

Properties and Class Methods

Robert Smallshire
Twitter: @robsmallshire
rob@sixty-north.com



Presenter

Austin Bingham
Twitter: @austin_bingham
austin@sixty-north.com



pluralsight 
hardcore dev and IT training



class attributes

versus

instance attributes



CARGO
CONTAINER

CARGO
CONTAINER

CARGO
CONTAINER

CARGO
CONTAINER

CARGO
CONTAINER

CG 044 283 01 MAX LOAD TARE HALFW 21004501

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!



Assignment to attributes

`self.attr = something`

always creates an instance
attribute, never a
class attribute



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

MSCU

M. G. W.
TARE

NET
CU.CAP.

98 8

22G1

HJCU
8281

MAX GROSS
TARE
PAYLOAD
CUBE

30,490 KG
67,200 LB
22,490 KG
49,400 LB
28,240 KG
62,260 LB
22.1 CBM
116.6 CFT

HANJIN
SHIPPING
CO.,LTD.

CAXU
629
22G

MAX GROSS
TARE

NET
CU.CAP.

CBM

NET
CU.CAP.

NET
CU.CAP.

MAX
GROSS

NET
CU.CAP.

NET
CU.CAP.

MAX
GROSS

NET
CU.CAP.

NET
CU.CAP.

L

ISO 6346 BIC code

CSQU3054383

Owner

code

Serial

number

Category

identifier

Check

digit



static methods with inheritance





class methods with inheritance



encapsulation using the `@property` decorator



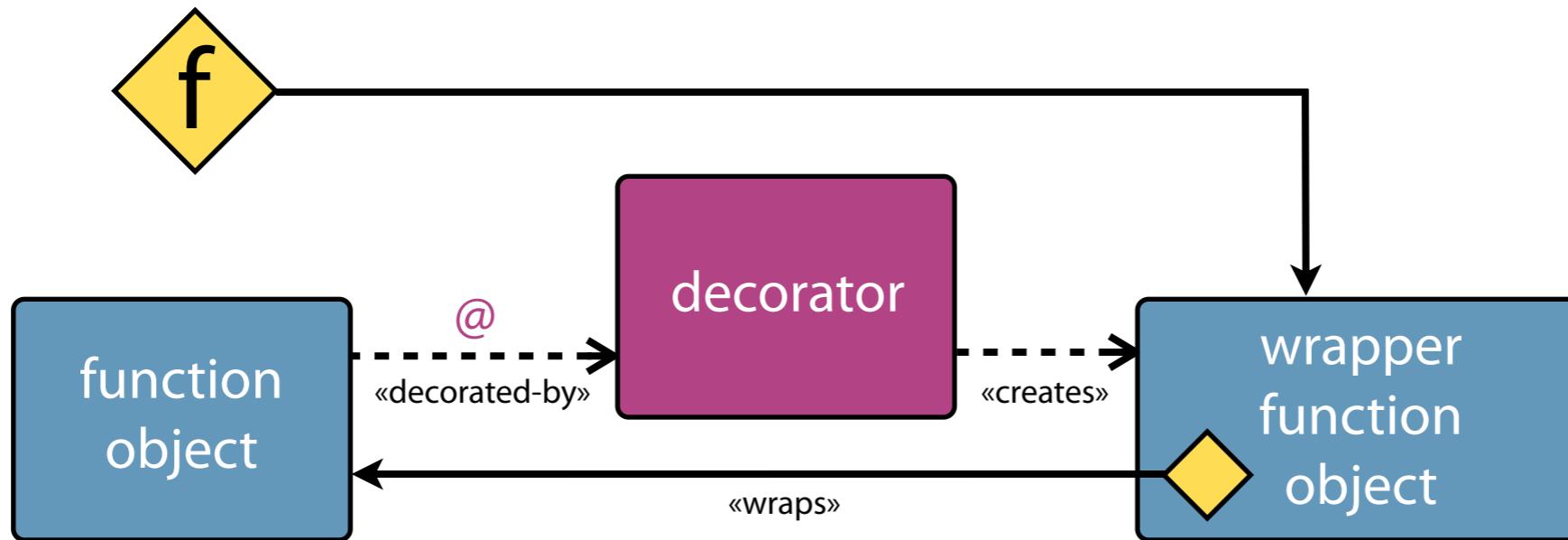
python™

≠



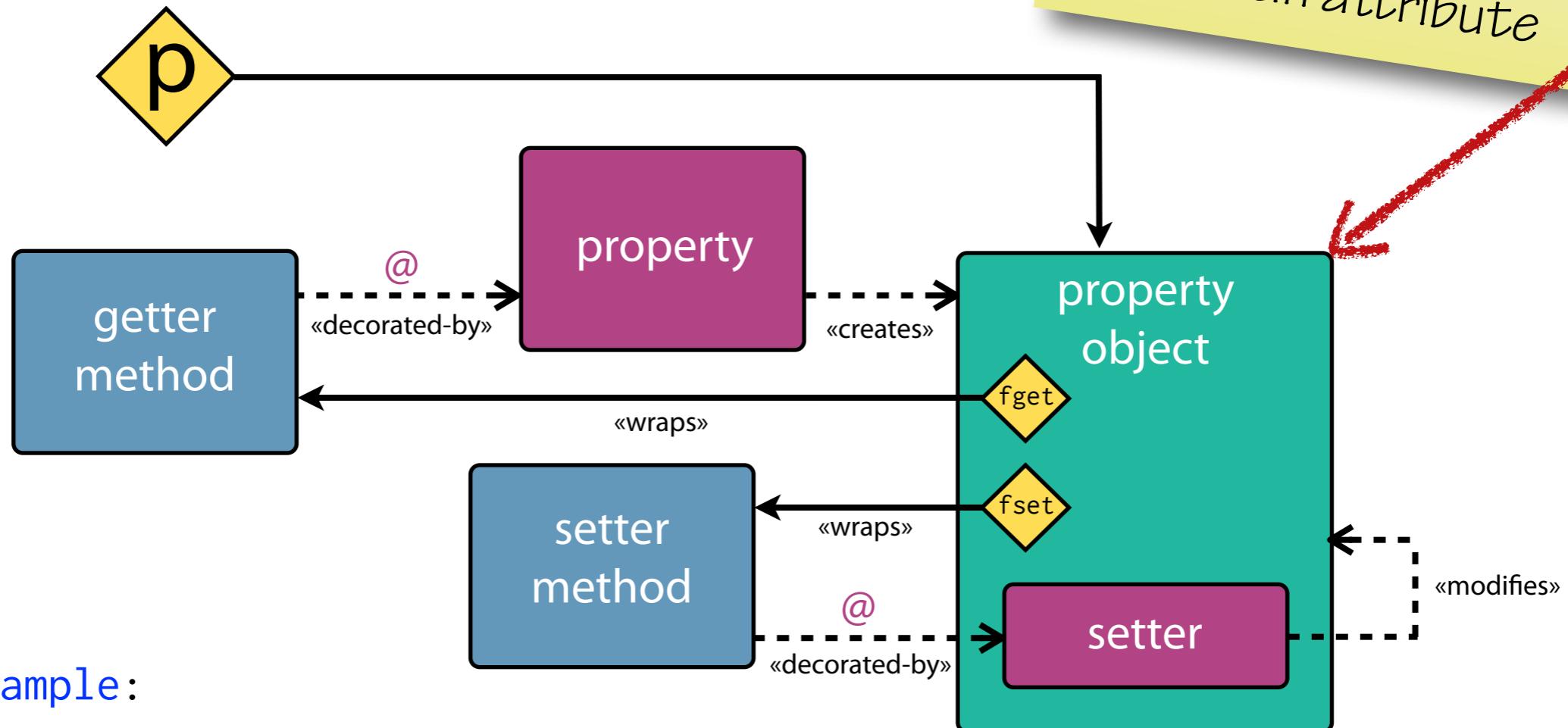
Java™

Decorator recap



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

The @property decorator



class Example:

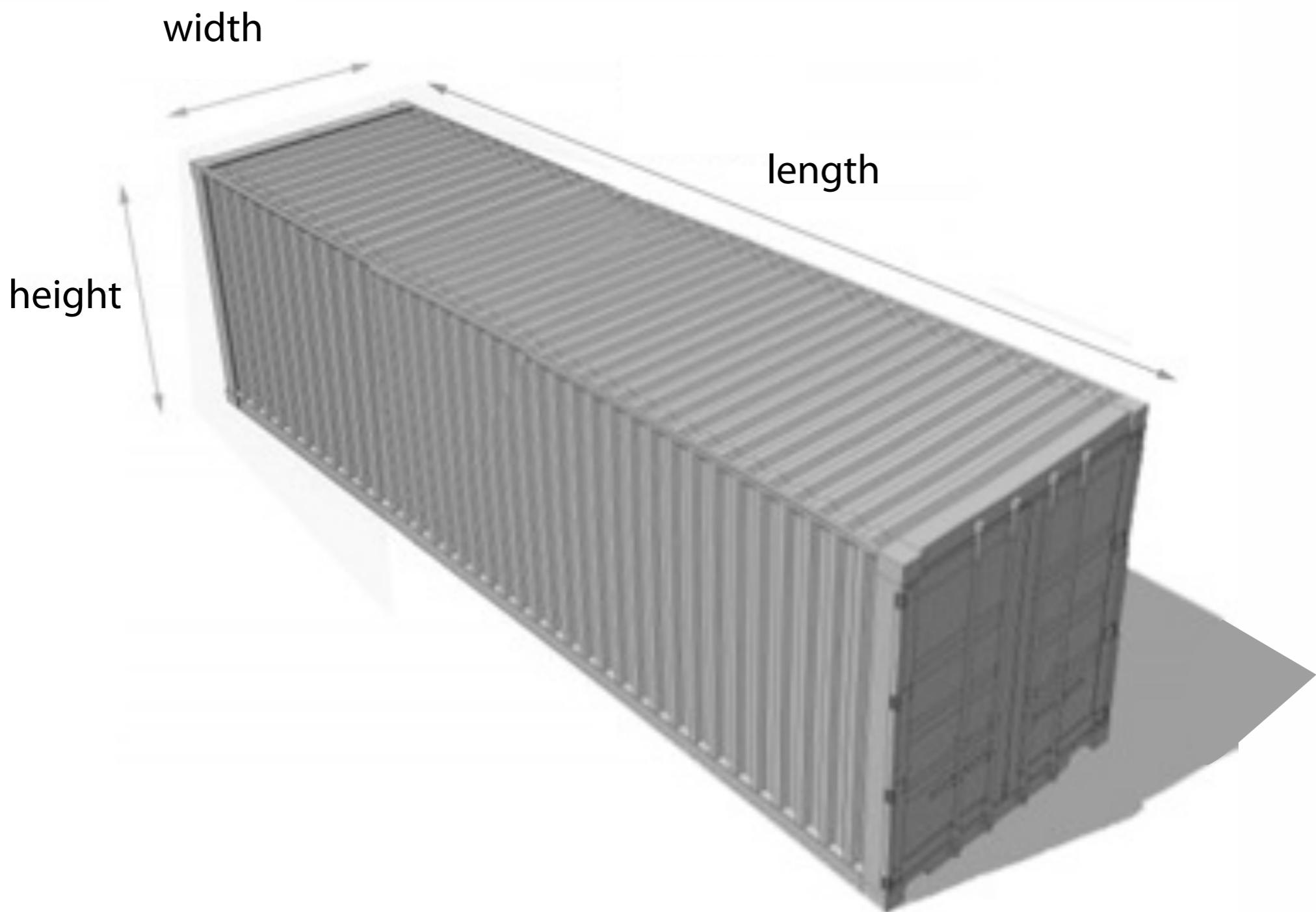
```
@property  
def p(self):  
    return self._p
```

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





Inheritance interaction with the `@property` decorator



 Carrier

THINLINE

2
7
1
1
0
0
4
4



Chained relational operators

a < b < c

equivalent to

(a < b) and (b < c)

Duck Tails

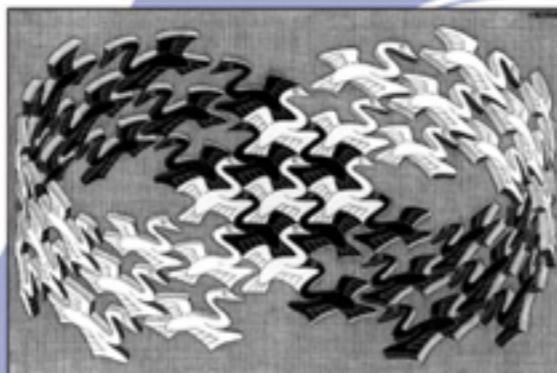
Properties and the Template Method Pattern



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

TEMPLATE METHOD

TEMPLATE METHOD

325

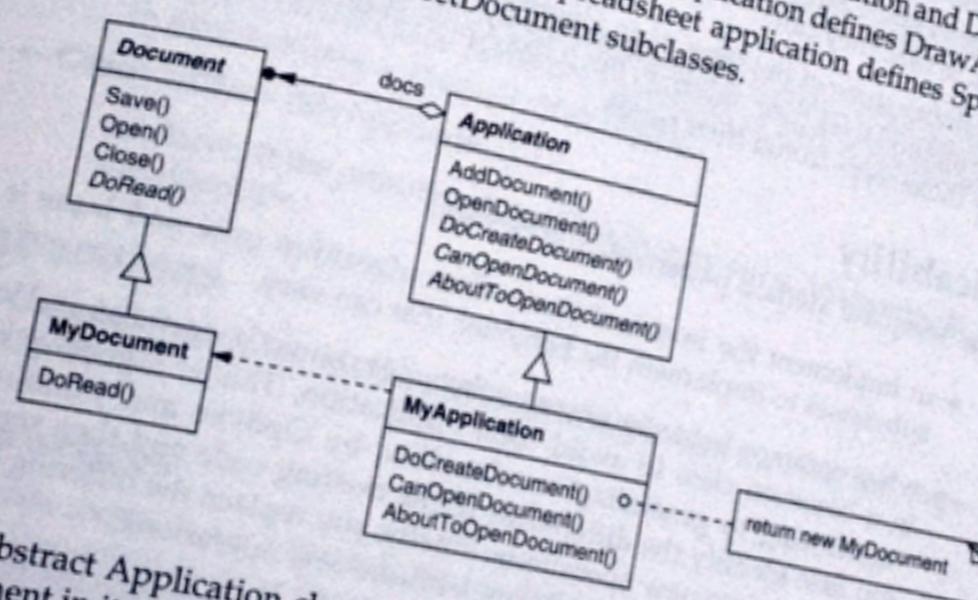
Class Behavioral

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Motivation

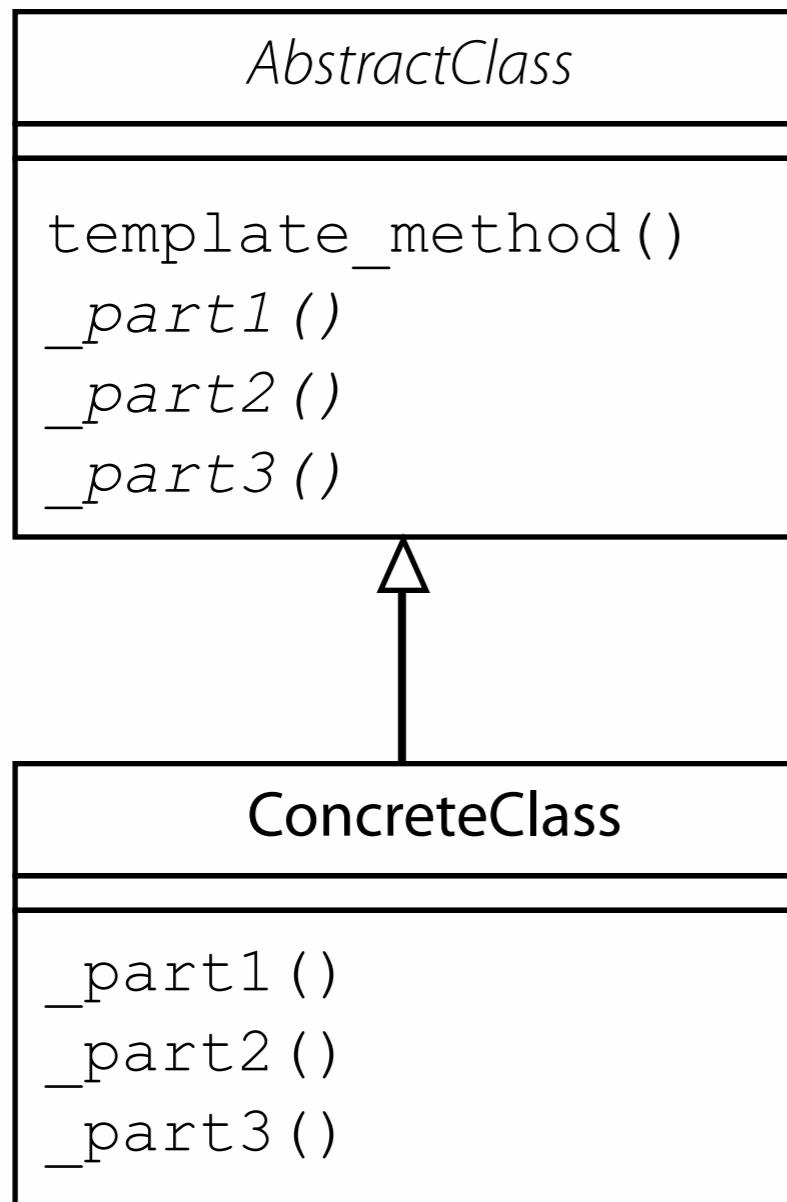
Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file. Applications built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

TEMPLATE METHOD



```
class AbstractClass:

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

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

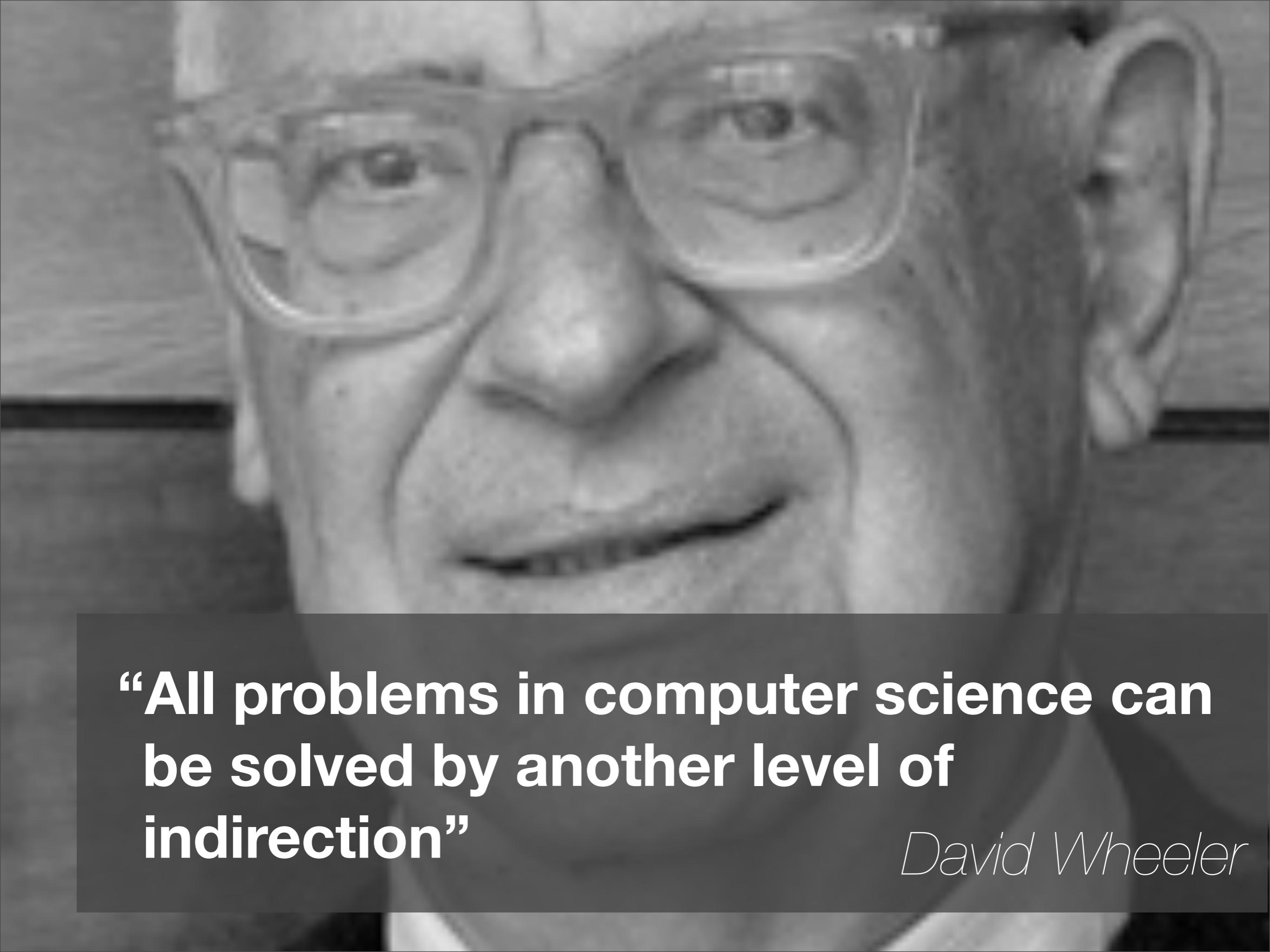
    def _part3(self):
        # Optionally override this
        print("Done!")

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

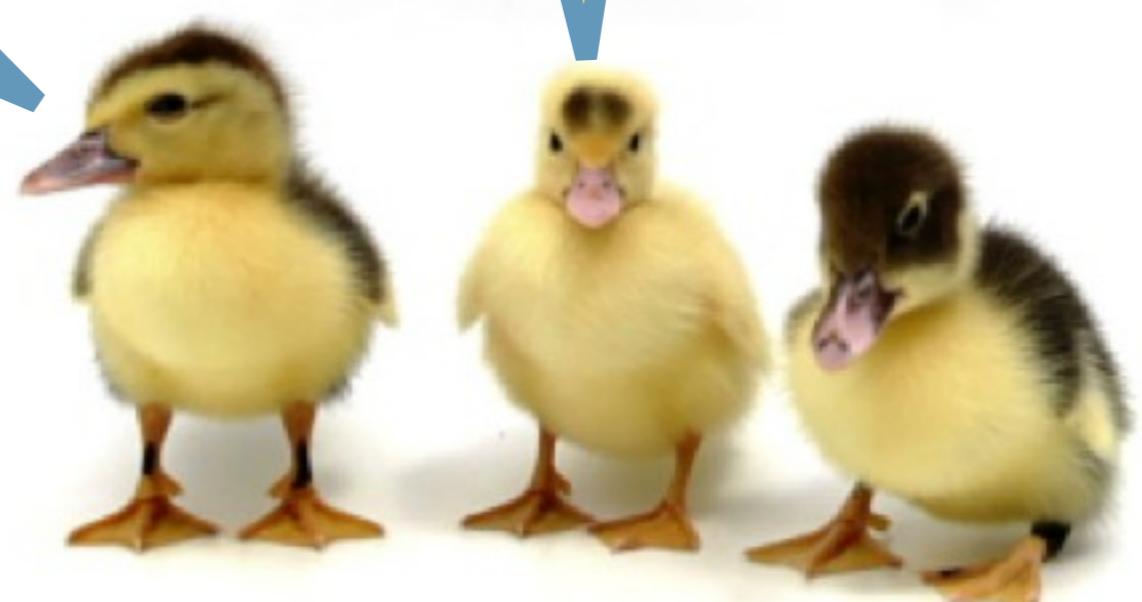
Duck Tails

"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





Properties and Class Methods

```
class Foo:

    a_class_attribute = 0

    def __init__(self):
        self.an_instance_attribute = 42
        Foo.a_class_attribute = 64
        self.a_class_attribute = "actually makes an instance attribute"

    @staticmethod
    def a_static_method():
        return "No args, just lives in the class"

    @classmethod
    def a_class_method(cls):
        cls.a_class_attribute += 1

    @classmethod
    def a_named_constructor(cls):
        return cls()

    @property
    def a_property(self):
        return self.an_instance_attribute

    @a_property.setter
    def a_property(self, value):
        self.an_instance_attribute = value
```