master DSLS

Programming 2

1. Introduction and SOLID principles

Bart Barnard

1

1. Introduction

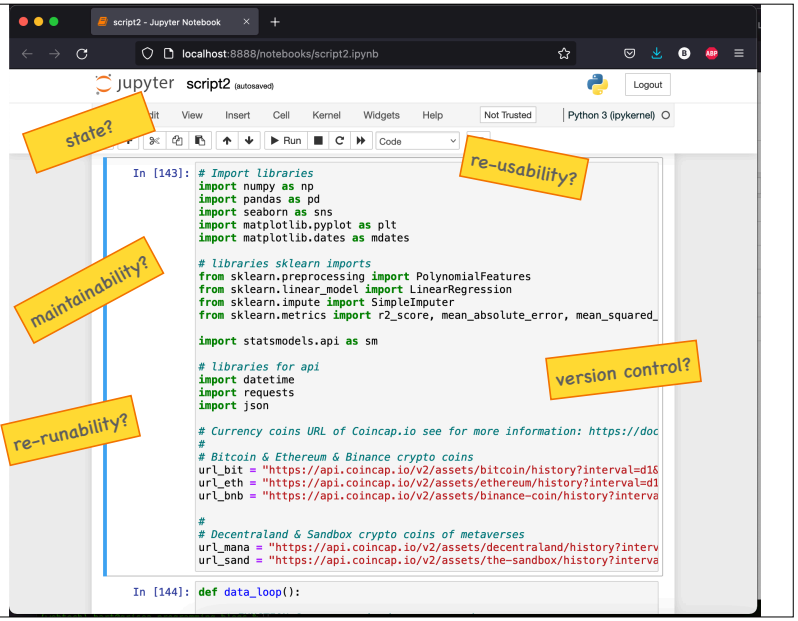
2

– Planning of the period –

week	date	subjects	form	assignment
1	2 March	Introduction, SOLID principles	L	
	3 March	static code analysis	L/T	code reading exercise
2	9 March	classes and instances, methods	L	
	10 March	constructors and destructors	L	
3	17 March	creating complex classes	T	json exercise
4	22 March	dunders, testing, git	L	
	24 March	test driven development	T	unit test exercise
5	30 March	functions as parameters	L	
	31 March	dependency injection	L	
6	5 Apil	code from scratch, modules	T	picturebuilder exercise
	7 April	UML and design patterns	L	
7	12 April	Lessons learned, wrap up	T	refactoring own code base
	13 April	presentations	T	

L = Lecture, T = Tutorial

3



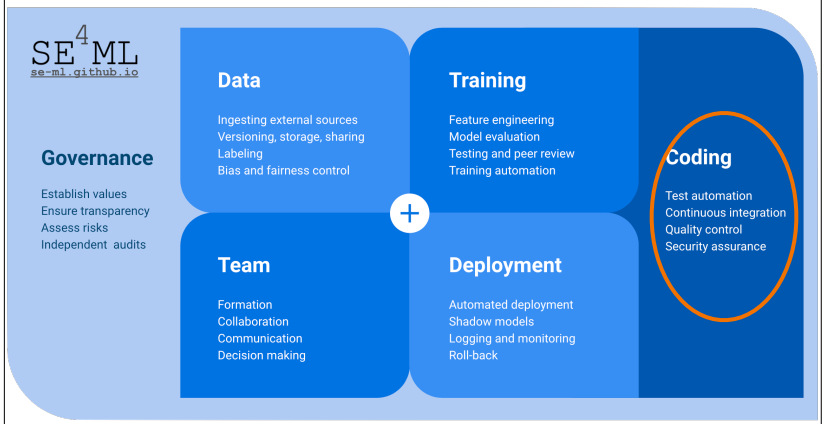
4

> I hope we don't get a generation of programmers that put python on their resume but all they know is how to open a notebook and call some pandas functions.
>
> – Guido van Rossum

> You can't be an ~~AI~~ data science expert these days and not have some grounding in software engineering.
>
> – Grady Booch

https://computingthehumanexperience.com/

5



https://se-ml.github.io/

6

2. SOLID principles

7

SOLID

- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

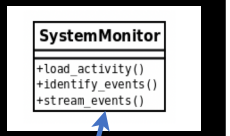
8

Single Response Principle

- *A class should have one, and only one, reason to change*
- A class should concentrate on *Doing One Thing*.
- If you can change different parts for different reasons that you should separate these.
- The smaller the class, the better

Bad design example

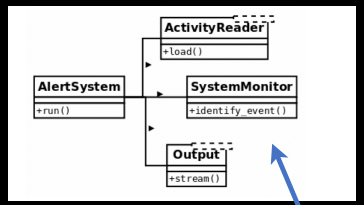
```
class SystemMonitor:
    def load_activity(self):
    """Get the events from a source, to be processed."""

    def identify_events(self):
    """Parse the source raw data into events (domain objects)."""

    def stream_events(self):
    """Send the parsed events to an external agent."""
```



Part of an application that is in charge of reading information about events from a source (this could be log files, a database, or many more sources), and identifying the actions corresponding to each particular log.

The problem with this class is that it defines an interface with a set of methods that correspond to actions that are orthogonal: each one can be done independently of the rest.

Solution: separation of concerns



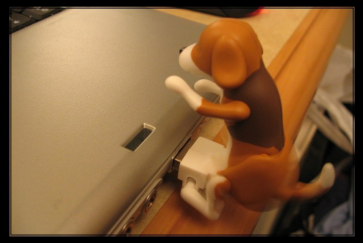
- Three new classes. Each class encapsulates a specific set of methods that are independent of the rest.
- The same behaviour is achieved by using an object that will interact with instances of three new classes, using those objects as collaborators.
- Changes are now local, the impact is minimal, and each class is easier to maintain and even more reusable.

Single Response Principle

NB: A class can (and usually does) have multiple methods as long as they correspond to the same logic that that class is in charge of handling.

This is more a question of ontology and methodology than of technique.

Open Close Principle



Open Closed Principle

You don't need to rewire your MoBo to plug in "Mr Happy"

- Software entities (classes, modules, functions, etc) should be *open for extension*, but *closed for modification*
- You should be able to *extend* a classes behaviour, without *modifying* it
- Good practice: change a class behaviour by *composition* (or, less good, *inheritance*)

Bad design example

```python
class AreaCalculator:
    """class with list of shapes that calculates total area"""

    def __init__(self, shapes):

        assert isinstance(shapes, list), "`shapes` should be of type `list`."
        self.shapes = shapes

    @property
    def total_area(self):
        """calculate area of a rectangle"""
        total = 0
        for shape in self.shapes:
            total += shape.width * shape.height

        return total
```

- What will happen if you want to extend the class to calculate the area of different shapes?
- How can you extend without modifying the method `total_area`?

Solution: do only one thing

A software problem should be decomposed into smaller subproblems, until the entity does exactly one thing.

`if else` statements in logic are *usually* not a good sign

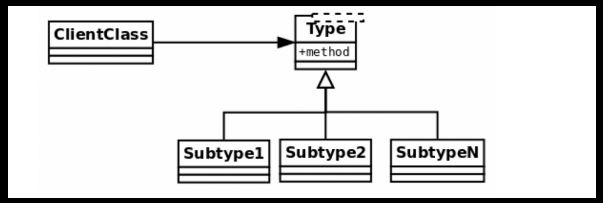
```python
@property
def total_area(self):
    total = 0
    for shape in self.shapes:
        if assert isinstance of Rectangle:
            total += shape.width * shape.height
        if assert isinstance of Circle:
            total += 2 * math.pi * (shape.radius)**2
    return total
```

```python
class Rectangle(Shape):

    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

class AreaCalculator(object):

    def __init__(self, shapes):
        self.shapes = shapes

    @property
    def total_area(self):
        total = 0
        for shape in self.shapes:
            total += shape.area
        return total
```


Liskov Substitution Principle

- Derived classes must be *substitutable* for their base classes.
- Functions that use pointers to base classes must be able to use objects of derived classes without knowing it
- Subclasses should behave nicely when used instead of their base class

Liskov Substitution Principle



- The main idea behind LSP is that, for any class, a client should be able to use any of its subtypes indistinguishably, without even noticing, and therefore without compromising the expected behaviour at runtime.
- This means that clients are completely isolated and unaware of changes in the class hierarchy.
- More formally, this is the original definition of Liskov's substitution principle: if ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced by objects of type ***S***, without breaking the program.

Derived classes must be substitutable for their base classes.

```python
class Event:
    def meets_condition(self, event_data: dict) -> bool:
        return False

class LoginEvent(Event):
    def meets_condition(self, event_data: list) -> bool:
        return bool(event_data)

#mypy error: Argument 1 of "meets_condition" incompatible with supertype "Event"
```

- You cannot derive square from rectangle, change calculation weight into height since it does not matter and down the road inherit from Square when you want to calculate rectangle. It will give wrong output.
- You can not derive from a class and overwrite the method

Liskov Substitution Principle

- *Make fine grained interfaces that are client specific.*
- Clients should not be forced to depend upon interfaces that they do not use
- Keep interfaces small
- Don’t pollute interfaces with a lot of methods

Method pollution

```python
class AbstractWorker(metaclass = ABCMeta):

    @abstractmethod
    def work(self):
        pass

    @abstractmethod
    def eat(self):
        pass

class Worker(AbstractWorker):

    def work(self):
        print("I'm normal worker. I'm working.")

    def eat(self):
        print("Lunch break....(5 secs)")
        time.sleep(5)
```

A robot is a worker but does not eat….

Solution: Use Interface Segregation

```python
class Workable(metaclass = ABCMeta):

    @abstractmethod
    def work(self):
        pass

class Eatable(metaclass = ABCMeta):

    @abstractmethod
    def eat(self):
        pass

class AbstractWorker(Workable, Eatable):
    pass

class Worker(AbstractWorker):

    def work(self):
        print("I'm normal worker. I'm working.")

    def eat(self):
        print("Lunch break....(5 secs)")
        time.sleep(5)
```

```python
class Robot(Workable):

    def work(self):
        print("I'm a robot. I'm working....")

    # No need for implementation of `eat` which
    #is not neccessary for a `Robot`.
```

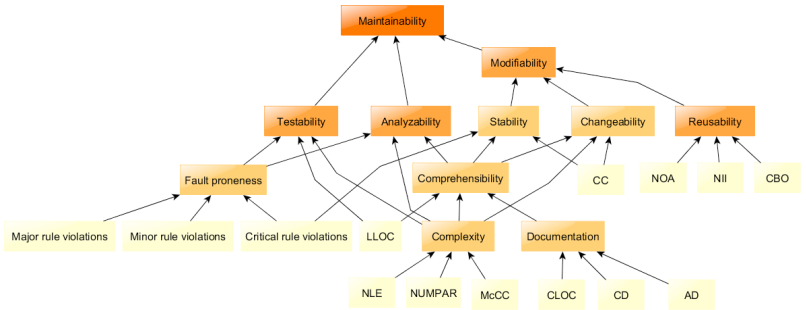

Dependency Inversion Principle

- *Depend on abstractions, not on concretions.*
- A high level module should not depend on low level modules. Both should depend upon abstractions. Abstraction should not depend upon details, details should depend upon abstraction

- Use lots of interfaces and abstractions
- Use python ABC for defining interfaces
- Write code that is decoupled

3. Static code analysis

SQuaRE (Software product Quality Requirements and Evaluation)



https://en.wikipedia.org/wiki/ISO/IEC_9126#Developments

Three levels of software analysis

Unit Level
Analysis that takes place within a specific program or subroutine, without connecting to the context of that program.

Technology Level
Analysis that takes into account interactions between unit programs to get a more holistic and semantic view of the overall program in order to find issues and avoid obvious false positives.

System Level
Analysis that takes into account the interactions between unit programs, but without being limited to one specific technology or programming language.

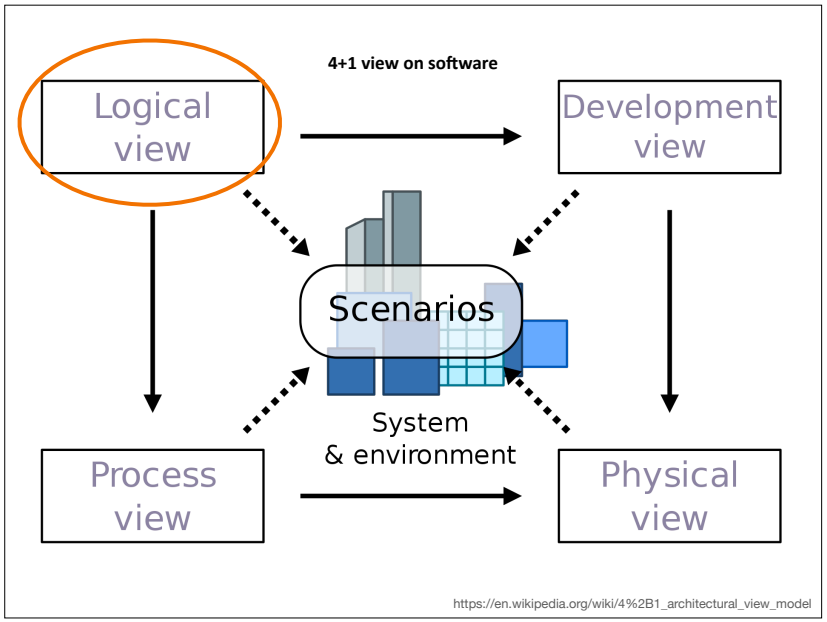
25

First questions of statical code analysis

How many modules?
How many classes?
On average, how many methods per class?
On average, how many lines per method?

What is the relationship between the classes?

26



27