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 warrior is a kind of player.
A staff is a kind of weapon.
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!

class Weapon(object):

  def get_weapon(self):
    return self.w

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

class Staff(Weapon):

  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 Sword(Weapon):

  def get_weapon(self):
    return self.w

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

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):

Let’s incorporate these requirements. What do we do?

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?
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

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
9 / 30

OK, so now we get an error:

def 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.

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
10 / 30

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.

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
11 / 30

Let's try a different idea:

class Wizard(Player):
    def get_weapon(self):
        if not isinstance(self.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

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
12 / 30

What do we do?

We'll get back to this. First, let's consider other problems too.

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
13 / 30

Let's assume we magically solved the previous problem.

Now consider how the code could evolve:

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

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
14 / 30

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 maintenance difficult and error-prone.
⇒ Can cause other unforeseen problems—code smells often bite back!

Mehrdad Niknami (UC Berkeley)  
CS 61A/CS 98-52  
15 / 30
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.

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

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

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:

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

Maybe we made poor assumptions?

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

Object-Oriented Design

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 Cosaan called `Wield` that holds a Player and a Weapon. Evaluate Cosaanos in the context of Rules, producing Effects.
- Make Rules for evaluating different Cosaanos, like `Wield`. These would modify any produced Effects as desired.

Object-Oriented Design

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.

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

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.
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 even be a programmer!

---

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.

---

Object-Oriented Design

Another, simpler scenario: how would you code **breadth-first-search**?

```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
```

---

Object-Oriented Design

Let’s make it a class instead:

```python
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)
            yield v
```

---

Object-Oriented Design

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.