Object Oriented Design

Bartosz Telenczuk, Niko Wilbert

Advanced Scientific Programming in Python

Summer School 2011, St. Andrews

Overview

1. General Design Principles
2. Object Oriented Programming in Python
3. Object Oriented Design Principles
4. Design Patterns

General Design Principles

Disclaimer

Good software design is a never ending learning process.

What really counts is your motivation to improve and question your code.

So this talk is mostly a teaser to get you interested ;-)

Unless explicitly stated all examples in the talk are implemented in Python 2.6

Good Software Design

Writing software is just like **writing prose**: it is not only about the information but also about the **style**!

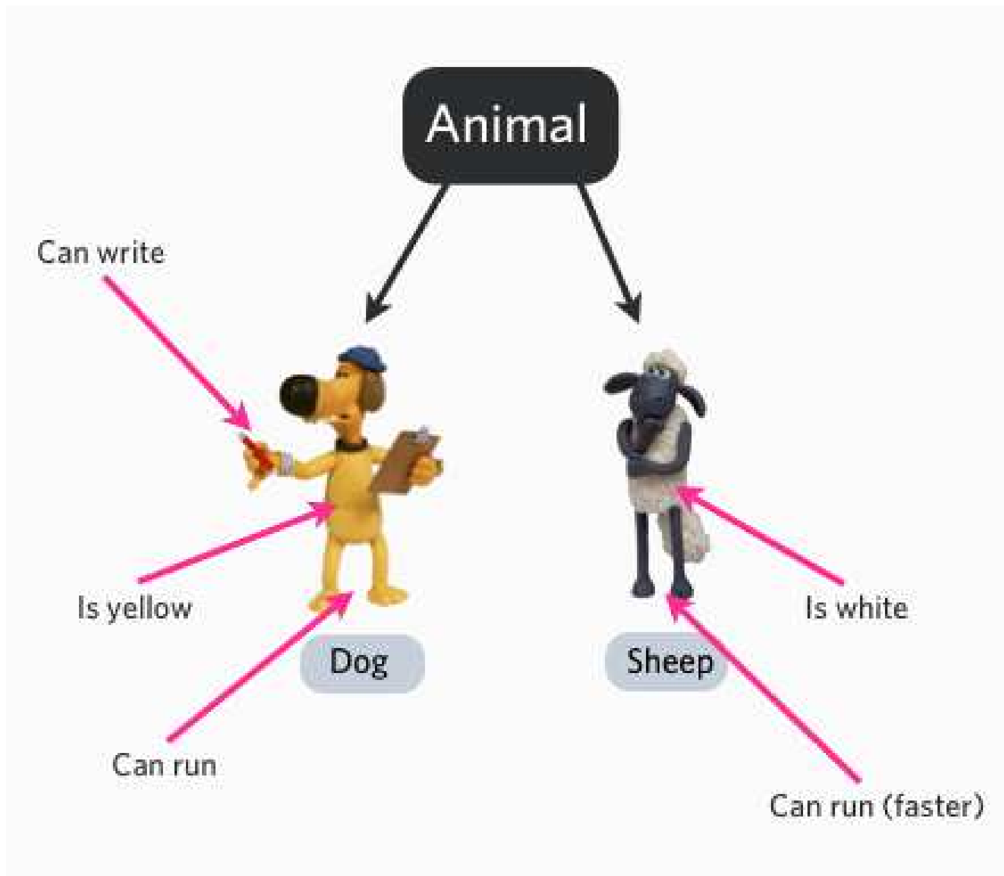
Let's start with two simple general principles:

KIS
: Keep it simple.
Overengineering and excessive fudging both violate this.

DRY
: Don't repeat yourself.
(Sure path to a maintenance nightmare.)

Start simple, use just as much forward planing as needed and **REVISE** (refactor) $\rightarrow$ Agile development.

Object Oriented Programming (in Python)



Object Orientated Programming

Objects

combine state (data) and behavior (algorithms).

Encapsulation

Objects decide what is exposed to the outside world (by their *public interface*) and hide their implementation details to provide *abstraction*.
The abstraction should not *leak* implementation details.

Classes (in classic oo)

define what is common for a whole class of objects, e.g.:
"Snowy **is a** dog" can be translated to
"The Snowy object is an *instance* of the dog class."
Define once how a dog works and then reuse it for all dogs.
Classes correspond to variable types (they are *type objects*).

Object Orientated Programming (II)

Inheritance

“**a** dog (subclass) **is a** mammal (parent/superclass)”
A subclass *is derived from* / *inherits* / *extends* a parent class.
It reuses and extends it, and it can *override* parts that need specialization.

Liskov substitution principle: “What works for the Mammal class should also work for the dog class”.

Polymorphism

Provide common way of usage for different classes, pick the correct underlying behavior for a specific class.
Example: the $+$ operator for real and complex numbers.

Python OO 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 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 OO 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=None):
        self.data = data

class MRIDataset(Dataset):
    def __init__(self, data=None, parameters=None):
        # here has the same effect as calling
        # Dataset.__init__(self)
        super(MRIDataset, self).__init__(data)
        self.parameters = parameters

mri_data = MRIDataset(data=[1,2,3])
```

Note: In Python 3 `super(B, self)` becomes `super()`

Python OO Basics (III)

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

```
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 OO Basics (IV)

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

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

    def get_x(self):
        return self._x

    def set_x(self, x):
        self._x = x

    x = property(get_x, set_x)

    # define getter using decorator syntax
    @property
    def y(self):
        return self._y

v1 = My2Vector(1, 2)
x = v1.x  # use the getter
v1.x = 4  # use the setter
x = v1.y  # use the getter
```

OO Principles in Python

- Python is a **dynamically typed** language, which means that the type of a variable is only known when the code runs.
- **duck typing**:
  If it talks like a duck, walks like a duck, it must be a duck.
- in special cases strict type checking can be performed via `isinstance` function.
- Python relies on convention instead of enforcement.
  If you want to create a giant mess, Python isn't going to stop you.
- No attributes are really private, use a single underscore to signal that an attribute is for internal use only (encapsulation).

Important: document your code (e.g. which arguments can be passed to a function).

Python Example (Listing 1)

```python
import random

class Die(object):  # derive from object for new style classes
    """Simulate a generic die."""

    def __init__(self, sides=6):
        """Initialize and roll the die.

        sides -- Number of faces, with values starting at one (default is 6).
        """
        self._sides = sides  # leading underscore signals private
        self._value = None  # value from last roll
        self.roll()

    def roll(self):
        """Roll the die and return the result."""
        self._value = 1 + random.randrange(self._sides)
        return self._value

    def __str__(self):
        """Return string with a nice description of the die state."""
        return "Die with %d sides, current value is %d." % (self._sides, self._value)

class WinnerDie(Die):
   """Special die class that is more likely to return a 1."""

   def roll(self):
        """Roll the die and return the result."""
        super(WinnerDie, self).roll()  # use super instead of Die.roll(self)
        if self._value == 1:
            return self._value
        else:
            return super(WinnerDie, self).roll()
```

Python Example (II)

```
>>> die = Die()
>>> die._sides  # we should not access this, but nobody will stop us
6
>>> die.roll
<bound method Die.roll of <Die object at 0x03AE3F70>>
>>> for _ in range(10):
...     print die.roll()
2 2 6 5 2 1 2 6 3 2

>>> print die  # this calls __str__
Die with 6 sides, current value is 2.
>>> winner_die = WinnerDie()
>>> for _ in range(10):
...     print winner_die.roll(),
2 2 1 1 4 2 1 5 5 1
>>>
```

Advanced Kung-Fu

Python OO might seem primitive at first. But the dynamic and open nature means that there is enough rope to hang yourself.

Some Buzzwords to give you an idea:

- **Multiple inheritance** (deriving from multiple classes) can create a real mess. You have to understand the **MRO** (Method Resolution Order) to understand `super`.
- You can modify classes at runtime, e.g., overwrite or add methods (**monkey patching**).
- **Class decorators** can be used to modify the class (like function decorators but for classes).
- **Metaclasses** are derived from `type`, their instances are classes! They are more flexible but harder to use than decorators.

...and there is more.

Try to avoid all of this unless you really need it! (KIS)

Object Oriented Design Principles

OO Design

How do you decide what classes should be defined and how they interact?

- OO design is highly **nontrivial**: take a step back and start with pen and paper.
- classes and their inheritance in a good design often have no correspondence to real-world objects.

Design Principles

OO design principles tell you in an abstract way what a good OO design should look like.

1. Identify the aspects of your application that vary and separate them from what stays the same (**encapsulation**).

2. Program to an **interface**, not an implementation.

3. Favor **composition** over inheritance.

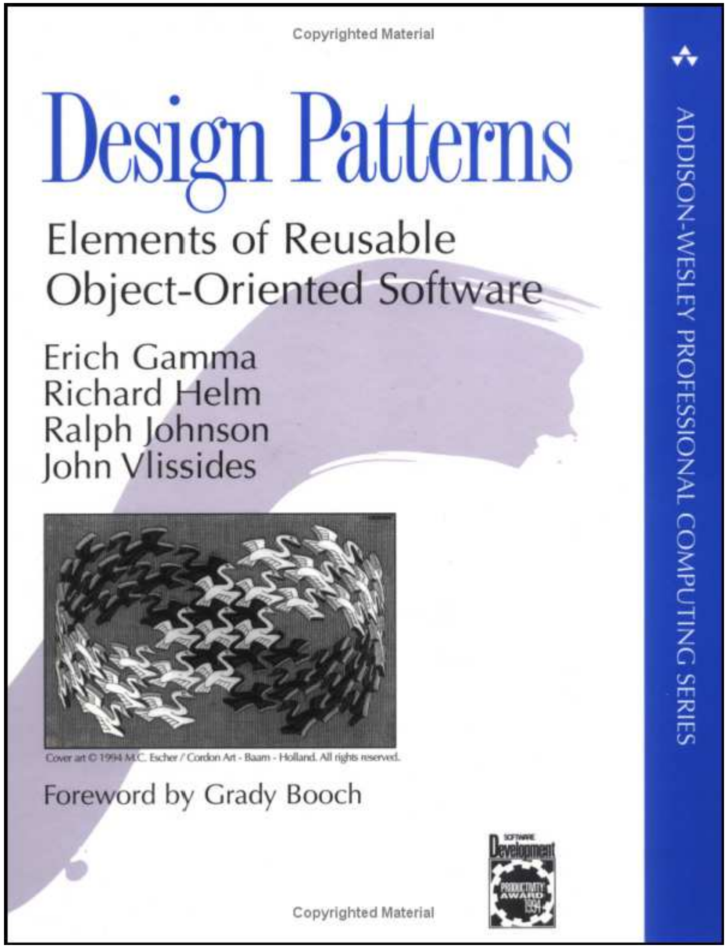
4. Strive for **loosely coupled** designs between objects that interact.

5. Classes should be open for extension, but closed for modification (**Open-Closed Principle**).

Design Patterns

Origins

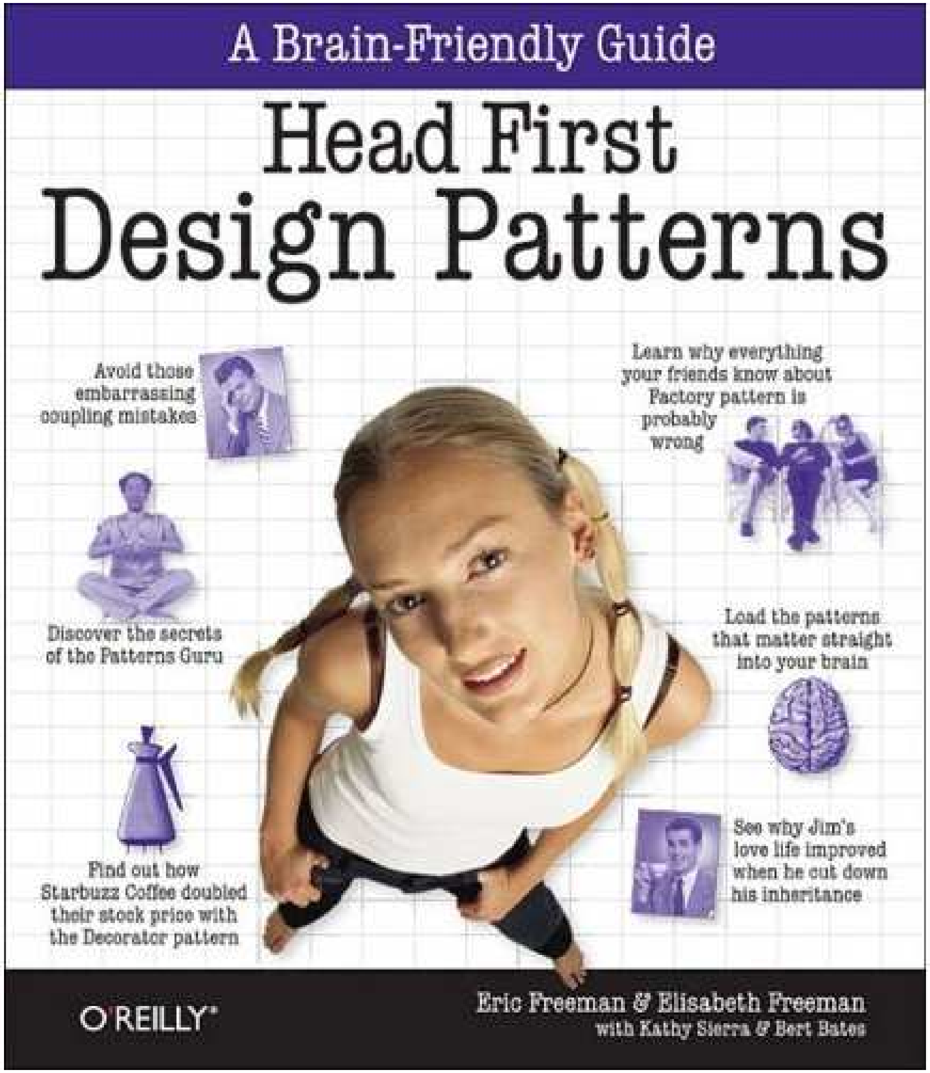
It started with this book (1995):



"*Design Patterns. Elements of Reusable Object-Oriented Software.*"
(GoF, "Gang of Four")

Learning Design Patterns

Easier to read and more modern (uses Java):



Iterator Pattern

Problem

How would you iterate elements from a collection?

My first (inept) try:

```
>>> 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 = {'a': 1, 'b': 2, 'c': 3}
>>> for i in range(len(my_collection)):
...     print my_collection[i],
# What will happen here?
```

This violates one of the design principles! (Which one??)

Idea: Separate (encapsulate) iteration handling from the container definition!

Description

We will need to:

- store the elements in a collection (*iterable*),
- manage the iteration over the elements by means of an *iterator* object which keeps track of the elements which were already delivered

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

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

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

Example (Listing 2)

```python
class MyIterable(object):
    """Example iterable that wraps a sequence."""

    def __init__(self, items):
        """Store the provided sequence of items."""
        self.items = items

    def __iter__(self):
        return MyIterator(self)

class MyIterator(object):
    """Example iterator that is used by MyIterable."""

    def __init__(self, my_iterable):
        """Initialize the iterator.

        my_iterable -- Instance of MyIterable.
        """
        self._my_iterable = my_iterable
        self._position = 0

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

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

Example (II)

First, lets perform the iteration manually:

```
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."
```

A more elegant solution is to use the Python for-loop:

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

In fact Python lists are already *iterables*:

```
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!
- Many **native** and third-party data types implement the Iterator Pattern!
- Iterator Pattern is an example of programming to the **interface** not to implementation.
- *iterator* and *iterable* have only **single responsibilities**: handling iteration and providing a container for data, respectively.

Decorator Pattern

Starbuzz Coffee (Listing 3)

```python
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 "normal coffee"

    def get_cost(self):
        return 3.00

class Tee(Beverage):
    def get_description(self):
        return "tee"
    def get_cost(self):
        return 2.50

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!
```

Any solutions?

We have the following requirements:

- adding new ingredients like soy milk should be easy and work with all beverages,
- anybody should be able to add new custom ingredients without touching the original code (open-closed principle),
- there should be no limit to the number of ingredients.

Decorator Pattern (Listing 4)

```python
class Beverage(object):

    def get_description(self):
        return "beverage"

    def get_cost(self):
        return 0.00

class Coffee(Beverage):
    #[...]

class BeverageDecorator(Beverage):

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

class Milk(BeverageDecorator):

    def get_description(self):
        #[...]

    def get_cost(self):
        #[...]

class Sugar(BeverageDecorator):
    #[...]

coffee_with_milk = Milk(Coffee())
coffee_with_milk_and_sugar = Sugar(Milk(Coffee()))
tea_with_milk = Milk(Tea())
```

Decorator Pattern

Decorator Pattern in general means:

- Decorators **implement** the interface of components they decorate.
- Decorators **wrap** the components and modify their behaviour.
- Decorators are applied at **run-time**.
- Multiple decorators can be "**stacked**".

Decorators provide an alternative for sub-classing for extending objects (**composition-over-inheritance**)

Decorators follow the **open-closed principle** by allowing to extend functionality without modification of the existing code.

Strategy Pattern

Duck Simulator (Listing 5)

```
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."


class RedheadDuck(Duck):

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


class RubberDuck(Duck):

    def quack(self):
        print "Squeak!"

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

Problem

Oh snap! The `RubberDuck` is able to fly!

Looks like we have to override all the flying related methods.

But if we want to introduce a `DecoyDuck` as well we will have to override all three methods again in the same way (DRY).

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

Solution (Listing 6)

```
class FlyingBehavior(object):
    """Default flying behavior."""

    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 quack(self):
        print "Quack!"

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

    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()
```

Solution (II) (Listing 7)

```
class NonFlyingBehavior(FlyingBehavior):
    """FlyingBehavior for ducks that are unable to fly."""

    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()

    def quack(self):
        print ""

    def display(self):
        print "Looks almost like a real duck."
```

Strategy Pattern

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

Strategy Pattern in general 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.

Note: If the behavior has only a single method we can simply use a Python function (Strategy Pattern is “invisible” in Python).

Closing Notes on Patterns

More on Patterns

Caution: Use patterns only where they fit naturally. Adapt them to your needs (not the other way round).

Some other famous and important patterns:

- Observer
- Singleton (can use some Python-Fu here)
- Template
- Composite
- and more...

Combine patterns to solve complex problems.

Example: The *Model-View-Controller* (MVC) pattern is the most famous example for such *compound patterns* (used by GUI APIs and many web frameworks).

Acknowledgements

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

The illustration images were downloaded from Google image search, please inform if they infringe copyright to have them removed.

Adapter Pattern

Teaching Turkey to Quack

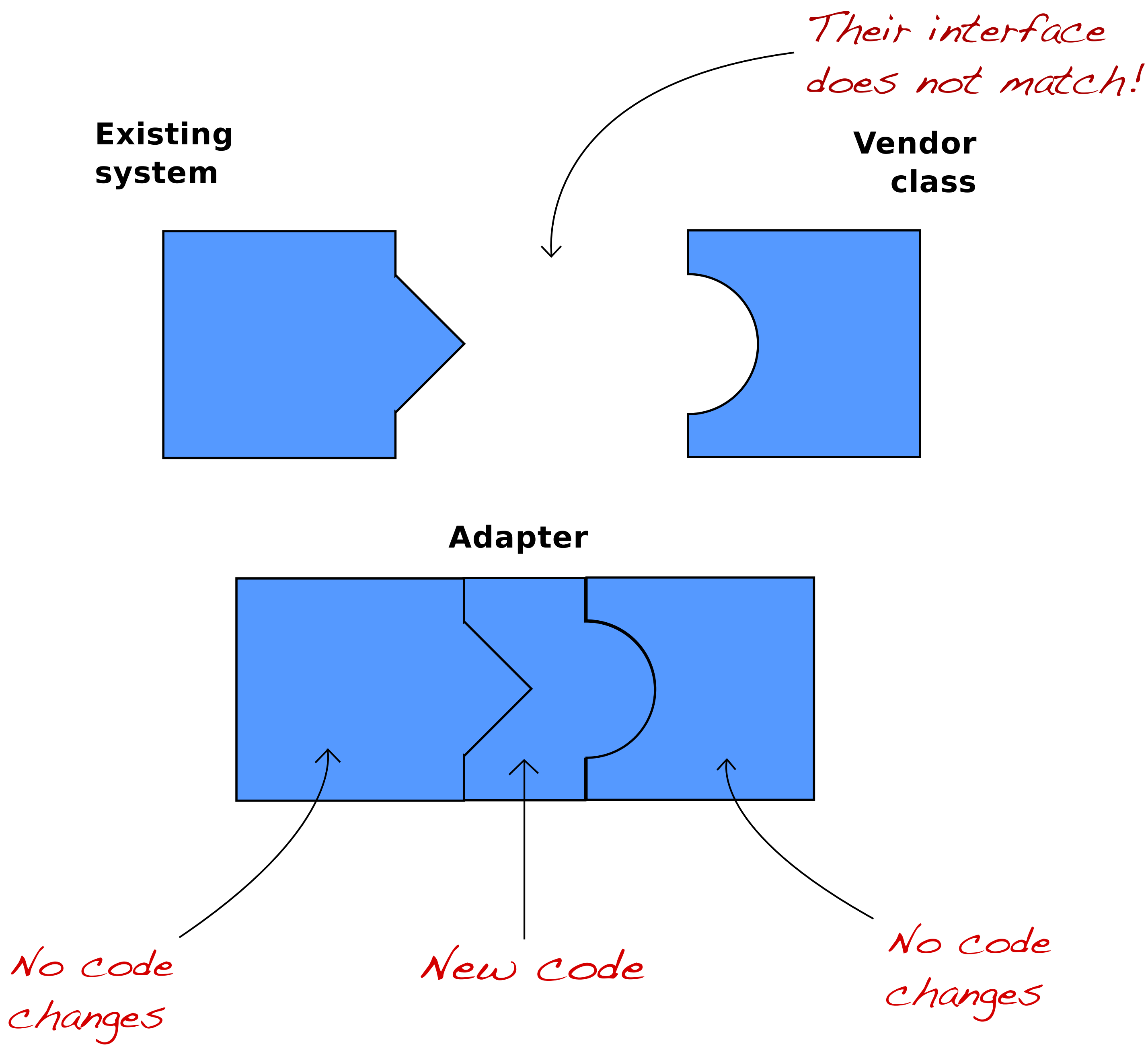
Lets say we obtained the following class from our collaborator:

```
class Turkey(object):

    def fly_to(self):
        print "I believe I can fly..."

    def gobble(self, n):
        print "gobble " * n
```

How to integrate it with our Duck Simulator: turkeys can fly and gobble but they can not quack!

Solutions?

Incompatible interface? No problem!



Being adaptive (Listing 8)

```
class TurkeyAdapter(object):

    def __init__(self, turkey):
        self.turkey = turkey
        self.fly_to = turkey.fly_to  #delegate to native Turkey method
        self.gobble_count = 3

    def quack(self):                         #adapt gobble to quack
        self.turkey.gobble(self.gobble_count)

>>> turkey = Turkey()
>>> turkeyduck = TurkeyAdapter(turkey)
>>> turkeyduck.fly_to()
I believe I can fly...
>>> turkeyduck.quack()
gobble gobble gobble
```

Adapter Pattern applies several good design principles:

- uses **composition** to wrap the adaptee (Turkey) with an altered interface,
- binds the client to an **interface** not to an implementation

Being adaptive: mixin

Flexible, good use of multiple inheritance:

```
class Turkey2Duck(object):

    def __init__(self):
        self.gobble_count = 3
    def quack(self):
        return self.gobble(self.gobble_count)

class TurkeyDuck(Turkey2Duck, Turkey):
    pass

>>> turkeyduck = TurkeyDuck()
>>> turkeyduck.fly_to()
I believe I can fly...
>>> turkeyduck.quack()
gobble gobble gobble
```