Credits: Mostly a direct Python adaptation of “Wizards and Warriors”, a series by Eric Lippert, a principal developer of the C# compiler.
Object-Oriented Design

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. ;)

Mehrdad Niknami (UC Berkeley)
Software engineering is a difficult discipline
Software engineering is a difficult discipline... unlike what you may think.
Software engineering is a difficult discipline... unlike what you may think. Programming models and software design are nontrivial endeavors.
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.
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 + ...
Object-Oriented Design

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 precondition**: 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.
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.
In OOP (and arguably programming in general), every procedure needs:

- A *pre-condition*: assumptions it makes

Specifying interfaces correctly is crucial and difficult.
In OOP (and arguably programming in general), every procedure needs:

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

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

Object-Oriented Design

Let’s jump in!

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.
Let’s jump in!

Here’s a scenario:
Object-Oriented Design

Let’s jump in!

Here’s a scenario:

A wizard is a kind of player.
Let’s jump in!

Here’s a scenario:

A **wizard** is a kind of **player**.

A **warrior** is a kind of **player**.
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.
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**.
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**.
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?*
**Object-Oriented Design**

*We know OOP, so let’s use it!*

**Question:** What classes do we need?

```python
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):
    ...
```
We know OOP, so let’s use it!

**Question:** What classes do we need?

```python
class Weapon(object):
    ...
```

```python
class Staff(Weapon):
    ...
```

```python
class Sword(Weapon):
    ...
```

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

```python
class Wizard(Player):
    ...
```

```python
class Warrior(Player):
    ...
```
We know OOP, so let’s use it!

**Question:** What classes do we need?

```python
class Weapon(object):
    ...

class Staff(Weapon):
    ...
```

```python
class Player(object):
    ...

class Wizard(Player):
    ...

class Warrior(Player):
    ...
```
We know OOP, so let’s use it!

**Question:** What classes do we need?

class Weapon(object):
    ...

class Staff(Weapon):
    ...

class Sword(Weapon):
    ...
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):
    ...
We know OOP, so let’s use it!

**Question:** What classes do we need?

```python
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):
    ...
```

Mehrdad Niknami (UC Berkeley)
Object-Oriented Design

*We know OOP, so let’s use it!*

**Question:** What classes do we need?

```python
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...

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.
Awesome, we’re done!

Oops...
Awesome, we’re done!

Oops... a new requirement has appeared!
Awesome, we’re done!

Oops... a new requirement has appeared! Or rather, two requirements:
Awesome, we’re done!

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

- A Warrior can only use a Sword.
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.
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!!
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.
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.

```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?
Object-Oriented Design

Obviously, we need to enforce the types somehow. How about this?
Obviously, we need to enforce the types somehow. How about this?

class Player(object):
    @abstractmethod
def get_weapon(self): raise NotImplementedError()
    @abstractmethod
def set_weapon(self, w): raise NotImplementedError()
Obviously, we need to enforce the types somehow. How about this?

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?
Obviously, we need to enforce the types somehow. How about this?

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...)
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.
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.
Consider:

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

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

...really??
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*?!
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.
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?
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:
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
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()!
OK, so now we get an error:

```python
together = [Wizard(), Warrior()]
for player in together:
    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.
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.*
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()`. 
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`. 
Object-Oriented Design

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`. 
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!
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.

Mehrdad Niknami (UC Berkeley)
Let’s try a different idea:

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
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?
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:**
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
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
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?
What do we do?

We’ll get back to this. First, let’s consider other problems too.
Let’s assume we magically solved the previous problem.

```python
class Monster:
    ...:

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).
Clearly we need something like a Player.attack() method.
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). Clearly we need something like a `Player.attack()` method.
Let’s assume we magically solved the previous problem.

Now consider how the code could evolve:

class Monster(object): ...

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 assume we magically solved the previous problem.

Now consider how the code could evolve:

class Monster(object): ...

class Werewolf(Monster): ...
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): ...
```
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!
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.*
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?
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)
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
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
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?
```
Let’s codify the attack method:

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

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

How does this look? Do you see a problem?

Mehrdad Niknami (UC Berkeley)
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?
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?
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 maintenance difficult and error-prone. Can cause other unforeseen problems—code smells often bite back!
Problem 2(a): `isinstance` is exactly what you need to avoid in OOP!
Problem 2(a): `isinstance` is exactly what you need to avoid in OOP!
- OOP uses dynamic dispatch for polymorphism, not conditionals
**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(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?
Object-Oriented Design

**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?
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.*
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.
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.
Solving problem 2(a) (avoiding instanceof)

“Dispatch” means “deciding which method to use”.
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).
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*
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.
Solving problem 2(a) (avoiding isinstance):

```python
# Visitor pattern
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): ...
    def wizard_defend(self, wizard): ...

class Vampire(Monster):  # visitee
    def warrior_defend(self, warrior): ...
    def wizard_defend(self, wizard): ...
```

Solving problem 2(a) (avoiding `isinstance`):

“Visitor pattern”—simulate double dispatch via single dispatch:

```python
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
```
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
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?
Visitor pattern solves problem 2(a) (and popular), but bad idea here:

- 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.)
Visitor pattern solves problem 2(a) (and popular), but bad idea here:

- **Problem 2(b)** still there (symmetry still broken)
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
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

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.)
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?
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 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?!
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.)*
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.
~ 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.
~ Words of Wisdom #1 ~

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

~ 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?
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?
Objective:

- Code should be “DRY”: *Don’t Repeat Yourself*
Objective:

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

1. OOP is a solution
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.
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
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.
Solution: We’re missing a very fundamental class.
Solution: We’re missing a very fundamental class. Any ideas?
**Solution:** We’re missing a very fundamental class. Any ideas?

⇒ We need a “Rule” class.
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.
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:
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...
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...
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.
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
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
What do we know about **effects**?

- Effects include doing nothing (no-op, or “nop”)

- 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
What do we know about **effects**?

- Effects include doing nothing (no-op, or “nop”)
- Effects include mutating game state
What do we know about effects?

- Effects include doing nothing (no-op, or “nop”)
- Effects include mutating game state
- Effects include playing audio, video, ...
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 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
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.
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
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
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.
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.
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*.
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.
Previous problems no longer exist:

- Players possess weapons? OK, make Player class with weapon field. Nothing else—\textit{that's all}. Player's only job is to maintain its state.
- Make a Command called \textit{Wield} that holds a Player and a Weapon. Evaluate Commands in the context of Rules, producing Effects.
- Make Rules for evaluating different Commands, like \textit{Wield}. These would modify any produced Effects as desired.
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!
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 Rule 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!
What problems have we solved?

- Arbitrary choices are no longer made
Object-Oriented Design

What problems have we solved?

- Arbitrary choices are no longer made
- Location of rule in code is obvious and unique
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
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
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!
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
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
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.

Mehrdad Niknami (UC Berkeley)
What just happened?
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**.

Mehrdad Niknami (UC Berkeley)
Object-Oriented Design

What just happened?

- We explicitly represented our code as data (Rule, Effect, ...)
- We made our design more flexible and scalable
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
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
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?
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.
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.

Mehrdad Niknami (UC Berkeley)
Takeaways:

- Think before you code.
Takeaways:

- Think before you code.
- Design choices have far-reaching ramifications on an entire project.
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.
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).
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?

```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
```

Mehrdad Niknami (UC Berkeley)
Another, simpler scenario: how would you code \texttt{breadth-first-search}?
Another, simpler scenario: how would you code **breadth-first-search**?

Probably similarly to this:
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:

```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)
            return v
```
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)
            return 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.
Let’s make it a class instead!

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

Note that making BreadthFirstSearcher a class is not obvious! Realizing this solution takes some thinking... and pays dividends.
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.