Python:
Beyond the Basics

Properties and Class Methods

Robert Smallshire
@robsmallshire
rob@sixty-north.com

Presenter

Austin Bingham
@austin_bingham
austin@sixty-north.com

pluralsight
hardcore dev and IT training

class attributes

versus

instance attributes


CARGO
CONTAINER
CARGO
CONTAINER
CARGO
CONTAINER
CARGO
CARGO
CG 044 233 01
MAX. LOAD
TARE
MAX.GW

PEP 20 : The Zen of Python by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

static methods

with the

`@staticmethod`

decorator

The static terminology is a relic from C and C++

```
// static_member_functions.cpp
#include <stdio.h>

class StaticTest
{
private:
     static int x;
public:                          Text
     static int count()
     {
          return x;
     }
};

int StaticTest::x = 9;

int main()
{
     printf_s("%d\n", StaticTest::count());
}
```

class methods

with the

@classmethod

decorator

Choosing

`@staticmethod` or `@classmethod`

No access needed to either *class* or *instance* objects.

Most likely an implementation detail of the class.

May be able to be moved to become a module-scope function

Requires access to the class object to call other class methods or the constructor.

class methods

for

named constructors

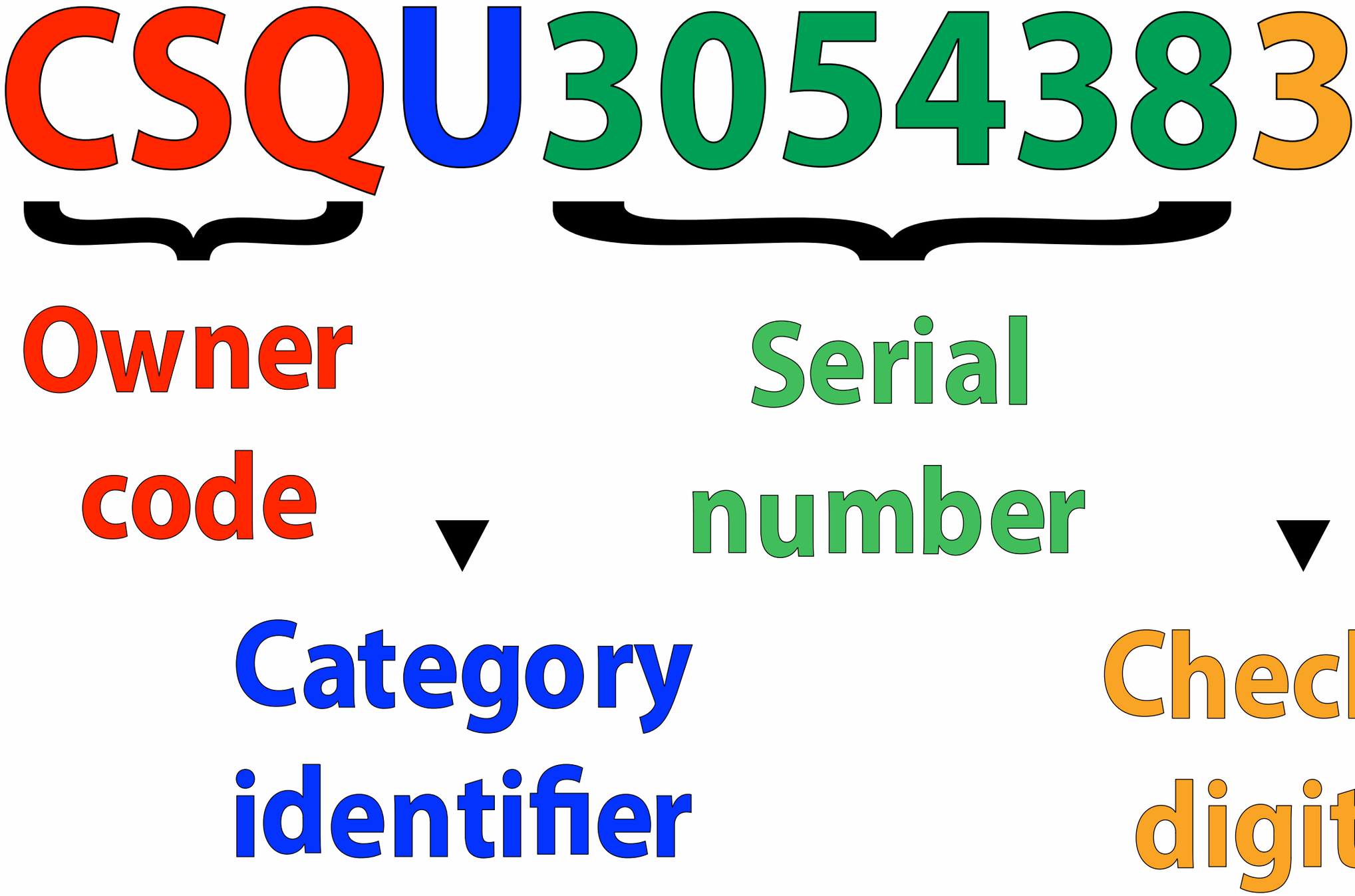
static methods

with

inheritance


CAXU
HJCU
8281 98 8
22G1
MAX.GROSS
TARE
NET
CU.CAP
MAX.GROSS
TARE
PAYLOAD
CUBE
HANJIN
SHIPPING
CO.,LTD.
MSCU
M. G. W.
TARE
NET
CU.CAP.

ISO 6346 BIC code




MAERSK
SEALAND
INGER

class methods

with

inheritance

encapsulation

using the

`@property`

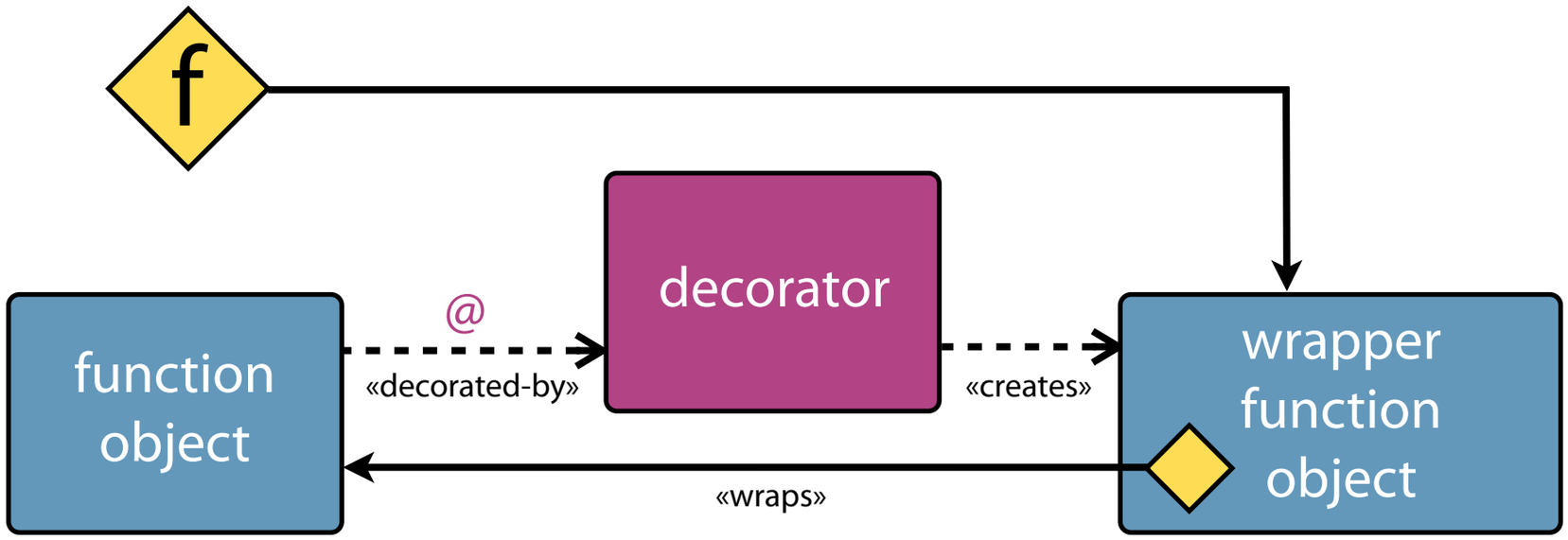
decorator


python™

≠

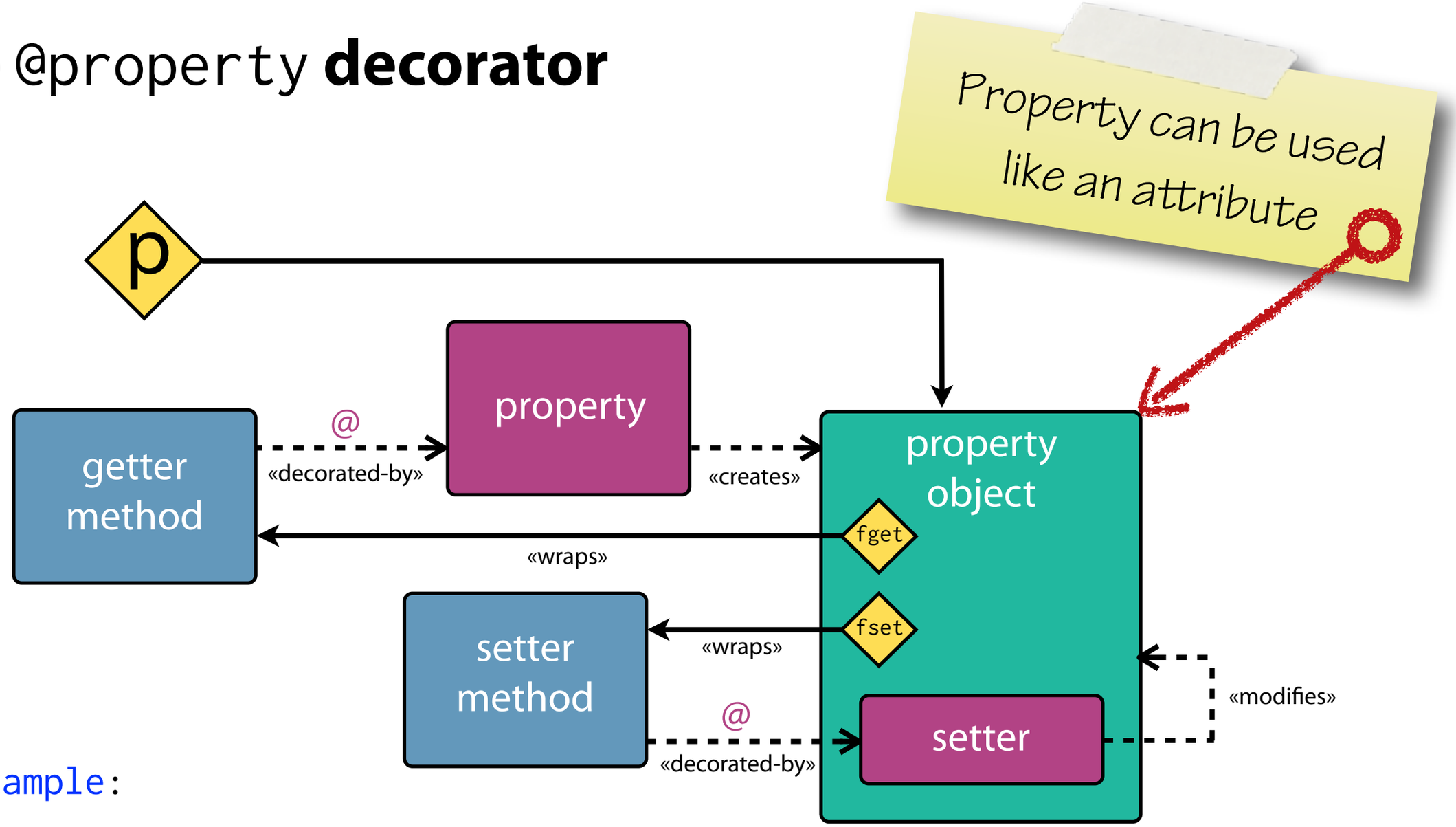

Java™

Decorator recap




```
@decorator
def f():
    do_something()
```

The `@property` decorator




```
class Example:

    @property
    def p(self):
        return self._p

    @p.setter
    def p(self, value):
        self._p = value
```

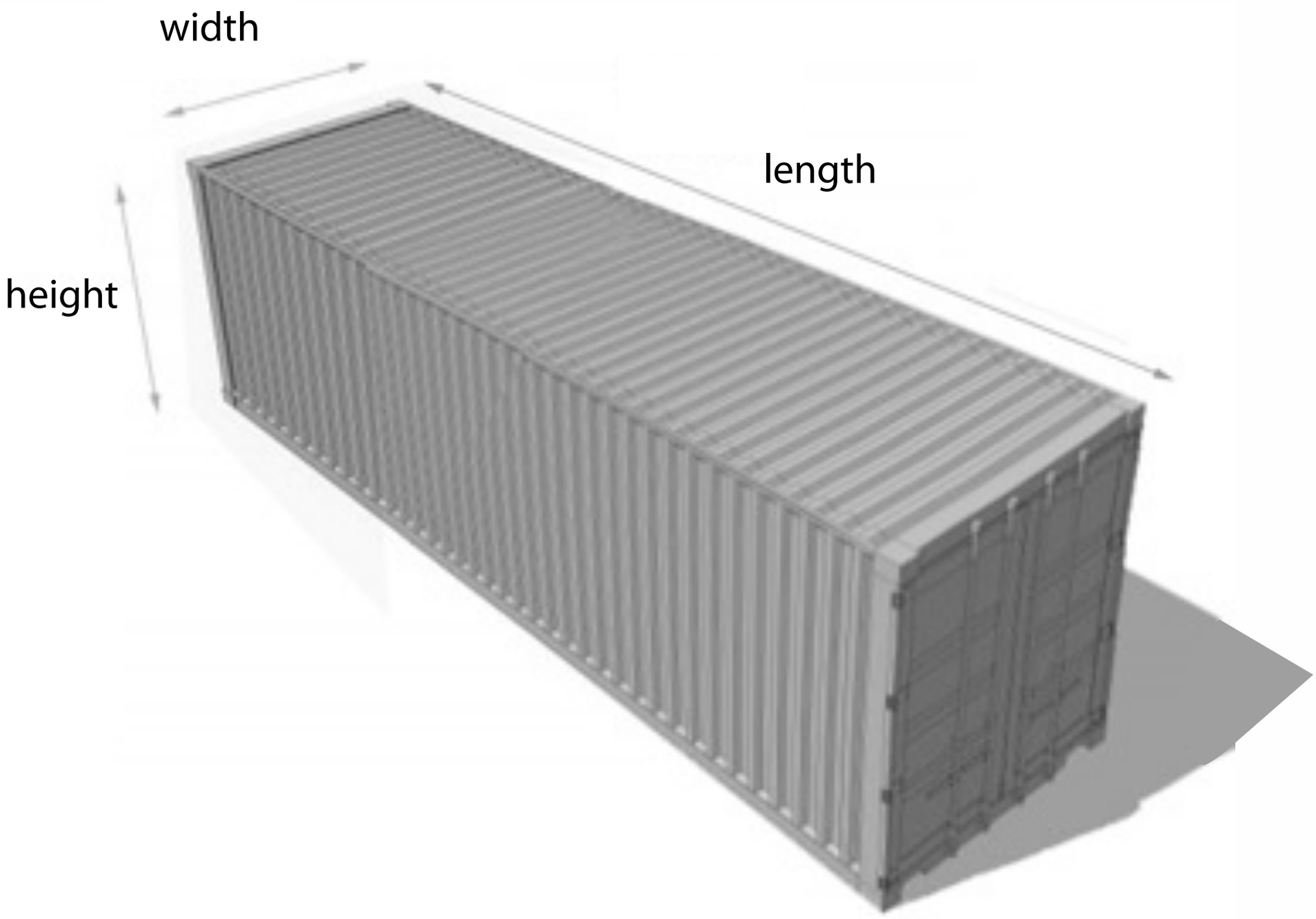
Monday, 24 February, 14

Inheritance

interaction with the

@property

decorator


width
length
height


Carrier
THINLINE
2710041

Chained
relational operators

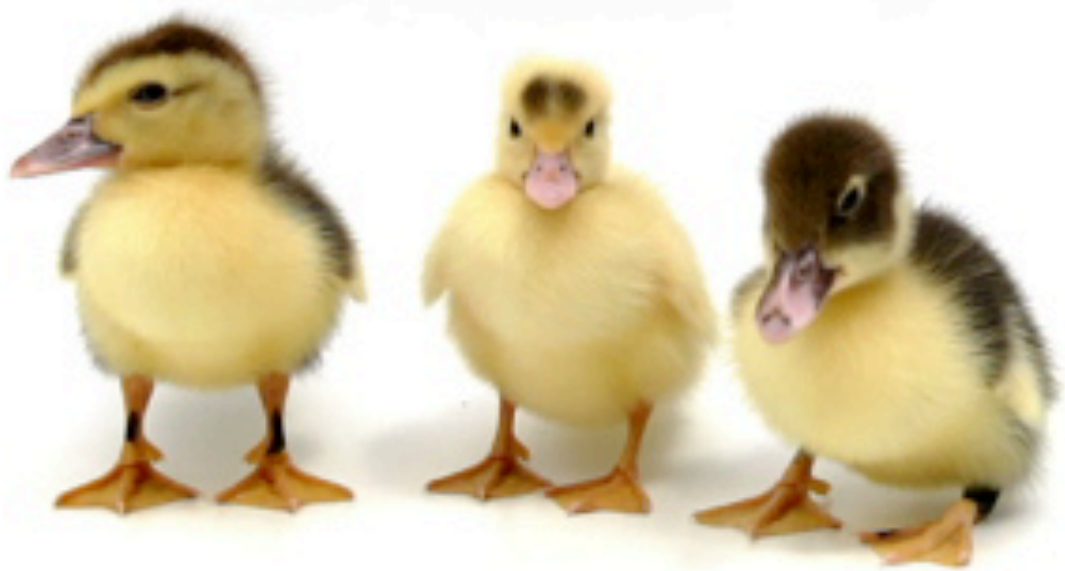
```
a < b < c
```

equivalent to

```
(a < b) and (b < c)
```

Tales of real-world Python
– better in practice than in theory –
just like Duck Typing.

Duck Tails



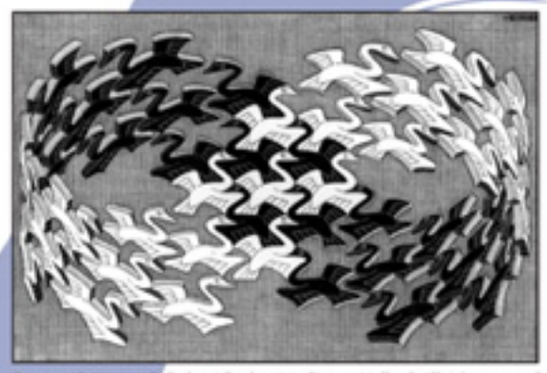
Duck Tails

Properties and the Template Method Pattern

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

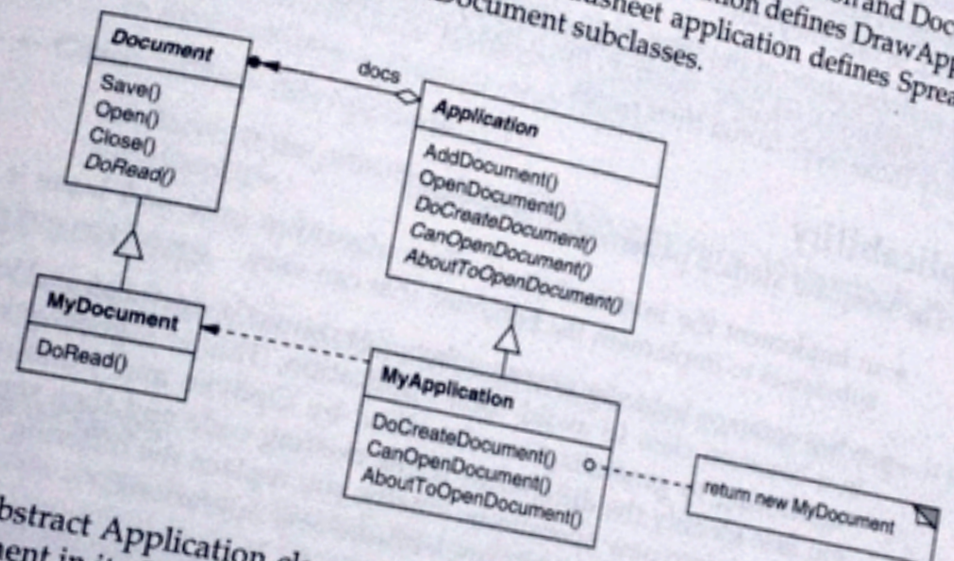
Intent

Define the skeleton of an algorithm in an operation, deferring some steps t
subclasses. Template Method lets subclasses redefine certain steps of an algorithm
without changing the algorithm's structure.

Motivation

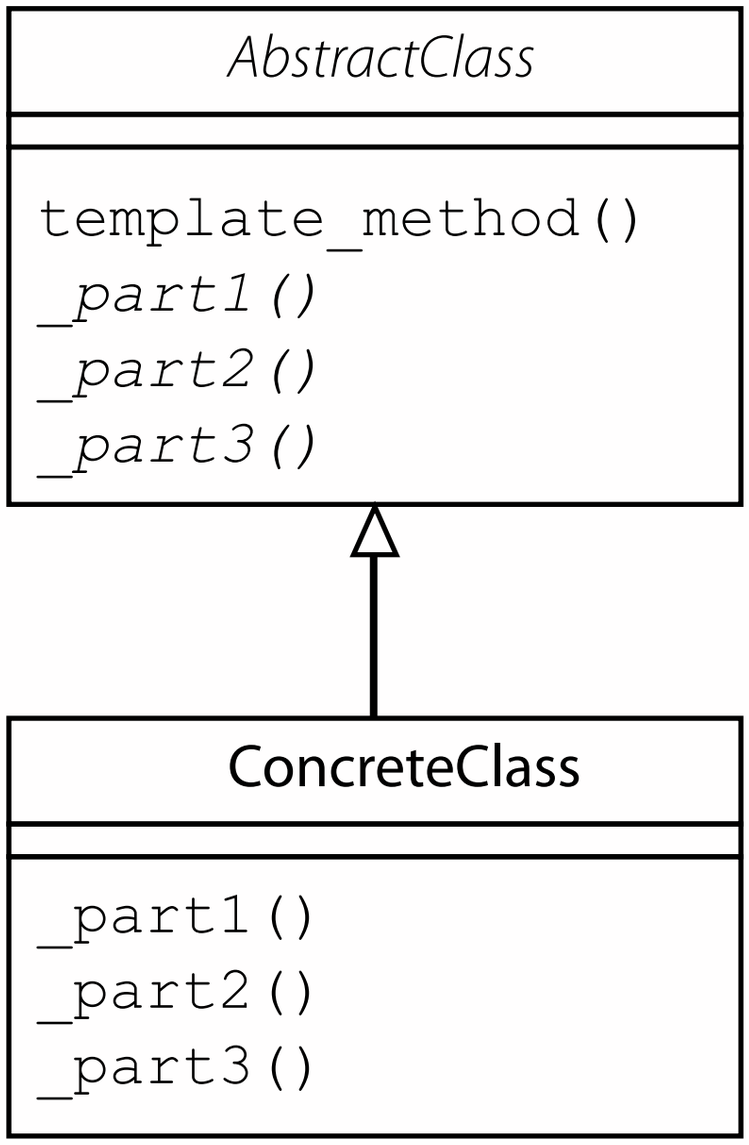
Consider an application framework that provides Application and Document
classes. The Application class is responsible for opening existing documents stored
in an external format, such as a file. A Document object represents the information
in a document once it's read from the file.

Applications built with the framework can subclass Application and Document to
suit specific needs. For example, a drawing application defines DrawApplication
and DrawDocument subclasses; a spreadsheet application defines Spreadsheet-
Application and SpreadsheetDocument subclasses.



The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation:

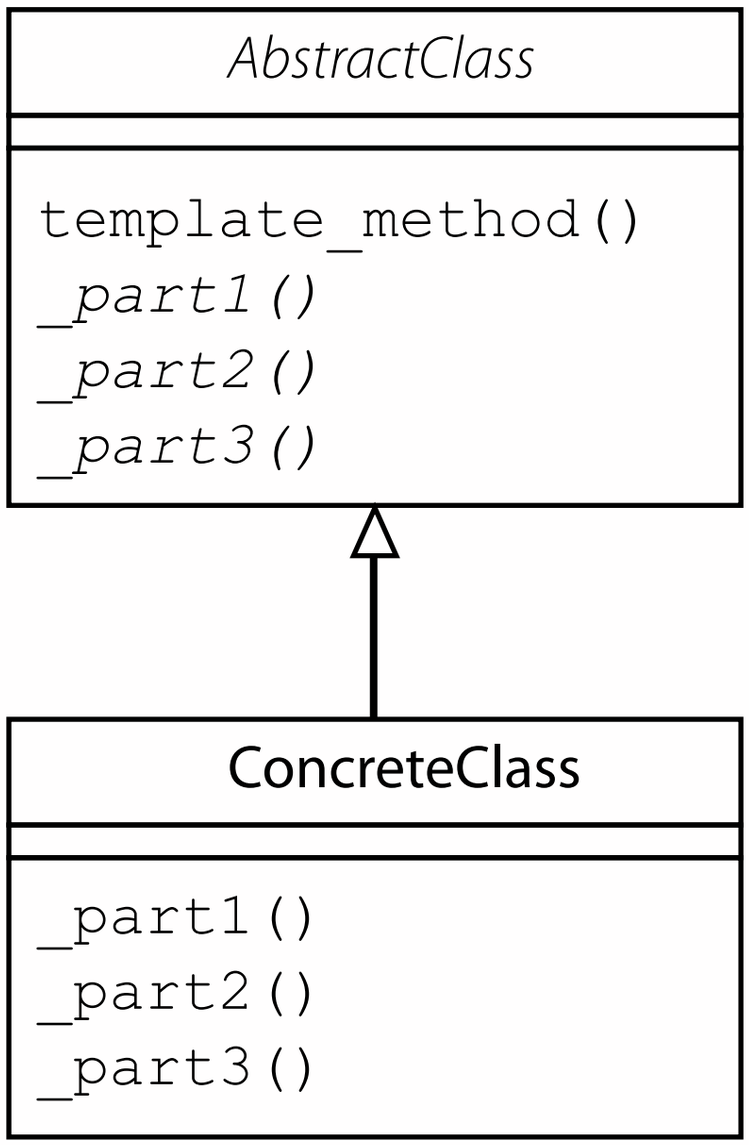
```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }
```




```python
class AbstractClass:

    def template_method(self):
        self._part1()
        self._part2()
        self._part3()

    def _part2(self):
        raise NotImplementedError("Override this method")

    def _part3(self):
        # Optionally override this
        print("Done!")
```

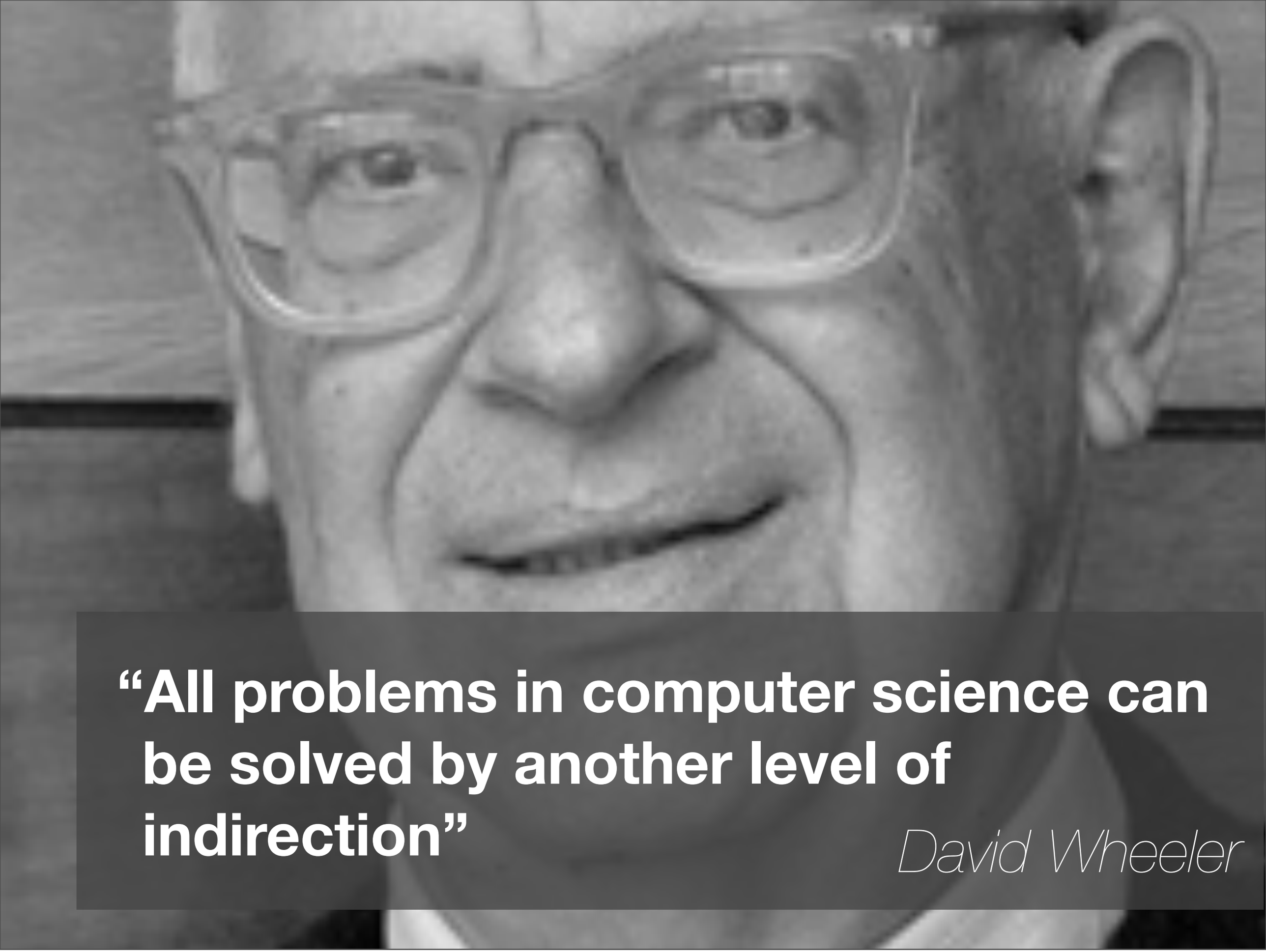
```python
class ConcreteClass(AbstractClass):

    def _part1(self):
        print("About to perform action")

    def _part2(self):
        perform_action()

    def _part3(self):
        print("Action performed!")
```

Template Method




```
class AbstractClass:

    def template_method(self):
        self._part1()
        self._part2()
        self._part3()

    def _part2(self):
        raise NotImplementedError("Override this method")

    def _part3(self):
        # Optionally override this
        print("Done!")
```

```
class ConcreteClass(AbstractClass):

    def _part1(self):
        print("About to perform action")

    def _part2(self):
        perform_action()

    def _part3(self):
        print("Action performed!")
```



“All problems in computer science can be solved by another level of indirection”

David Wheeler

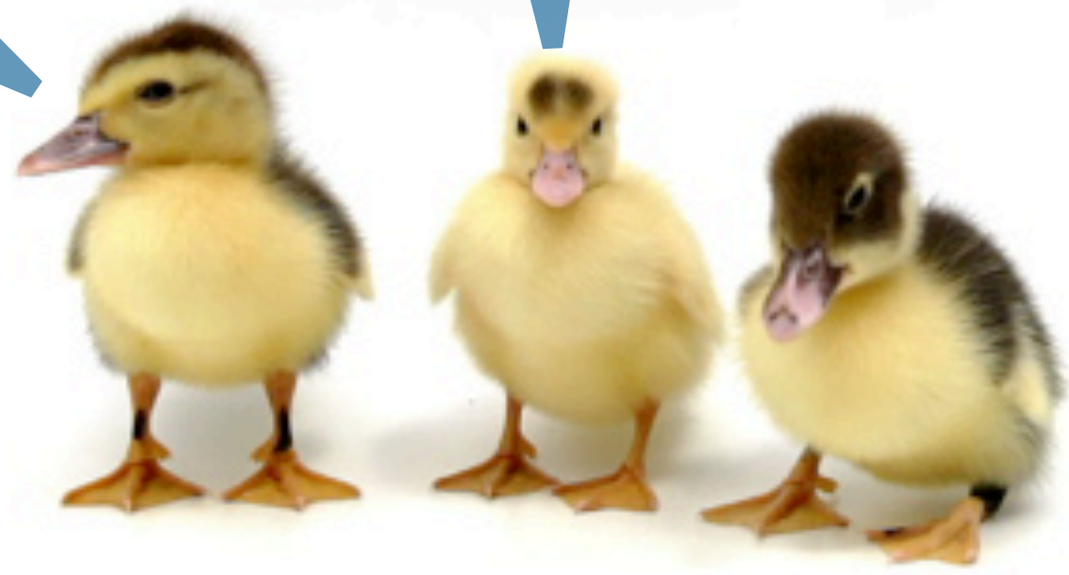
Duck Tails

"All problems in computer science can be solved by another level of indirection"

David Wheeler

"...except for the problem of too many layers of indirection"

Kevlin Henney



Summary: Properties and Class Methods

- Shown the differences between class attributes and instance attributes.
- Illustrated how class attributes are shared amongst all instances.
- Demonstrated how to access class attributes by fully qualifying with the class name.
- Shown that attempting to assign to a class attribute through the instance reference `self` actually creates a new instance attribute.
- Used `@staticmethod` to decorate methods within a class which depend on neither class nor instance objects.
- Used `@classmethod` to decorate methods which operate on the class object.
- Implemented named constructors using class methods.
- Shown how inheritance interacts with static and class methods.
- Demonstrated class- and static-method polymorphism by invoking through the `self` instance.

Summary: Properties and Class Methods

- Shown how to create read-only and read-write properties using the `@property` decorator.
- Used the *template method* design pattern to override properties without recourse to esoteric syntactical constructs.