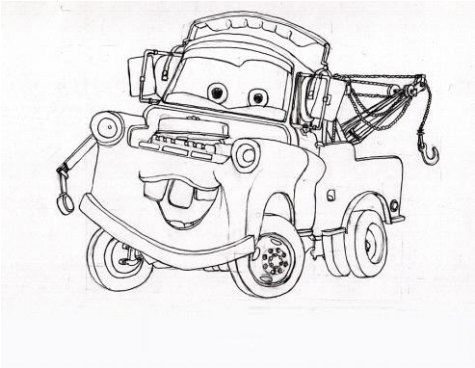
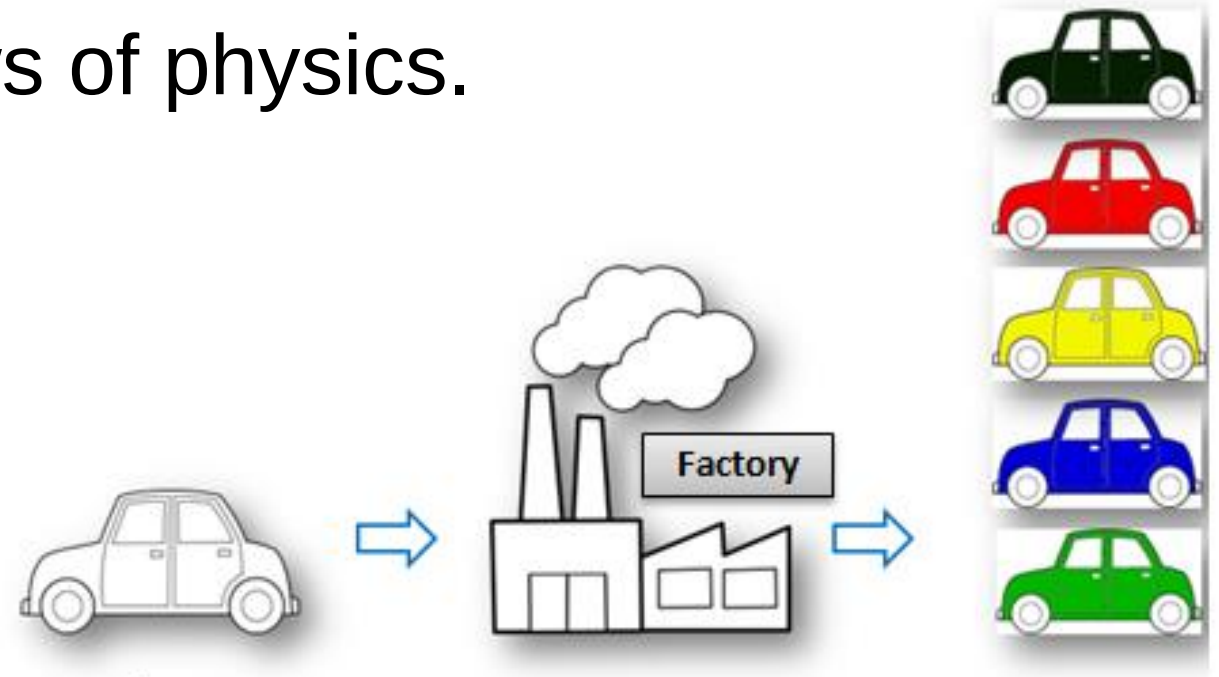
SCNU - UOA
OBJECT-ORIENTED PROGRAMMING

Design Pattern

Patterns in Engineering

Car designers don't design cars from scratch using the laws of physics.





Patterns

- Christopher Alexander, American Architect
- Patterns serve as an aid to design cities and buildings.

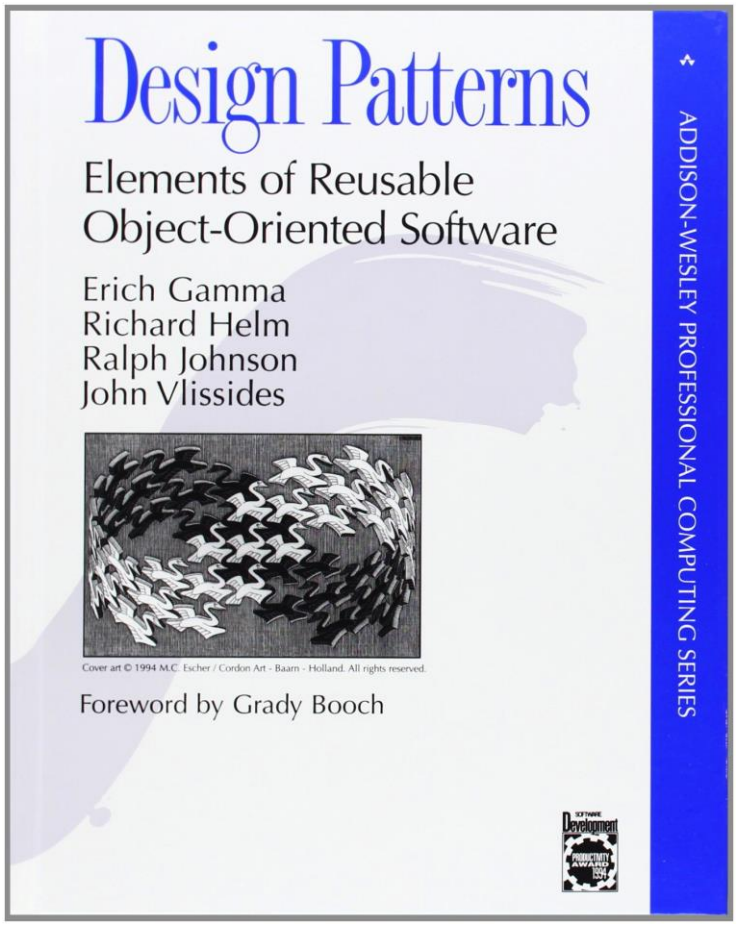
A pattern is a recurring solution to a standard problem.
模式是对标准问题的重复解决方案

模式→重复使用

The Gang of Four (GoF) Book

- ***Design Patterns: Elements of Reusable Object-Oriented Software***, Addison-Wesley Publishing Company, 1994.

Written by the ”Gang of Four”:



Dr. Erich Gamma

Dr. Richard Helm

Dr. Ralph Johnson

Dr. John Vlissides

The Gang of Four (GoF) Book

- Book catalogues ***23 different patterns*** as solutions to different classes of problems, in C++ & Smalltalk languages.
- The problems and solutions are broadly applicable, used by many people over many years.

It defines its 23 patterns in three categories:

- **Creational Patterns:** Deal with initializing and configuring classes and objects. **创建型模式**
- **Structural Patterns**: Deal primarily with the static composition and structure of classes and objects.**结构型模式**
- **Behavioural Patterns**: Deal with dynamic interactions among societies of classes and objects. How they distribute responsibility.**行为型模式**

GoF Patterns

- *Creational Patterns*
  - **Factory Method**
- *Structural Patterns*
  - **Decorator**
- *Behavioural Patterns*
  - **Iterator**

Iterator Pattern

迭代器模式

Iterator Pattern

```
1  alist = [1,2,3,]
2  for e in alist:
3      print(e)
```

列表、元组、字典：
可迭代对象

```
1  adict = {'a':1, 'b':2, 'c':3}
2  for e in adict:
3      print(e)
```

```
1  atuple = (1,2,3)
2  for e in atuple:
3      print(e)
```

Iterator Pattern

iter() next() method 迭代器方法

The iter() method is used to convert to convert an iterable to iterator.

This presents another way to iterate the container i.e access its elements. It uses next() for accessing values.

```
1 alist_iter = iter(alist)
```

```
1 next(alist_iter)
```

```
1
```

```
1 next(alist_iter)
```

```
2
```

```
1 next(alist_iter)
```

```
3
```

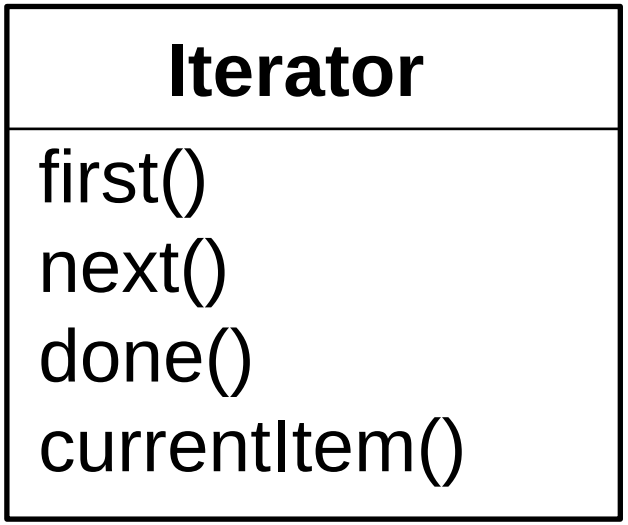
```
1 next(alist_iter)
```

```
---------------------------
StopIteration
<ipython-input-25-6637c
----> 1 next(alist_iter)
```

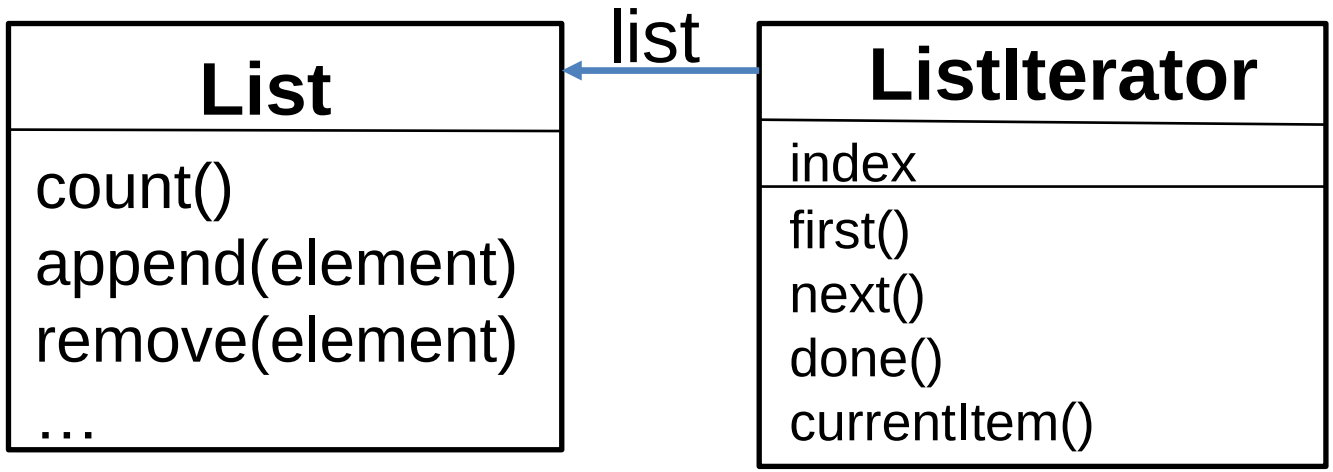
GoF Version of *Iterator*

- Generic pattern:

通用迭代器模式



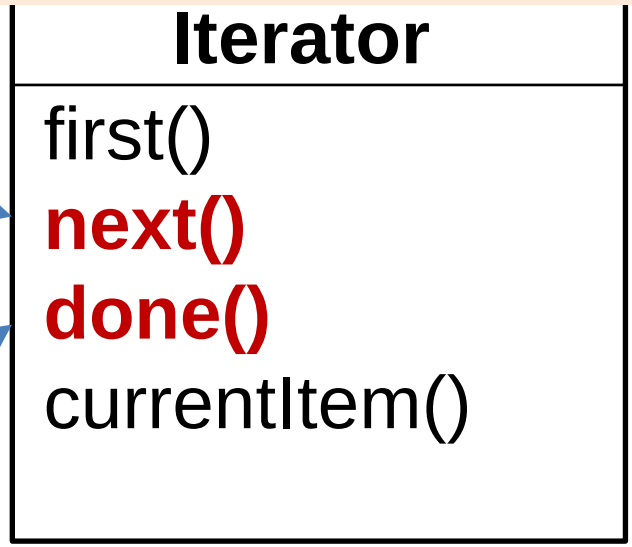
- Example case:



```
>>> x = iter([1, 2, 3])
>>> x
<listiterator
>>> next(x)
1
>>> next(x)
2
>>> next(x)
3
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Built-in function iter() – takes an iterable object and returns an iterator.

Built-in method next() on iterator gives next element.



If no more elements, iterator raises `StopIteration` exception.

Iterables vs *Iterators*

An ***ITERABLE*** is:

iterable可迭代对象；
iterator迭代器对象；

- anything that can be looped over

  (i.e. you can loop over a **string or file**)

- or anything that can appear on the right-side of a for-loop:

```
for x in iterable: ...
```

Iterables vs *Iterators*

iterable可迭代对象；

An ***ITERABLE*** is:

- or anything you can call with `iter()` that will **return an *ITERATOR***: `iter(obj)#获取迭代器`
- or an object that defines `__iter__` that returns a fresh *ITERATOR*, or it may have a `__getitem__` method suitable for indexed lookup.

Iterables vs *Iterators*

- An ***ITERATOR*** is:

iterator迭代器对象；

  - an object with state that **remembers** where it is during iteration;
  - with a `__next__` method that:
    1. returns the next value in the iteration,
    2. updates the state to point at the next value,
    3. signals when it is done by raising `StopIteration`

Iterables vs *Iterators*

- An ***ITERATOR*** is:

iterator迭代器对象；

  - and that is self-iterable (meaning that it has an `__iter__` method that returns **`self`**).

The builtin function **`next()`** calls the `__next__` method on the object passed to it.

内置函数next()调用传递给迭代器对象的__next__方法

Iterables vs *Iterators*

An ***ITERABLE*** is: iterable可迭代对象；

- anything that can be looped over

An ***ITERATOR*** is: iterator迭代器对象；

- an object with state that **remembers** where it is during iteration;

可迭代对象：任何能够被循环的对象；
迭代器对象：在迭代过程中能够记忆自身状态的对象；

```
class CapitalIterator:
    def __init__(self, string):
        self.words = string.split()
        self.index = 0
    def __next__(self):
        if self.index == len(self.words):
            raise StopIteration()
        word = self.words[self.index]
        self.index += 1
        return word.capitalize()
    def __iter__(self):
        return self
```

```
class CapitalIterable:
    def __init__(self, string):
        self.string = string
    def __iter__(self):
        return CapitalIterator(self.string)
```

```
iterable=CapitalIterable('south china normal')

iterator = iter(iterable)
while True:
    try:
        print(next(iterator))
    except StopIteration:
        break
```

Python gives us a much simpler way of doing this:

```
for i in iterable:
    print(i)
```

South China Normal

Although this form doesn't look at all object-oriented - there is a **LOT** of *OO cleverness* going on behind the scenes!

Generators Python的迭代器模式

- Python function that behaves like an *iterator*.
- Built around use of the **`yield`** keyword:
  1. Similar to **`return`** statement
  2. **Exits** the function and **returns** a value.
  3. Next time function is called (using `next()`) it starts again from where it left off (on the line after the yield statement)!

Generators

- Python wraps up functions containing **`yield`** - to create a ***generator object***.
- If we don't need a list/set/dictionary - generators are more efficient than comprehensions.

```
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1
```

Generators

```
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1
```

```
>>> gen = firstn(3)
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most
recent call last):
  File "python", lin
1, in <module>
StopIteration
```

airplane

automobile

bird

cat

deer

dog

frog

horse

ship

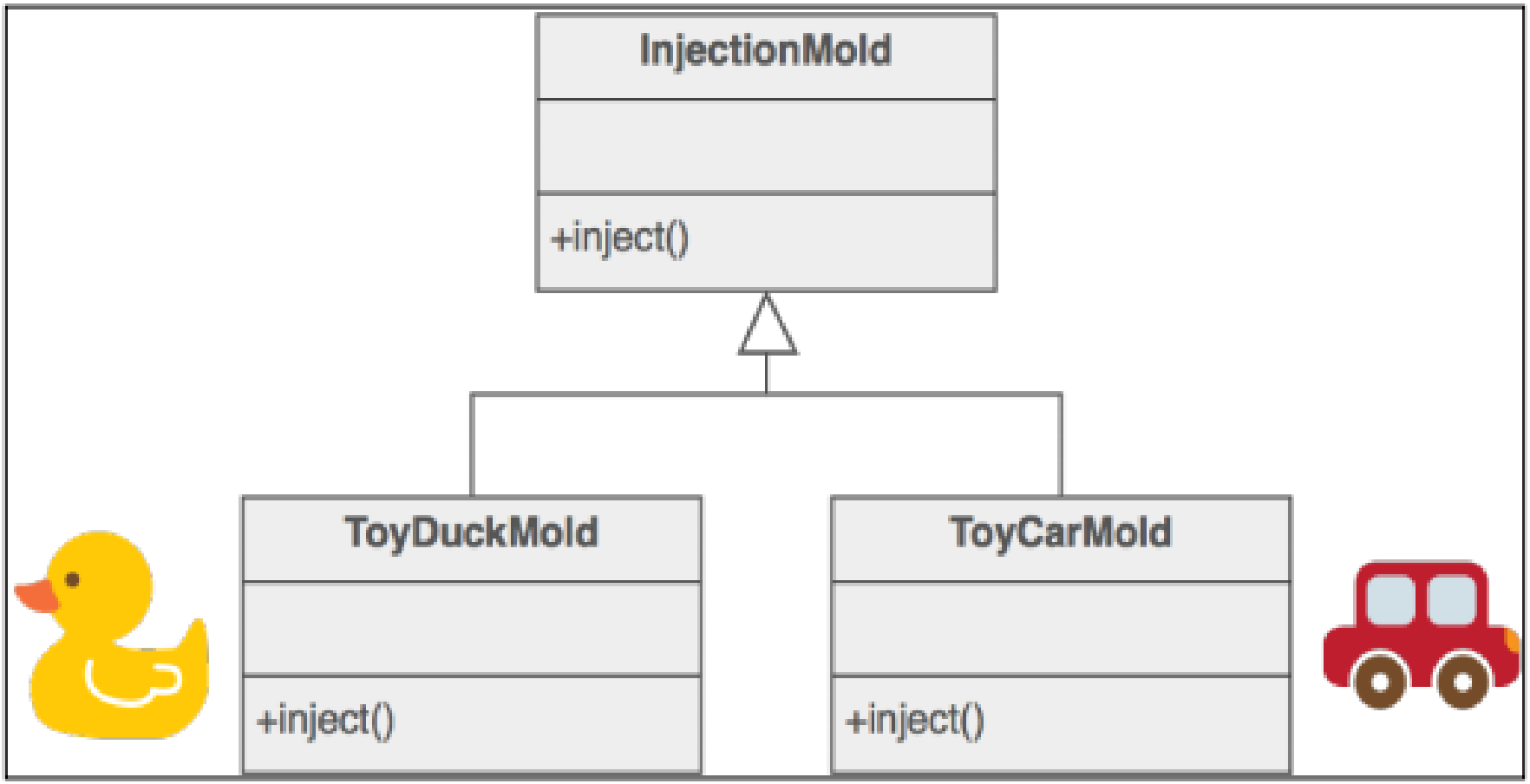
truck

GoF Patterns

- *Creational Patterns* 创建型
  - Factory Method
- *Structural Patterns* 结构型
  - Decorator
- *Behavioural Patterns* 行为型
  - Iterator

Factory Pattern

Factory Pattern



Factory Pattern

- **Factory Method**

  In the Factory Method, we execute a single function, passing a parameter that provides information about *what* we want. We are not required to know any details about *how* the object is implemented and *where* it is coming from. 不需要知道对象是如何实现

Factory Pattern

- Functional programming 函数式编程

```
1  def role(rtype):
2      if rtype is 'player':
3          print('It is a player!')
4      elif rtype is 'enemy':
5          print('It is an enemy!')
6      else:
7          print('Unkown!')
8
9  role('player')
```

任务：修改角色

函数式编程：麻烦，修改过程容易出错；

```
It is a player!
```

Factory Pattern

- 一般面向对象做法（1）

```
1  class role:
2      def show(self):
3          print('Unkown!')
4
5  class player(role):
6      def show(self):
7          print('It is a player!')
8
9  class enemy(role):
10     def show(self):
11         print('It is an enemy!')
12
```

Factory Pattern

- 一般面向对象做法（2）

```
13  class factory:
14      @classmethod
15      def build(self, rtype):
16          if rtype is 'player':
17              product = player()
18          elif rtype is 'enemy':
19              product = enemy()
20          else:
21              product = role()
22          return product
```

Factory Pattern

- 一般面向对象做法（3）

```
13  class factory:
14      @classmet
15      def build(s
16          if rtype
17              produ
18          elif rtyp
19              produ
20          else:
21              produ
22          return p
```

```
r1 = factory.build('player')
r1.show()

r2 =  factory.build('player')
r2.show()
```

```
player!
player!
```

Factory Pa

- 一般面向对象做

```
13  class factory:
14      @classmetho
15      def build(sel
16          if rtype is
17              product
18          elif rtype i
19              product = enemy()
20
21
22
```

```
r1 = factory.build('player')
r1.show()

r2 = factory.build('player')
r2.show()
```

```
player!
player!
```

优点：如果要换敌人，只需要换参数

缺点：构造角色大量重复factory.build，如果要修改成敌人角色，需要重复修改字符串，容易出错

Factory Pattern

- A better design: **Factory Pattern**

```
1  class role:
2      def show(self):
3          print('Unkown!')
4
5  class player(role):
6      def show(self):
7          print('It is a player!')
8
9  class enemy(role):
10     def show(self):
11         print('It is an enemy!')
12
```

更好的方式：

工厂模式

保留角色类不变

- A better design: **Factory Pattern**

```
class factory:
    def build(self):
        pass

class player_factory(fact
    def build(self):
        return player()

class enemy_factory(factory):
    def build(self):
        return enemy()
```

```
factory1 = player_factory()

r1 = factory1.build()
r1.show()
r2 = factory1.build()
r2.show()
```

```
player!
player!
```

工厂类抽象化——
派生出player工厂、enemy工厂

- A better design: **Factory Pattern**

```
class factory:
    def build(self):
        pass

class player_factory(factory):
    def build(self):
        return player()

class enemy_factory(factory):
    def build(self):
        return enemy()
```

```
factory1 = player_factor

r1 = factory1.build()
r1.show()
r2 = factory1.build()
r2.show()
```

```
player!
player!
```

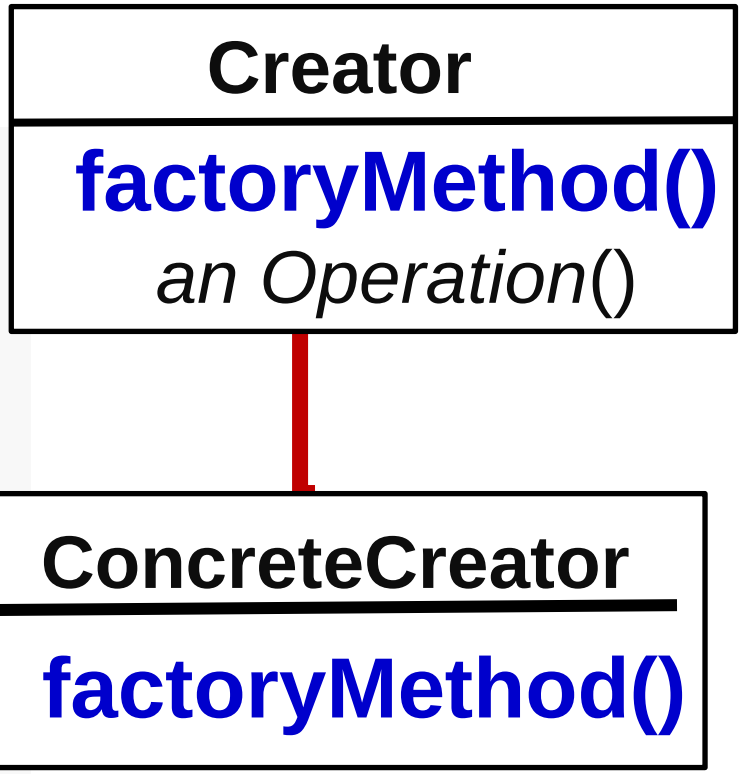
更好的面向对象设计：工厂模式

Factory Pattern

```
class factory:
    def build(self):
        pass

class player_factory(factory):
    def build(self):
        return player()

class enemy_factory(fac
    def build(self):
        return enemy()
```

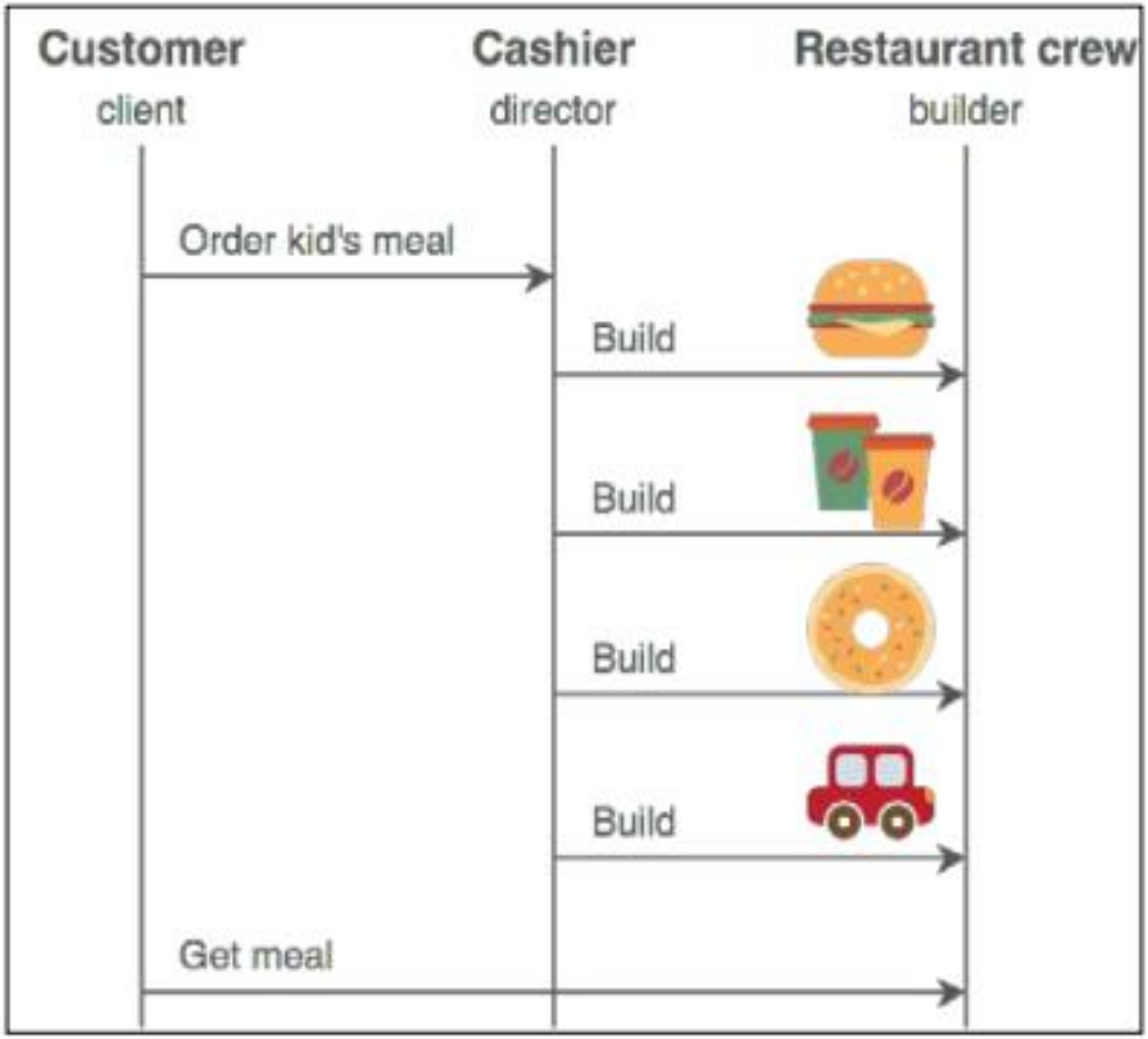


抽象工厂：定义接口build
实体工厂：实现接口

统一接口，模块化开发，
分工解耦，职责分离；

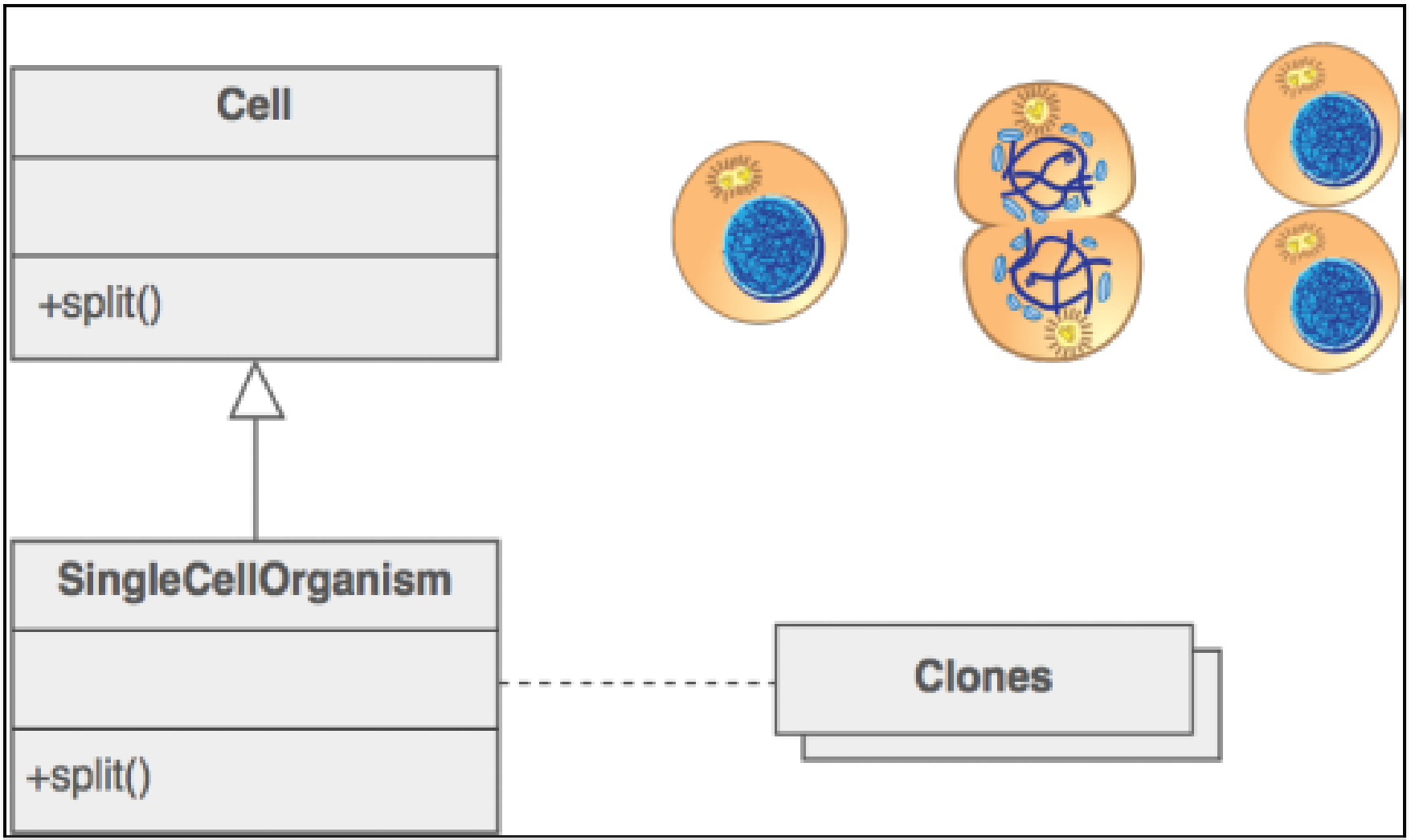
Builder Pattern

Builder Pattern



Prototype **Pattern**

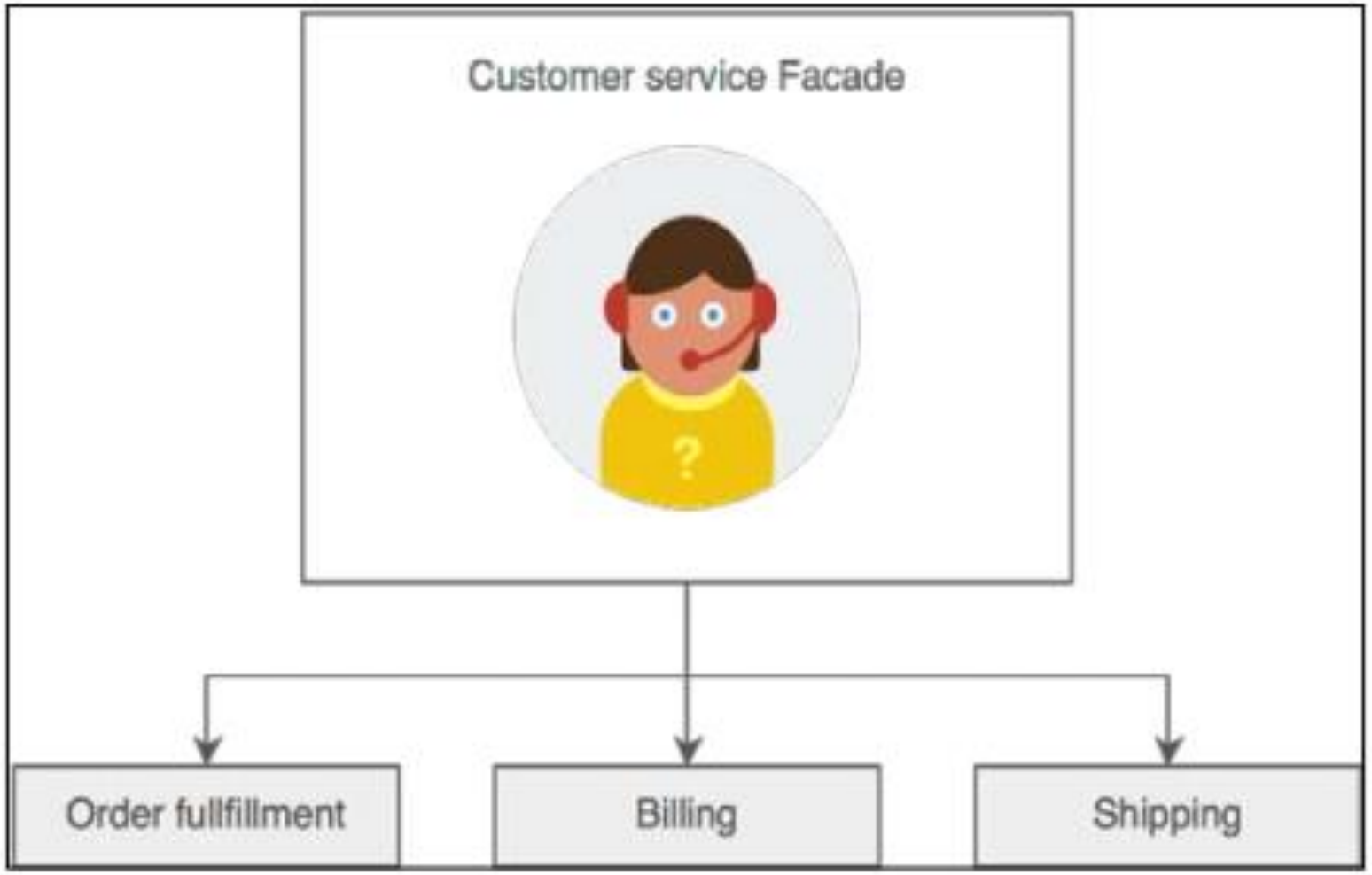
Prototype Pattern



Facade Pattern

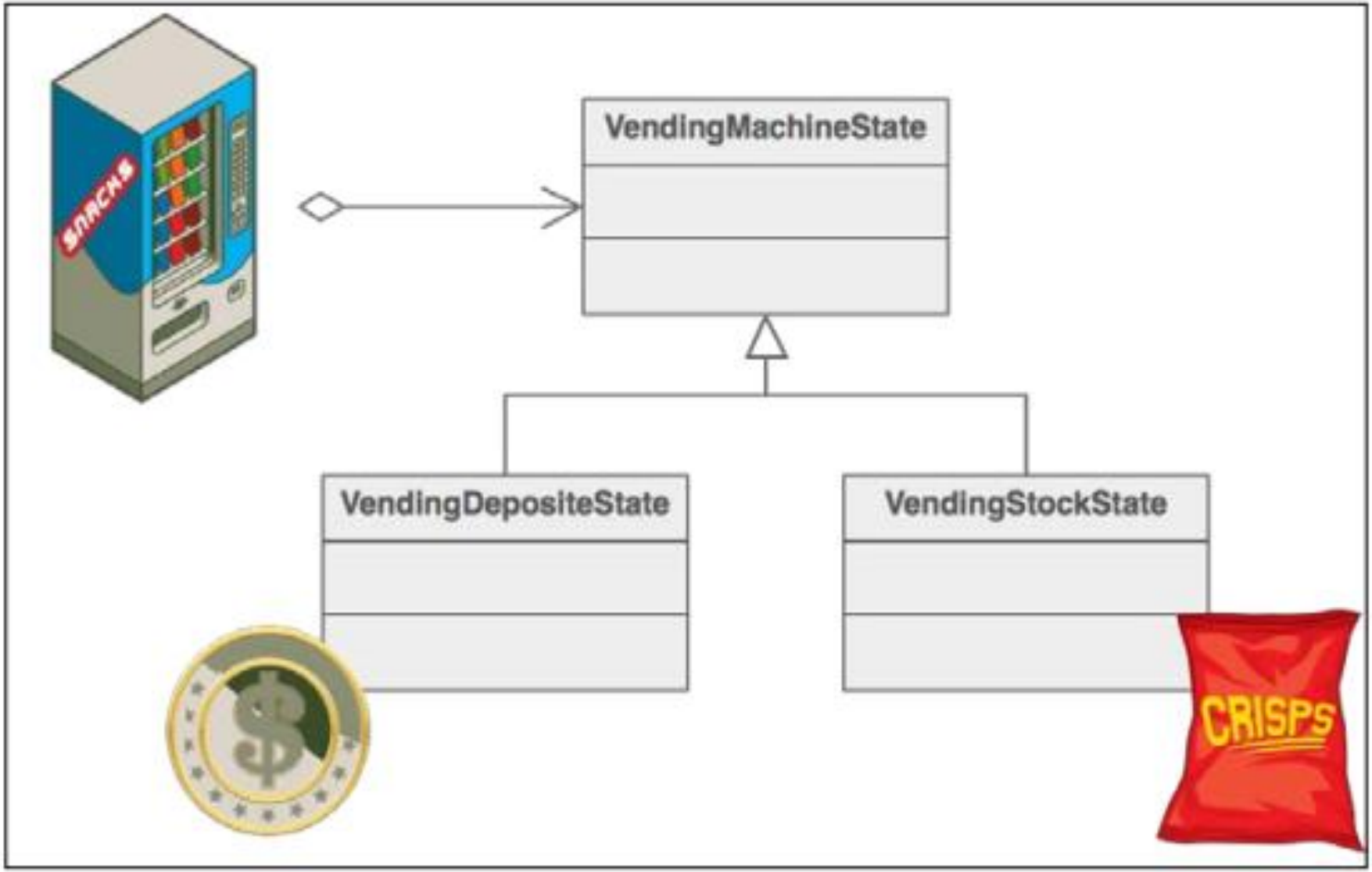
外观模式 or 享元模式

Facade Pattern



State Pattern

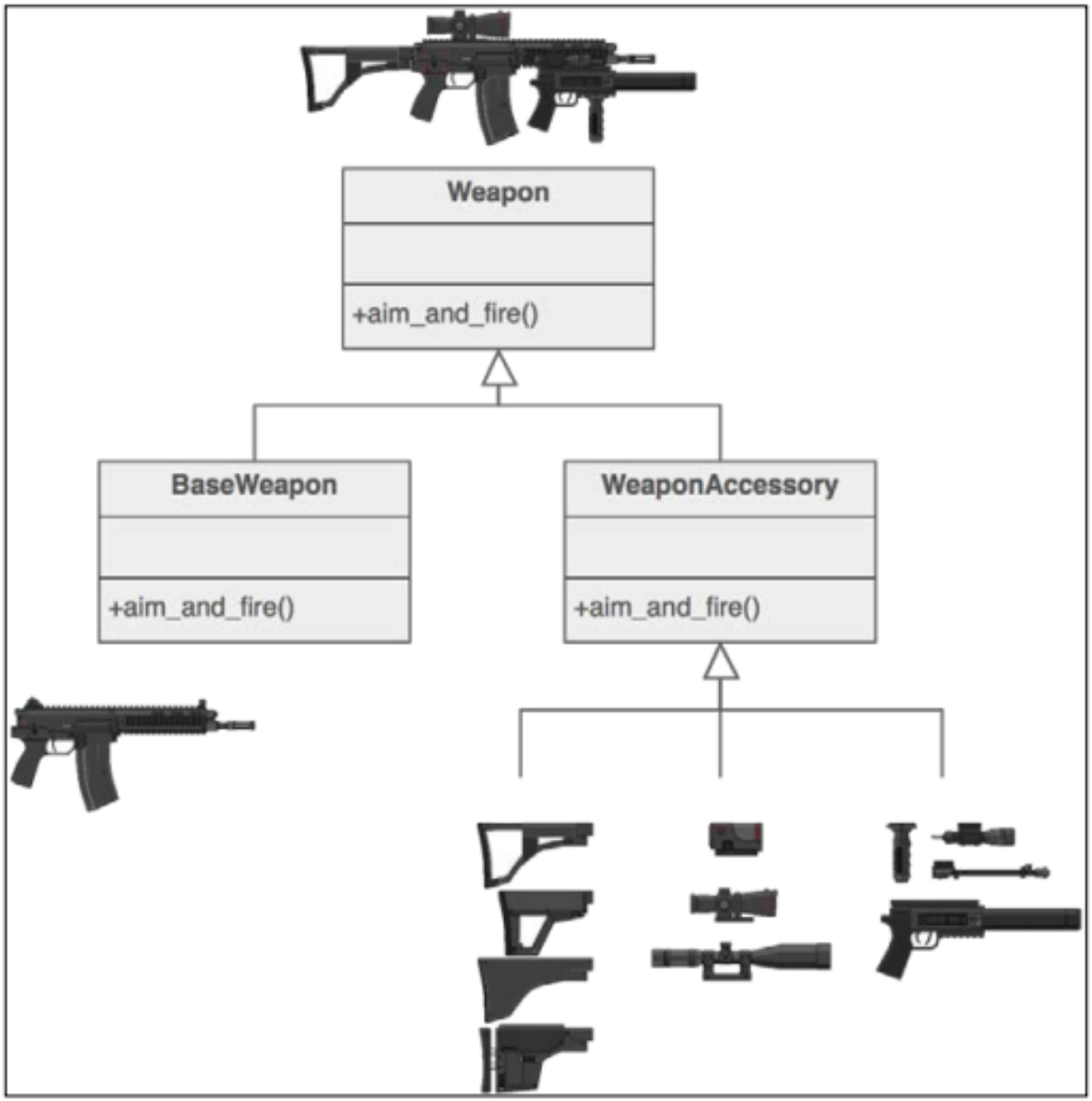
State Pattern



Decorator pattern

装饰器模式

Decora


Weapon
+aim_and_fire()
BaseWeapon
+aim_and_fire()
WeaponAccessory
+aim_and_fire()

Decorator

```
def foo():
    print('foo function')

def new_foo():
    print('testing')
    print('foo function')

foo() #原函数
print('*'*20)
new_foo()#新函数
```

#How to 'decorate' a function?

```
foo function
********************

testing
foo function
```

Decorator

```
def decorator(func):
    print('testing')
    func()

decorator(foo)
```

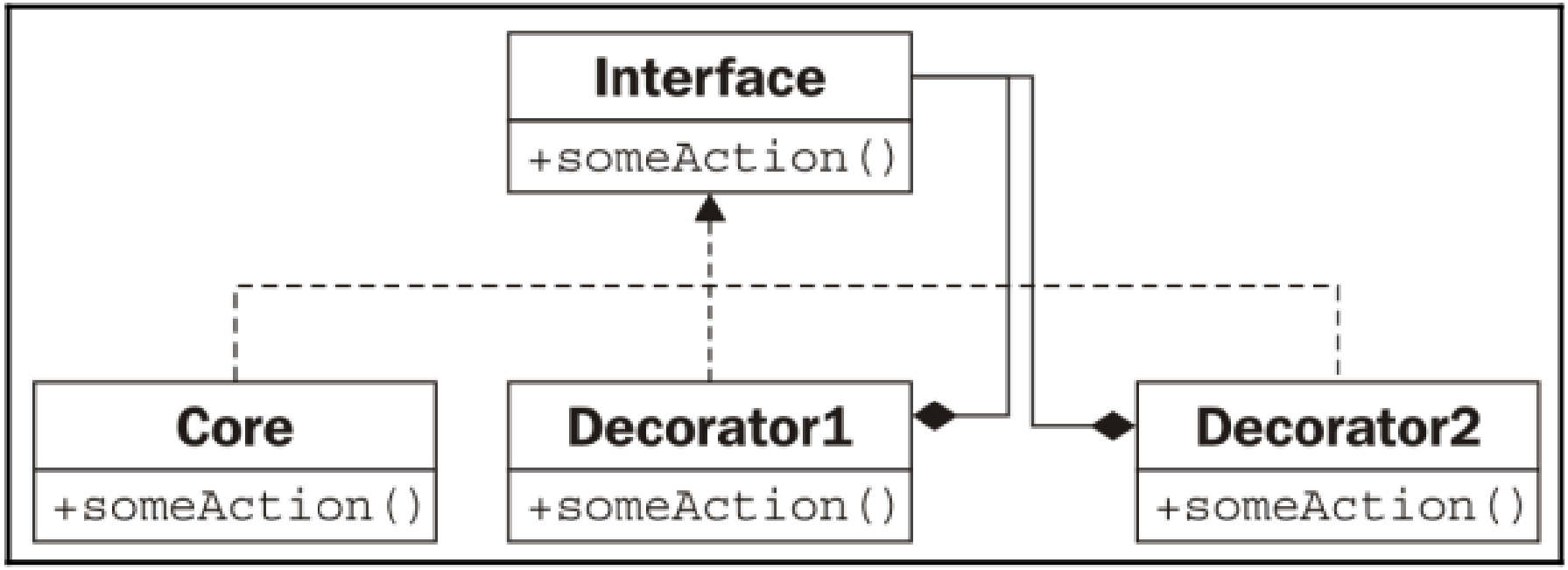
函数名-传参

函数名-调用

简单装饰

```
testing
foo function
```

Decorator pattern



Decorator pattern

```
class person:
    def __init__(self, name):
        self.name = name
        self.clothes = None


def Finery(obj):
    if not obj.clothes:
        print('{} is wearing {}!'.format(obj.name, '
    else:
        print('{} is wearing {}!'.format(obj.name, c
```

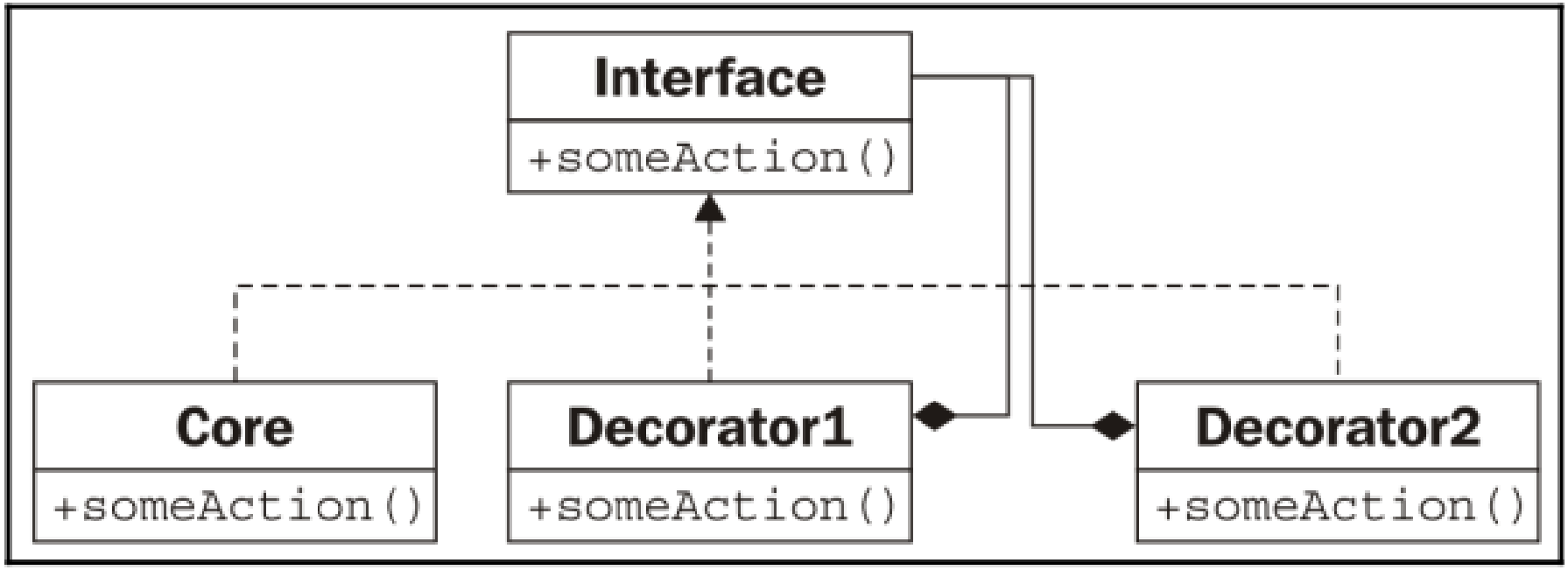
Decorator pattern


```
def decorator(func, clothes):
    def deco(obj):
        obj.clothes = clothes
        func(obj)
    return deco
```

```
18 Andy = person('Andy')
19 Finery(Andy)
20 print('*********')
21 Tshirt = decorator(Finery, 'Tshirt')
22 Tshirt(Andy)
23 print('#######')
24 Suit = decorator(Finery, 'Suit')
25 Suit(Andy)
```

```
Andy is wearing Nothing!
*********
Andy is wearing Tshirt!
#######
Andy is wearing Suit!
```

Decorator pattern



Decorator

- 函数也是对象，可调用对象。
- 函数可作为普通变量、参数、返回值等。
- 高阶函数：
  1. 接受一个或多个函数作为参数
  2. 输出一个参数

Decorator

```
def play():
    print('Hello')
#区别play( )和play
```

- **play( )：运行play函数**
- **play是函数对象**→作为变量、参数、返回值等

Decorator

- 闭包的概念：当某个**函数被当成对象返回**时，**夹带了外部变量**，就形成了一个闭包。

```
def make(msg):
    def printer():
        print(msg)  # 外部变量
    return printer # 返回函数

printer = make('Hello!')
printer()
```

```
Hello!
```

```
drink = make('Cola')
drink()
```

Decorator

- 闭包的应用：装饰器

```
def decorator(func): #被装饰函数名func-
    def defunc():          #装饰后函数
        print('testing')   #添加装饰
        func()             #运行被装饰函数
    return defunc          #return装饰后函数名
```

```
def decorator(func): #被装饰函数名func-
    def defunc():          #装饰后函数
        print('testing')   #添加装饰
        func()             #运行被装饰函数
    return defunc          #return装饰后函数名

def foo():
    print('foo function')

foo = decorator(foo)#新函数
foo()
```

```
testing
foo function
```

Python Decorators

- Python has its own "***decorators***" - these are not the same as the GoF *Decorator* pattern.
  - A decorator in Python is any callable Python object that is used to modify a function or a class.

#Python的@装饰符可以修改函数、类。

Python Decorators

No need to type

```
foo = our_decorator(foo)
```

@ - decorator syntax

#Python的@装饰符可以修改函数、类。

```
def decorator(func): #被装饰函数名func--无括号
    def defunc():        #装饰后函数
        print('testing')  #添加装饰
        func()            #运行被装饰函数
    return defunc     #return装饰后函数名

@decorator
def foo():
    print('foo function')

foo()
```

foo = decorator(foo)

```
testing
foo function
```

Python Decorators

- Application

```
1  import time
2  def func1():
3      time.sleep(3)
4      print("in the func1")
5  def func2():
6      time.sleep(2)
7      print("in the func2")
```

```python
import time
def func1():
    start_time = time.time()
    time.sleep(3)
    print("in the func1")
    stop_time = time.time()
    print("this program run %ss"%(stop_t
```

```python
def func_new1():
    start_time = time.time()
    func1()
    stop_time =time.time()
    print("this program run %ss"%(stop_time-
```

Decorator

装饰器timmer

```
1  import time
2  def timmer(func):
3      def deco():
4          start_time = time.time()
5          func()
6          stop_time = time.time()
7          print("the func run time is %ss
8      return deco
```

```python
import time
def timmer(func):
    def deco():
        start_time = time.time()
        func()
        stop_time = time.time()
        print("the func run time is %ss
    return deco
```

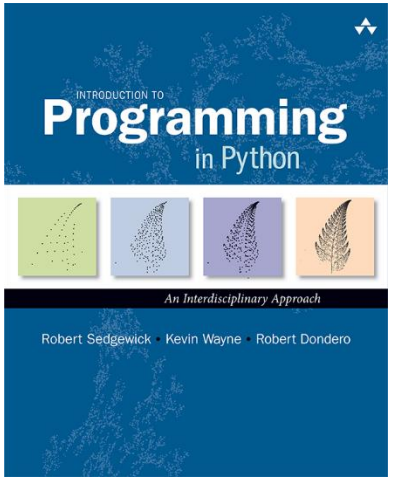
```
func1 = timmer(func1)
func1()
```

Python Decorator @

```
10  @timmer #func1=timmer(func1)
11  def func1():
12      time.sleep(3)
13      print("in the func1")
14
15  func1()
```

```
in the func1
the func run time is 3.0038695335388 2s
```

- **Name:** ***Decorator*** **(aka Wrapper 包装)**
- **Recurring Problem:** How can you augment objects with new responsibilities dynamically, when subclassing is impractical?
- **Solution:**
  1. Implement one or more decorator classes;
  2. Enclose an object within a decorator object with a similar interface.
- **Consequences:**
  - Behaviours can be added/removed at run-time; avoids subclass explosion.



OBJECT-ORIENTED PROGRAMMING

Thank you!