Credits: Mostly a direct Python adaptation of “Wizards and Warriors”, a series by Eric Lippert, a principal developer of the C# compiler.
Software engineering is a difficult discipline... unlike what you may think. Programming models and software design are nontrivial endeavors. Object-oriented programming is no exception to this. OOP is far more than mere encapsulation + polymorphism + ... If you’ve never really struggled with OOP, you haven’t really seen OOP. ;)}
In OOP (and arguably programming in general), every procedure needs:

- A *pre-condition*: assumptions it makes
- A *post-condition*: guarantees it provides

These describe the procedure’s *interface*.

After all, if you knew nothing about a function, you couldn’t use it.

Often we hand-wave these without specifying them:

- Sometimes we’re lucky and get it right! And everything works.
- Other times we it bites us back later... and we don’t even realize.

**Specifying interfaces correctly is crucial and difficult.**

Let’s see some toy examples.
Let’s jump in!

Here’s a scenario:

A wizard is a kind of player.  A staff is a kind of weapon.

A warrior is a kind of player.  A sword is a kind of weapon.

A player has a weapon.

⇒ How do we model this problem?
We know OOP, so let's use it!

Question: What classes do we need?

class Weapon(object):
    ...

class Staff(Weapon):
    ...

class Sword(Weapon):
    ...

class Player(object):
    ...
    def get_weapon(self):
        return self.w
    def set_weapon(self, w):
        self.w = w

class Wizard(Player):
    ...

class Warrior(Player):
    ...
Awesome, we’re done!

Oops… a **new requirement has appeared**! Or rather, two requirements:

- A Warrior can only use a Sword.
- A Wizard can only use a Staff.

*How unexpected!!*

Let’s incorporate these requirements. **What do we do?**
Obviously, we need to enforce the types somehow. How about this?

```python
class Player(object):
    @abstractmethod
def get_weapon(self): raise NotImplementedError()
    @abstractmethod
def set_weapon(self, w): raise NotImplementedError()

class Wizard(Player):
    def get_weapon(self): return self.w
    def set_weapon(self, w):
        assert isinstance(w, Staff), "weapon is not a Staff"
        self.w = w

class Warrior(Player): ...
```

Is this good? (Hint: no...) What is the problem?
Consider:

```python
players = [Wizard(), Warrior()]
for player in players:
    player.set_weapon(weapon)
```

Oops: `AssertionError`: `weapon` is not a `Staff`

...really?? Picking up the wrong weapon is a `bug`?!

No, it isn’t the programmer’s fault. **Raise an error instead.**
OK, so how about this?

class Wizard(Player):
    def get_weapon(self):
        return self.w
    def set_weapon(self, w):
        if not isinstance(w, Staff):
            raise ValueError("weapon is not a Staff")
        self.w = w
OK, so now we get an error:

```python
players = [Wizard(), Warrior()]
for player in players:
    player.set_weapon(weapon)
```

ValueError: weapon is not a Staff

But we declared every Player has a set_weapon()!

⇒ Player.set_weapon() is a lie. It does not accept a mere Weapon.
We say this violates the **Liskov substitution principle** (LSP):

> When an instance of a superclass is expected, any instance of any of its subclasses should be able to substitute for it.

However, there’s no single consistent type for \( w \) in `Player.set_weapon()`. Its correct type depends on the type of `self`.

In fact, for `set_weapon` to guarantee anything to the caller, the caller must already know the type of `self`.

But at that point, we have no abstraction! Declaring a common `Player.set_weapon()` method *provides no useful information.*
Let’s try a different idea:

```python
class Wizard(Player):
    def get_weapon(self):
        if not isinstance(w, Staff):
            raise ValueError("weapon is not a Staff")
        return self.w

    def set_weapon(self, w):
        self.w = w
```

Thoughts? **Bad idea:**

- Wizard is now lying about what weapons it accepts
- We’ve planted a ticking time bomb
- We’ve only shifted the problem around
What do we do?

We’ll get back to this. First, let’s consider other problems too.
Object-Oriented Design

Let’s assume we magically solved the previous problem.

Now **consider how the code could evolve**: 

```python
class Monster(object): ...
class Werewolf(Monster): ...
class Vampire(Monster): ...
```

**New rule!** *A Warrior is likely to miss hitting a Werewolf after midnight.*

How do we represent this?

- Classes represent nouns (things); methods represent verbs (behavior)
- We’re describing a behavior
- Clearly we need something like a `Player.attack()` method
Let’s codify the attack method:

```python
class Player(object):
    def attack(self, monster):
        ... # generic stuff

class Warrior(Player):
    def attack(self, monster):
        if isinstance(monster, Werewolf):
            ... # special rules for Werewolf
        else:
            Player.attack(self, monster) # generic stuff
```

How does this look?

Do you see a problem?
Problem 2(a): `isinstance` is exactly what you need to avoid in OOP!

- OOP uses dynamic dispatch for polymorphism, not conditionals
- Caller may not even know all possibilities to be tested for

Problem 2(b): Why the asymmetry between Warrior and Werewolf?

- Why put mutual interaction logic in Warrior instead of Werewolf?
- Again: arbitrary symmetry breakage is a code smell—indicating a potentially deeper problem.

Can lead to code fragmentation: later logic might just as easily end up in Werewolf, suddenly multiplying the number of places such logic is maintained, making maintainance difficult and error-prone.

Can cause other unforeseen problems—code smells often bite back!
Solving problem 2(a) (avoiding isinstance)

“Dispatch” means “deciding which method to use”.

With classes, we get single dispatch: dispatching based on a single argument (self).

Fundamentally, we want double dispatch: deciding what method to call based on the Player and Monster arguments.
Object-Oriented Design

Solving problem 2(a) (avoiding isinstance):
“Visitor pattern”—simulate double dispatch via single dispatch:

class Warrior(Player):  # visitor
    def attack(self, monster):
        return monster.warrior_defend(self)  # request visit

class Wizard (Player):  # visitor
    def attack(self, monster):
        return monster.wizard_defend(self)  # request visit

class Werewolf(Monster):  # visitee
    def warrior_defend(self, warrior): ...  # accept visit
    def wizard_defend(self, wizard): ...  # accept visit

class Vampire (Monster):  # visitee
    def warrior_defend(self, warrior): ...  # accept visit
    def wizard_defend(self, wizard): ...  # accept visit
Visitor pattern solves problem 2(a) (and popular), but bad idea here:

- **Problem 2(b)** still there (symmetry still broken)
- **Too much code**—simple idea, but painful to write
- **Convoluted/confusing**—difficult to reason about

Worst of all: **not scalable** (and **ugly!!!**)

- What if attack also depended on Location, Weather, etc.?
- Visitor pattern for quadruple-dispatch?? Do you seriously want to?!

(P.S.: *Even true multiple-dispatch would have its own problems.*)

⇒ **Is there a fundamentally different, superior solution?**
Object-Oriented Design

～ Words of Wisdom #1 ～

Recognize when you’re fighting your code/framework.
Then stop doing it.
It might be trying to tell you something.

～ Words of Wisdom #2 ～

If your design is convoluted, you might be missing a noun.

～ Words of Wisdom #3 ～

Elegant solutions often solve multiple problems at once.

*Let’s take a step back and re-examine our assumptions & goals.*
Objective:

- Code should be “DRY”: *Don’t Repeat Yourself*
- More generally: code should be easy to read, write, and maintain
- Constraints and logic should be expressed in code somehow

Assumptions:

1. OOP is a solution
2. Represent every “entity” (noun) with a class: player, monster, etc.
3. Represent every “behavior” (verb) with a method

*Maybe we made poor assumptions?*
Solution: We’re missing a very fundamental class. Any ideas?

⇒ We need a “Rule” class.

In fact, our class hierarchy completely missed our program’s objective, which was to maintain state consistency against modification attempts.

Instead of coding blindly, we should’ve started with our real concerns:

- Users provide sequences of commands...
- ...to be evaluated in the context of rules and current game state...
- ...to produce effects.
Object-Oriented Design

What do we know about **effects**?

- Effects include doing nothing (no-op, or “nop”)
- Effects include mutating game state
- Effects include playing audio, video, ...
- Effects include combinations of other effects

What do we know about **rules**?

- Rules can determine effects based on the player, action, etc.
- Rules can be *invariants*: conditions that must never be violated
- Rules can determine “default” command behavior
- Rules can affect (weaken/strengthen/override/etc.) other rules
Previous problems no longer exist:

- Players possess weapons? OK, make Player class with weapon field. Nothing else—*that’s all*. Player’s only job is to maintain its state.

- Make a Command called *Wield* that holds a Player and a Weapon. Evaluate Commands in the context of Rules, producing Effects.

- Make Rules for evaluating different Commands, like Wield. These would modify any produced Effects as desired.
Object-Oriented Design

What problems have we solved?

- Arbitrary choices are no longer made
- Location of rule in code is obvious and unique
- No more LSP violations and ticking time bombs
- Solution is scalable to more sophisticated rules

**Bonus:** separating out Rules actually solves more problems!

- We can put rules into a database and pass them around if needed
- We can write engines to test rules in different orders, for validation
- We can write rules in a simpler domain-specific language (DSL)
  No more need to know codebase—*or to even be a programmer!*
Object-Oriented Design

What just happened?

- We explicitly represented our code as data (Rule, Effect, ...)
- We made our design more flexible and scalable
- We made our design more elegant
- We made our design easier to understand and maintain

How did we achieve this? By not coding blindly.
Object-Oriented Design

Takeaways:

- Think before you code.
- Design choices have far-reaching ramifications on an entire project.
- Constantly watch out for code smells and unnecessary oddities.
- Software engineering can require genuine thinking and insight. Take it seriously. Don’t naively assume it’s “beneath” you as a theorist or systems programmer (or whatever).
- Fundamentally poor decisions may not make themselves obvious. If you don’t actively re-evaluate your design decisions, you may never notice problems.
Another, simpler scenario: how would you code `breadth-first-search`?

Probably similarly to this:

```python
def breadth_first_search(v):
    i = 0
    queue = [v]
    while i < len(queue):
        v = queue[i]
        i += 1
        queue.extend(v.children)
        yield v
```
Let’s make it a class instead:

class BreadthFirstSearcher(object):
    def __init__(self, v):
        (self.i, self.queue) = (0, [v])
    def next(self):
        while self.i < len(self.queue):
            v = self.queue[self.i]
            self.i += 1
            self.queue.extend(v.children)
        return v
Let’s make it a class instead!

Why make a whole class for BFS?? Does anybody do this?!

Well, maybe because we can now very easily:

- **Inspect** the queue while iterating
- **Modify** the queue if desired
- **Save and restore** the iterator state
- **Copy/fork** the iterator mid-way and continue it on multiple graphs

Note that making BreadthFirstSearcher a class is **not obvious**!

Realizing this solution takes some thinking... and pays dividends.