Structural and Behavioral design patterns

Benedikt Hegner
(CERN)

With thanks to Jacek Generowicz

From Stefan's slides yesterday:

Strong typing:	operation upon an object must be defined
Weak typing:	can perform operations on any object
Static typing:	names bound to types (classes) at compile time
Dynamic typing:	names bound to objects at run time
Static binding:	names bound to objects at compile time
Dynamic binding:	names bound to objects at run time

C++, Java:	strong+static typing + dynamic binding
Python:	strong+dynamic typing
Perl:	weak+dynamic typing
Fortran, C:	strong+static typing + static binding (except casts)

One has to disentangle various ‘types’

```
<variable> = <value>
```

Variable might have a type

Value might have a type

```
object->method
```

is the actual code already known at compile time?

Strong typing:

```
int a = 2
string b = "2"

concatenate(a, b)          # Type Error
add(a, b)                  # Type Error
concatenate(str(a), b) # Returns "22"
add(a, int(b))             # Returns 4
```

Concerns types/values

Weak typing:

```
a = 2
b = "2"

concatenate(a, b) # Returns "22"
add(a, b)              # Returns 4
```

Static typing (e.g. C++):

Concerns variables/names

```
int a;

a = 2;

a = “foo”;      # Type error
```

Dynamic typing (e.g. : python)

```
a = 2;

a = “foo”;      # Perfectly works
```

Static binding:

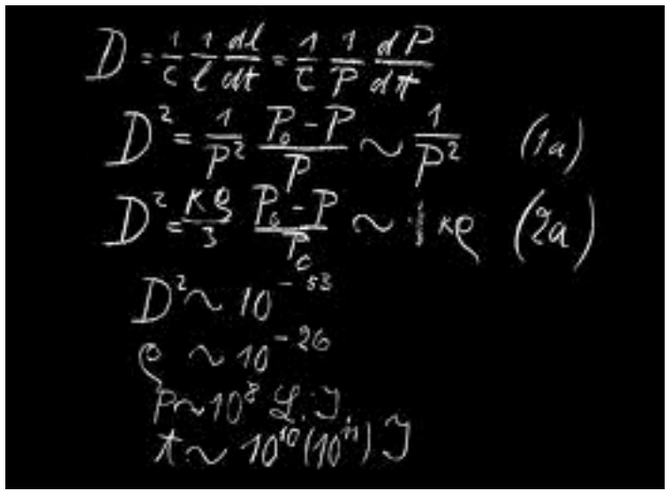
```
class A{
    int doSomething(){return 1;}
};

class B : A{
    int doSomething(){return 2;}
};

A* a = new B();
a->doSomething;   # returns “1”
```

Dynamic binding:

```
class A{
    virtual int doSomething(){return 1;}
};

class B : A{
    int doSomething(){return 2;}
};

A* a = new B();
a->doSomething;   # returns “2”
```

Concerns polymorphism

The Blackboard Pattern

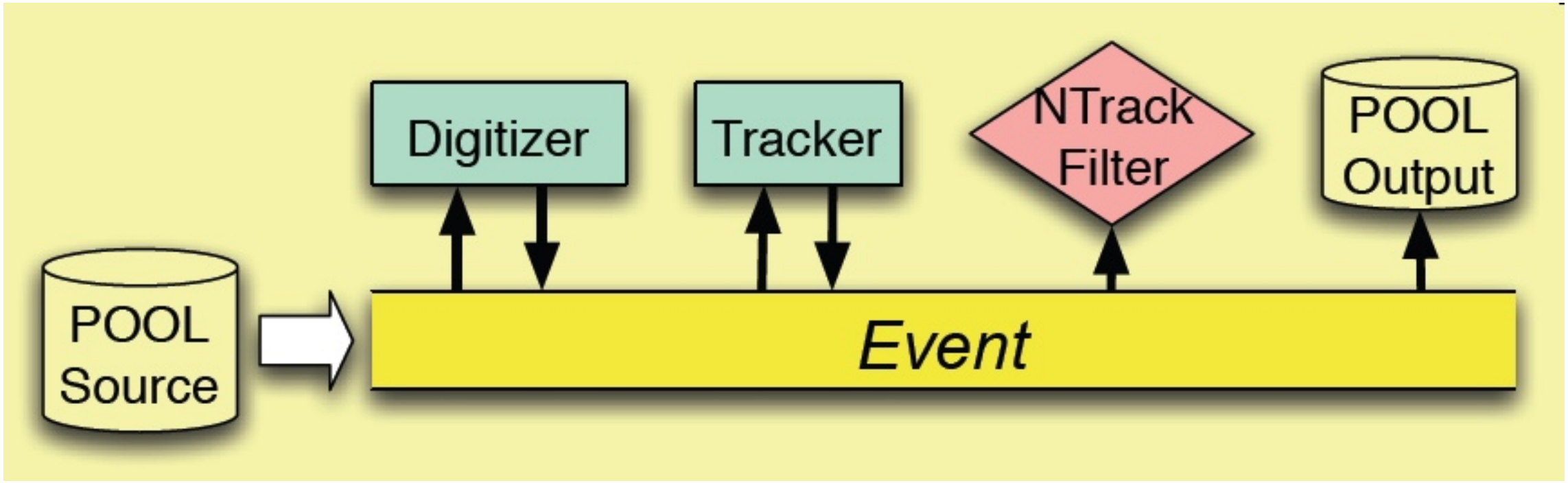


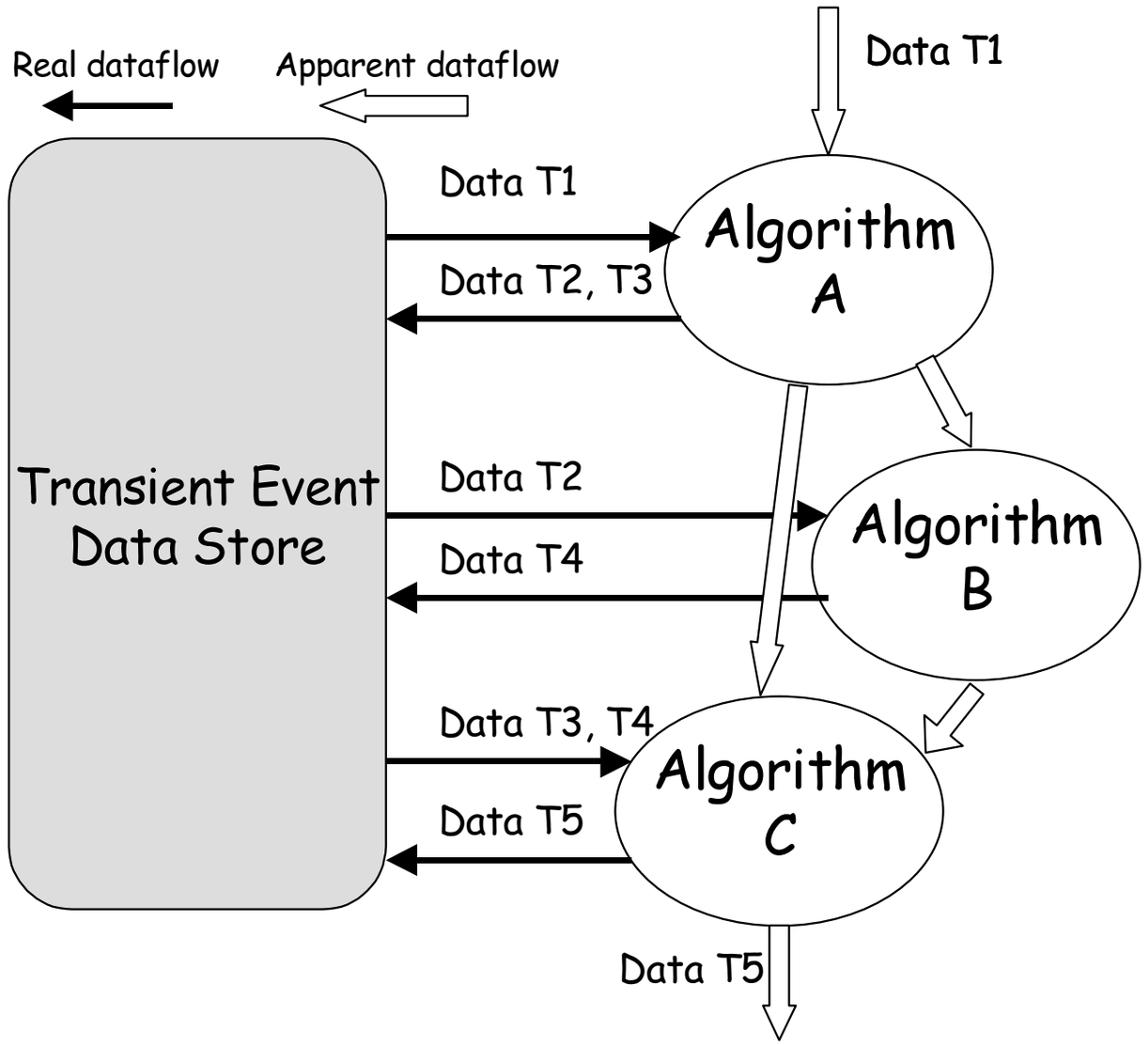
Very early in the software design the LHC experiments decided to decouple algorithms from reconstructed objects

- Modules/Algorithms create data objects
- These get stored into a common place and can be accessed by other modules/algorithms
- The data objects are inherently dumb and can't do any advanced things

- The reason has been historical as previously procedures have been operating on a Fortran **COMMON** blocks. And people just went on that way.

- Only later one really understood the advantage of this idea...


Digitizer
Tracker
NTrack
Filter
POOL
Output
POOL
Source
Event

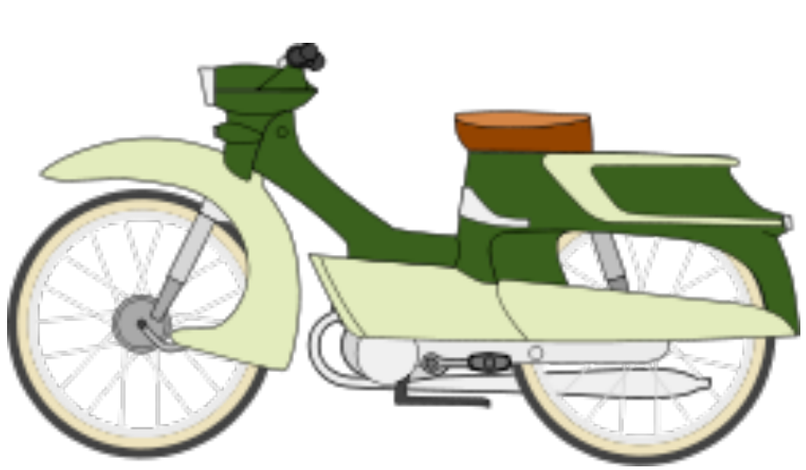

Real dataflow
Apparent dataflow
Data T1
Transient Event
Data Store
Data T1
Data T2, T3
Data T2
Data T4
Data T3, T4
Data T5
Algorithm
A
Algorithm
B
Algorithm
C
Data T5

Design a ‘blackboard’ to store and retrieve data

What checks or policies have to be put in so that write actions don’t interfere with each other?

Why is it good to decouple the algorithms from the created data?

The Bridge Pattern



```
class Kettler:

    def pedals(self):
        ...

    def handlebar(self):
        ...


class VWBeetle:

    def engine(self):
        ...

    def steering_wheel(self):
        ...


class HarleyDavidson:

    def engine(self):
        ...

    def handlebar(self):
        ...
```

Find a way to make all vehicles usable by the same interface

Assume you don't have a chance to convince e.g. VW to change their beetle class

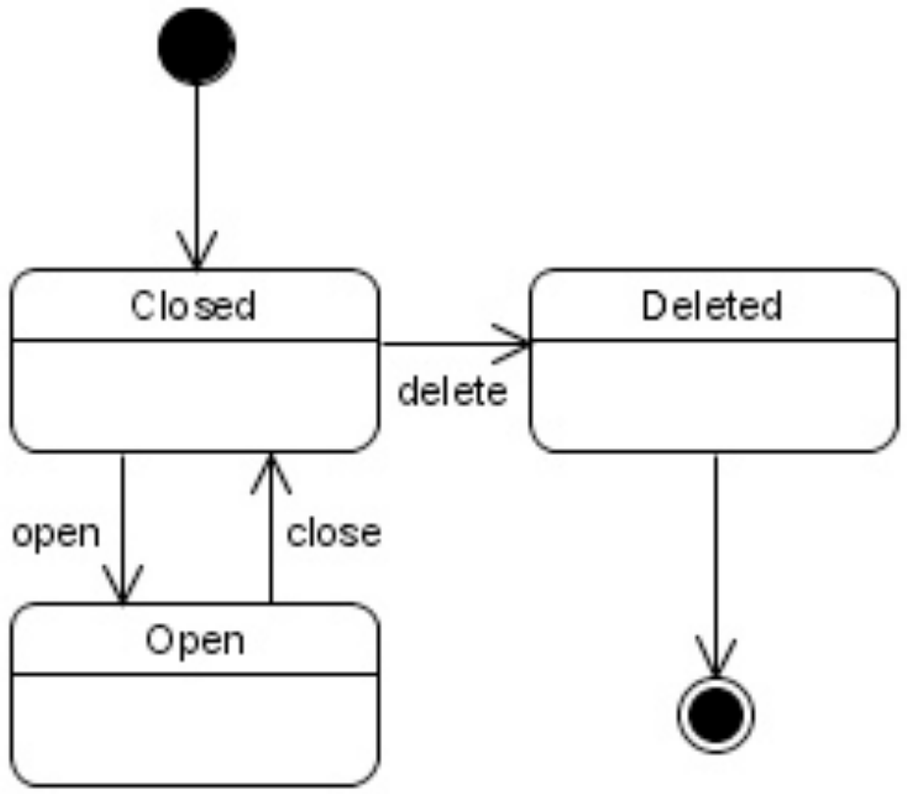
```
class Vehicle (Vehicle):

    def drive(self, impl):
        self.impl = impl

    def drive(self):
        <not implemented>


class Bike(Vehicle):

    def drive(self):
        self.impl.pedals()
        self.impl.handlebar()
```

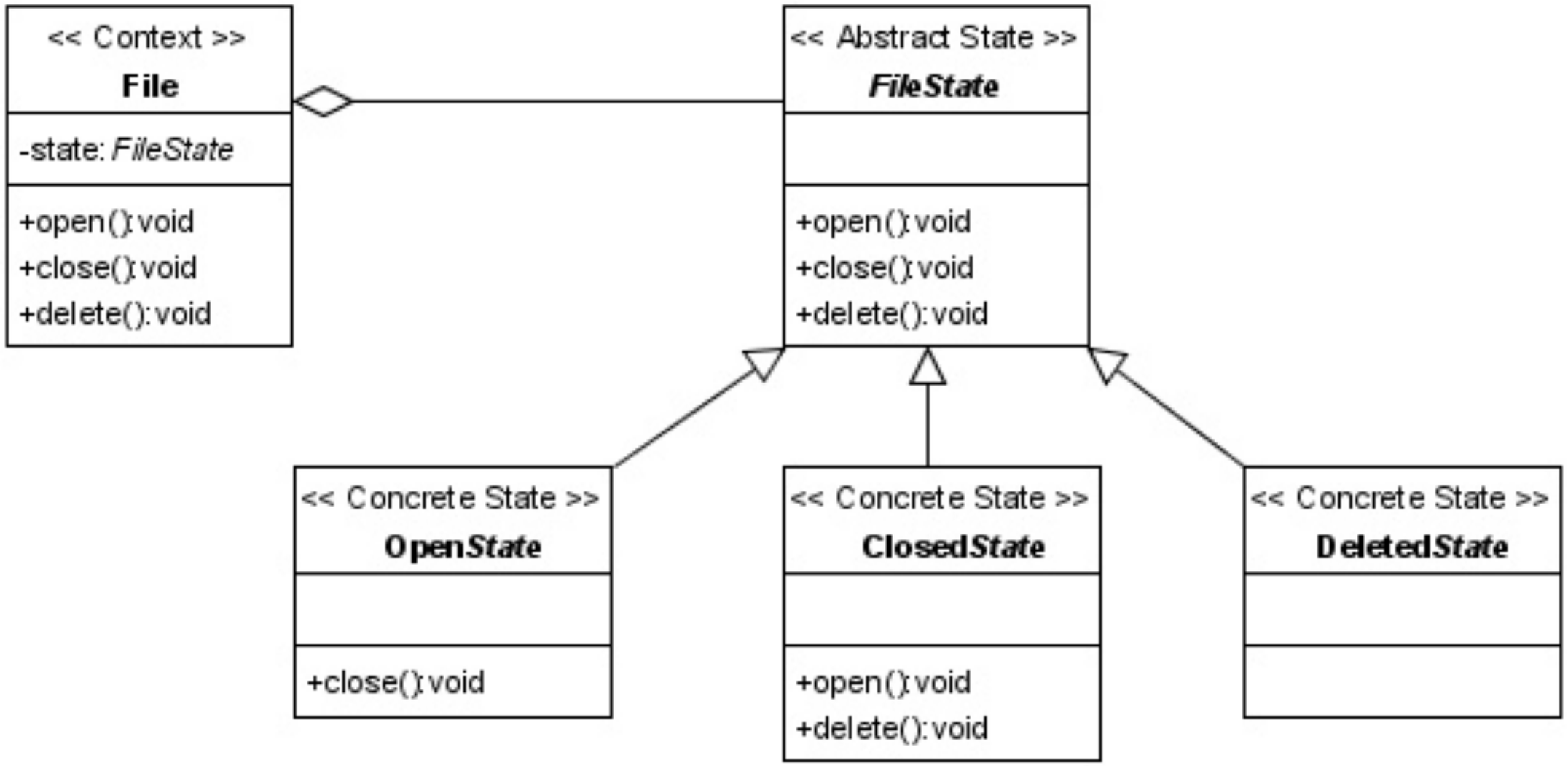
Good example for abstraction and making different classes interchangeable

The State Pattern

States appear all over the place

- Often the behaviour of an object depends on what happened before
  - For example only an opened file allows you to write to it
  - But we don't care which exact steps lead to the file being open (bulk opening, individual opening)
  - We only care about the current **`state`** of the object

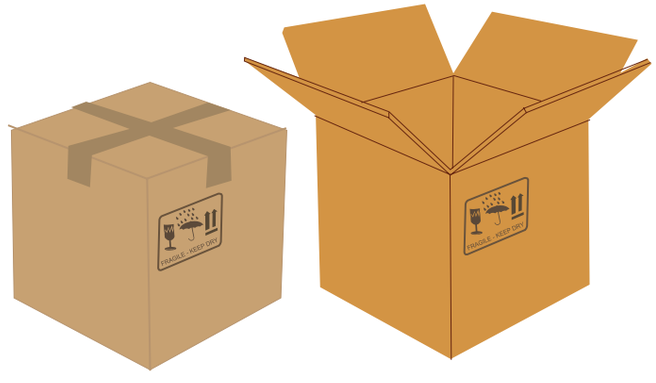



<< Context >>
File
-state: FileState
+open(): void
+close(): void
+delete(): void
<< Abstract State >>
FileState
+open(): void
+close(): void
+delete(): void
<< Concrete State >>
OpenState
+close(): void
<< Concrete State >>
ClosedState
+open(): void
+delete(): void
<< Concrete State >>
DeletedState

```
class OpenState:

    @staticmethod
    def close(box):
        box.state = box.closedState

    @staticmethod
    def open(box):
        print "Already open”

    @staticmethod
    def is_empty(box):
        return True
```

```
class ClosedState:

    @staticmethod
    def close(box):
        print "Already closed”

    @staticmethod
    def open(box):
        box.state = box.openState

    @staticmethod
    def is_open(box):
        return False
```

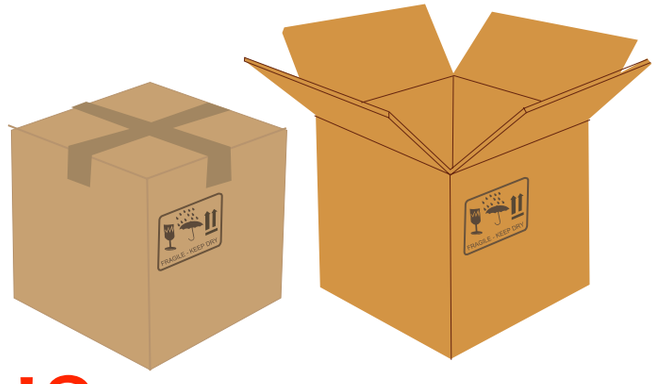
```
class Box:

    openState = OpenState()
    closedState = ClosedState()

    def __init__(self):
        self.state = self.closedState

    def close(self):
        self.state.close()

    def open(self):
        self.state.open()

    def is_empty(self):
        return self.state.is_open(self)
```



```
class OpenState:

    @staticmethod
    def close(box):
        box.state = box.closedState

    @staticmethod
    def open(box):
        print "Already open”

    @staticmethod
    def is_empty(box):
        return True
```

```
class ClosedState:

    @staticmethod
    def close(box):
        print "Already closed”

    @staticmethod
    def open(box):
        box.state = box.openState

    @staticmethod
    def is_open(box):
        return False
```

```
class Box:

    openState = OpenState()
    closedState = ClosedState()

    def __init__(self):
        self.state = self.closedState

    def close(self):
        self.state.close()

    def open(self):
        self.state.open()

    def is_empty(self):
        return self.state.is_open(self)
```



Paradox?

States are stateless!?

Design a framework to allow the usage of a local batch farm of various computers

A user should be able to inspect, start, stop, cancel, resubmit, ...

Give it a thought about who is responsible for the state transition

The Facade Pattern

Sometimes you have a very complicated system, which you want to hide from the user

- Using hot water in the shower, you don't have to think about switching on the boiler, opening support valves, changes in gas mixture, switching on a fuse... (*)

- You don't care if one component gets replaced

- In design pattern terms the simplified interface to a complicated system is called **`Facade`**

- This concept is rather common sense and an example for not to overrate design patterns for their 'brilliance'

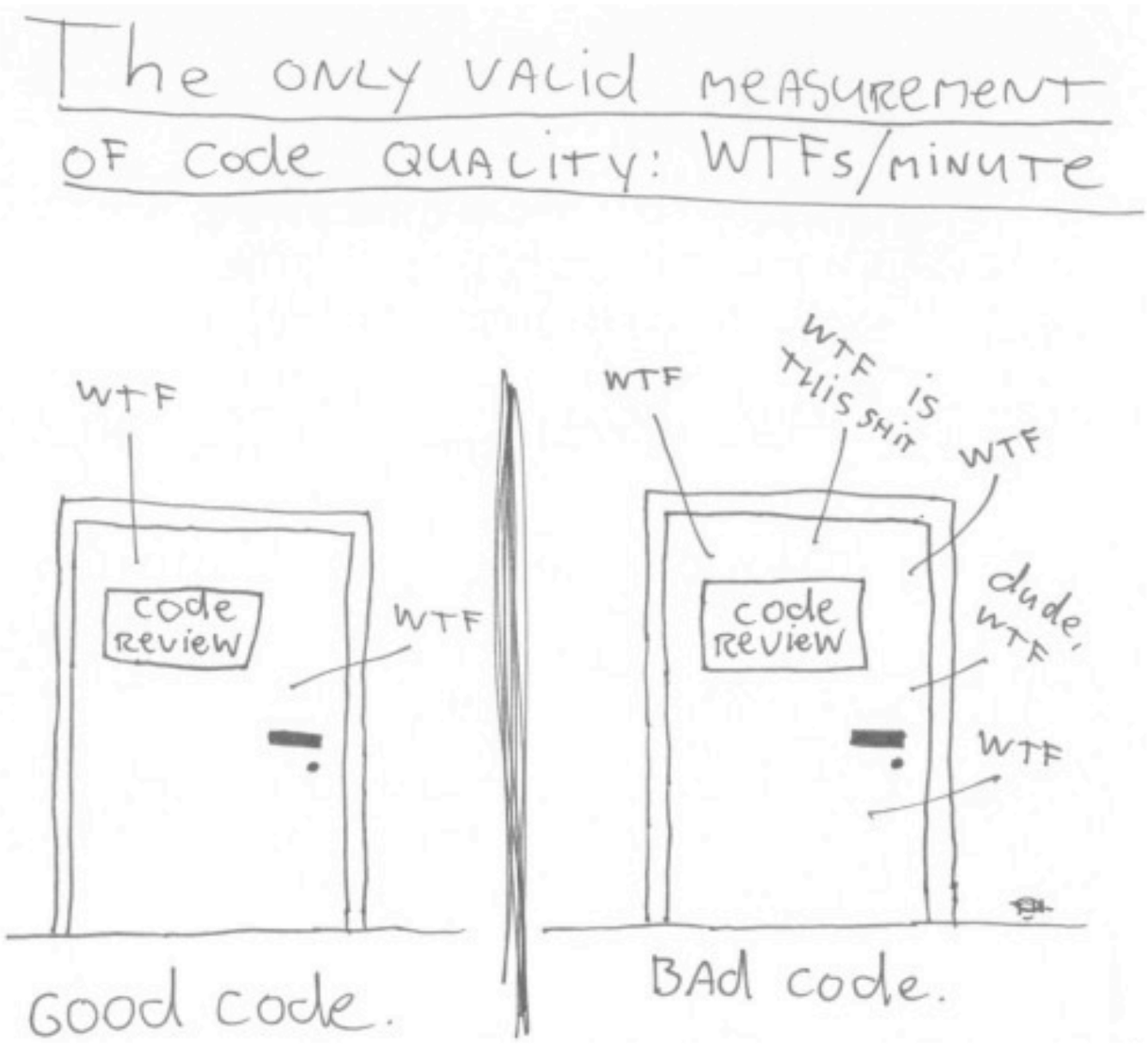
- The real name of it is just **`encapsulation`**.

(*) if you ever lived in one of the french apartments near CERN, that's unfortunately not true

What you should take home as a message:

Think before you type!

And keep the WTF frequency low!



That’s it folks!