



Technische
Universität
Braunschweig



Clean Code and Refactoring

Sven Marcus, 11.07.22

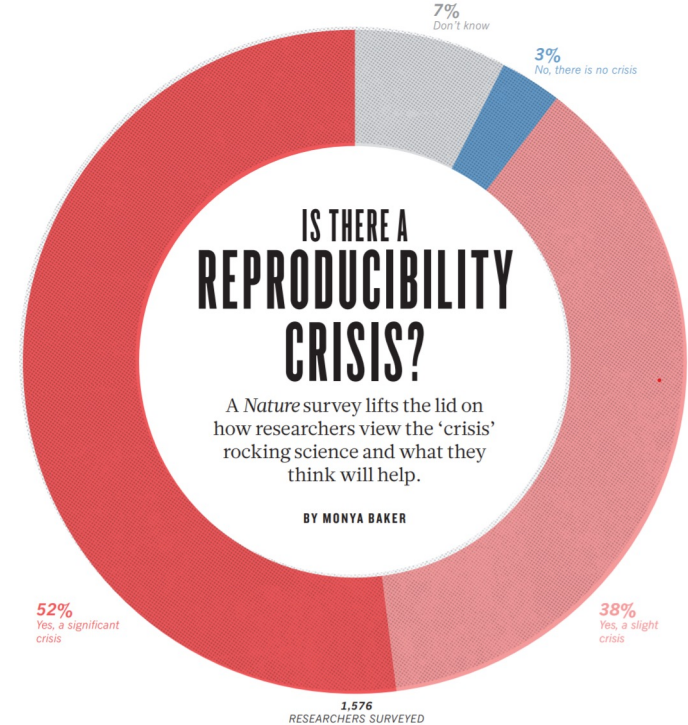
Content

- Motivation
- Clean code and Refactoring
- Python tips and tricks
- Let's get to work:
improving the readability and structure of an existing heat transfer simulation

Motivation

Motivation

- 84% of scientists say that developing software is essential to their research. [1]
- Incorrect publications have led to a reproducibility and credibility crisis. [2, 3]



[1] Jo Hannay et al. "How Do Scientists Develop and Use Scientific Software?"

[2] Monya Baker. "1,500 scientists lift the lid on reproducibility".

[3] Zeeya Merali. "Computational science: Error, why scientific programming does not compute".

Motivation

Researchers

- often lack knowledge about principles and practices of the software engineering discipline [3, 4]
- don't gain reputation for developing software
- are pressed to publish results as fast as possible [5]

[4] Lucas Joppa et al. "Troubling Trends in Scientific Software Use".

[5] Mark De Rond and Alan N Miller. "Publish or perish: bane or boon of academic life?"

Consequences

Common problems of research software

- Low code quality
- Hard to understand
- Neither published nor documented

Broken Window Effect



[6] George L Kelling, James Q Wilson et al. "Broken windows".

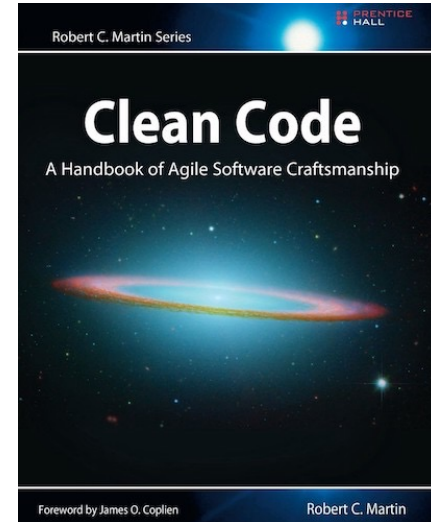
Motivation

“Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write.” – Robert C. Martin

Clean Code and Refactoring

What is clean code?

- Term coined by Robert C. Martin
- No strict rules, but a set of principles to make code easy to understand, extend and adapt



Clean Code

“Clean code always looks like it was written by someone who cares.”

– Michael Feathers

“Clean code reads like well-written prose.”

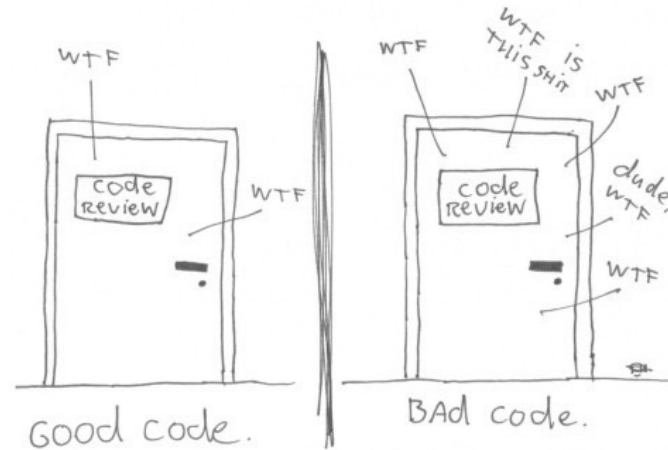
– Grady Booch

“Clean code can be read, and enhanced by a developer other than its original author.”

– Dave Thomas

Clean Code

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

Refactoring

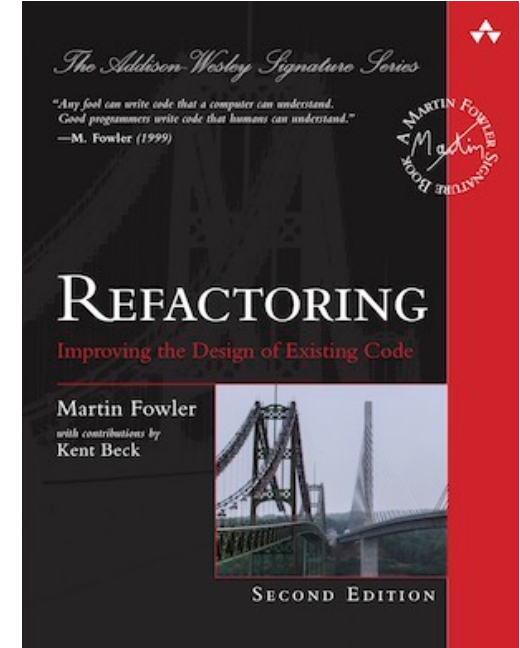
noun: *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*

verb: *to restructure software by applying a series of refactorings without changing its observable behavior.*

<https://refactoring.com/>

Refactoring

- Focus on improving the readability and structure of existing code
- Assigns clear names to the code changes
- Shares many ideas with Clean Code



Refactoring

*“Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.”*

“The point of refactoring isn’t to show how sparkly a code base is - it is purely economic. We refactor because it makes us faster - faster to add features, faster to fix bugs.”

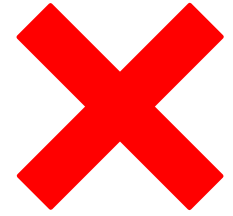
– Martin Fowler

Names

Use meaningful names

- Classes, variables, functions, modules, binaries, libraries...
- The name should describe the **purpose**

Names



```
1 var = 7 # num days in week
```

Names



```
1 number_of_days_in_week = 7
```

Name of the Refactoring:
Rename Variable

Names



```
1 def copy_chars(a1, a2):  
2     for i in range(0, len(a1)):  
3         a2[i] = a1[i]
```

Names

Even if the function is simple,
it can be hard to understand what it does for a user



```
(a1: Any, a2: Any) → None  
copy_chars()
```

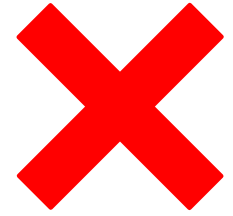
Names

Make meaningful distinctions to make the code understandable



```
1 def copy_chars(source, destination):  
2     for i in range(0, len(destination)):  
3         destination[i] = source[i]
```

Names



```
1 def genymdhms(): # generate date, year, month, day, minutes
```

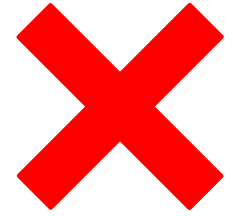
Names

- Use names that are pronounceable and searchable
- Avoid encodings (unless they are commonly known)



```
1 def generate_timestamp():
```

Names



```
1 def do_the_next_thing():
```


Names



Use domain names

```
1 def apply_neumann_boundary_condition():
```

Name of the Refactoring:
Change function declaration, Rename function

Comments

„Don't comment bad code - rewrite it.“

- Brian W. Kernighan and P.J. Plaugher

*„The proper use of comments is to compensate for our failure to express ourself in code.
Note that I used the word failure. I meant it. Comments are always failures.“*

- Robert C. Martin

Comments

If a comment must be used, it should describe the ***why*** and not the ***what*** or ***how***

Comments



```
1  # check if i is a prime number
2  prime = True
3  for j in range(i):
4      if i % j == 0:
5          prime = False
6          break
```

Comments



Express your intention in code instead of comments


```
1  def is_prime(number: int) → bool:
2      for divisor in range(number):
3          if number % divisor == 0:
4              return False
5
6      return True
```

Comments



```
1 class Container:
2
3     def __init__(self, logger) → None:
4         # The logger associated with this container
5         self.logger = logger
```

Comments



```
1 class MyClass:
2     # ----- CONSTRUCTOR -----
3     def __init__(self):
4         self.value = 0
5
6     # ----- SETTER -----
7     def set_value(self, new_value):
8         self.value = new_value
```

Comments



Don't require the use of doc comments in private functions if they don't add value



Use doc comments in public APIs

```
1 def _is_sad(maybe_sad: str) → bool:
2     """Checks if string is sad
3
4     Args:
5         maybe_sad(str): a string
6
7     Returns:
8         bool
9     """
10    return ":((" in maybe_sad or "T_T" in maybe_sad
11
12
13 def make_happy(sad_string: str) → str:
14     """Makes a sad string happy
15
16     Args:
17         sad_string(str): A string with sad smileys
18
19     Returns:
20         str
21     """
22     if not _is_sad(sad_string):
23         return sad_string
24
25     return sad_string.replace(":((", ":)").replace("T_T", "^_^")
```

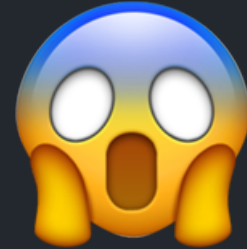

Comments

```
1 def load_config(parser: ConfigParser, config_path: str) → list[str]:  
2     """  
3     Loads configuration from a config file.  
4     Falls back to the defaults file on fail.  
5     """  
6     config = parser.read(config_path)  
7     if not config:  
8         config = parser.read(".defaults")  
9  
10    return config
```



Comments

```
1 def load_config(parser: ConfigParser, config_path: str) → list[str]:  
2     """  
3     Loads configuration from a config file.  
4     Falls back to the defaults file on fail.  
5     """  
6     config = parser.read(config_path)  
7     if not config:  
8         config = parser.read(".defaults")  
9  
10    return config
```



LIES!!!

Good Comments

```
1  # We're doing this, because of ...           Explanation of intent
2
3  # This code could lead to a dead lock         Warning
4  # which I wasn't able to prevent so far
5
6  # TODO: pass additional parameter to implement XYZ model
7
8  # Cutting out spaces here is important, because ... Amplification
```

Functions

```
1  a = [1, 2, 3, 4]
2  with open("log_a.txt", "w") as f:
3      for num in a:
4          f.write(str(num) + "\n")
5
6  b = [1, 2, 3, 4]
7  with open("log_b.txt", "w") as f:
8      for num in b:
9          f.write(str(num) + "\n")
```



Functions

DRY – DON'T REPEAT YOURSELF!



```
1 def write_list_to_file(a_list: list[int], filename: str) → None:
2     with open(filename, "w") as f:
3         for num in a_list:
4             f.write(str(num) + "\n")
5
6 write_list_to_file(a, "log_a.txt")
7 write_list_to_file(b, "log_b.txt")
```

Name of the Refactoring: Extract Function

Functions

DRY – DON'T REPEAT YOURSELF!



Functions

Duplication is dangerous, because

- it increases the chance of making mistakes.
- when requirements change, all duplicated code sections have to be adjusted. It becomes easy to forget one of the implementations.

“Duplication may be the root of all evil in software.”

– Robert C. Martin

Challenge:
Spot the duplication!

Functions

```
1  def multiply_elements():
2      my_list = [1, 2, 3, 4, 5, 6]
3      product = 1
4      for element in my_list:
5          product = product * element
6
7      return product
8
9
10 def sum_elements():
11     some_list = [4, 6, 2, 7, 8, 1]
12     total = 0
13     for value in some_list:
14         total = total + value
15
16     return total
```

Functions

Duplications!

Differences

```
1  def multiply_elements():
2      my_list = [1, 2, 3, 4, 5, 6]
3      product = 1
4      for element in my_list:
5          product = product * element
6
7      return product
8
9
10 def sum_elements():
11     some_list = [4, 6, 2, 7, 8, 1]
12     total = 0
13     for value in some_list:
14         total = total + value
15
16     return total
```

Functions

Extract the duplications and parameterize the differences!

Functions

```
1 BinaryOperator = Callable[[Number, Number], Number]
2
3
4 def accumulate(
5     iterable: Iterable[Number],
6     operator: BinaryOperator,
7     initial: Number,
8 ) → Number:
9     for element in iterable:
10         initial = operator(initial, element)
11
12     return initial
```

Functions

```
1 def multiply(a, b):  
2     return a * b  
3  
4  
5  
6 def add(a, b):  
7     return a + b  
8
```

```
1 def multiply_elements():  
2     my_list = [1, 2, 3, 4, 5, 6]  
3     return accumulate(my_list, multiply, 1)  
4  
5  
6 def sum_elements():  
7     some_list = [4, 6, 2, 7, 8, 1]  
8     return accumulate(some_list, add, 0)
```

Functions



```
1 @dataclass
2 class TemperatureData:
3     datetime: datetime
4     value_in_kelvin: float
5
6 def write_celsius_values_in_timeframe_to_log(data: list[TemperatureData], start: datetime, end: datetime) → None:
7     celsius_values: list[float] = []
8     for d in data:
9         if d.datetime is not None and not math.isnan(d.value_in_kelvin):
10             if start ≤ d.datetime ≤ end:
11                 celsius_values.append(d.value_in_kelvin - 273.15)
12
13     with open("logfile.txt", "w") as f:
14         f.writelines(celsius_values)
```

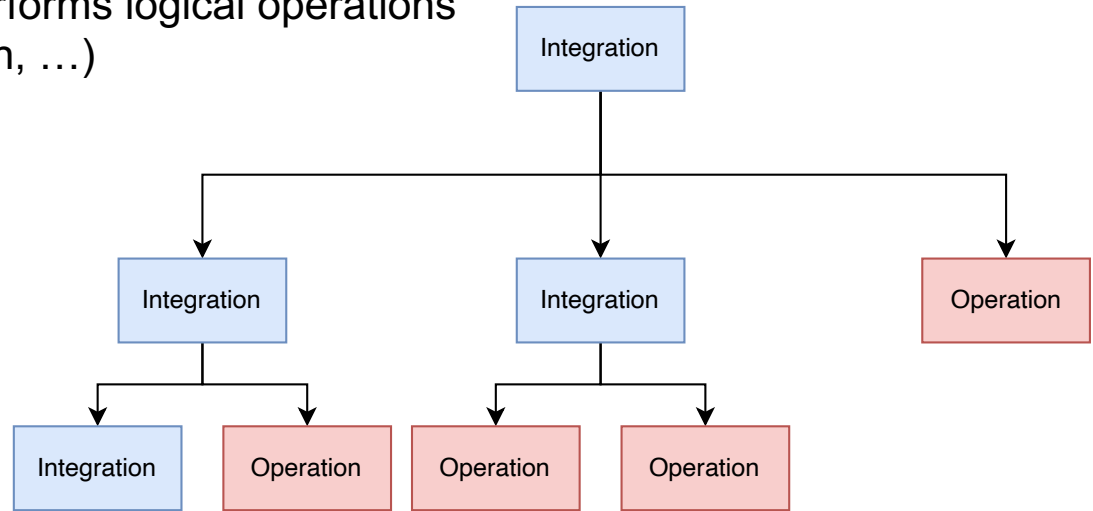
Functions

Functions should only have one task!

Functions

The Integration-Operation-Segregation-Principle (IOSP)

- Functions should perform **only integrations** or **only operations**
- An integration is a function that calls other integration or operation functions
- An operation is a function that performs logical operations (e.g. if statements, for loops, math, ...)



Functions

```
1 def write_celsius_values_in_timeframe_to_log(data: list[TemperatureData], start: datetime, end: datetime) → None:
2     filtered_data = filter(is_valid, data)
3     filtered_data = filter(in_timeframe(start, end), filtered_data)
4     celsius_values = map(value_in_celsius, filtered_data)
5     write_list_to_file(celsius_values, "logfile.txt")
6
7 def is_valid(d: TemperatureData) → bool:
8     return d.datetime is not None and not math.isnan(d.value_in_kelvin)
9
10 TimeFrameFilter = Callable[[TemperatureData], bool]
11
12 def in_timeframe(start: datetime, end: datetime) → TimeFrameFilter:
13     def _in_timeframe(d: TemperatureData) → bool:
14         return start ≤ d.datetime ≤ end
15
16     return _in_timeframe
17
18 def value_in_celsius(data: TemperatureData) → float:
19     return data.value_in_kelvin - 273.15
20
21 def write_list_to_file(a_list: list[Any], path: str) → None:
22     with open(path, "w") as f:
23         f.writelines(a_list)
```



Functions

Function Arguments

- Attempt to use no more than 3 arguments
- Use custom data structures to combine arguments
- Many arguments are an indicator that a function does too much
- Avoid boolean flags as they don't express intent

Functions



```
1 def throw_trajectory(v0, h, a, t, g=10.0): ???  
2     x = v0 * math.cos(a) * t  
3     return x, -0.5 * g / (((v0 * math.cos(a)) ** 2)) * x**2 + math.tan(a) * x + h
```

Functions

```
1 def throw_trajectory(initial_speed, initial_height, throwing_angle, time, gravity=10.0):
2     x = horizontal_position(initial_speed, throwing_angle, time)
3     y = vertical_position(x, initial_speed, initial_height, throwing_angle, gravity)
4     return (x, y)
5
6 def horizontal_position(initial_speed, throwing_angle, time):
7     return initial_speed * math.cos(throwing_angle) * time
8
9 def vertical_position(
10     horizontal_position, initial_speed, initial_height, throwing_angle, gravity
11 ):
12     a, b = coefficients_of_throw_function(initial_speed, throwing_angle, gravity)
13     return quadratic_function(a, b, initial_height, horizontal_position)
14
15 def quadratic_function(a, b, c, x):
16     return a * x ** 2 + b * x + c
17
18 def coefficients_of_throw_function(initial_speed, throwing_angle, gravity):
19     a = -0.5 * gravity / (((initial_speed * math.cos(throwing_angle)) ** 2))
20     b = math.tan(throwing_angle)
21     return a, b
```

Functions

- Steps are clearer now, follows IOSP
- Still a lot of parameters that make the code hard to read

Functions

- Grouped related parameters together into custom data class
- Tradeoff:
easier to read,
but a lot more code!

Names of the Refactorings:
Extract Function,
Replace Parameter with Query

```
1 @dataclass
2 class Throw:
3     speed: float
4     height: float
5     angle: float
6
7 def throw_trajectory(throw, time, gravity=10.0):
8     x = horizontal_position(throw, time)
9     y = vertical_position(throw, gravity, x)
10    return (x, y)
11
12 def horizontal_position(throw, time):
13     return throw.speed * math.cos(throw.angle) * time
14
15 def vertical_position(throw, gravity, horizontal_position):
16     a, b = coefficients_of_throw_function(throw, gravity)
17     return quadratic_function(a, b, throw.height, horizontal_position)
18
19 def quadratic_function(a, b, c, x):
20     return a * x ** 2 + b * x + c
21
22 def coefficients_of_throw_function(throw, gravity):
23     a = -0.5 * gravity / (((throw.speed * math.cos(throw.angle)) ** 2))
24     b = math.tan(throw.angle)
25     return a, b
```


Functions



```
1 def print_shape_area(shape):
2     if isinstance(shape, Square):
3         print(f"The area is {shape.side_length ** 2}")
4     elif isinstance(shape, Circle):
5         print(f"The area is {shape.radius ** 2 * math.pi}")
6     elif isinstance(shape, Rectangle):
7         print(f"The area is {shape.width * shape.height}")
```

Functions

Name of the Refactoring:
Replace Conditional with
Polymorphism




```
1 class Shape(Protocol):
2     def area(self) → float:
3         pass
4
5 @dataclass
6 class Square:
7     side_length: float
8
9     def area(self) → float:
10         return self.side_length ** 2
11
12
13 def print_shape_area(shape: Shape):
14     print(f"The area is {shape.area()}")
```


Formatting

- Code is read much more often than written
- Well formatted code helps us to focus on the essentials

Formatting



```
1 def important_function():
2     hello = ("hello" +
3             "world"
4             )
5
6 def other_function(num):
7     return [
8         1, 2,
9         3,
10
11         num]
```

Formatting



```
1 def important_function():  
2     hello = "hello" + "world"  
3  
4  
5 def other_function(num):  
6     return [1, 2, 3, num]
```

Formatting

```
1  def very_important_function(  
2      too: int,  
3      many: float,  
4      args: str,  
5      to: str,  
6      keep: tuple[int, int],  
7      track: dict[str, str],  
8      of: str,  
9  ):  
10     pass
```

Formatting

Look out for

- vertical size
- horizontal size
- consistent spacing

Data Structures, Classes and Objects

Semantic difference between objects and data structures

- objects hide their data behind abstractions and expose functions to operate on the data
- data structures expose their data and have no functions

Data Structures, Classes and Objects

A data structure representing a point

```
1 class Point:
2     def __init__(self, x: int, y: int) → None:
3         self.x = x
4         self.y = y
```

Data Structures, Classes and Objects

In Python we can use the `@dataclass` decorator

```
1  @dataclass
2  class Point:
3      x: int
4      y: int
```


Data Structures, Classes and Objects

A protocol describing *behavior* of a point

```
1 class Point(Protocol):
2     def x(self) → int:
3         pass
4
5     def y(self) → int:
6         pass
7
8     def set_carthesian(self, x: int, y: int) → None:
9         pass
10
11    def theta(self) → int:
12        pass
13
14    def radius(self) → int:
15        pass
16
17    def set_polar(self, radius: int, theta: int) → None:
18        pass
```

Data Structures, Classes and Objects



Law of Demeter

A module should not know about the internals of objects it manipulates

```
1 gui.get_circle().get_center().set_new_coordinates(point)
```

Data Structures, Classes and Objects



```
1 gui.move_circle_to(point)
```

Data Structures, Classes and Objects




A fluent interface does not expose the internals of a class

```
1 house_builder.floor().walls().floor().walls().roof().build()
```

Data Structures, Classes and Objects

- Classes should be small
- Classes should have a single responsibility
- A lot of member variables, each only used by a single method, can indicate that a class does too much

Data Structures, Classes and Objects



```
1 class TrafficSimulation:
2     vehicles: list[Vehicle]
3     gui: GUI
4
5     def simulate_traffic(self):
6         """Do complex traffic sim here"""
7
8     def render(self):
9         for vehicle in self.vehicles:
10             self.gui.render_rectangle(vehicle.position, vehicle.length, vehicle.width)
```

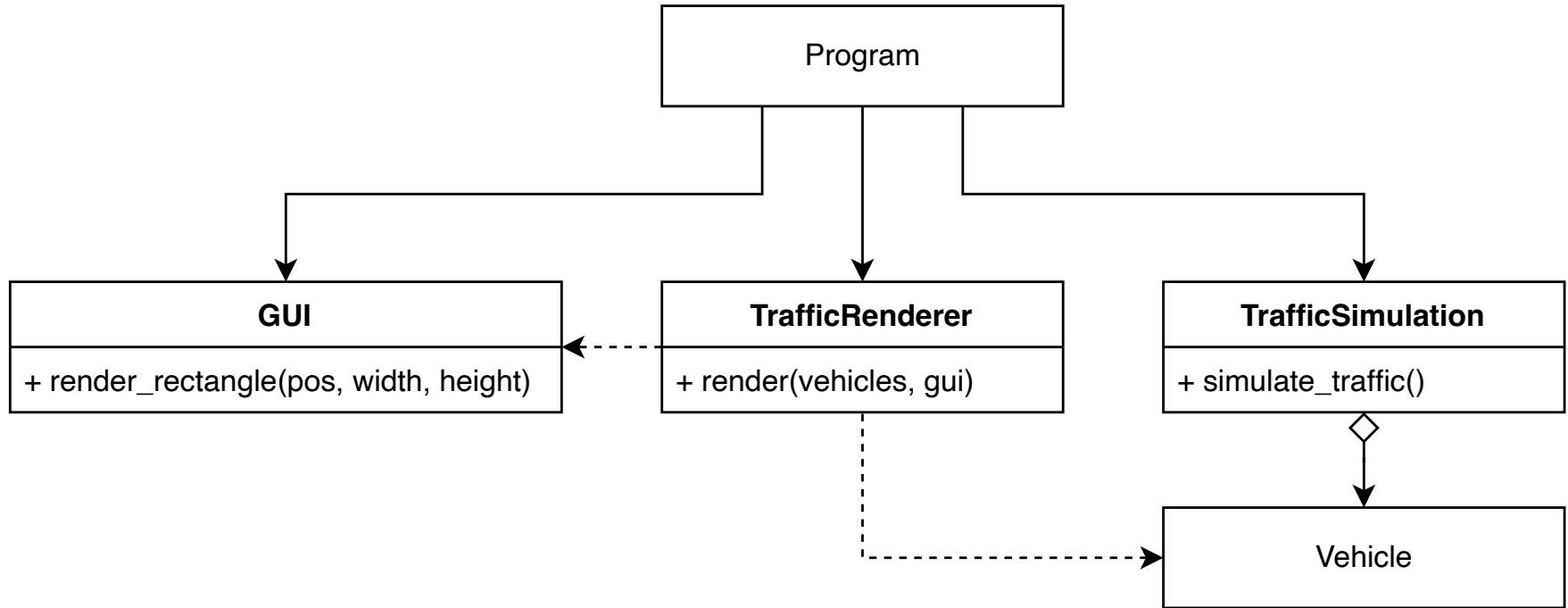
Data Structures, Classes and Objects

We have to modify TrafficSimulation every time

- something in the simulation logic changes
- we want to change the way vehicles are rendered
- the public interface of the GUI class changes

Multiple reasons to change = Multiple responsibilities!

Data Structures, Classes and Objects



Python tips and tricks

Type hinting

Python supports **type hints** since version 3.5

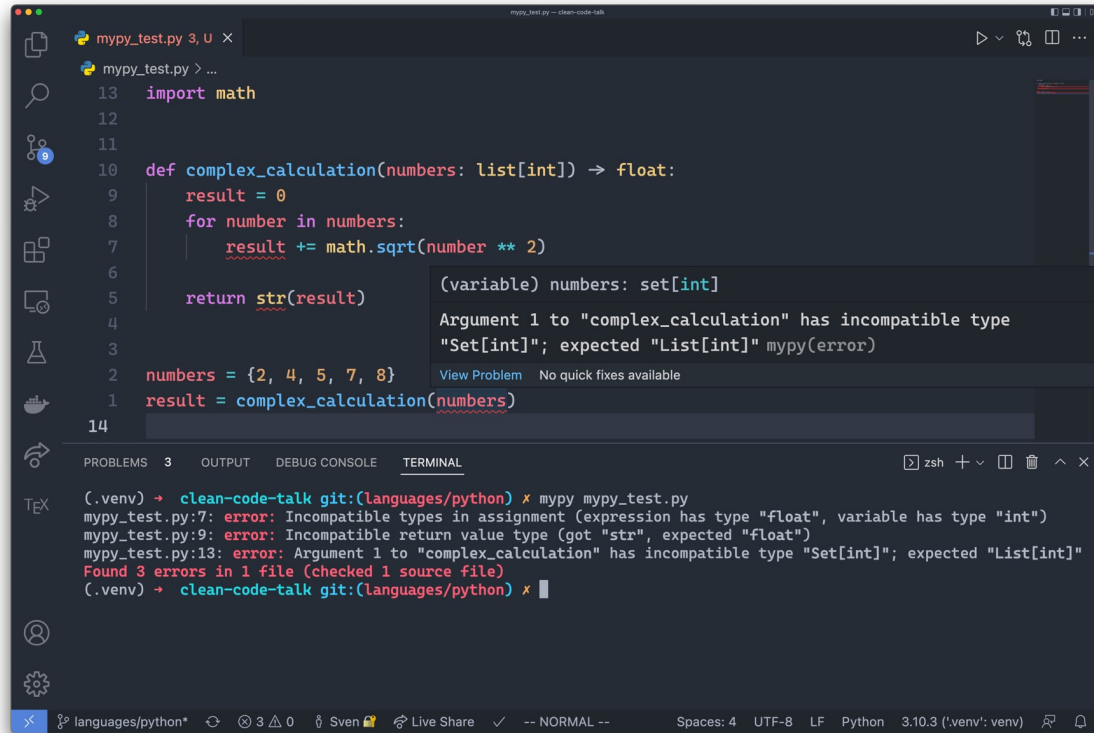
- Type hints communicate which types work with your functions
- Readers and users of your code won't have to guess which types are expected
- Use at least in interfaces intended to be used by others
- Recommendation: use type hints everywhere, they will help you understand your older code as well!

Type hinting

```
1 BinaryOperator = Callable[[Number, Number], Number]
2
3
4 def accumulate(
5     iterable: Iterable[Number],
6     operator: BinaryOperator,
7     initial: Number,
8 ) → Number:
9     for element in iterable:
10         initial = operator(initial, element)
11
12     return initial
```

Type hinting

Use *mypy* to check correct type usage across your code base



The screenshot shows a VS Code editor window with a file named `mypy_test.py`. The code contains a function `complex_calculation` that takes a `list[int]` and returns a `float`. The function body has a `result` variable of type `float` that is incremented by `math.sqrt(number ** 2)` for each `number` in `numbers`. The function returns `str(result)`. Below the function, there is a list `numbers = {2, 4, 5, 7, 8}` and a call to `complex_calculation(numbers)`. A tooltip is visible over the `numbers` variable, showing the error: `(variable) numbers: set[int]` and `Argument 1 to "complex_calculation" has incompatible type "Set[int]"; expected "List[int]" mypy(error)`. The bottom panel shows the `PROBLEMS` tab with three errors listed: `mypy_test.py:7: error: Incompatible types in assignment (expression has type "float", variable has type "int")`, `mypy_test.py:9: error: Incompatible return value type (got "str", expected "float")`, and `mypy_test.py:13: error: Argument 1 to "complex_calculation" has incompatible type "Set[int]"; expected "List[int]"`. The terminal shows the command `clean-code-talk git:(languages/python) x mypy mypy_test.py` and the output of the mypy command.

```
mypy_test.py 3, U x
mypy_test.py > ...
13 import math
12
11
10 def complex_calculation(numbers: list[int]) -> float:
9     result = 0
8     for number in numbers:
7         result += math.sqrt(number ** 2)
6
5     return str(result)
4
3
2 numbers = {2, 4, 5, 7, 8}
1 result = complex_calculation(numbers)
14

(variable) numbers: set[int]
Argument 1 to "complex_calculation" has incompatible type
"Set[int]"; expected "List[int]" mypy(error)
View Problem No quick fixes available

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL
(.venv) -> clean-code-talk git:(languages/python) x mypy mypy_test.py
mypy_test.py:7: error: Incompatible types in assignment (expression has type "float", variable has type "int")
mypy_test.py:9: error: Incompatible return value type (got "str", expected "float")
mypy_test.py:13: error: Argument 1 to "complex_calculation" has incompatible type "Set[int]"; expected "List[int]"
Found 3 errors in 1 file (checked 1 source file)
(.venv) -> clean-code-talk git:(languages/python) x
```

Formatting

Use a PEP8 conformant formatter like *autopep8* or *black*
Recommendation: *black*

Let's get to work!

Cleaning up a 2D heat transfer simulation

2D Heat Transfer

Numerical scheme: Forward in time, central in space

Temperature for next time step:

$$T_{i,j}^{t+1} = T_{i,j}^t + \kappa * \Delta t * \left[\frac{T_{i+1,j}^t - 2T_{i,j}^t + T_{i-1,j}^t}{(\Delta x)^2} + \frac{T_{i,j+1}^t - 2T_{i,j}^t + T_{i,j-1}^t}{(\Delta y)^2} \right]$$

2D Heat Transfer

Boundary Conditions

Dirichlet: constant value

$$T_{i,j}^{t+1} = \text{const}$$

Neumann: sets a fixed gradient for one direction

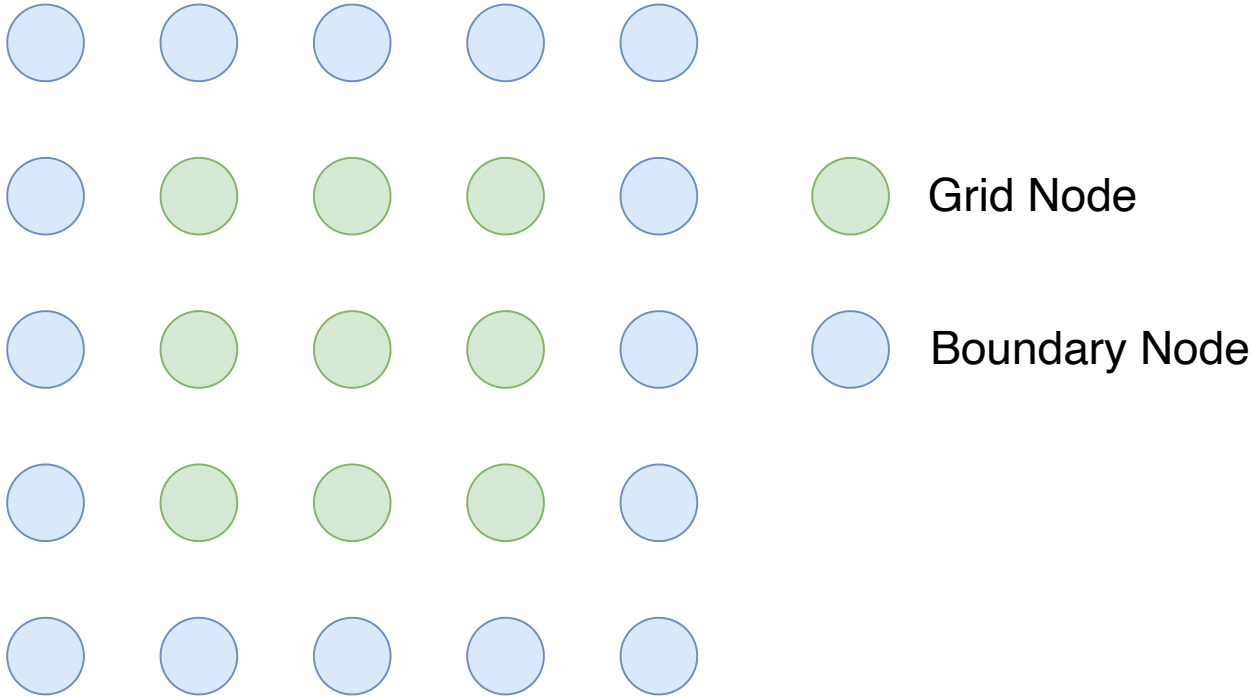
$$\text{North: } T_{i,j}^{t+1} = T_{i-2,j}^t + 2 * \text{grad} * \Delta y$$

$$\text{South: } T_{i,j}^{t+1} = T_{i+2,j}^t - 2 * \text{grad} * \Delta y$$

$$\text{West: } T_{i,j}^{t+1} = T_{i,j-2}^t + 2 * \text{grad} * \Delta x$$

$$\text{East: } T_{i,j}^{t+1} = T_{i,j+2}^t - 2 * \text{grad} * \Delta x$$

2D Heat Transfer



2D Heat Transfer

