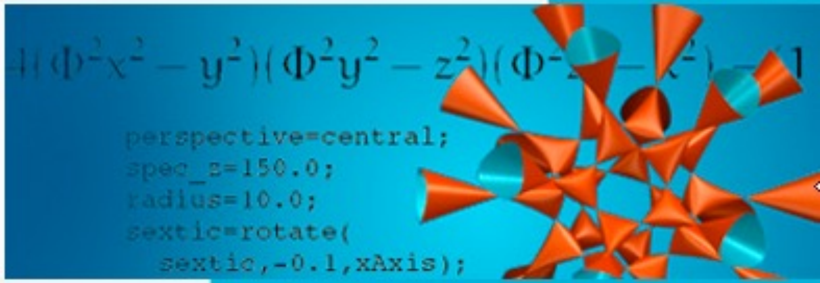
Philipps Universität Marburg

Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



```
perspective=central;
spec_z=150.0;
radius=10.0;
sextic=rotate(
   sextic,-0.1,xAxis);
```

[Script 3.3 - 3.4]

Draft program - how does it work?

- Recognize from:
  - What is relevant?
  - What data can be entered?
  - What data should be output?
- Is a functionality already available?
- Design recipes
  - Proven procedures

Recipe: Testing

- Random checks for correctness
- Calls of defined function and comparison with expected result
- Defining tests together with function definition

Automate tests

- Define tests persistently (in file)
- Evaluation of printout and comparison of results should not require any manual steps
- Then
  - Tests are not lost
  - Tests can be repeated
- Repeat all tests after each code change
  - Ensure that no errors have been introduced

Automate tests

- What happens if the test is successful/failed?
- How do you write down tests?
- Example: (+ 2 3) must add up to 5
  - (check-expect (+ 2 3) 5)

Function in BSL: check-expect
Behavior: Success message on console
or error message in dialog.

Tests

- Conversion Fahrenheit to Celsius

```
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

(define (f2c f)
        (* 5/9 (- f 32)))
```

Fractions are literals in Racket.

Do you notice anything?

Tests may precede function definitions. "check-*" are special cases.

- All checks are executed when Start is pressed

Best Practices

- Write tests before implementation
  - Focus on specification
  - No influence due to programming errors

- Also write tests after implementation
  - Addressing marginal cases
  - For example, are all cases of cond expression tested?

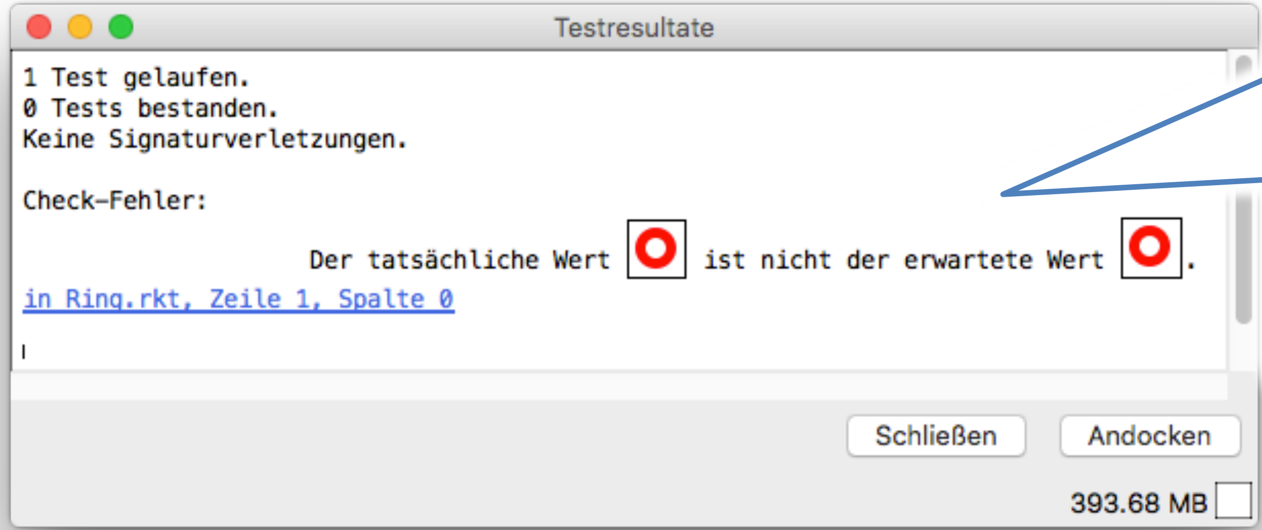
- Other comparisons: check-within, check-range, etc.

Best Practices

- Is this test successful?

(check-expect (ring 5 10 "red"))

```
(define (ring innerradius outerradius color)
        (overlay (circle inner radius "solid" "white")
                 (circle outerradius "solid" color)))
```



```
1 Test gelaufen.
0 Tests bestanden.
Keine Signaturverletzungen.

Check-Fehler:
                    Der tatsächliche Wert ⭘ ist nicht der erwartete Wert ⭘.
in Ring.rkt, Zeile 1, Spalte 0
```

Question of the representation of values/ implementation details

Best Practices

- Checking the properties of results

```
(check-expect (image-width (ring 5 10 "red")) 20)
```

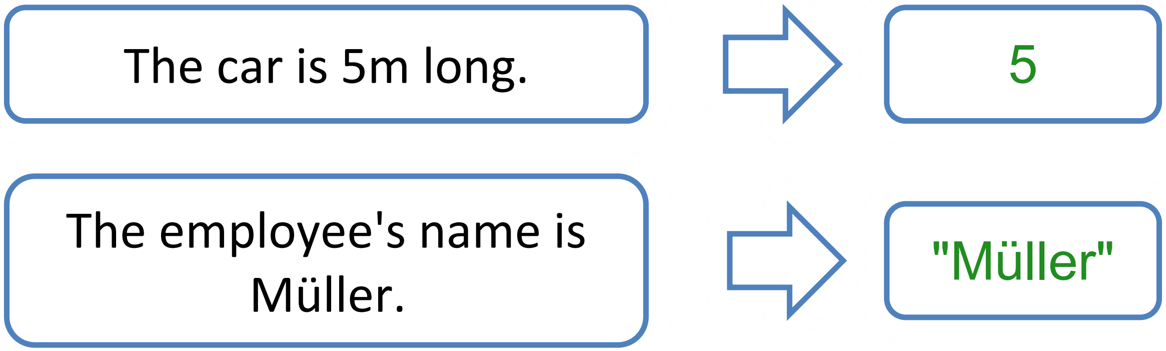
```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels from the left
(check-expect (add-image 5 "hello" (empty-scene 100 100))
   (place-image (text "hello" 10 "red") 10 5 (empty-scene 100 100)))

(define (add-image y s img)
   (place-image (text s 10 "red") 10 y img))
```

Expected value corresponds to partially reduced function call. **That is a coincidence!** Implementation of add-image not determined by this.

Recipe: Information and data

- Program
  - Processing of information
  - Production of information
- "Information" = knowledge + meaning
  - Knowledge and meaning not directly accessible to computers
- Representation through data/values



The car is 5m long. → 5

The employee's name is Müller. → "Müller"

Values

- You can't see the importance of a value

- What is the meaning of 5?
  - Length of a car?
  - Price of the meal?
  - Grade of the thesis?
  - Current temperature?

Data definition

- Class of data
  - Data type: Set of values that can be interpreted in the same way

- Name
  - Indicates the interpretation

- Examples

```
; Distance is a Number.
; interp. the number of pixels from the top margin of a canvas

; Temperature is a Number.
; interp. degrees Celsius
```

Data definition

- Previously: Numbers, strings, images, truth values
- Representation of information based on existing data types

```
; Temperature is a Number.
; interp. degrees Celsius
Examples:
(define sunny-weather 25)
(define bloody-cold -5)
```

Through examples: Ensure that values are representable

Can be used in tests

Recipe: Function definition

- Sequence of steps for the design of a function

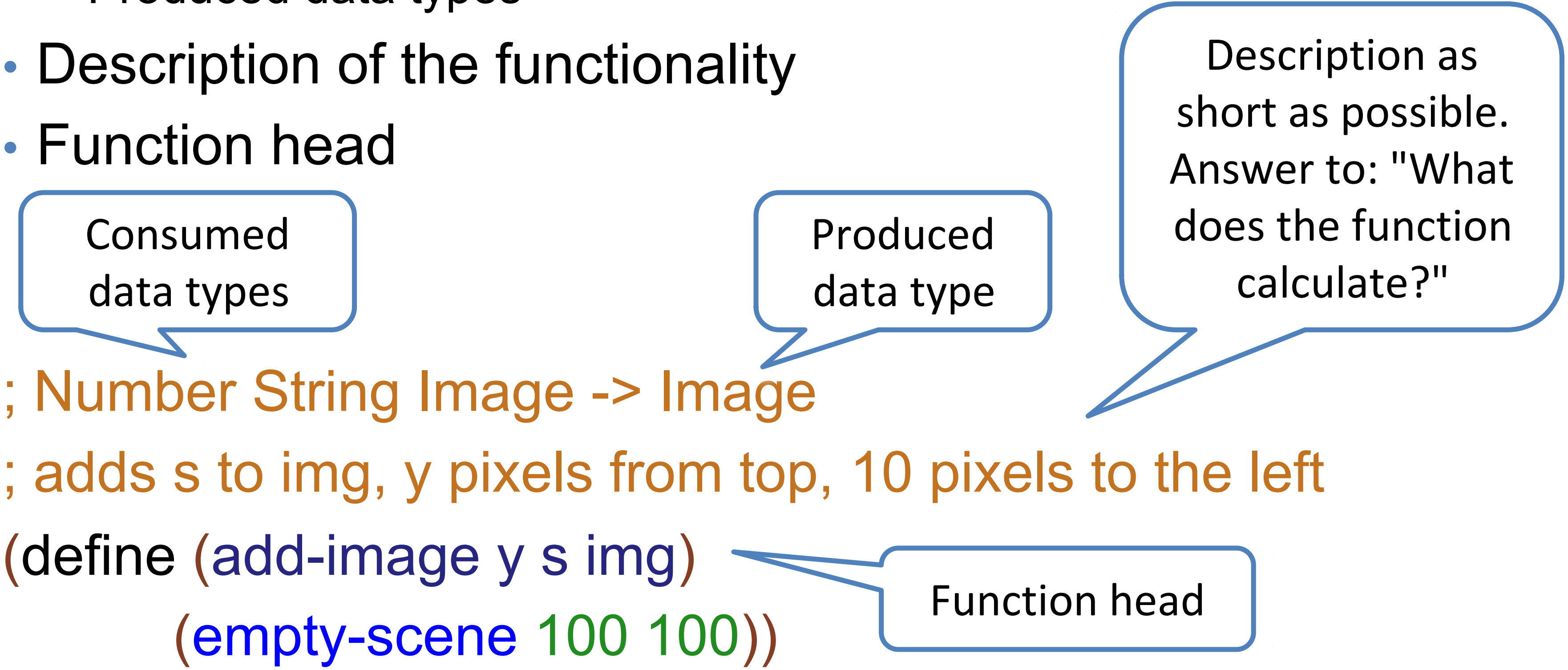
1. Information representation
2. Signature
3. Tests
4. Divide main function into sub-functions
5. Implement function body
6. Execute tests
7. Post-processing

Information representation

- What information is relevant?
- Input
- Issue
- Data definition including examples

Signature

- In BSL: Comment
  - Consumed data types
  - Produced data types
- Description of the functionality
- Function head



Consumed data types

Produced data type

Description as short as possible. Answer to: "What does the function calculate?"

```
; Number String Image -> Image
; adds s to img, y pixels from top, 10 pixels to the left
(define (add-image y s img)
        (empty-scene 100 100))
```

Function head

Signature

- Function head
  - Function definition with define: "Header"
  - Body expression produces dummy value: "Stub"

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(define (add-image y s img)
        (empty-scene 100 100))
```

Signature

- Functional head
  - Function definition with define: "Header"
  - Body expression produces dummy value: "Stub"

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(define (add-image y s img)
        (empty-scene 100 100))
```

Use parameter names in description

Function name

Name of the input parameters

Signature

- Function head
  - Function definition with define: "Header"
  - Body expression produces dummy value: "Stub"

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(define (add-image y s img)
        (empty-scene 100 100))
```

Stub does superficially correct: no syntax error, no runtime error

Tests

- Between task description and header
- By means of check-*
- Part of the documentation:
  Expected behavior
- Automatic execution:
  If successful, also assurance of the behavior documented by tests

Tests First Principle

```
Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) 0)
```

- Program execution now fails:
  Returned dummy value does not correspond to documented behavior

Divide main function into sub-functions

- What can be used to implement the function?
  - Input data
  - Auxiliary functions
- Replace the dummy body with "Template"

```
Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) ... len ...)
```

Template

Implement function body

- Replace template with expression, that fulfills the specification
- Specification
  - Signature
  - Job description
  - Tests

```
Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) (* len len))
```

Execute tests

- Automated tests: Click on Start
- Test successful: finished
- Test fails
  - Test case defined incorrectly?
  - Function implementation faulty?
  - Repair until tests successful

Really?

It is also possible that the tests are too weak. It is therefore always necessary to check whether the tests are sufficient.

Post-processing

- Review
  - Signature correct?
  - Task description correct?
  - Do the signature and task description match the implementation?

- Test coverage
  - Tests are successful, but ...
  - ... is everything tested?
  - Is all code executed during tests?
  - Are all edge cases tested?

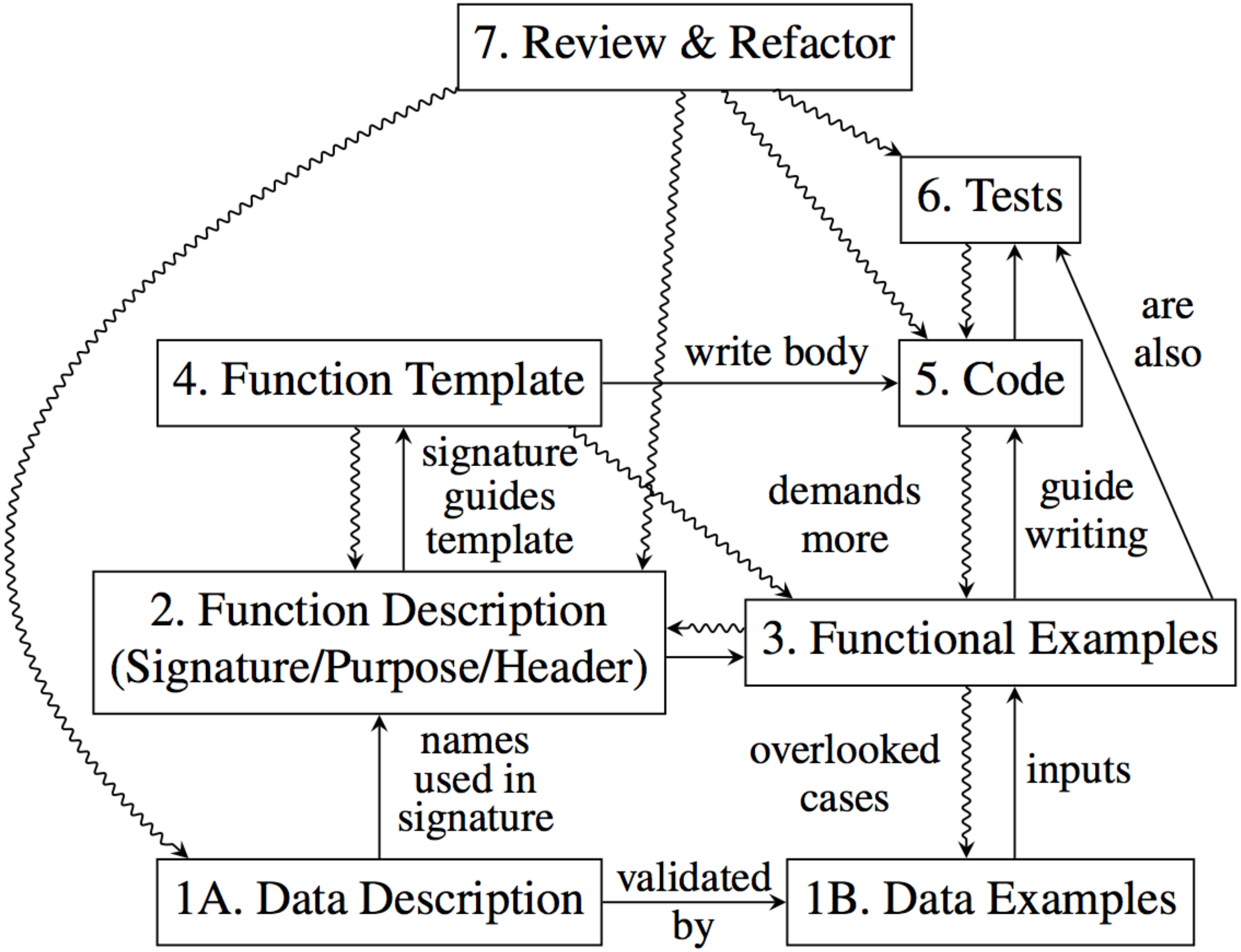
Post-processing

- Does implementation correspond to the template?

- Improving the structure: "Refactoring"
  - Delete function and constant definitions that are no longer used
  - Search for redundancies and replace them with function and constant definitions
  - Simplifying conditional expressions

- Re-run tests to ensure that functionality has not been changed
  - **Modification of functionality must be done separately from refactorings!**

Design recipe



Recipe: Programs with many functions

- Apply design recipe for each function
- Use functions and constants defined in the function template
- Typical: Top-Down
  - Based on the main function
  - Division into auxiliary functions
  - Create a "wish list"
    - Header: Signature, task description, function name
    - Work through one after the other
    - Until list empty

Stepwise Refinement

- Top-down approach is also called "stepwise refinement"
  - A large design problem is broken down into many small ones
  - Step by step
  - Until small problem can be solved concretely
- Disadvantages
  - Top-level design decisions influence bottom-level auxiliary functions
    - For example, are all the required arguments available?
  - Testing only possible at a late stage
    - Remedy: Define stub so that tests are successful

Test stub

```
Number -> Number
; computes the area of a cube with side length len
(check-expect (area-of-cube 3) 54)
(define (area-of-cube len) (* 6 (area-of-square len)))

Number -> Number
; computes the area of a square with side length len
(check-expect (area-of-square 3) 9)
(define (area-of-square len) ( if (= len 3) 9 (error "not yet implemented")))
```

Test stub

```
Number -> Number
; computes the area of a cube with side length len
(check-expect (area-of-cube 3) 54)
(define (area-of-cube len) (* 6 (area-of-square len)))

Number -> Number
; computes the area of a square wit          length len
(check-expect (area-of-square 3) 9)
(define (area-of-square len) ( if (= len 3) 9 (error "not yet
imp
```

Returns value such that test of area-of-cube is successful.

Protection in the event that a function is called from another context.

Force runtime error

Information Hiding

- Design in layers and stepwise refinement
  - Replace implementation with function call
  - Program becomes easier to understand

- Further advantages
  - Functions can be maintained independently
  - Reuse of functions

"Information hiding
or "secret principle"

- Basis
  - Caller only has knowledge of the specification:
  - Signature, task description, tests
  - Implementation is hidden

Information Hiding

```
(string-append
   (body "Tillman" "Rendel")
   "your GNB account manager")
```

Is this program correct?

```
String String -> String
; generates the pretense of tax refund for the victim fst last
(check-range (string-length (body "Tillman" "Rendel")) 50 300)

(defi    ody fst lst) "" )
```

Implementation may change. Must only adhere to the specification.

You can rely on documented properties.