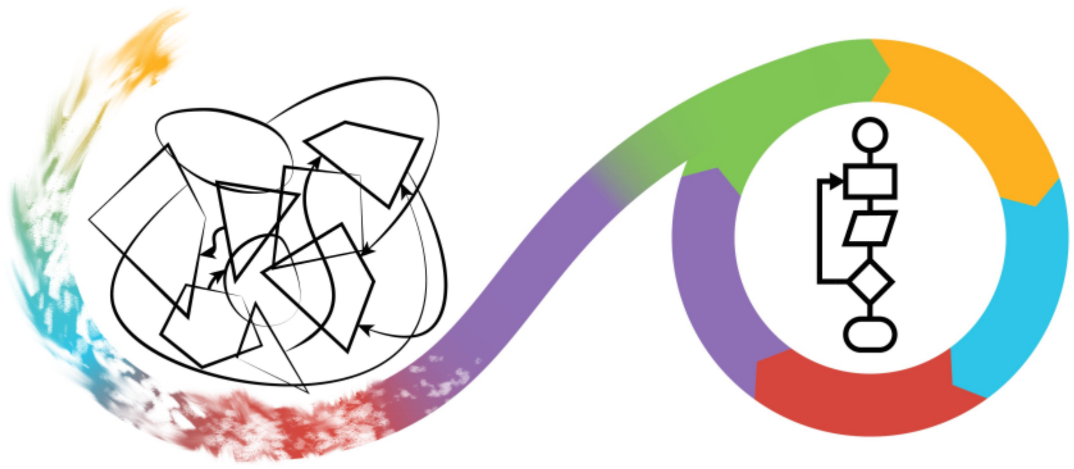
Algorithms

Richèl Bilderbeek

1 The Big Picture

https://github.com/UPPMAX/programming_formalisms/blob/main/tdd/tdd_lecture/tdd_lecture.qmd



1.1 Breaks

Please take breaks: these are important for learning. Ideally, do something boring (**newport2016deep?**)!

1.2 Schedule

Day	From	To	What
Thu	12:00	13:00	Lunch
Thu	13:00	14:00	Project: improve class code

Day	From	To	What
Thu	14:00	14:15	Break
Thu	14:15	15:00	Lecture: Function design
Thu	15:00	15:15	Break
Thu	15:15	15:45	Project: improve function code
Thu	15:45	16:00	Reflection

2 Algorithms

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.1 Problem

How do I write functions [1] that are:

- easy to use

- correct
- fast [2]

References;

- [1] For now, we use `algorithm == function`, as the definition of an algorithm is 'a step-by-step procedure for solving a problem or accomplishing some end' [3],
- [2] pick any vague definition
- [3] https://www.merriam-webster.com/dictionary/algorithm

2.2 What is a good function?

A good function:

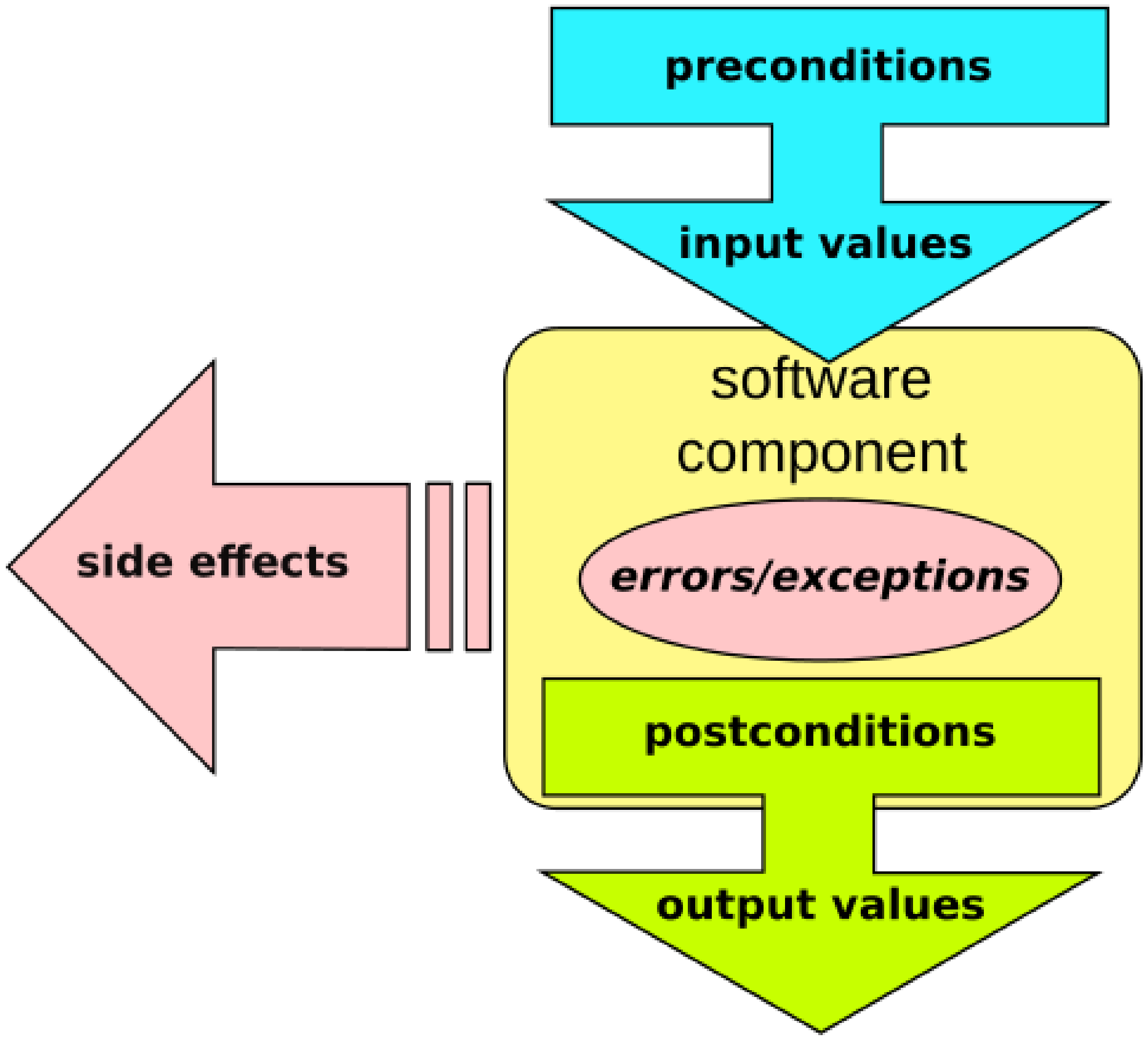
. . .

- Is documented
- Has a clear name
- Has a clear interface
- Does one thing correctly
- Is tested
- Gives clear error messages
- Fast iff needed

2.3 References

- C++ Core Guidelines on functions
- R Tidyverse style guidelines on functions
- The Hitchhiker's Guide to Python on general concepts
- PEP 20: The Zen of Python

2.4 Design by contract



Source: Wikipedia

2.5 A good function is documented

```
def sort_1(x):
  """Sort list `x` in-place.

  Returns nothing
  """

def sort_2(x):
  """Sort list `x`.

  Returns the sorted list.
  """

assert sort_1.__doc__
assert sort_2.__doc__
```

2.6 A good function has a clear name

> There are only two hard things in Computer Science: cache invalidation and naming things
>
> Phil Karlton

2.7 Examples of bad function names?

Could you give examples of bad function names?

. . .

- `calculate`: calculates what?

. . .

- `calc_bfgt`: calculates what??

. . .

- `prime`: a prime number is a data type. What does this function do?

. . .

- `needleman_wunch`: this is a technique to get a DNA alignment.

Figure 1: https://www.karlton.org/karlton/images/with-fish.jpg

. . .

2.8 Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *        x
```

How would you call the algorithm that detects the location of the `*` (but not of the `x`, as the `*` comes earlier)?

2.9 Solutions 1

- `get_first_mismatch_pos`, `get_first_mismatch_locus`, `find_first_mismatch_pos`, `find_first_mismatch_locus`
- you answer that …
    1. starts with a verb
    2. is as English as possible
    3. only uses common abbreviations

2.10 Example 2

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *        *
```

How would you call the algorithm that detects all the locations of the `*`s?

2.11 Solutions 2

- `find_mismatch_positions`, `find_mismatch_loci`, `get_mismatch_positions`, `get_mismatch_loci`
- you answer that …
  1. starts with a verb
  2. is as English as possible
  3. only uses common abbreviations

2.12 Example 3

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

How would you call the algorithm that makes the sequences have as much similarities as possible, by possibly inserting a `-`

```
AAACCCGGGTTT
ATACC-GGGTTT
```

2.13 Solutions 3

- `align_seqs`, `align_sequences`, `align_dna_sequences`, `align_dna_seqs`, `calc_aligned_seqs`, `get_aligned_seqs`
- you answer that …
  1. starts with a verb
  2. is as English as possible
  3. only uses common abbreviations

2.14 A good function has a clear name

- F.1: "Package" meaningful operations as carefully named functions
- Use verbs, strive for names that are concise and meaningful

2.15 A function has a clear interface 1/3

Comment on this function from Pythonpool:

```
i=2

def Prime(no, i):
    if no == i:
        return True
    elif no % i == 0:
        return False
    return Prime(no, i + 1)
```

. . .

Function names start with lowercase character, name does not start with a verb, input is not checked, clumsy interface:

```
assert Prime(2, 2)
assert Prime(3, 2)
assert Prime(3, 3) # Nothing stops me!
assert not Prime(4, 2)
assert Prime(5, 2)
```

The classic on refactoring is (1).

2.16 A function has a clear interface 2/3

Comment on this function again:

```
def is_prime(no, i = 2):
    assert isinstance(no, int)
    assert isinstance(i, int)
    if no == i:
        return True
    elif no % i == 0:
        return False
    return is_prime(no, i + 1)
```

. . .

- Clumsy interface:

```
assert is_prime(2)
assert is_prime(2, 2) # Nothing stops me!
assert is_prime(3)
```

```python
assert not is_prime(4)
assert is_prime(5)
```

2.17 A function has a clear interface 3/3

Comment on this function again:

```python
def is_prime(no):
    if not isinstance(no, int):
        raise TypeError("'no' must be integer")
    return is_prime_internal(no)

def is_prime_internal(no, i = 2):
    assert isinstance(no, int)
    assert isinstance(i, int)
    if no == i:
        return True
    elif no % i == 0:
        return False
    return is_prime_internal(no, i + 1)

assert is_prime(2)
assert is_prime(3)
assert not is_prime(4)
assert is_prime(5)
```

. . .

I think it is OK, please correct me :-)

2.18 A function does one thing correctly

F.2: A function should perform a single logical operation, hence don't:

```python
def do_x_and_y(): pass

do_x_and_y()
```

Do:

```
def do_x(): pass

def do_y(): pass

do_x()
do_y()
```

You rarely need `and` in a function name. Possible exception: mean and standard deviation

2.19 What is a good function?

A good function:

- **Has a clear name**
- **Does one thing correctly**
- Is tested
- Gives clear error messages
- Is documented
- Fast iff needed

2.20 A good function is tested

- F.2: A function should perform a single logical operation: A function that performs a single operation is simpler to understand, test, and reuse.
- Joint Strike Fighter Coding Standards, section 3: Testability: Source code should be written to facilitate testability

2.21 Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

The function `align_dna_seqs` aligns two DNA sequences to this:

```
AAACCCGGGTTT
ATACC-GGGTTT
```

Which tests would you write?

2.22 Solutions 1

```
assert align_dna_seqs(
  "AAACCCGGGTTT",
  "ATACCCGGGTAT"
  ) == {
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  }
assert align_dna_seqs(
  {
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  }
) ==
  {
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  }
```

```
expect_equal(
  align_dna_seqs(
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  ),
  c(
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  )
)
expect_equal(
  align_dna_seqs(
    c(
      "AAACCCGGGTTT",
      "ATACCCGGGTAT"
    )
  ),
  c(
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  )
)
```

2.23 What is a good function?

A good function:

- **Has a clear name**
- **Does one thing correctly**
- **Is tested**
- Gives clear error messages
- Is documented
- Fast iff needed

2.24 The role of `assert` within functions

Within functions, `assert` is used for:

- as a stub
- to do things in debug mode only
- to document assumptions a developer makes

2.25 `assert` differs between debug and release

```
$ cat assert.py
assert 1 == 2

$ python -O assert.py

$ python assert.py
Traceback (most recent call last):
  File "/home/richel/assert.py", line 1, in <module>
    assert 1 == 2
AssertionError
```

2.26 `assert` as a stub

```python
def align(dna_sequences):
    """Align the DNA sequences"""
    assert len(dna_sequences) == 2 # TODO
```

2.27 `assert` in debug mode

```python
def align_two_dna_sequences(dna_sequences):
    """Internal function to align two DNA sequences"""
    assert len(dna_sequences) == 2 # TODO
```

Superior to documentation, as it cannot be ignored.

'Assert liberally to document internal assumptions and invariants' (2) chapter 68.

2.28 `assert` to document assumptions a developer makes

```python
def align_two_dna_sequences(dna_sequences):
    """Align the DNA sequences"""
    # ....
    results = ["AAAA", "AAC-"] # Should be result of calculation
    assert len(results[1]) == len(results[2])
```

2.29 Recursive algorithms

- Iterative: use a for-loop
- Recursive: a function that calls itself

2.30 Example 1: factorial

`n`	`n!`
0	1
1	1
2	2 * 1 = 2

n	n!
3	3 * 2 * 1 = 6
4	4 * 3 * 2 * 1 = 24
5	5 * 4!
n	n * (n-1)!

2.31 Exercise 1: factorial

- Develop a function to get the factorial of a number
- If the function used a for-loop, create another function that uses recursion, or the other way around

```
assert calc_factorial_iterative(13) ==
  calc_factorial_recursive(13)
```

```
expect_equal(
  calc_factorial_iterative(13),
  calc_factorial_recursive(13)
)
```

2.32 Example 2

Fibonacci sequence:

N	0	1	2	3	4	5	6	7	8	9	10
Fn	0	1	1	2	3	5	8	13	21	34	55

2.33 Example 2

N	Fn
0	0
1	1
2	1
3	Fn(1) + Fn(2)
n	Fn(n - 2) + Fn(n - 1)

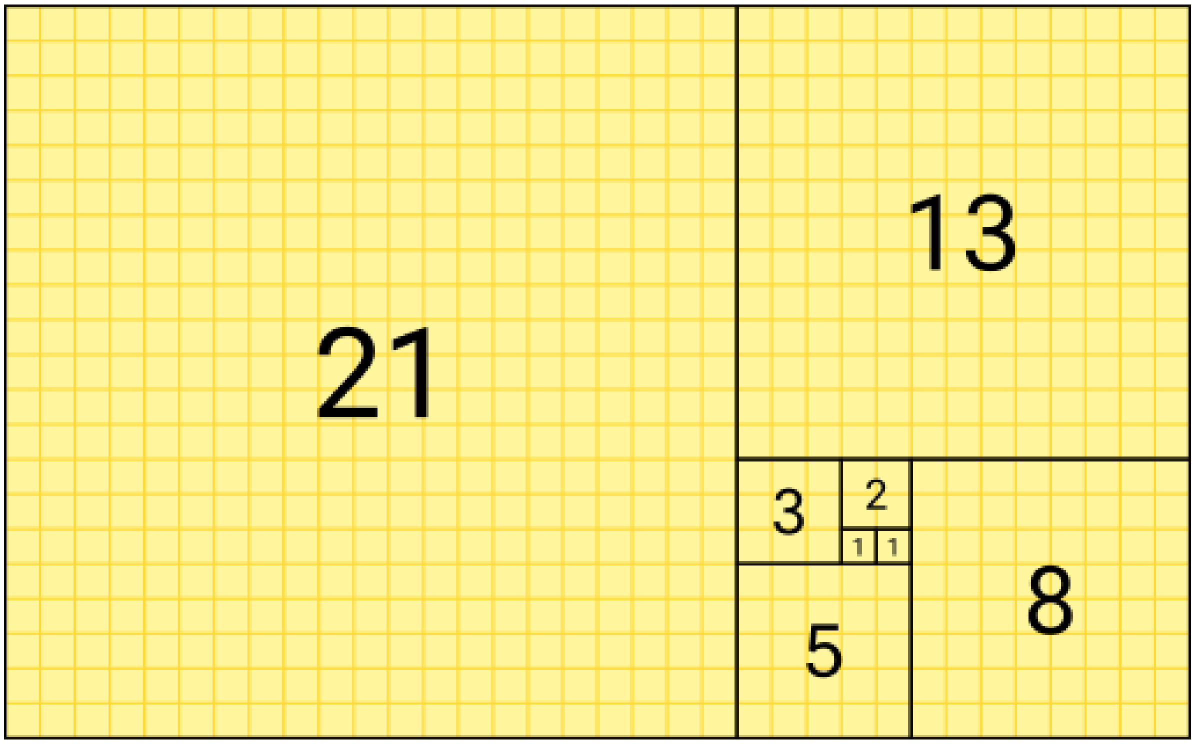


Figure 2: https://en.wikipedia.org/wiki/File:Fibonacci_Squares.svg

2.34 Exercise

- Develop a function to get the nth value in the Fibonacci sequence
- If the function used a for-loop, create another function that uses recursion, or vice versa

```
assert get_nth_fibonacci_iterative(13) ==
  get_nth_fibonacci_recursive(13)
```

```
expect_equal(
  get_nth_fibonacci_iterative(13),
  get_nth_fibonacci_recursive(13)
)
```

2.35 Recap

- Function design is hard
- Documentation helps
- TDD helps
- assert helps in debug mode only

2.36 Appendix

2.37 Links

- Course materials of 2022.

1. Fowler M. Refactoring. Addison-Wesley Professional; 2018.

2. Sutter H, Alexandrescu A. C++ coding standards: 101 rules, guidelines, and best practices. Pearson Education; 2004.