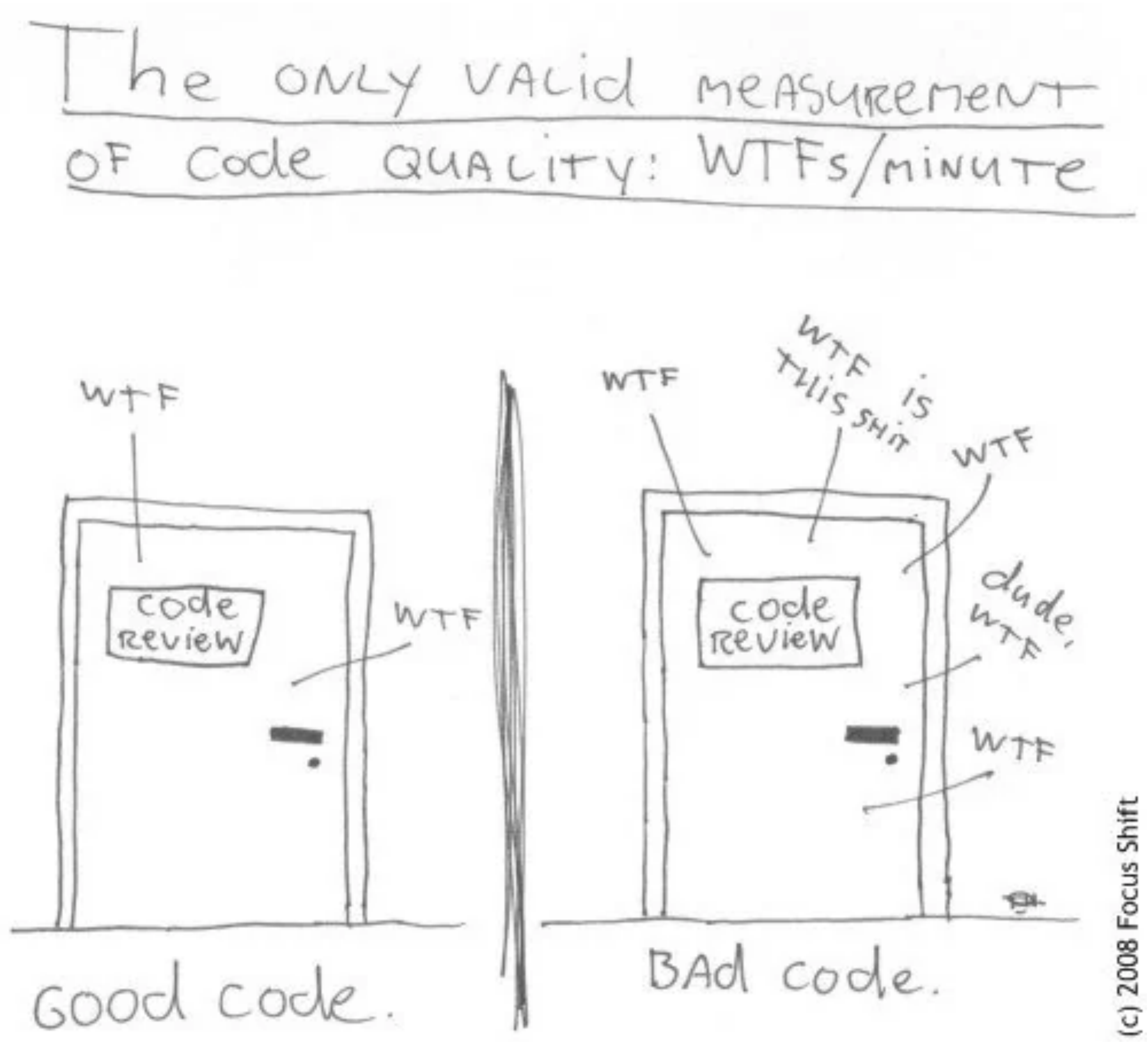
COL1000: Introduction to Programming

Specifications & Debugging

Subodh Sharma | Lec 26 | Oct 17

Specifications



Specifications

Specifications

- States what the code **must** do (and not how).

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s’, s.t. s’[I] = s[(i-r)mod n]**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s', s.t. s'[I] = s[(i-r)mod n]**
  - They must be **observable**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s’, s.t. s’[I] = s[(i-r)mod n]**
  - They must be **observable**
    - **Bad: Returns None and updates an internal var so the system reads s as rotated**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s', s.t. s'[I] = s[(i-r)mod n]**
  - They must be **observable**
    - **Bad: Returns None and updates an internal var so the system reads s as rotated**
    - **Good: len(s') = len(s); rotated right by k positions modulo len(s); No mutation of s; raise exception if s is not str**

Specifications (Cont.)

Specifications (Cont.)

- States what the code **must** do (and not how).

Specifications (Cont.)

- States what the code **must** do (and not how).
  - They must be **verifiable/checkable**

Specifications (Cont.)

- States what the code **must** do (and not how).
  - They must be **verifiable/checkable**
    - **Bad:** Should be **efficient** and should **feel like** the original string but rotated.

Specifications (Cont.)

- States what the code **must** do (and not how).
  - They must be **verifiable/checkable**
    - **Bad:** Should be **efficient** and should **feel like** the original string but rotated.
    - **Good: rotate(rotate(s,k), -k) = s;**

Specifications (Cont.)

- States what the code **must** do (and not how).
    - They must be **verifiable/checkable**
        - **Bad:** Should be **efficient** and should **feel like** the original string but rotated.
        - **Good: rotate(rotate(s,k), -k) = s;**
    - They must be **complete and minimal**

Specifications (Cont.)

- States what the code **must** do (and not how).
  - They must be **verifiable/checkable**
    - **Bad:** Should be **efficient** and should **feel like** the original string but rotated.
    - **Good: rotate(rotate(s,k), -k) = s;**
  - They must be **complete and minimal**
    - **Bad: rotated string is of the same length (not complete); must use exactly one modulo (not minimal)**

Specifications (Cont.)

- States what the code **must** do (and not how).
  - They must be **verifiable/checkable**
    - **Bad:** Should be **efficient** and should **feel like** the original string but rotated.
    - **Good: rotate(rotate(s,k), -k) = s;**
  - They must be **complete and minimal**
    - **Bad: rotated string is of the same length (not complete); must use exactly one modulo (not minimal)**
    - **Good: No I/O, no i/p mutation; length match, character multiset preserved; input validations - s:str, k:int**

Methods to Capture Specifications

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed
- **Preconditions**: Conditions that must be true about the inputs for a function

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed
- **Preconditions**: Conditions that must be true about the inputs for a function
- **Postconditions:** Conditions that must be true about the outputs of a function

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed
- **Preconditions**: Conditions that must be true about the inputs for a function
- **Postconditions:** Conditions that must be true about the outputs of a function
- **State Invariants, Exceptions:** True at some control-location in the program

Methods to Capture Specifications

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed

```python
def add_items(a: int, b: int) -> int: #Input/Output spec through typing hints
    """
    This function is ANNOTATED to only accept and return integers.
    """
    return a + b

# --- Correct usage (according to hints) ---
print(f"Correct call: {add_items(5, 10)}")

# --- Incorrect usage (according to hints) ---
# This call will RUN perfectly, but mypy will flag it.
print(f"Incorrect call: {add_items('hello', ' world')}")
```

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed

```
def add_items(a: int, b: int) -> int: #Input/Output spec through typing hints
    """
    This function is ANNOTATED to only accept and return integers.
    """
    return a + b

# --- Correct usage (according to hints) ---
print(f"Correct call: {add_items(5, 10)}")

# --- Incorrect usage (according to hints) ---
# This call will RUN perfectly, but mypy will flag it.
print(f"Incorrect call: {add_items('hello', ' world')}")
```

```
bash-3.2$ python3 ./lec26.py
Correct call: 15
Incorrect call: hello world
bash-3.2$
```

Methods to Capture Specifications

- **Typing hints, Function Signatures** — Function return values and parameters must be typed

```
def add_items(a: int, b: int) -> int: #Input/Output spec through typing hints
    """
    This function is ANNOTATED to only accept and return integers.
    """
    return a + b

# --- Correct usage (according to hints) ---
print(f"Correct call: {add_items(5, 10)}")
```

```
bash-3.2$ python3 ./lec26.py
Correct call: 15
Incorrect call: hello world
bash-3.2$
```

```
bash-3.2$ mypy lec26.py
lec26.py:31: error: Argument 1 to "add_items" has incompatible type "str"; expected "int"  [arg-type]
lec26.py:31: error: Argument 2 to "add_items" has incompatible type "str"; expected "int"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

Methods to Capture Specifications

Methods to Capture Specifications

- PreConditions & PostConditions

Methods to Capture Specifications

- **PreConditions & PostConditions**

```python
def average(nums: list[float]) -> float:

    #Preconditions
    assert isinstance(nums, list), "nums must be a list"
    assert all(isinstance(x, (int, float)) for x in nums), "nums must contain numbers"
    assert len(nums) > 0, "nums must be non-empty"

    result = sum(nums) / len(nums)

    # Postcondition
    assert abs(result * len(nums) - sum(nums)) < 1e-9, "Postcondition failed"

    return result
```

Unit Testing from Pre/Post Conditions

Unit Testing from Pre/Post Conditions

- Specification based testing

Unit Testing from Pre/Post Conditions

- **Specification based testing**

```
def rotate(s: str, k: int) -> str:

    if not s:
        return s
    k %= len(s)
    return s[-k:] + s[:-k]
```

my_rotate.py

Unit Testing from Pre/Post Conditions

- **Specification based testing**

```python
def rotate(s: str, k: int) -> str:

    if not s:
        return s
    k %= len(s)
    return s[-k:] + s[:-k]
```

my_rotate.py

```python
import pytest
from my_rotate import rotate

def test_length_preserved():
    assert len(rotate("hello", 7)) == 5

def test_identity_when_k_multiple_of_n():
    s = "spoon"
    for k in (0, 5, 10, -5):
        assert rotate(s, k) == s

def test_composition_inverses():
    s = "abcdef"
    k = 2
    assert rotate(rotate(s, k), len(s)-k) == s

def test_negative_k():
    assert rotate("abcd", -1) == "bcda"
```

test_rotate.py

Unit Testing from Pre/Post Conditions

- **Specification based testing**

```python
def rotate(s: str, k: int) -> str:

    if not s:
        return s
    k %= len(s)
```

```python
import pytest
from my_rotate import rotate

def test_length_preserved():
    assert len(rotate("hello", 7)) == 5

def test_identity_when_k_multiple_of_n():
    s = "spoon"
    for k in (0, 5, 10, -5):
        assert rotate(s, k) == s

def test_composition_inverses():
```

```
bash-3.2$ pytest
============================================ test session starts ============================================\
=
platform darwin -- Python 3.11.6, pytest-8.4.2, pluggy-1.6.0
rootdir: /Users/svs/svs-research/classes-IITD/col100-intro-to-compscience/2025/lectures/lec26
collected 4 items

test_rotate.py ....                                                                                    [100%]

============================================= 4 passed in 0.01s =============================================\
```

Conclusions: Specifications

Conclusions: Specifications

- For complex problems, specification and solution go hand in hand
  - Problems are divided into parts, and each part is separately specified
  - Usually the solution of one part becomes the given input for another part

Conclusions: Specifications

- For complex problems, specification and solution go hand in hand
  - Problems are divided into parts, and each part is separately specified
  - Usually the solution of one part becomes the given input for another part
- Knowing the nature of problem can help specify problems and guide solutions
  - Decision Problems (Is there any solution with the given property?)
  - Search Problems (Find one or more solutions among all <u>candidates</u> with the property)
  - Counting Problems (number of candidates with the property)
  - Optimization Problems (best solution satisfying the property in terms of some <u>metric</u>) .

Conclusions: Specifications

Conclusions: Specifications

- Ensure that the input and output are clear: constraints the **input data type, size, or range**

Conclusions: Specifications

- Ensure that the input and output are clear: constraints the **input data type, size, or range**
- Establish that output for every possible input can be verified to meet the specification
    - If multiple correct solutions for the same input are possible, see if all must be produced

Conclusions: Specifications

- Ensure that the input and output are clear: constraints the **input data type, size, or range**
- Establish that output for every possible input can be verified to meet the specification
  - If multiple correct solutions for the same input are possible, see if all must be produced
- Model the problem in mathematical terms, with numbers and symbols
  - Use only standard notation and commonly understood axioms