Schedule

Theme	Description
Introduction to testing	We will present the motivation (why) and basics (how) behind testing.
Break	
Demo: unit tests and refactoring in Python and R	We will show some code examples of written tests. This includes discussing the Python and R packages suitable for testing. Through an example code we show the process of refactoring and adding tests. Here we also discuss the cases when testing is appropriate with the involvement of the participants.
Break	
Practical: refactor and test code in groups	Participants will apply presented packages and write tests for either their own or example functions provided by us.
Practical: discussion	Insights and questions will be discussed with the whole room.
(Optional) Automated tests via GitHub Actions	If time allows, we will demonstrate how tests can be automated through popular open-source platforms, such as GitHub Actions
Wrap-up	We will summarize the main aspects of testing and answer all remaining questions from the participants.

Before starting

https://github.com/ferenckata/nbd24_testing

Why testing?

How are you testing your own code?

How are you testing your own code?

- Here we are going to discuss:
  - Systematical
  - Reusable
  - Automated

  tests that aim to ensure the correct implementation.
- Not the model / science / biology.

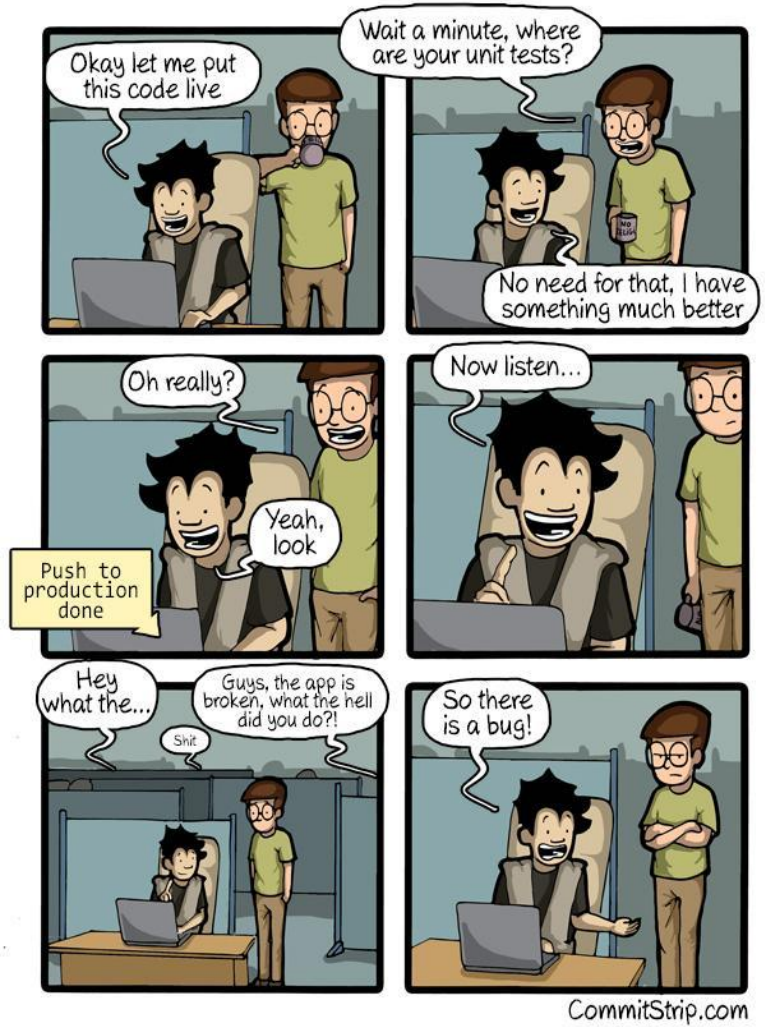
Bit more clarification before we begin

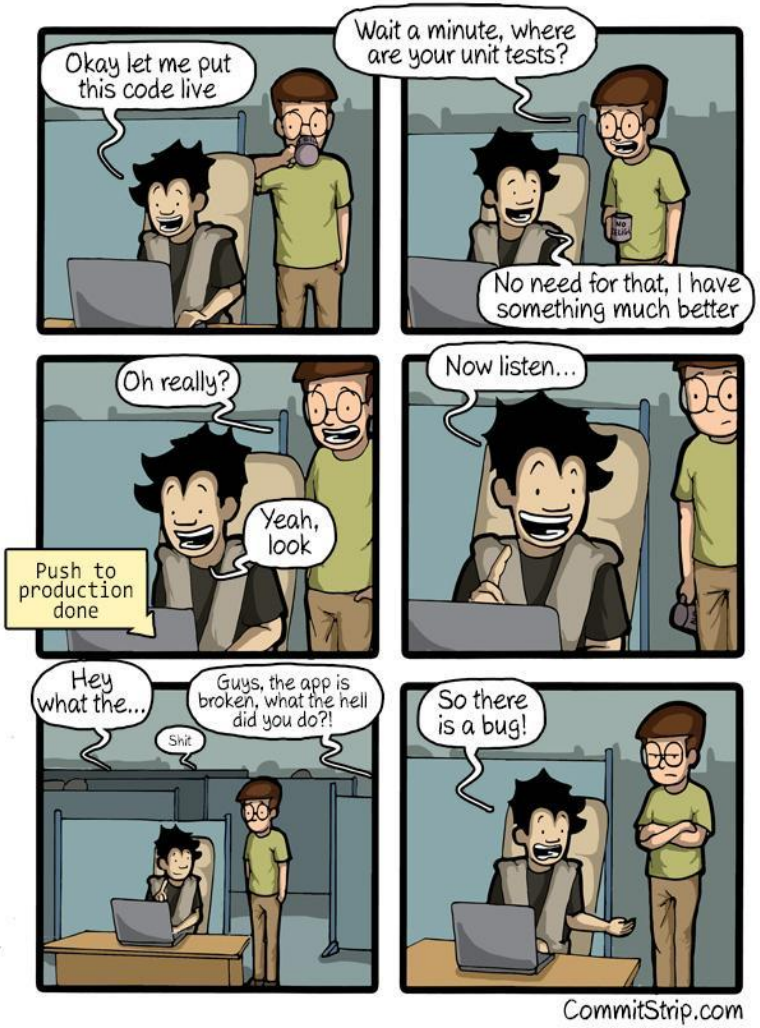
Testing the implementation

- **How does my code behave given a specific input?**
- Collected in a test folder outside of source
- Does not run with the main code (a.k.a. not “shipped”)

Sanity checks

- **Does the input complies with the code?**
- a.k.a. defensive programming
- Written in the source code
- Runs when the main code runs


Okay let me put this code live
Wait a minute, where are your unit tests?
No need for that, I have something much better
Oh really?
Yeah, look
Push to production done
Now listen...
Hey what the...
Shit
Guys, the app is broken, what the hell did you do?!
So there is a bug!
CommitStrip.com



≈

Improving software quality in bioinformatics through teamwork

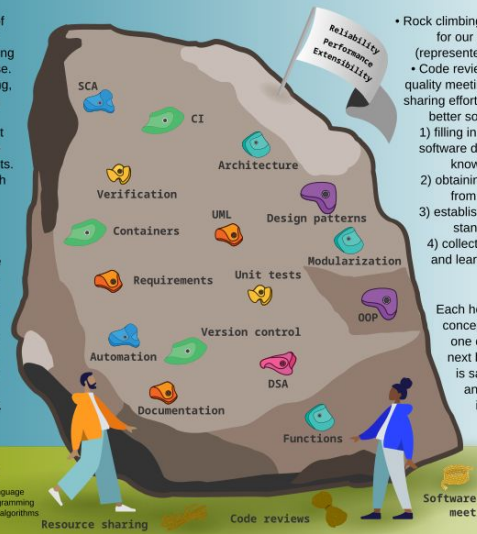
Katalin Ferenc[1], Ieva Rauluseviciute[1], Ladislav Hovan[1], Vipin Kumar[1], Marieke Kuijjer[1,2,3], Anthony Mathelier[1,4,5]

1 Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway
2 Department of Pathology, Leiden University Medical Center, 2333 ZA Leiden, The Netherlands
3 Leiden Center for Computational Oncology, Leiden University Medical Center, 2333 ZA Leiden, The Netherlands
4 Department of Medical Genetics, Institute of Clinical Medicine, University of Oslo and Oslo University Hospital, Oslo, Norway
5 Centre for Bioinformatics, Faculty of Mathematics and Natural Sciences, University of Oslo, Oslo, Norway

Abstract: Since high-throughput techniques became a staple, computational algorithms and scientific software generation boomed. **However, more recently, it has been noted that bioinformatics software, is falling short of what one would expect from software intended to produce robust scientific insights.** Code can be hard to test (or independently verify), reuse, and maintain. **Relying on such code can directly lead to poor reproducibility of scientific results.** Industry addresses comparable shortcomings through a team-based organization for code production. Teamwork directly impacts code quality through knowledge sharing, redundancy, and standards. It prompts us to incorporate team activity elements into software development process. **We established a learning community centered on discussing various aspects of software development and building up a shared software engineering knowledge base**. Regular members improved their coding skills, became more efficient bioinformaticians, and obtained detailed knowledge about peers' work.

• A community of bioinformatics practitioners sharing technical expertise.
• Learning, sharing, and discussing software.
• Learning about basic and more advanced concepts.
• Reviewing each other's code.

Our experience compels us to advocate for improving the software development culture within bioinformatics through local collective effort.



• Rock climbing as a metaphor for our activities (represented as ropes).
• Code reviews, software quality meetings, resource-sharing efforts help us write better software by:
1) filling in gaps in our software development knowledge
2) obtaining feedback from peers
3) establishing shared standards
4) collecting scripts and learning tools

Each hold represents a concept, using which one can reach the next hold. Climbing is safer, quicker, and more fun in teams.

SCA: static code analysis
CI: continuous integration
UML: unified modeling language
OOP: object-oriented programming
DSA: data structures and algorithms

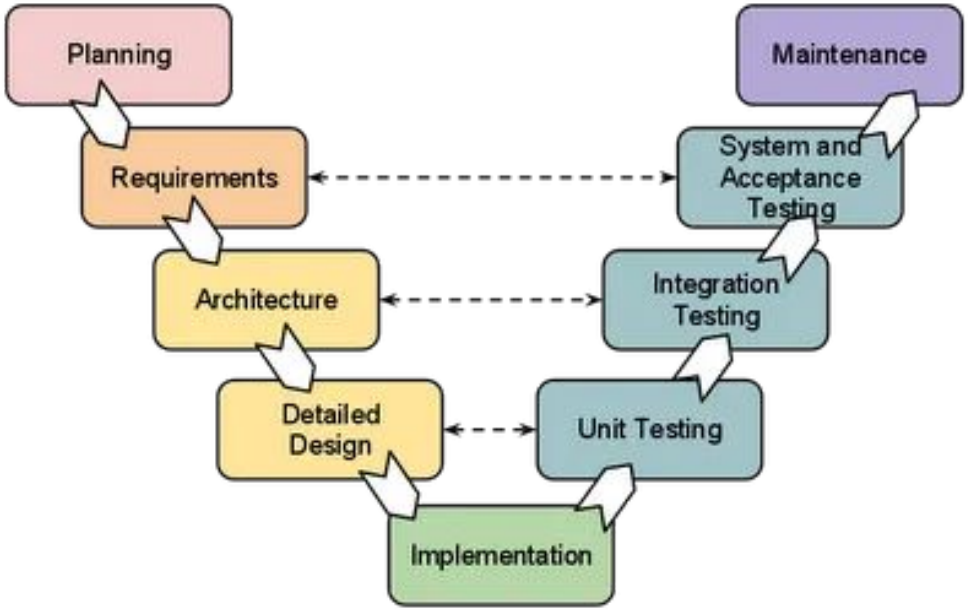
Software quality meetings:
• focusing on basic computer science and software engineering concepts;
• understanding the trade-offs and achieving good quality software;
• members share their expertise.

Code reviews:
• discussing concrete code examples;
• alternative group meeting, where the code base is the focus;
• increase in technical expertise;
• establishing standards for coding practices.

Resource sharing:
• recordings of the software quality meetings;
• shared repository of scripts for recurring tasks within the team.

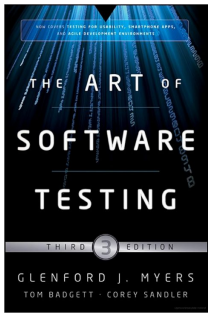
The Research Council of Norway · Scientia Fellows · KREFTFORENINGEN · HELSE SØR-ØST · NORDIC EMBL PARTNERSHIP FOR MOLECULAR MEDICINE · EMBL

Software engineering, ideal workflow



Principles of Testing

- Intent of finding errors - you should expect to find errors
- The expected result of a test case needs to be defined
- Test results should actually be checked
- Test cases must cover invalid and unexpected input conditions
- Create permanent test cases (regression testing - to not bring back old bugs when adding new features)
- Errors tend to cluster - if you found errors already, there's a good chance to get more
- Untestable code is usually of poor quality



Myers, Glenford J., Corey Sandler, and Tom Badgett. *The art of software testing*. Hoboken, N.J: John Wiley & Sons, 2012.

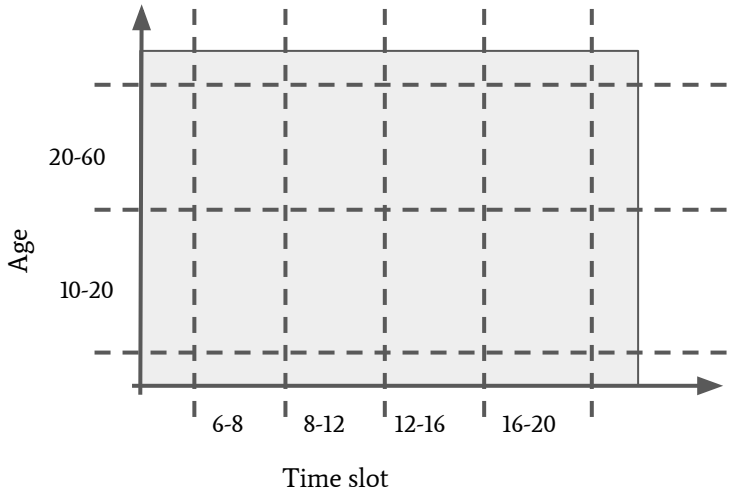
Levels of testing

- **Unit test:** Test individual piece of code (method or class)
- **Integration test:** When putting the units together, one has to make sure that their interaction does not produce error
- **Regression test:** Whenever any part of the system changes (maybe just fixing a bug) all test should run again
- **Acceptance test:** Testing with a customer to make sure that the application actually does what the customer wants

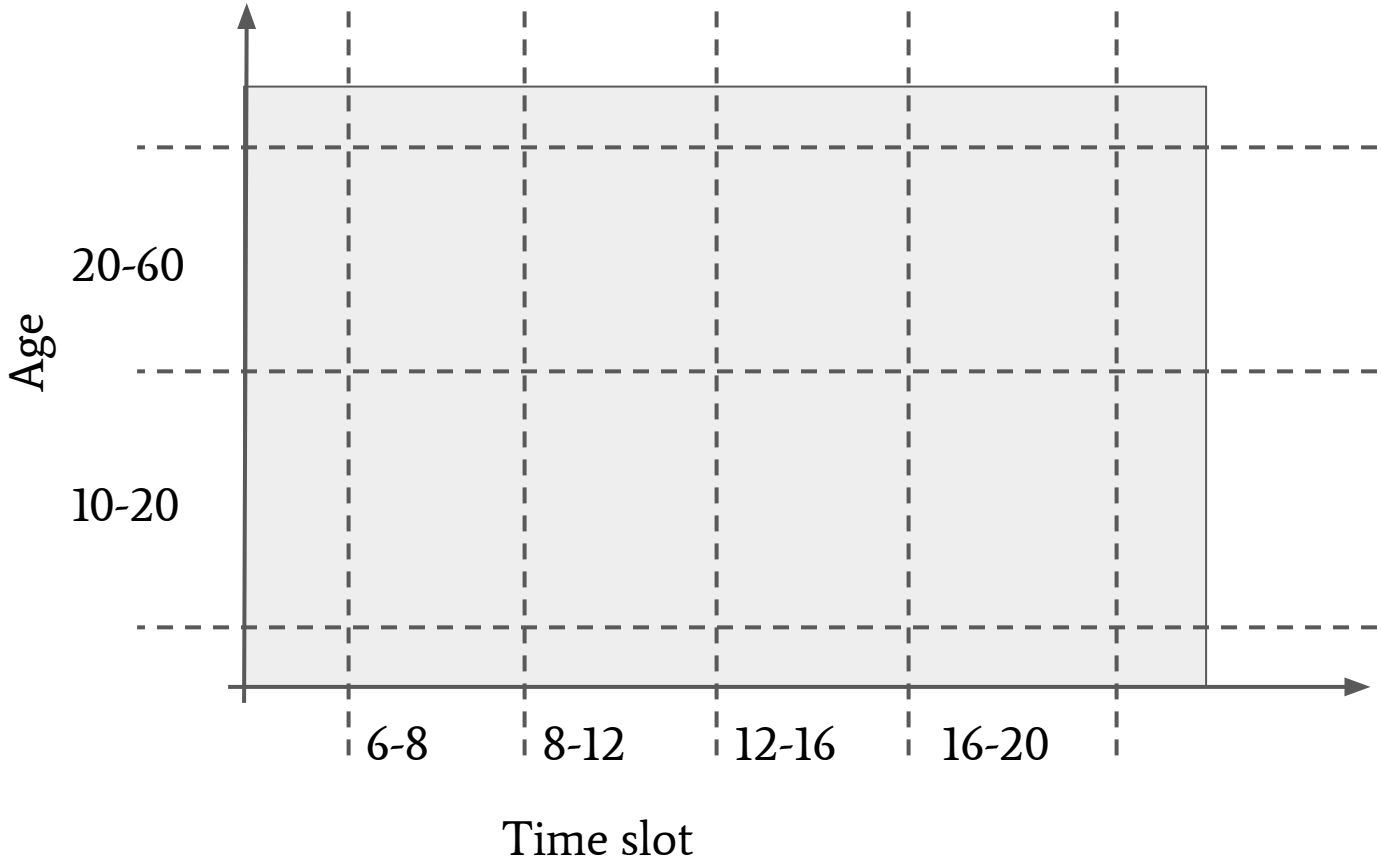
What to test?

Example:

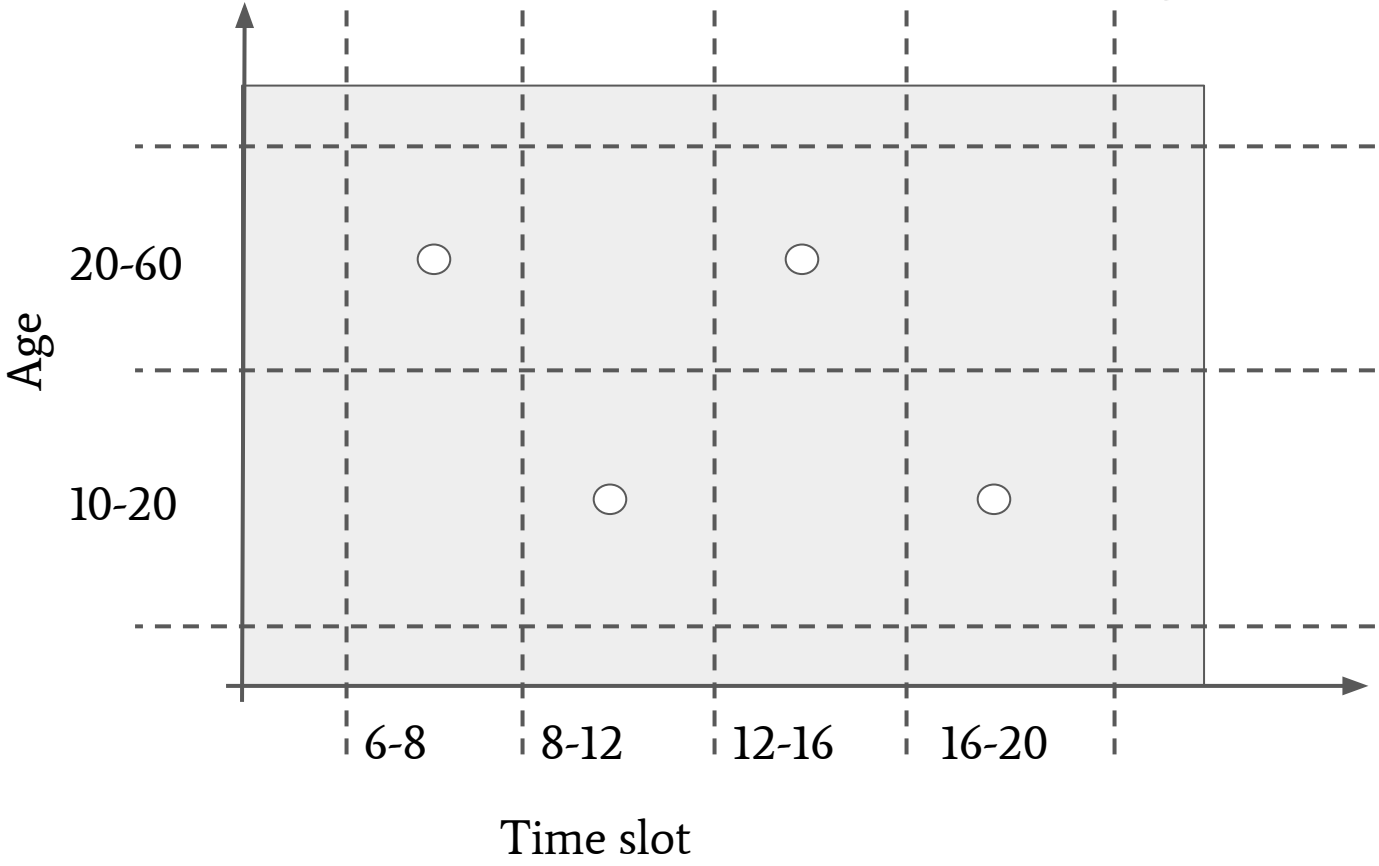
- There is a code that calculates the entrance fee for a swimming pool. There are two different age groups with different prices and 4 time slot per day that also affects the price.
- How would you test your system?



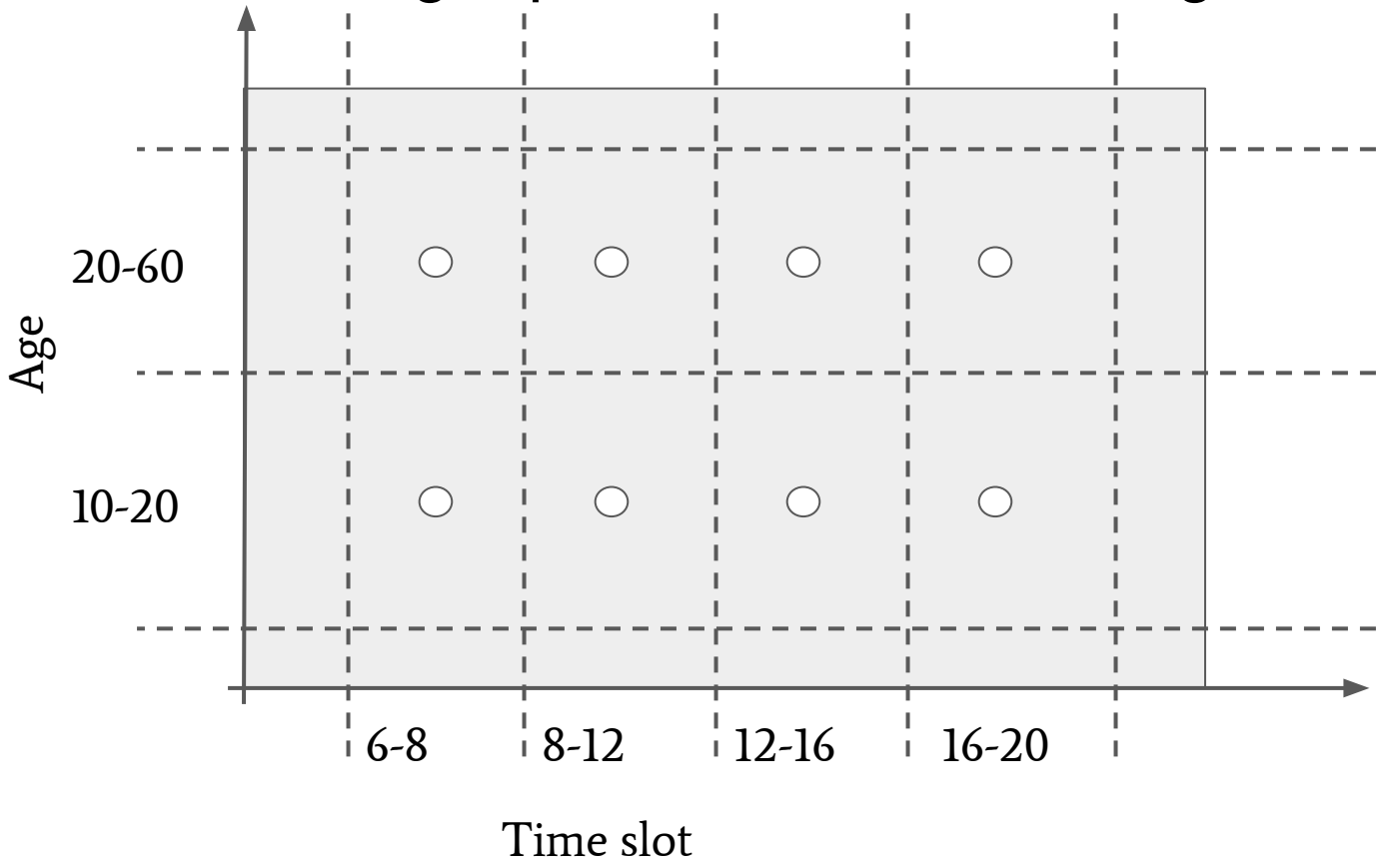
What to test?



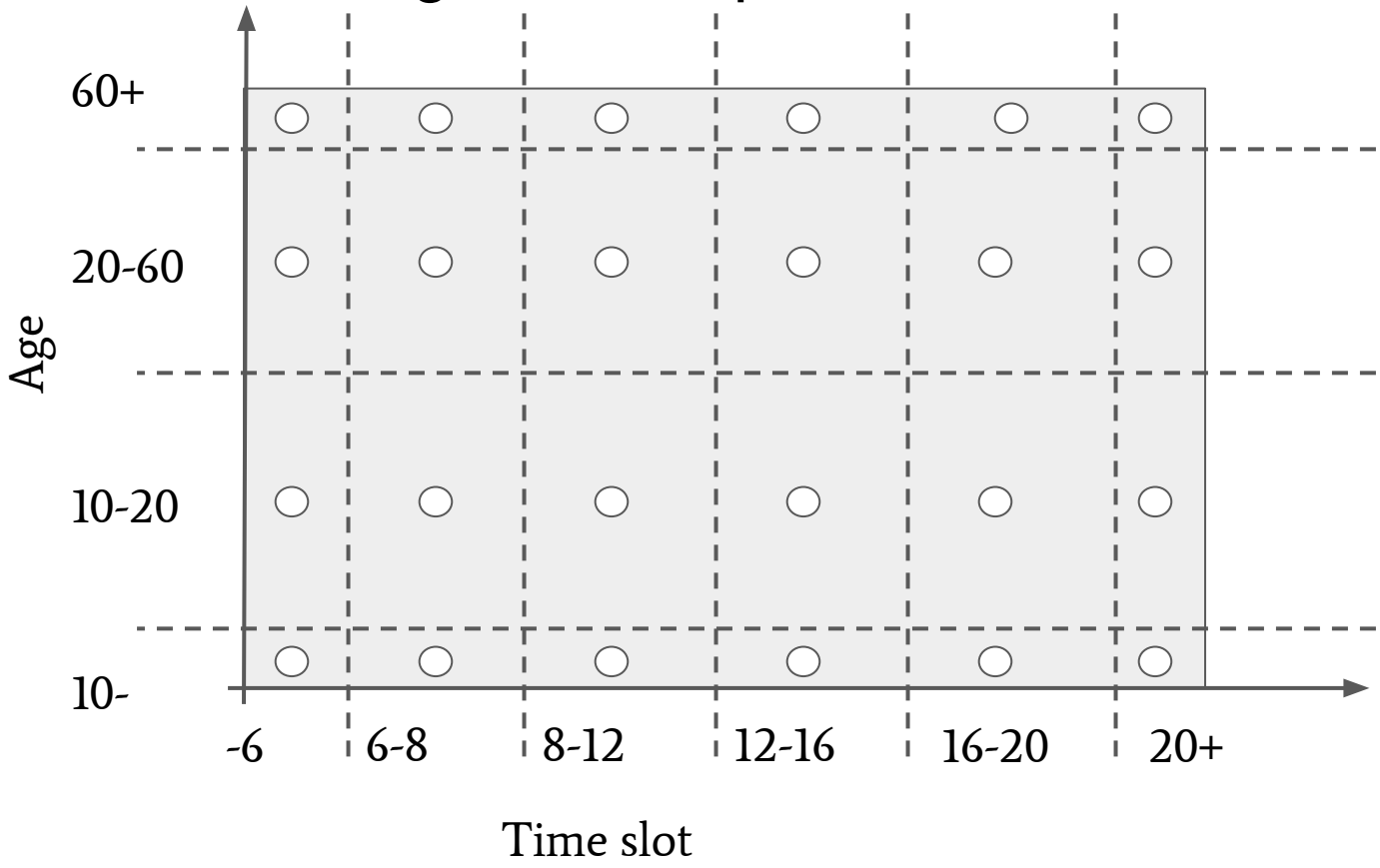
What to test? - Weak equivalence class testing



What to test? - Strong equivalence class testing



What to test? - Strong robust equivalence class testing



How is it relevant for us?

How is it relevant for us?

- What if my code has only a few functions?
  - Test those functions
  - Consider organizing your code to be more modular
  - Assertions can be useful on their own when processing data (eg. assertthat::assert_that(dim(enhancers)[1]>0))

How is it relevant for us?

- What if my code has only a few functions?
  - Test those functions
  - Consider organizing your code to be more modular
  - Assertions can be useful on their own when processing data (eg. assertthat::assert_that(dim(enhancers)[1]>0))
- How can I test my big data analysis code?
  - Maybe you don’t need extensive testing, especially if using borrowed functions (eg. base R, sklearn)
  - However, you can consider simulating or randomizing data to ensure you understand the behaviour of the tools

How is it relevant for us?

- What if my code has only a few functions?
  - Test those functions
  - Consider organizing your code to be more modular
  - Assertions can be useful on their own when processing data (eg. assertthat::assert_that(dim(enhancers)[1]>0))
- How can I test my big data analysis code?
  - Maybe you don’t need extensive testing, especially if using borrowed functions (eg. base R, sklearn)
  - However, you can consider simulating or randomizing data to ensure you understand the behaviour of the tools
- When should I bother with this at all?
  - Quite easy to transform your ad-hoc tests to unit tests, but probably not needed during experimenting with tools and ideas
  - At some point (eg. when you settled on a method) it is a good idea to refactor your code and while doing so, it is a perfect opportunity to include tests

Some examples

- Python
  - Let’s take a look
  - https://docs.python.org/3/library/unittest.html
- R
  - Let’s take another look
  - https://r-pkgs.org/testing-basics.html

Let’s start testing!

	Python / R
A	Usually does scripting
B	Usually writes functions
C	Advanced level

https://github.com/ferenckata/nbd24_testing

Further info

- Continuous integration
- Test driven development
- Mocking
- Simulated data
- Debugging (what to do when you found a bug)