Object Oriented Design

Niko Wilbert TNG TECHNOLOGY CONSULTING

with contributions from Bartosz Telenczuk

Advanced Scientific Programming in Python

Summer School 2013, Zurich

Disclaimer

Good software design is a never ending learning process.
(so this is just a teaser)

Deeply rooted in practical experience.
(mostly pain and failure)

Not an exact science,
it's about tradeoffs and finding the right balance.

Note:
The examples are implemented in Python 2.7.
They do not always conform to PEP8 (due to limited space).
They are only meant to highlight certain points.

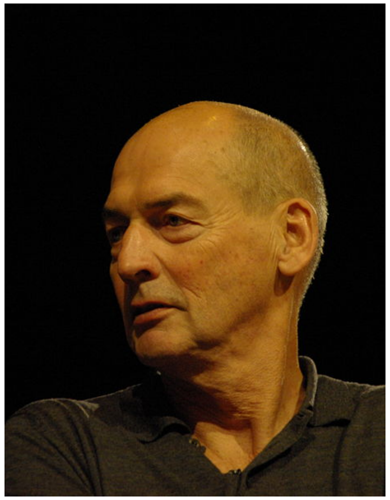
Overview

General Design Principles

The Problem of Scale

"Beyond a certain critical mass, a building becomes a BIG Building. Such a mass can no longer be controlled by a singular architectural gesture, or even by any combination of architectural gestures. The impossibility triggers the autonomy of its parts, which is different from fragmentation: the parts remain committed to the whole."

Rem Koolhaas in "Bigness and the Problem of Large"



Effective Software Design

Two simple general principles (we'll come back to this):

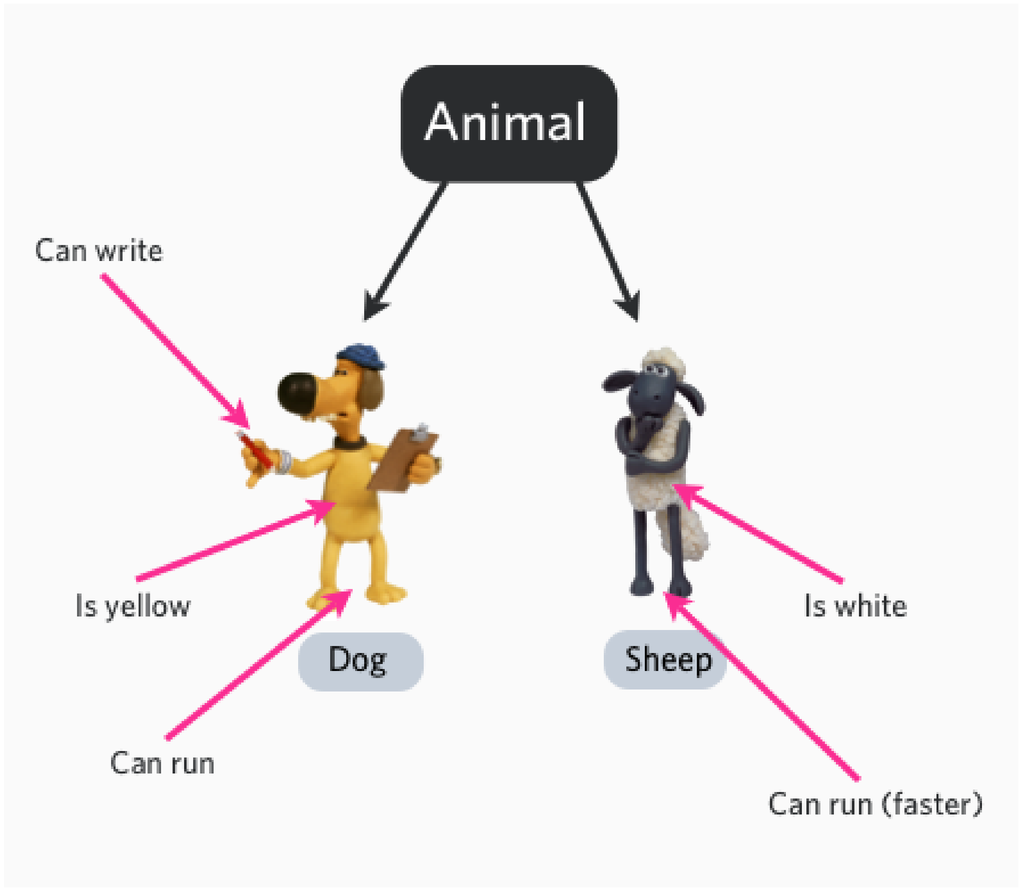
KIS (Keep It Simple)
No Overengineering, no Spaghetti code.

DRY (Don't Repeat Yourself)
Code duplication equals bug reuse.

Iterative Development: (*Agile Development*)

- One cannot anticipate every detail of a complex problem.
- Start simple (with something that works), then improve it.
- Identify emerging patterns and continuously adapt the structure of your code. (*Refactoring*, for which you want *Unittests*)

Object Oriented Programming (in Python)



Object Orientated Programming

Objects

Combine state (data) and behavior (algorithms).

Encapsulation

Only what is necessary is exposed (*public interface*) to the outside.
Implementation details are hidden to provide *abstraction*.
Abstraction should not *leak* implementation details.
Abstraction allows us to break up a large problem into understandable parts.

Classes

Define what is common for a whole class of objects, e.g.:
"Snowy **is a** dog"
= "The Snowy object is an *instance* of the dog class."
Define once how a dog works and then reuse it for all dogs.
The class of an object is its *type* (classes are *type objects*).

Object Orientated Programming (II)

Inheritance

"a dog (*subclass*) **is a** mammal (*parent* / *superclass*)"
Subclass *is derived from* / *inherits* / *extends* a parent class.

Override parts with specialized behavior and *extend* it with additional functionality.

Liskov substitution principle: What works for the parent class should also work for any subclass.

Polymorphism

Different subclasses can be treated like the parent class, but execute their specialized behavior.

Example: When we let a mammal make a sound that is an instance of the dog class, then we get a barking sound.

Object Orientation in Python

- Python is a *dynamically typed* language, which means that the type (class) of a variable is only known when the code runs.
- *Duck Typing*: No need to know the class of an object if it provides the required methods.

  "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
- Type checking can be performed via the `isinstance` function, but generally prefer duck typing and polymorphism.
- Python relies on convention and documentation instead of enforcement.

  No enforced private attributes, use a single underscore to signal that an attribute is not intended for public use (encapsulation).

Python Object Orientation Basics

- All classes are derived from `object` (*new-style classes*).

```
class Dog(object):
    pass
```

- Python objects have data and function attributes (*methods*).

```
class Dog(object):
    def bark(self):
        print "Wuff!"

snowy = Dog()
snowy.bark()  # first argument (self) is bound to this Dog instance
snowy.a = 1  # added attribute a to snowy
```

- Always define your data attributes first in `__init__`.

```
class Dataset(object):
    def __init__(self):
        self.data = None
    def store_data(self, raw_data):
        ... # process the data
        self.data = processed_data
```

Python Object Orientation Basics (II)

- Class attributes are shared across all instances.

```
class Platypus(Mammal):
    latin_name = "Ornithorhynchus anatinus"
```

- Use `super` to call a method from a superclass.

```
class Dataset(object):
    def __init__(self, data):
        self.data = data

class MRIDataset(Dataset):
    # __init__ does not have to follow the Liskov principle
    def __init__(self, data, parameters):
        # here has the same effect as calling
        # Dataset.__init__(self)
        super(MRIDataset, self).__init__(data)
        self.parameters = parameters

mri_data = MRIDataset([1,2,3], {'amplitude': 11})
```

Note: In Python 3 `super(B, self)` can be written `super()`.

Python Object Orientation Basics (III)

- *Special* / *magic* methods start and end with two underscores ("dunder") and customize standard Python behavior (e.g., operator overloading).

```python
class My2Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return My2Vector(self.x+other.x, self.y+other.y)

v1 = My2Vector(1, 2)
v2 = My2Vector(3, 2)
v3 = v1 + v2
```

Python Object Orientation Basics (IV)

- *Properties* allow you to add behavior to data attributes:

```python
class My2Vector(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        print "returning x, which is {}".format(self._x)
        return self._x

    def set_x(self, x):
        print "setting x to {}".format(x)
        self._x = x

    x = property(get_x, set_x)

v1 = My2Vector(1, 2)
x = v1.x   # uses the getter, which prints the value
v1.x = 4   # uses the setter, printing the value
```

Helps with refactoring (can replace direct attribute access with a property).

Advanced Kung-Fu

There many advanced techniques that we didn't cover:

- *Multiple inheritance* (deriving from multiple classes) can create a real mess. Need to understand the MRO (Method Resolution Order) to understand `super`.
- Modify classes and objects at runtime, e.g., overwrite or add methods (*monkey patching*).
- *Class decorators* can be used to augment a class.
- *Abstract Base Classes* allow us to register classes as subclasses without actual inheritance (overriding `isinstance`).
- *Metaclasses* (derived from `type`), their instances are classes. Great way to dig yourself a hole when you think you are clever.

Try to avoid these, in most cases you would regret it. (KIS)

Stop Writing Classes!

Good reasons for not writing classes:

- A class is a tightly coupled piece of code, can be an obstacle for change. Complicated inheritance hierarchies hurt.
- Tuples can be used as simple data structures, together with stand-alone functions.
  Introduce classes later, when the code has settled.
- `collections.namedtuple` can be used as an additional intermediate step (can use `__slots__` to keep the lower memory footprint when switching to classes).
- Functional programming can be very elegant for some problems, coexists with object oriented programming.

(see "Stop Writing Classes" by Jack Diederich)

Functional Programming

Pure functions have no side effects.
(mapping of arguments to return value, nothing else)

Great for parallelism and distributed systems.
Also great for unittests and TDD (Test Driven Development).

It's interesting to take a look at functional programming languages (e.g., Haskell) to get a fresh perspective.

Functional Programming in Python

Python supports functional programming to some extend, for example:

- Functions are just objects, pass them around!

```python
def get_hello(name):
    return "hello " + name

a = get_hello
print a("world") # prints "hello world"

def apply_twice(f, x):
    return f(f(x))

print apply_twice(a, "world") # prints "hello hello world"
```

Functional Programming in Python (II)

- Functions can be nested and remember their context at the time of creation (*closures*, *nested scopes*).

```python
def get_add_n(n):
    def _add_n(x):
        return x + n
    return _add_n

add_2 = get_add_n(2)
add_3 = get_add_n(3)

add_2(1) # returns 3
add_3(1) # returns 4
```

Object Oriented Design Principles and Patterns

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

How to do Object Oriented Design right?

How to come up with a good structure for classes and modules?

- KIS & iterate. When you see the same pattern for the third time then it might be a good time to create an abstraction (refactor).
- Sometimes it helps to sketch with pen and paper.
- Classes and their inheritance often have no correspondence to the real-world, be pragmatic instead of perfectionist.
- *Design principles* tell you in an abstract way what a good design should look like (most come down to *loose coupling*).
- *Testability* (with unittests) is a good design criterium.
- *Design Patterns* are concrete solutions for reoccurring problems.

Some Design Principles

- One class, one single clearly defined responsibility (*cohesion*).
- Principle of least knowledge (*Law of Demeter*):
  Each unit should have only limited knowledge about other units.
  Only talk to your immediate friends.
- Favor *composition* over inheritance.
  Inheritance is **not** primarily intended for code reuse, its main selling point is polymorphism.

  Ask yourself: "Do I want to use these subclasses interchangeably?"
- Identify the aspects of your application that vary and separate them from what stays the same.

  Classes should be "open for extension, closed for modification" (*Open-Closed Principle*).
  You should not have to modify the base class.

Some Design Principles (II)

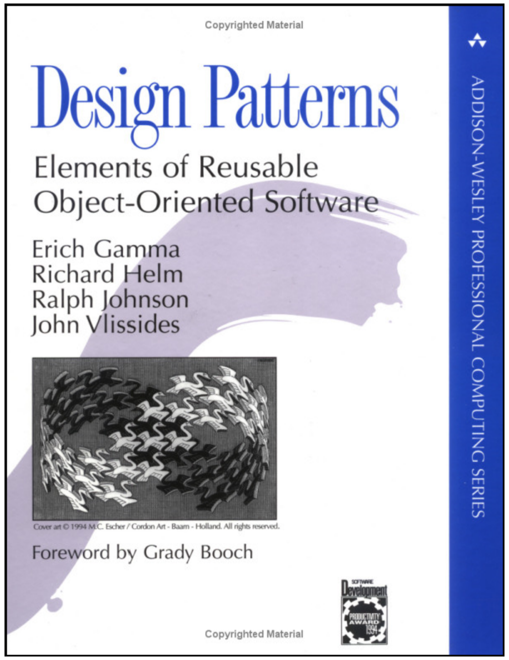
- Minimize the *surface area* of the interface.
  (*Interface Segregation principle*)
- Program to an interface, not an implementation.
  Do not depend upon concrete classes.
  Decouple class instanciation from use.
  (*Dependency Inversion*, *Dependency Injection*)

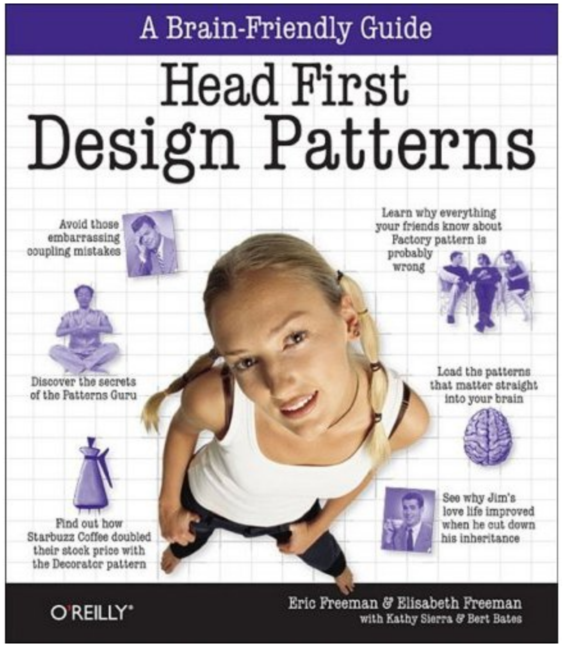
SOLID = Single Responsibility + Open-Closed + Liskov Substitution + Interface Segregation + Dependency Inversion

Design Patterns

Started with “*Design Patterns. Elements of Reusable Object-Oriented Software.*” (1995), written by the “Gang of Four” (GoF).

Easier to read: “Head First Design Pattens” (uses Java)


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


A Brain-Friendly Guide
Head First Design Patterns
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates
O'REILLY

Examples

We'll now discus three popular patterns:

- Iterator Pattern
- Decorator Pattern
- Strategy Pattern

These are just three of many patterns, so go read the book ;-)

Standard pattern names simplify communication between programmers!

Iterator Pattern

Problem

How would you iterate over elements from a collection?

A first (inept) attempt (imitating C code):

```
>>> my_collection = ['a', 'b', 'c']
>>> for i in range(len(my_collection)):
...     print my_collection[i],
a b c
```

But what if `my_collection` does not support indexing?

```
>>> my_collection = {'#x1': 'a', '#x2': 'b', '#y1': 'c'}
>>> for i in range(len(my_collection)):
...     print my_collection[i],
# What will happen here?
```

Should we have to care about this when all we want is iterate?

Idea: Provide an abstraction for iteration handling that separates us from the collection implementation.

Description

What we want:

- Standard interface for collections that we can iterate over (we call these *iterables*),
- Iteration state (e.g., the position counter) should be decoupled form the container and should be encapsulated. Use an *iterator* object, which keeps track of an iteration and knows what the next element is.

What to implement:

The **iterator** has a `next()` method that returns the next item from the collection. When all items have been returned it raises a `StopIteration` exception.

The **iterable** provides an `__iter__()` method, which returns an iterator object.

Example

```
class MyIterable(object):
    def __init__(self, items):
        """items -- List of items."""
        self.items = items

    def __iter__(self):
        return _MyIterator(self)

class _MyIterator(object):
     def __init__(self, my_iterable):
        self._my_iterable = my_iterable
        self._position = 0

    def next(self):
        if self._position >= len(self._my_iterable.items):
            raise StopIteration()
        value = self._my_iterable.items[self._position]
        self._position += 1
        return value

    # in Python, iterators also support iter by returning self
    def __iter__(self):
        return self
```

Note: In Python 3 `next()` becomes `__next__()`.

Example (II)

Lets perform the iteration manually using this interface:

```python
iterable = MyIterable([1,2,3])
iterator = iter(iterable)  # or use iterable.__iter__()
try:
    while True:
        item = iterator.next()
        print item
except StopIteration:
    pass
print "Iteration done."
```

...or just use the Python for-loop:

```python
for item in iterable:
    print item
print "Iteration done."
```

In fact, Python lists are already iterables:

```python
for item in [1, 2, 3]:
    print item
```

Summary

- Whenever you use a for-loop in Python you use the power of the Iterator Pattern!
- Implement the iterable interface in your containers.
  Accept an iterable (or iterator) in your consumers.
- The iterator has a single responsibility, while the iterable does not have to keep track of the iteration (which isn't its business).
- Note that `__iter__` is semantically different for iterables and iterators (duck typing fail!).
- Normally one uses *generator functions* with `yield` instead of writing iterator classes.

Use case: Processing huge data sets in manageable chunks that can come from different sources (e.g., from local disk or from the network).

Decorator Pattern

Starbuzz Coffee

```
class Beverage(object):
    # imagine some attributes like temperature, amount left,...

    def get_description(self):
        return "beverage"

    def get_cost(self):
        return 0.00

class Coffee(Beverage):

    def get_description(self):
        return "coffee"

    def get_cost(self):
        return 3.00

class Tee(Beverage):

    def get_description(self):
        return "tee"

    def get_cost(self):
        return 2.50
```

example taken from “Head First Design Patterns”

Adding Ingredients: First Try

```
class Beverage(object):
    def __init__(self, with_milk, with_sugar):
        self.with_milk = with_milk
        self.with_sugar = with_sugar

    def get_description(self):
        description = str(self._get_default_description())
        if self.with_milk:
            description += ", with milk"
        if self.with_sugar:
            description += ", with_sugar"
        return description

    def _get_default_description(self):
        return "beverage"

    # same for get_cost...

class Coffee(Beverage):
    def _get_default_description(self):
        return "normal coffee"

    # and so on...
```

But what if we want more ingredients? Open-closed principle?

Adding Ingredients: Second Try

```python
class CoffeeWithMilk(Coffee):

     def get_description(self):
        return (super(CoffeeWithMilk, self).get_description() +
                ", with  milk")

     def get_cost(self):
        return super(CoffeeWithMilk, self).get_cost() + 0.30

class CoffeeWithMilkAndSugar(CoffeeWithMilk):

    # And so on, what a mess!
```

What we want:

- Adding a new ingredient like soy milk should not modify the original beverage classes.
- Adding new ingredients should be simple and work automatically across all beverages.

Solution: Decorator Pattern

```
class BeverageDecorator(Beverage):

    def __init__(self, beverage):
        super(BeverageDecorator, self).__init__()
        self.beverage = beverage

class Milk(BeverageDecorator):

    def get_description(self):
        return self.beverage.get_description() + ", with  milk"

    def get_cost(self):
        return self.beverage.get_cost() + 0.30

coffee_with_milk = Milk(Coffee())
```

Composition solves the problem.

Note: Do not confuse this with Python function decorators.

Strategy Pattern

Duck Simulator

```python
class Duck(object):

    def __init__(self):
        # for simplicity this example class is stateless

    def quack(self):
        print "Quack!"

    def display(self):
        print "Boring looking duck."

    def take_off(self):
        print "I'm running fast, flapping with my wings."

    def fly_to(self, destination):
        print "Now flying to %s." % destination

    def land(self):
        print "Slowing down, extending legs, touch down."
```

(example taken from “Head First Design Patterns”)

Duck Simulator (II)

```python
class RedheadDuck(Duck):

    def display(self):
        print "Duck with a read head."

class RubberDuck(Duck):

    def quack(self):
        print "Squeak!"

    def display(self):
        print "Small yellow rubber duck."
```

Oh, snap! The `RubberDuck` has same flying behavior like a normal duck, must override all the flying related methods.

What if we want to introduce a `DecoyDuck` as well? (DRY)

What if a normal duck suffers a broken wing?

Idea: Create a `FlyingBehavior` class which can be plugged into the `Duck` class.

Solution

```python
class FlyingBehavior(object):

    def take_off(self):
        print "I'm running fast, flapping with my wings."

    def fly_to(self, destination):
        print "Now flying to %s." % destination

    def land(self):
        print "Slowing down, extending legs, touch down."

class Duck(object):
    def __init__(self):
        self.flying_behavior = FlyingBehavior()

    def take_off(self):
        self.flying_behavior.take_off()

    def fly_to(self, destination):
        self.flying_behavior.fly_to(destination)

    def land(self):
        self.flying_behavior.land()

    # display, quack as before...
```

Solution (II)

```
class NonFlyingBehavior(FlyingBehavior):

    def take_off(self):
        print "It's not working :-("

    def fly_to(self, destination):
        raise Exception("I'm not flying anywhere.")

    def land(self):
        print "That won't be necessary."

class RubberDuck(Duck):
    def __init__(self):
        self.flying_behavior = NonFlyingBehavior()

    def quack(self):
        print "Squeak!"

    def display(self):
        print "Small yellow rubber duck."

class DecoyDuck(Duck):
    def __init__(self):
        self.flying_behavior = NonFlyingBehavior()

    # different display, quack implementation...
```

Analysis

The *strategy* in this case is the flying behavior.

- If a poor duck breaks its wing we do:
  `duck.flying_behavior = NonFlyingBehavior()`
  Flexibility to change the behaviour at runtime!
- Could have avoided code duplication with inheritance (by defining a `NonFlyingDuck`). Could make sense, but is less flexible.
- Relying less on inheritance and more on composition.

Strategy Pattern means:

- **Encapsulate** the different strategies in different classes.
- Store a strategy object in your main object as a data attribute.
- **Delegate** all the strategy calls to the strategy object.

For example, use this to compose data analysis algorithms.

Strategy Pattern with Functions

What if our behavior only needs a single method?

Stop writing classes!™ Use a function!

Standard examples:

- Sorting with a customized sort key:

```
>>> sorted(["12", "1", "2"], key=lambda x: int(x))
['1', '2', '12']
```

- Filtering with a predicate function:

```
>>> predicate = lambda x: int(x) > 2
>>> data = ["1", "2", "12"]
>>> [x for x in data if predicate(x)]
['12']
```

Closing Notes on Patterns

More on Patterns

Other famous patterns:

- *Observer*
- *Factory*
- *Model-View-Controller* (MVC)
  Compound of multiple patterns.

Warning: Some old patterns are nowadays often considered anti-patterns:

- *Singleton* (overused, replaced with dependency injection)
- *Template Method* (composition is generally better)

Wabi-sabi (Closing Notes / Cheesy Analogy)

"Wabi-sabi represents a comprehensive Japanese world view or aesthetic centered on the acceptance of transience and imperfection. The aesthetic is sometimes described as one of beauty that is imperfect, impermanent, and incomplete." (from Wikipedia)

Acknowledgements

Thanks to my employer for supporting this school and me.
We are hiring :-)

TNG TECHNOLOGY CONSULTING

www.tngtech.com

The examples were partly adapted from
“Head First Design Patterns” (O’Reilly) and
“Design Patterns in Python” `http://www.youtube.com/watch?v=0vJJlVBVTFg`

Image Sources

- CCTV building: Jakob Montrasio (CC BY)
- Rem Koolhaas image: Rodrigo Fernández (CC BY SA)
- Wasi-sabi bowl: Chris 73 / Wikimedia Commons (CC BY SA)

Please inform if any other illustration images infringe copyright to have them removed.