Object Oriented Design

Niko Wilbert

May 11, 2009

General Design Principles

Real Life Problems

Some typical real-life problems for scientists:

- Code duplication: You start with one script, then copy and modify it for a slightly different scenario. Maintenance nightmare!
- Technical debt: Program structure no longer matches the demands. But your supervisor wants new features so you keep adding dirty hacks. Time to refactor!
- over-engineering: Try to build the golden hammer for a problem, but end up with a pile of useless bloat.
- Opportunity for an interesting collaboration arises, but this means that other people will have to use your program.
- You just can't get the bugs out of your program.

Scientific Programming

When writing software you typically want the following characteristics:

Effectiveness
: Solve the problem with the minimal amount of effort.

Reliability
: Your results must be trustworthy.

Flexibility
: Requirements often change as new ideas arise.

Transparency
: Other scientists should be able to understand the software and even use it.

Not achieving these goals is a waste of time and potentially leads to bad science.

Effective Methods

Fortunately you are not alone - there are many tools and methods available to write good code, for example:

Effectiveness & Flexibility
: Design principles and patterns.

Transparency
: Clean design, coding conventions and documentation.

Reliability
: All of the above and unit testing.

Good design is the foundation for achieving all these goals.

import this

Python has its own set of guidelines, just execute `import this`:

```
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!
```

Code that follows this is often called "pythonic".

Object Orientation

Object Orientation Principles

Classes and Objects
: Combine data (attributes) and functions (methods) in classes. Instances of classes hold concrete data, instances are called objects.

Encapsulation
: Classes control what they expose to the outside world and they hide their implementation details (so they provide abstraction).

Inheritance
: A class can inherit methods from a parent class, reusing common features. A parent superclass is more general, an inherited / derived subclass overrides superclass features and is more specific.

Polymorphism
: Handling specific objects in a generic way. Internally use different implementations, depending on the class of the object (in Python this is less prominent, due to duck typing).

OO in Python

There are a couple of things that distinguish OO Python from Java (and other "strict" statically types languages):

- Python uses "duck typing" instead of interfaces (if it talks like a duck...). Can still be a good idea to define abstract base classes (see the new ABC's in 2.6)
- Relies on convention instead of enforcement.
  If you want to create a giant mess, Python isn't going to stop you.
- Nothing is really private, use an underscore to signal that something is for internal use only ("use at your own risk").
- The OO system in Python lets you access and modify almost everything (even at runtime).
  This allows you to do cool stuff, but use with caution.

Python Example


```
import random

class Die(object):
    """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
        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

    # TODO: replace this with a read-only property
    def get_value(self):
        """Return the current die value."""
        return value

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

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

Python Example 2


```
>>> import dice
>>> die = Die()
>>> die._sides
6
>>> die.roll
<bound method Die.roll of <dice.Die object at 0x03AE3F70>>
>>> for _ in range(10):
        print die.roll(),


2 2 6 5 2 1 2 6 3 2
>>> winner_die = dice.WinnerDie()
>>> for _ in range(10):
        print winner_die.roll(),


2 2 1 1 4 2 1 5 5 1
>>>
```

OO Design

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

- First of all realize that this is highly nontrivial!
  So take a step back and start with pen and paper.
- *Design principles* tell you in an abstract way what a good design should look like.
- *Design patterns* are concrete solutions for reoccurring problems. They satisfy the design principles and can be used to understand and illustrate them.
- Note that the classes and their inheritance in a good design often have no correspondence to real-world objects.

Design Principles

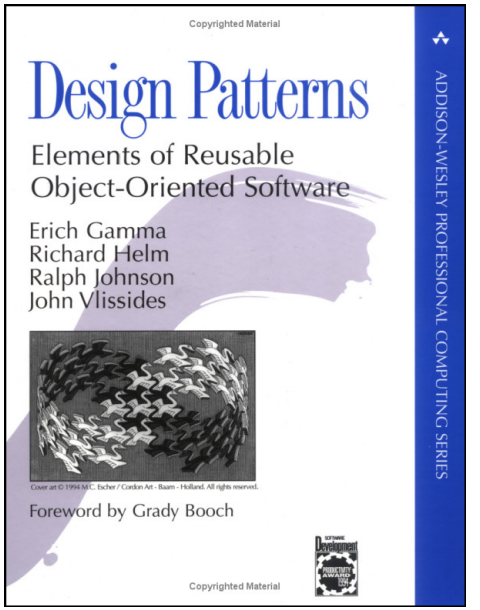
- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interface, not an implementation.
- Favor composition over inheritance.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension, but closed for modification. (Open-Closed Principle)
- Depend upon abstractions. Do not depend upon concrete classes. (Dependency Inversion Principle)
- Talk only to your immediate friends. (Principle of Least Knowledge)
- Don't call us, we'll call you. (Hollywood Principle)
- A class should have only one reason to change. (Single Responsibility)

from "Head First Design Patterns"

Design Patterns

Origins

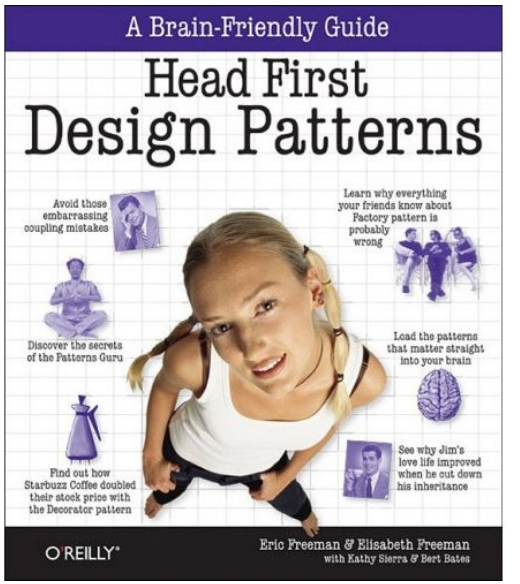
It all started with this book (1995):
"Design Patterns. Elements of Reusable Object-Oriented Software."
(GoF, "Gang of Four")



Actually the idea came from a book on architecture.
Obviously identifying patterns is not limited to software...

Learning Design Patterns

Easier to read and more modern (uses Java):



Many other books and websites also cover design patterns.

If you know the most common design patterns it gives a terminology to communicate effectively with other programmers!

Iterator Pattern

Description

- We want to iterate over different collections / containers of items. We call such a collection an *iterable*.
- We create an *iterator* object that manages the iteration (keeps track of where we are, which items have already been passed)
- The *iterator* has a `next()` method that returns an 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):
    """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 2


```
iterable = MyIterable([1,2,3])

## perform the iteration by hand:
iterator = iter(iterable)  # or use iterable.__iter__()
try:
    while True:
        item = iterator.next()
        print item
except StopIteration:
    pass
print "Iteration done."

## or use a for-loop:
for item in iterable:
    print item
print "Iteration done."

## iterator also supports iter:
iterator = iter(iterable)  # the old iterator has been used up!
for item in iterator:
    print item

## actually lists in Python are already iterables
for item in [1,2,3]:
    print item
```

Python Specifics

- Whenever you use a for-loop in Python you use the power of the iterator pattern!
  This is why they work with so many data types.
  (in Java this capability was only added later on)
- In Java one would define special interfaces for both iterators and iterables, in Python these are just conventions (duck typing).
- For convenience iterators also have an `__iter__` method, but it is semantically different (returns `self`).
  This is a case where duck typing can be dangerous.
  Do not confuse iterables (which are collections) and iterators (they manage an iteration)!
- Python also has *generators*, which are a special kind of iterator (use the `yield` keyword).

Strategy Pattern

Duck Simulator


```
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 read 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 (code duplication).
And what if a normal duck suffers a broken wing?

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

Solution


```
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 2


```
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."
```

Analysis

- 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`), but with additional behaviors gets complicated (requiring multiple inheritance).
- Relying less on inheritance and more on composition (good according to the design principles).

Strategy Pattern

The *strategy* in this case is the flying behavior.
Strategy pattern in general means:

Encapsulate the different strategies in different classes. Classes that use the strategy get a strategy object to which they delegate all the strategy calls.

Note that if the behavior only has a single method we can simply use a function. Therefore it is often said that the strategy pattern is invisible in Python.

Factory Method Pattern

Factories

Object creation sometimes requires more than instantiating a class. Area of many specializations or frequent code changes.

Makes sense to encapsulate the object creation and decouple it from the rest of the code.

Patterns to deal with this generally have *factory* in their name. The *factory method pattern* is a method in a class for this purpose.

Creating Duck Data


```
class DuckSimulator(object):
    """Class to create realistic duck data for machine learning."""

    def add_duck(self, duck):
        """Place a duck into the virtual pont."""

    def create_visual_duck_data(self):
        """Return VisualDuckData."""

    def create_audio_duck_data(self):
        """Return AudioDuckData."""


class DuckData(object):
    """Data set created from ducks."""

    def __iter__(self):
        """Return iterator for data as numpy arrays."""

    def get_duck_description(self):
        """Return the description of the duck."""

    def get_duck_parameters(self):
        """Return the paramters for the duck."""


class VisualDuckData(DuckData):

    def __iter__(self):
        # the iterator converts the stored images to numpy arrays

class AudioDuckData(DuckData):

    def __iter__(self):
        # the iterator decompresses the stored MP3 audio data
```

Problem

`DuckSimulator` is now a very complicated class because it has to deal with different modalities.
Depends on all the special `DuckData` classes.

Normally we will be only working with one modality at a time, so break this up.

Solution


```
class DuckSimulator(object):

    def add_duck(self, duck):
        """Place a duck into the virtual pont."""

    def create_duck_data(self):
        """Create DuckData."""
        # create boring data tables


class VisualDuckSimulator(DuckSimulator):

    def create_duck_data(length):
        """Create 3D VisualDuckData."""
        # here goes our 3D duck-rendering code


class AudioDuckSimulator(DuckSimulator):

    def create_duck_data(length):
        """Create AudioDuckData."""
```

Factory Method

The *factory method* in the example is `create_training_data` and is overriden by derived classes.

`DuckSimulator` no longer depends on all special `DuckData` classes (dependency inversion).

Depending on the size of the factory method implementations it might be wise to refactor this into a dedicated class, e.g. `DuckDataCreator`.

This brings us to the *abstract factory pattern*, were concrete factory classes are derived from one (abstract) class.

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 "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!
```

Second Attempt


```
class Beverage(Beverage):

    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"

    def get_cost(self):
        cost = self._get_default_cost()
        if self.with_milk:
            cost += 0.30
        if self.with_sugar:
            cost += 0.20
        return cost

    def _get_default_cost(self):
        return 0.00

class Coffee(Beverage):

    def _get_default_description(self):
        return "normal coffee"

    def _get_default_cost(self):
        return 3.00
```

Analysis

Second solution already looks much cleaner than the first.

`_get_default_description` like a factory method.

One monolithic class that depends on every little detail.
Hard to maintain (e.g. when adding soy milk as a new option).
Violates the open-closed principle.

Decorator pattern to the rescue!

Decorator


```
class Beverage(object):

    def get_description(self):
        return "beverage"

    def get_cost(self):
        return 0.00

class BeverageDecorator(Beverage):

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


class Coffee(Beverage):

    def get_description(self):
        return "normal coffee"

    def get_cost(self):
        return 3.00

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())
```

Notes

- Adding new ingredients like soy milk is now very easy and automatically works with all beverages.
- Anybody can define new custom ingredients without touching the original code.
- There is no limit to the number of ingredients, this design scales very well.

The decorator pattern is very popular, for example the Java IO library is completely built around it.

Caution: Do not confuse the decorator pattern with the Python *decorator syntax* for the wrapping or modification of functions or classes (2.6) when they are defined (not at runtime).

Closing Notes

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:

- other Factory Patterns (briefly mentioned)
- Command Pattern
- Observer
- Singleton
- Adapter
- Facade
- Composite

Combine patterns to solve complex problems. The Model-View-Controller (MVC) pattern is the most famous example for such *compound patterns*.

Acknowledgements

The examples were partly adapted from
“Head First Design Patterns” (O’Reilly)
and
“Building Skills in Python”
http://homepage.mac.com/s_lott/books/python