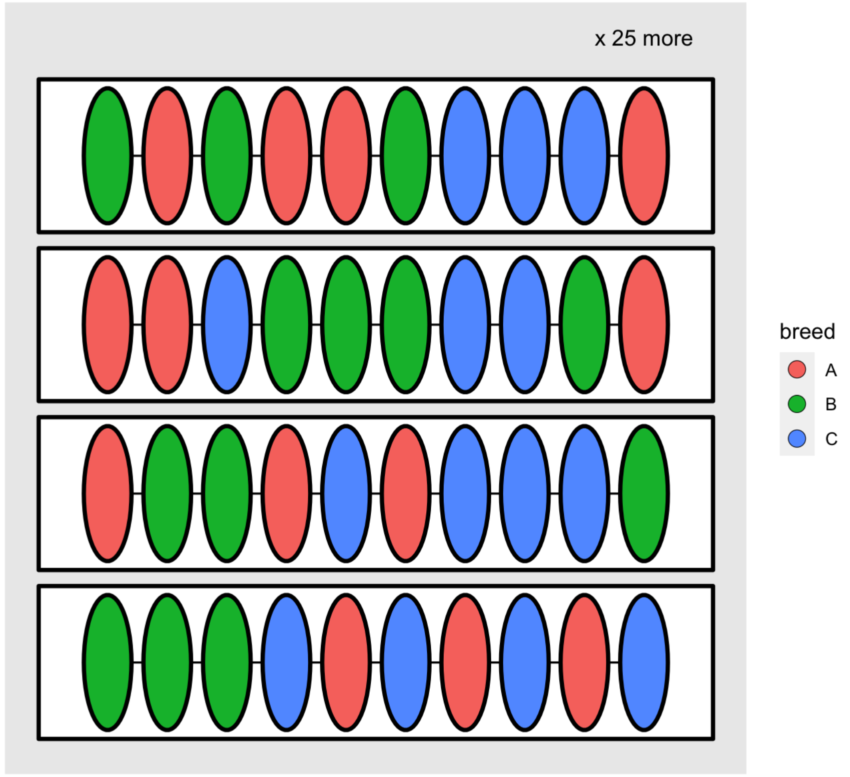
```
set.seed(2021)
start_design() %>%
  set_units(pen = 100,
            pig = nested_in(pen, 10)) %>
  set_trts(breed = c("A", "B", "C")) %>%
  allocate_trts(breed ~ pig) %>%
  randomise_trts() %>%
  serve_table() %>%
  autoplot()
```



deggust::autoplot() Part 10

- Is your design too big to fit in the plot?
- Subset it!

```
set.seed(2021)
start_design() %>%
  set_units(pen = 100,
            pig = nested_in(pen, 10)) %>
  set_trts(breed = c("A", "B", "C")) %>%
  allocate_trts(breed ~ pig) %>%
  randomise_trts() %>%
  serve_table() %>%
  dplyr::filter(pen %in% c("pen1", "pen2
  autoplot() +
  annotate("text", x = 10, y = 4.7, labe
```



🧆

Designing experiments using a web app with edibbleGUI

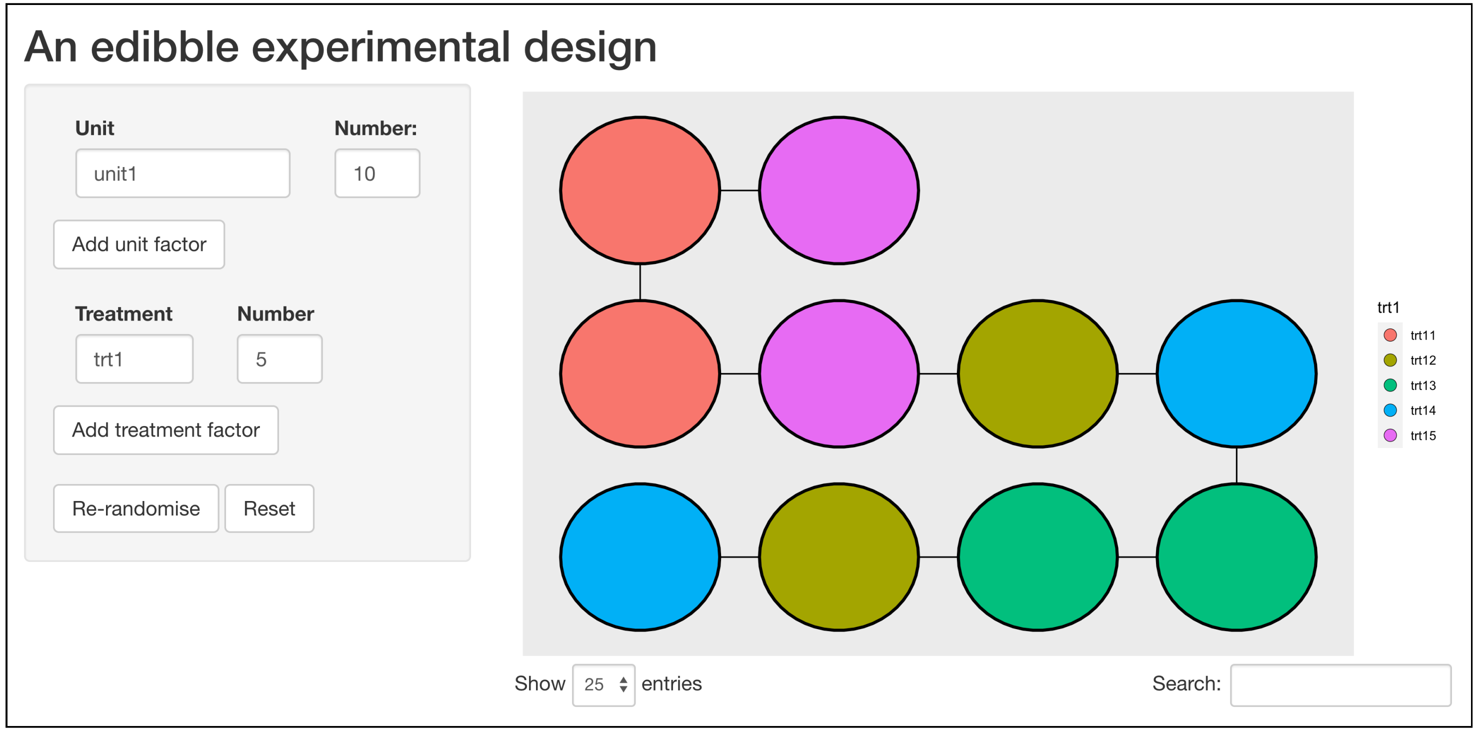
☠ Totally not ready

very bare bones, not enough meat for consumption

```r
remotes::install_github("emitanaka/edibbleGUI")

edibbleGUI::app()
```

❗ **Rapid development phase — use with caution** ❗

edibbleGUI::app()



An edibble experimental design

Unit: unit1

Number: 10

Treatment: trt1

Number: 5

- This app is bound to change.
- The development of the app will always lag from `edibble`.

Some plans:

- ☐ Show corresponding code
- ☐ Export design table
- ☐ Host app on the web
- ⁙ Better integration with `edibble`

🍹

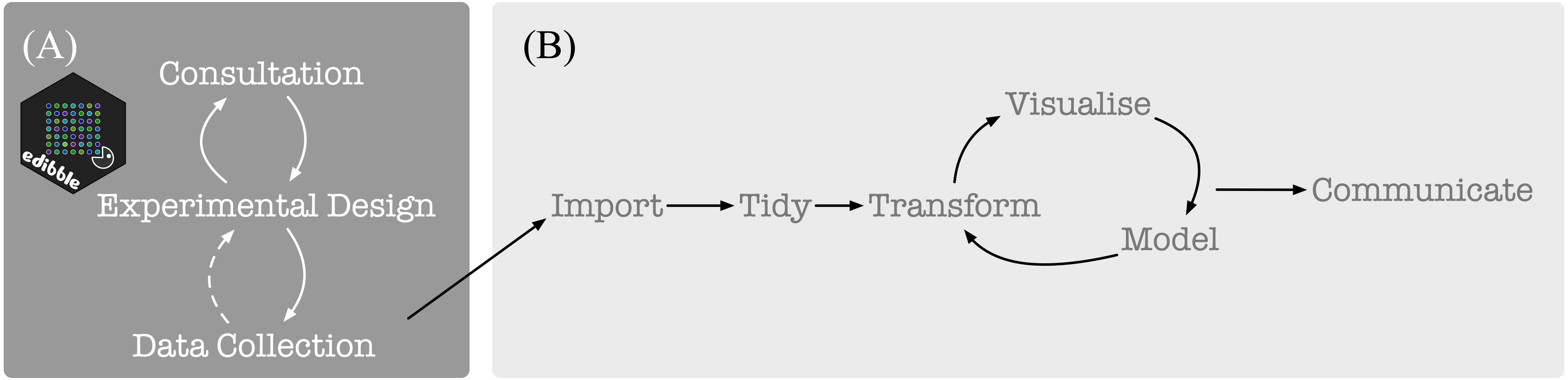
Future directions

Experimental design is different to statistical analysis

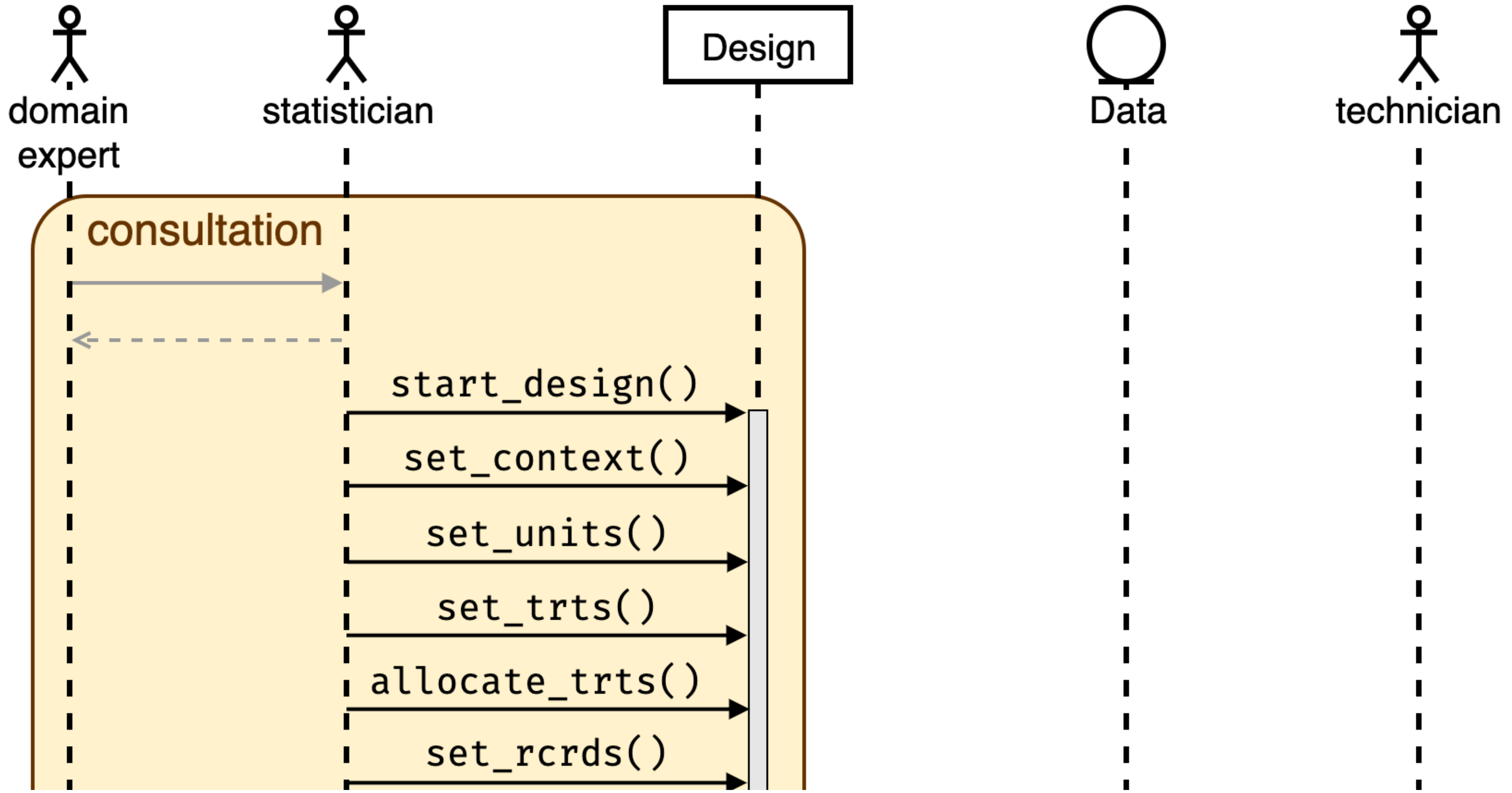
- Constructing an experimental design is different to analysis:
  - redoing an experiment is generally more expensive than redoing an analysis
  - often there is no "data" but "information" only"
    - taking into account experimental context is important
    - selecting a design from a list of known designs often means that you are not adapting the design to the context

What `edibble` does

- `tidyverse` does well for processes in (B)
- `edibble` aims to tackle (A)



Designing for the *whole* experiment



`edibble` developments

- All developments are open-source and transparent:
  - 🔗 https://github.com/emitanaka/edibble
  - 🔗 https://github.com/emitanaka/deggust
  - 🔗 https://github.com/emitanaka/edibbleGUI
- `edibble` and its extensions `deggust` and `edibbleGUI` are currently one-person effort
- Something not working? Feature request? Feel free to submit it in issues!
  - 🏴 https://github.com/emitanaka/edibble/issues
  - 🏴 https://github.com/emitanaka/deggust/issues
  - 🏴 https://github.com/emitanaka/edibbleGUI/issues
- 🚧 `edibble` is a work-in-progress with plans to submit the CRAN version later this year
- `edibble` will continue to improve — how *fast* it improves is another story!

This slide is made using the `xaringan` R-package and found at

emitanaka.org/slides/DAF2021/edibble

Thank you!

Emi Tanaka

Department of Econometrics and Business Statistics,
Monash University, Melbourne, Australia

emi.tanaka@monash.edu

@statsgen