

# SOLID Design Principles

AND THE LAW OF DEMETER.

COMP3522 OBJECT ORIENTED  
PROGRAMMING 2: WEEK 3

# SOLID

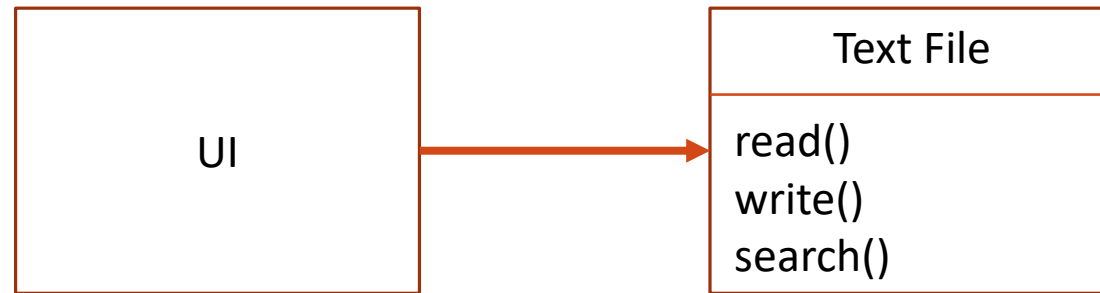
Software Development  
Is not a Jenga game



Mark Nijhof

# Recap: Remember this?

---



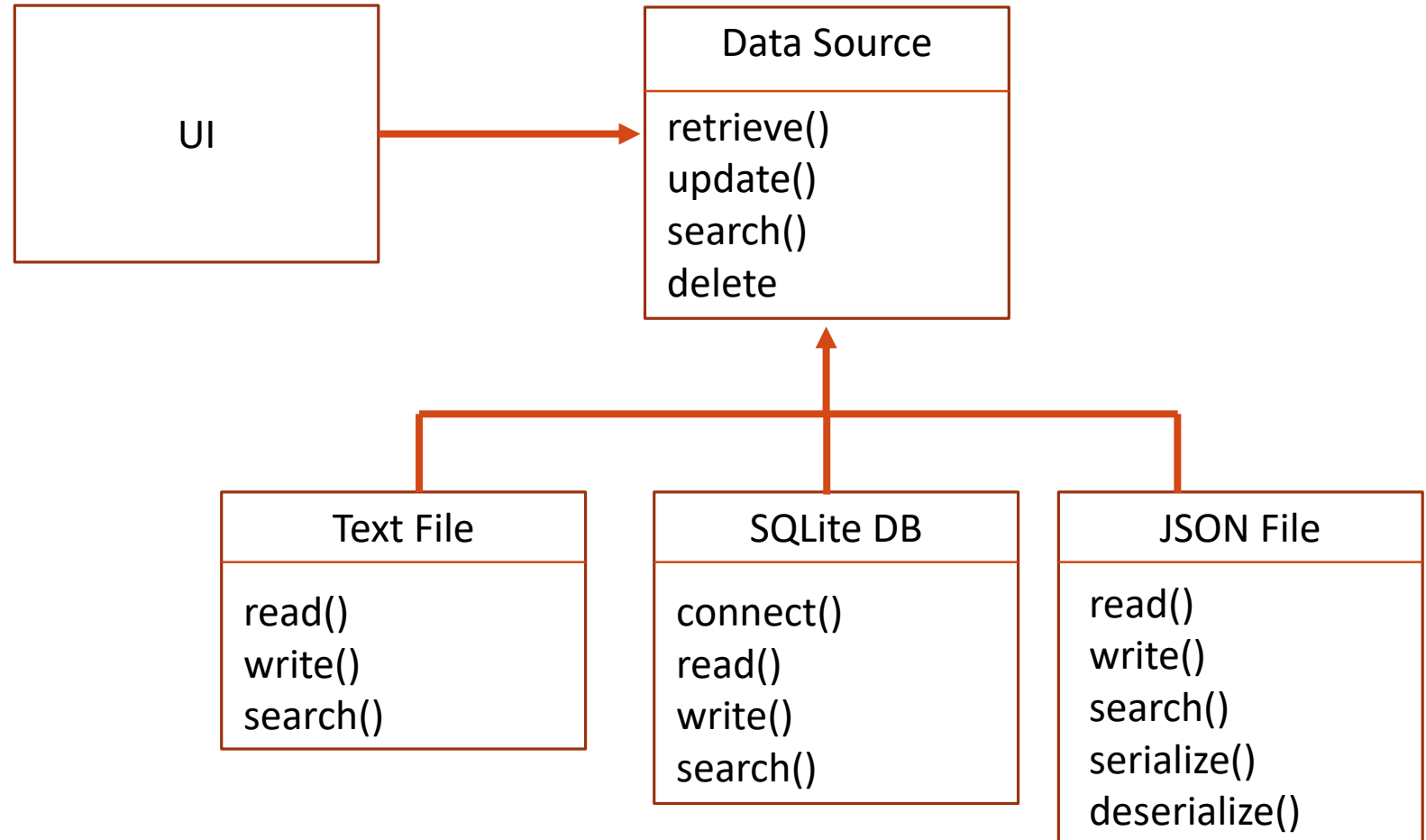
Say we were developing an app with a UI that was populated with some data from a text file.

After a few releases and years of development, for some reason we decide to switch out to a more secure source of data , perhaps an encrypted JSON file or even perhaps a SQLite Database. We would have to edit all the modules/classes that dealt with our app's UI!

# Recap: Decoupling dependencies

We can decouple our system to instead depend on a data source which is an **abstraction** that **hides** different data sources.

Our UI would just have to be **dependent on a common interface** provided by the Data Source and not be concerned with how that data source is implemented

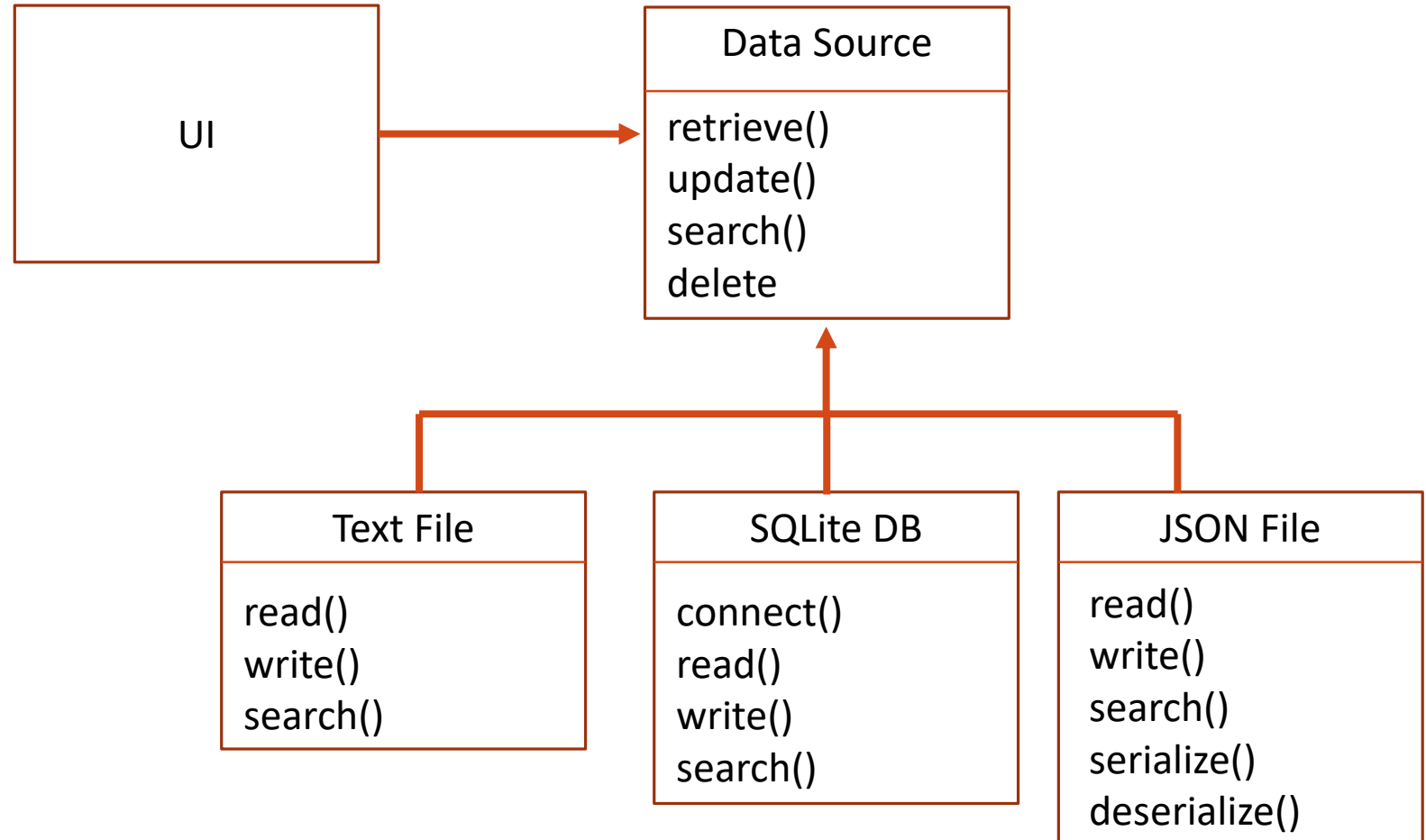


# Recap: Decoupling dependencies

---

We actually used at least 3 **SOLID Design Principles** here!

Today we're going to learn what these principles are and why are they so important.



# Recap: Dependencies and Coupling

---

What are they?

Why do we need to be aware of them?

Are they undesirable?

# Recap: Dependencies & Coupling

---

## Dependency:

When one entity depends on another entity.

If entity A uses entity B, then A is said to be dependent on B.

## Coupling:

If Entity A is coupled with Entity B then Entity A will not be able to function without Entity B, **even if it was replaced** with another similar Entity. Entity A would need to be refactored to use the new Entity

Coupling is a form of a **strong** dependency.

We usually want to avoid this as it hinders us from creating flexible, dynamic, maintainable code.



# SOLID Design Principles

---

- A set of guidelines to help us write Maintainable, decoupled, flexible Object Oriented Code
- Created by Robert C. Martin, also known as Uncle Bob amongst developers.
- Highly respected and has a bunch of books on writing clean code.



## **S**ingle Responsibility Principle

A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)



## **O**pen / Closed Principle

A software module (it can be a class or method) should be open for extension but closed for modification.



## **L**iskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.



## **I**nterface Segregation Principle

Clients should not be forced to depend upon the interfaces that they do not use.

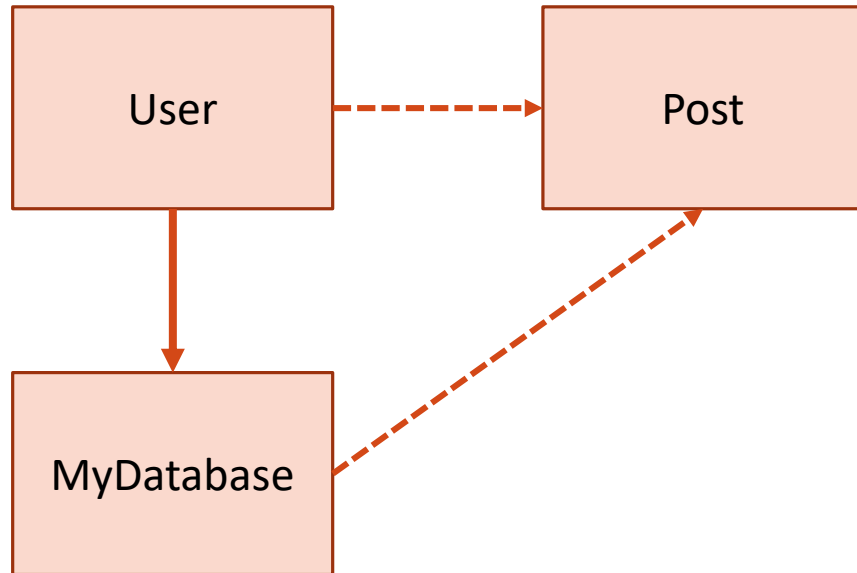


## **D**ependency Inversion Principle

Program to an interface, not to an implementation.

# Say we had a system where a user could write and create forum posts

---



```
class User:

    def __init__(username, password, name, email):

        self.username = username

        self._password = password

        self._name = name

        self._email = email

        self._database = MyDatabase()
```

## Why is this code bad?

The User class is very coupled and dependent. It carries out a number of different roles and responsibilities. Changing any part of this system will probably cause a change in the User class as well.

```
def post_to_forum( message, subject):

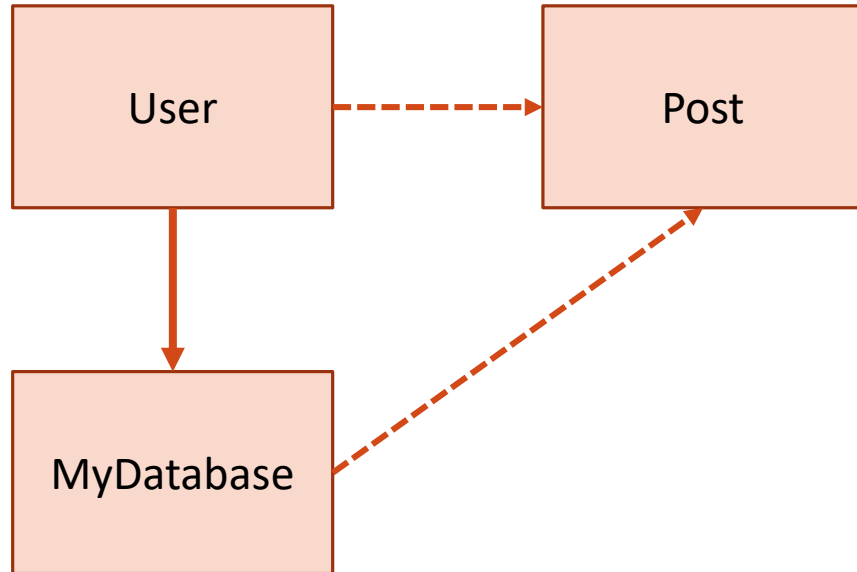
    post = Post(subject, message, self.username)

    self.database.add_simple_post(post)
```



# Say we had a system where a user could write and create forum posts

---



```
class User:
```

```
    def __init__(username, password, name, email):
```

```
        self.username = username
```

```
        self._password = password
```

```
        self._name = name
```

```
        self._email = email
```

```
        self._database = MyDatabase()
```

```
    def post_to_forum( message, subject):
```

```
        post = Post(subject, message, self.username)
```

```
        self.database.add_simple_post(post)
```

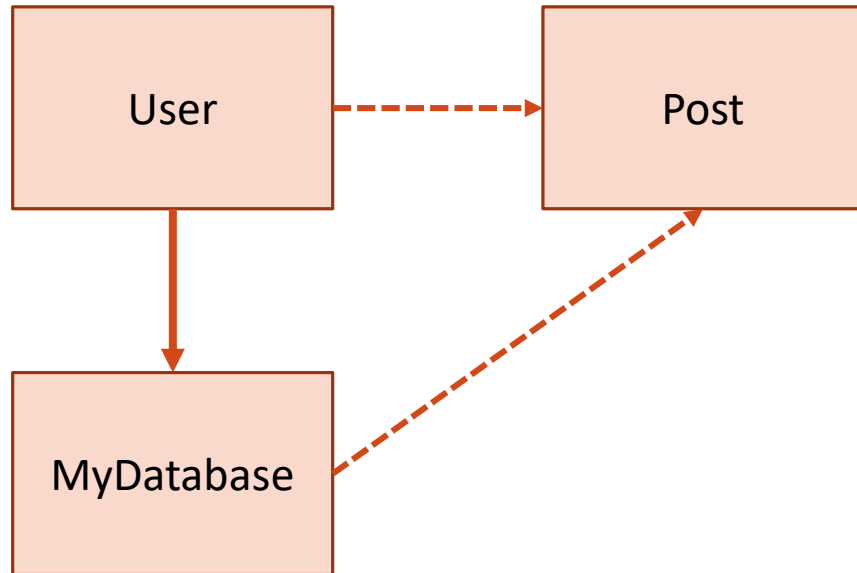
## Why is this code bad?

User class has code to:

- **be a user**

# Say we had a system where a user could write and create forum posts

---



```
class User:
```

```
    def __init__(username, password, name, email):  
        self.username = username  
        self._password = password  
        self._name = name  
        self._email = email  
  
        self._database = MyDatabase()
```

```
    def post_to_forum( message, subject):  
        post = Post(subject, message, self.username)  
        self.database.add_simple_post(post)
```

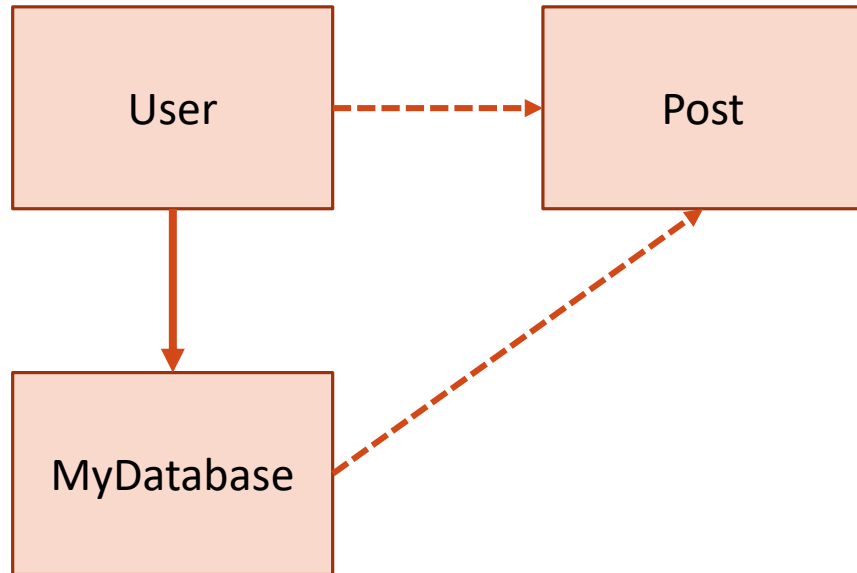
## Why is this code bad?

User class has code to:

- be a user
- **create a database**

# Say we had a system where a user could write and create forum posts

---



```
class User:
```

```
    def __init__(username, password, name, email):  
        self.username = username  
        self._password = password  
        self._name = name  
        self._email = email  
        self._database = MyDatabase()
```

```
    def post_to_forum( message, subject):
```

```
        post = Post(subject, message, self.username)  
        self.database.add_simple_post(post)
```

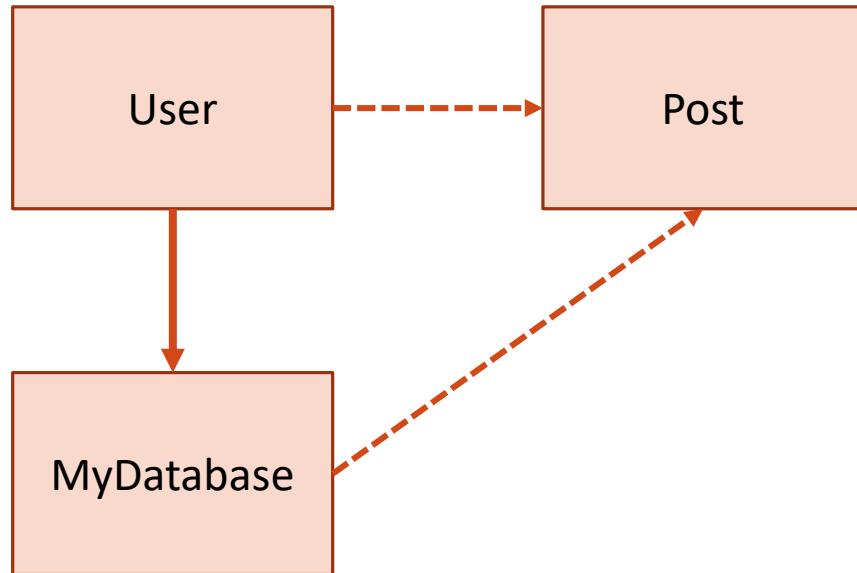
## Why is this code bad?

User class has code to:

- be a user
- create a database
- **Post to a forum**

# Say we had a system where a user could write and create forum posts

---



The **User** class should focus on only **being a user**

```
class User:

    def __init__(username, password, name, email):

        self.username = username

        self._password = password

        self._name = name

        self._email = email

        self._database = MyDatabase()

    def post_to_forum( message, subject):

        post = Post(subject, message, self.username)

        self.database.add_simple_post(post)
```

# Single Responsibility Principle

“The Single Responsibility Principle requires that each class is responsible for only one thing.”

Ask yourself, are there multiple reasons to change my class?

There should only be one reason to change your class. Each class should have one and only one responsibility.

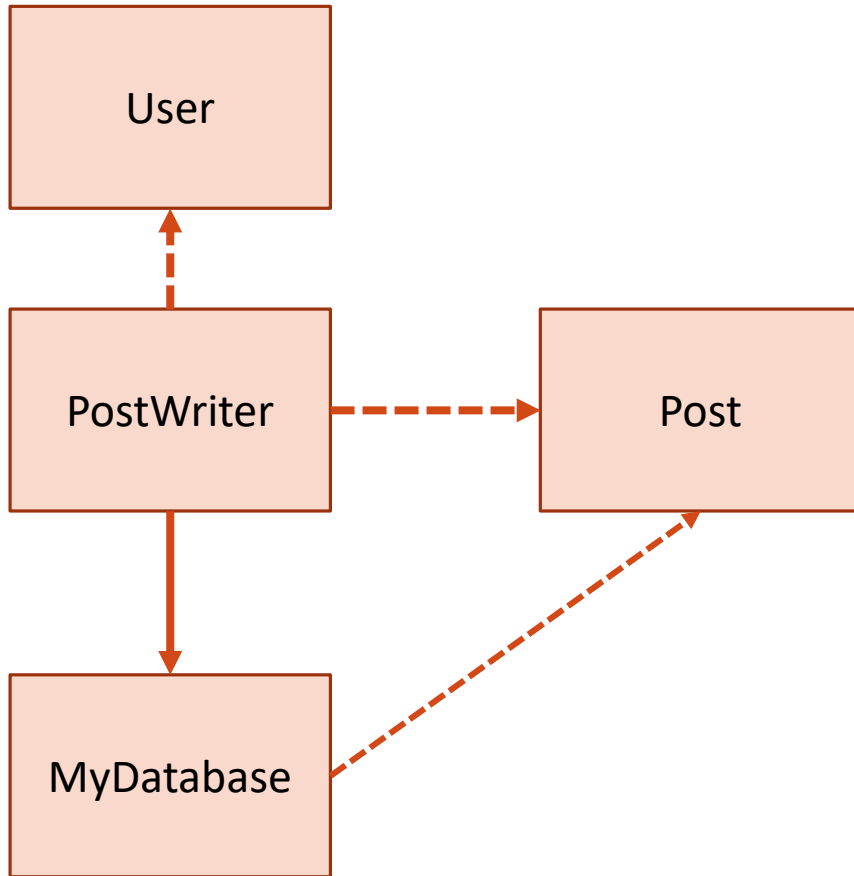


## SINGLE RESPONSIBILITY PRINCIPLE

**Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.**

# We Split the User Class! Pt.1

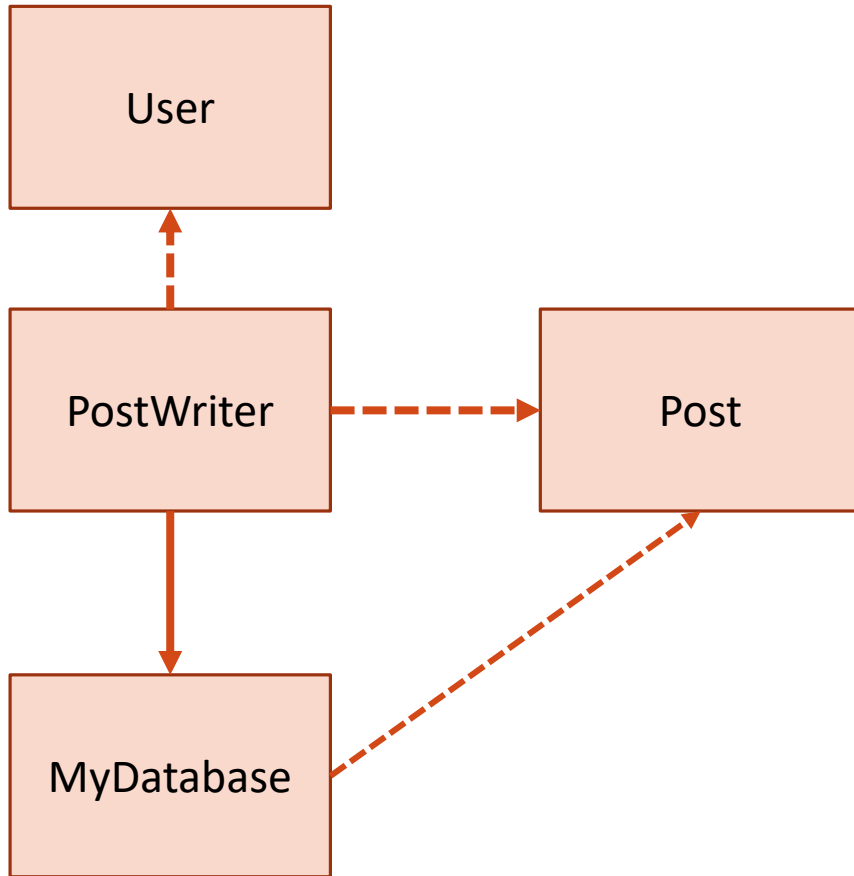
---



```
class User:
    def __init__(username, password, name, email):
        self.username = username
        self._password = password
        self._name = name
        self._email = email
```

# We Split the User Class! Pt.2

---



```
class PostWriter:
    def __init__(self, database):
        self.database = database

    def post_to_forum(user, message, subject):
        post = Post(subject, message, user.username)
        self.database.add_post(post)
```

# Now, Say we had a UI widget class....

---

Say we had a graphical user interface, where each UI element was a UI widget.



UIPanelWidget

```
class UIWidget:
    def draw_widget(self):
        if self._type == "Button":
            # complicated button draw logic here
        else:
            # complicated draw logic for a
            # generic UI box (Panel)
```



# Now over time, what if our UIPanelWidgetClass looked like this?

---

```
class UIPanelWidget:
    def draw_widget(self):
        if self._type == "Button":
            #complicated button draw logic here
        elif self._type == "Text Box":
            # complicated draw logic for a generic UI box
        elif self._type == "ScrollBar":
            # complicated draw logic for a generic UI box
        elif self._type == "Label":
            # complicated draw logic for a generic UI box
        else:
            # complicated draw logic for a generic UI box
```



This is terrible design! If we wanted to change the logic / algorithm behind drawing a button, we would need to edit this massive draw method.

An error or bug here would effect all the widgets on screen and crash our whole system!

# Open Closed Principle

---

*The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

Once a class is made, it's behaviour shouldn't be modified to support special cases. It should be **CLOSED for modification**.

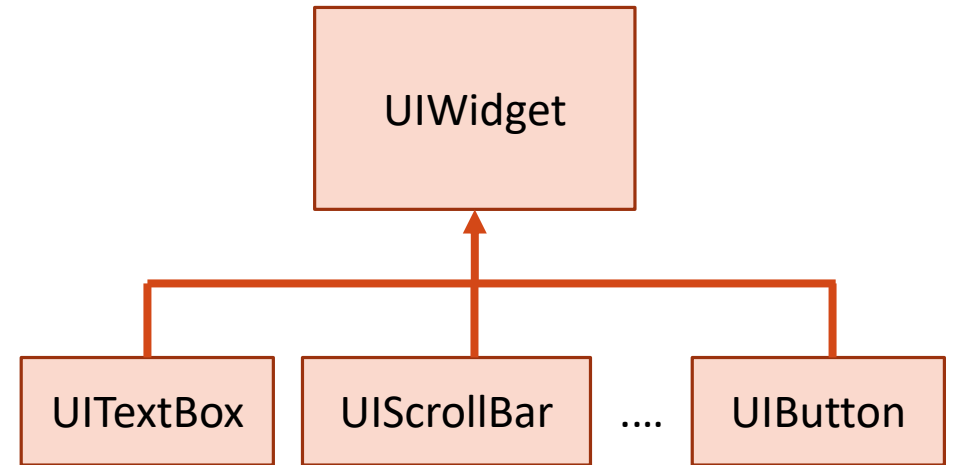
Instead, we extend (or inherit) from the class and override behaviours. That is, it's **OPEN for extension**.



# We use Inheritance (and overriding)!

---

```
class UIWidget(abc.ABC):  
    @abc.abstractmethod  
    def draw_widget(self):  
        pass  
  
class UITextBox(UIWidget):  
    def draw_widget(self):  
        #complicated draw code for text box goes here  
  
class UIScrollBar(UIWidget):  
    def draw_widget(self):  
        #complicated draw code for scroll bar goes here
```

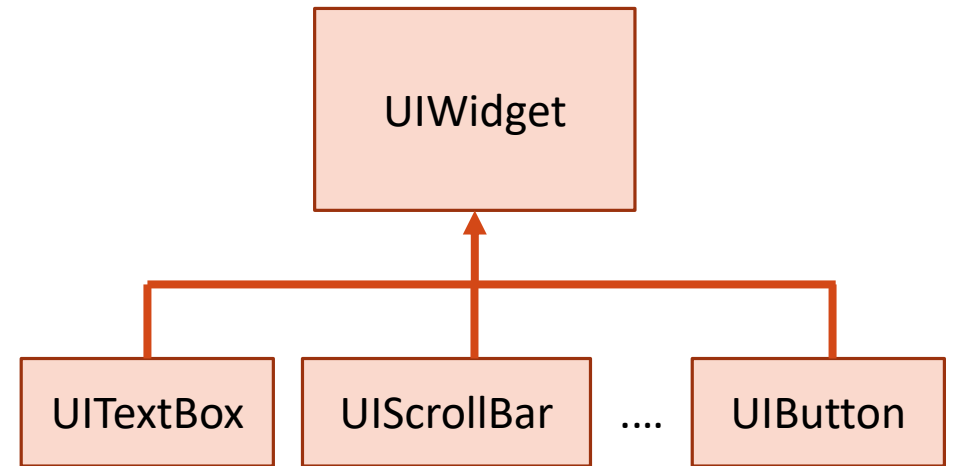


Now, any change to UIButton would not effect any other widget. Each widget class is modular and decoupled. It even follows the Single Responsibility Principle!

# We use Inheritance (and overriding)!

---

```
class UIWidget(abc.ABC):  
  
    @abc.abstractmethod  
    def draw_widget(self):  
        pass  
  
Class UITextBox(UIWidget):  
  
    def draw_widget(self):  
        #complicated draw code for text box goes here  
  
Class UIScrollBar(UIWidget):  
  
    def draw_widget(self):  
        #complicated draw code for scroll bar goes here
```



We used abstraction to do this. The UIWidget class is now an Abstract Base Class ensuring that each UI component that inherits from it has the same interface.

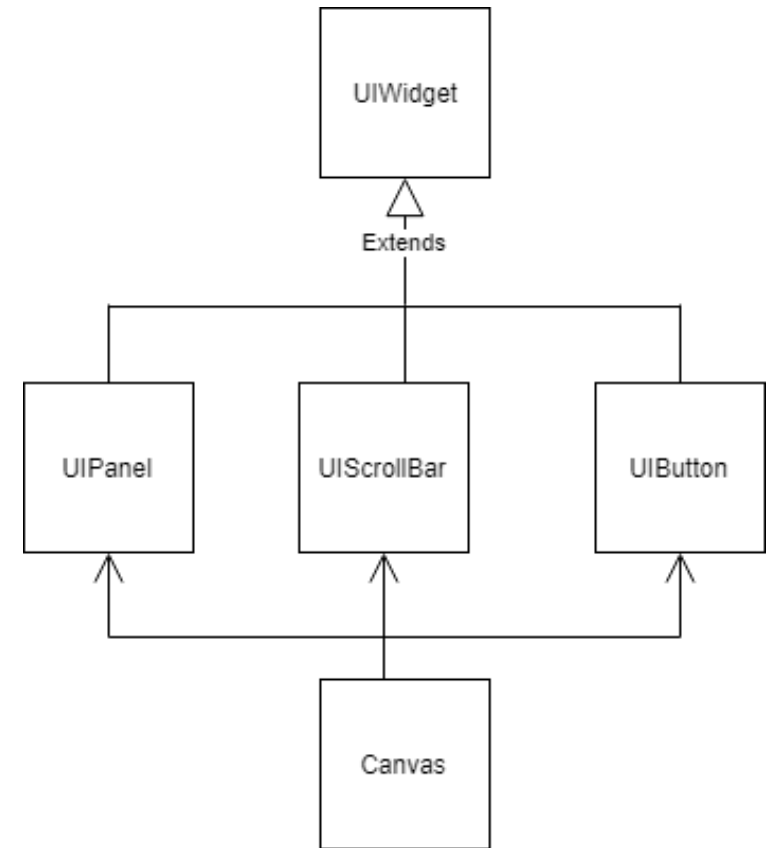
# Now how do we draw all these widgets?

---

Class Canvas:

```
def __init__(self, panel, scrollbar, ..., button):  
    self.panel = panel  
    self.scrollbar = scrollbar  
    self.button = button  
    ..  
  
def draw_screen(self):  
    self.panel.draw()  
    self.scrollbar.draw()  
    self.button.draw()  
    ..  
    ..
```

Say we have a Canvas Object that needs to draw the screen. Would this be a good way of drawing all the widgets?



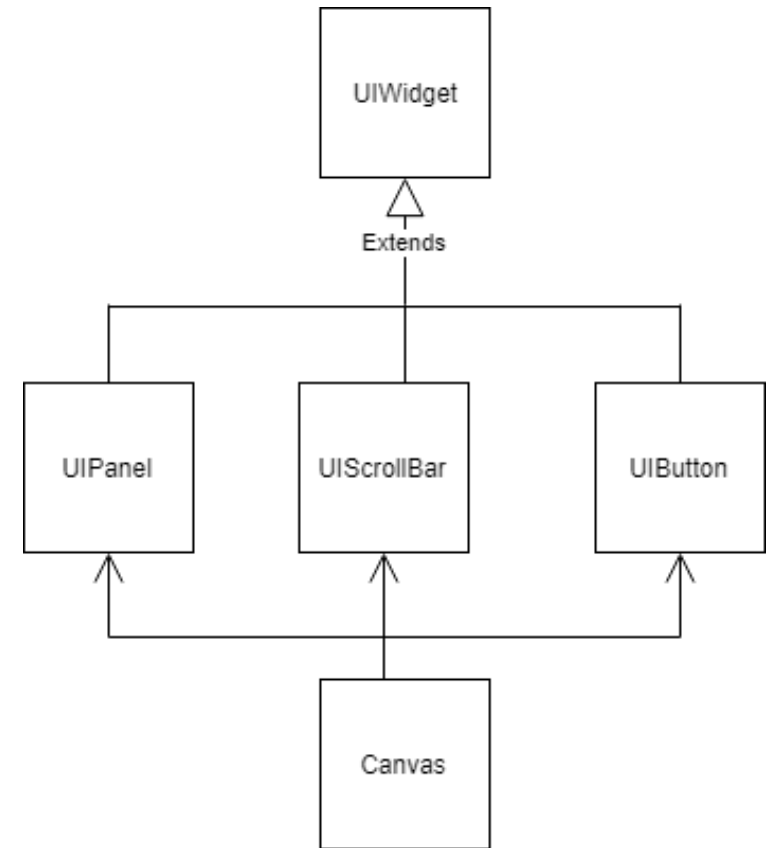
# Now how do we draw all these widgets?

---

Class Canvas:

```
def __init__(self, panel, scrollbar, ..., button):  
    self.panel = panel  
    self.scrollbar = scrollbar  
    self.button = button  
    ..  
  
def draw_screen(self):  
    self.panel.draw()  
    self.scrollbar.draw()  
    self.button.draw()  
    ..  
    ..
```

This would be terrible! Why inherit if you have to keep an explicit reference to each subtype?



# How would you solve this?

---

Thoughts?

# Liskov Substitution Principle

---

In programming, the Liskov substitution principle states that if  $S$  is a subtype of  $T$ , then objects of type  $T$  may be replaced (or substituted) with objects of type  $S$ .

Or

Objects in a program should be replaceable with instances of their base types without altering the correctness of that program.

Simply put, you should be able to refer to all the different UI components as its parent class, `UIWidget`





# In Python this is easy, since we don't worry about types!

---

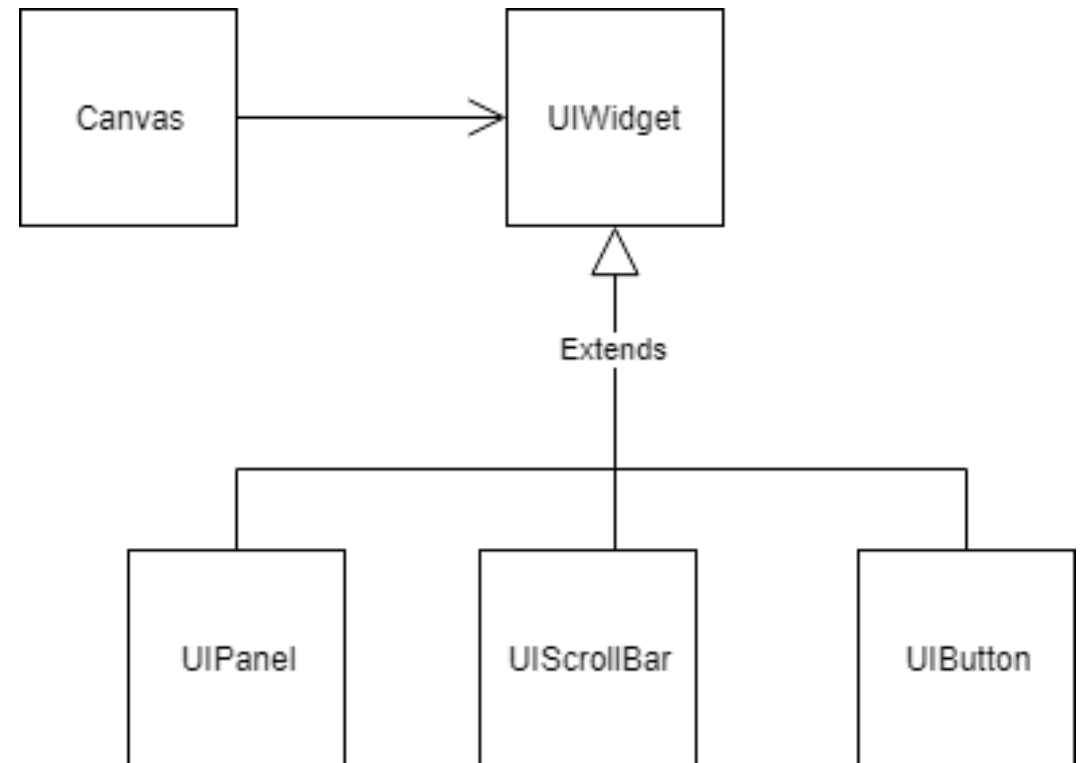
Class Canvas:

```
def __init__(self, widget_list):  
    self.ui_widgets = widget_list  
  
def draw_screen(self):  
    for widget in self.ui_widgets:  
        widget.draw()
```

Now we just have a widget list filled with all the different kinds of UI Widget objects.

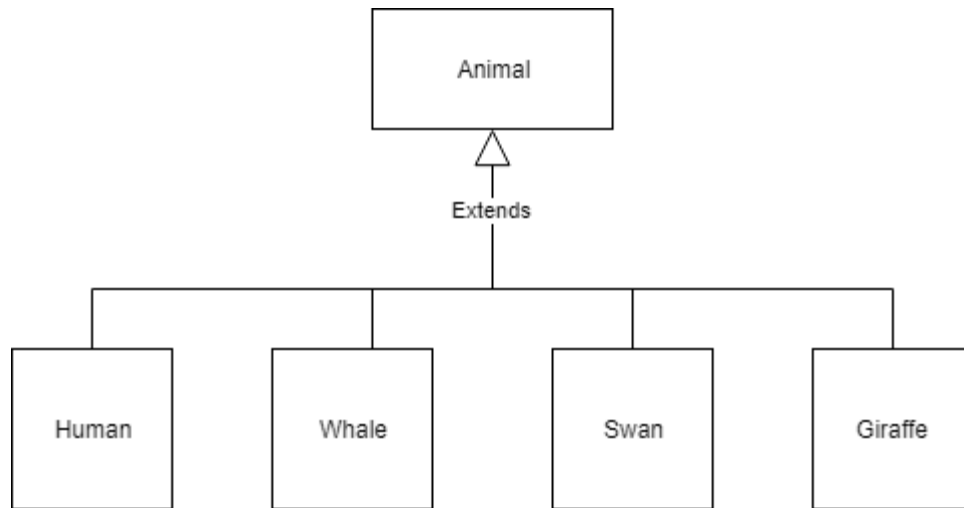
Since they all have the same interface (draw method) they can be treated as if they were their base class (UIWidget).

We don't care about the subtype.



# Now Say We Were Modeling A Zoo...

---



So we followed all our design principles and wrote the animal ABC Class.  
But this is terrible! **Now our Human class needs to override the fly method. Our Fish can walk!**

```
class Animal(abc.ABC):  
  
    def breathe(self):  
        # common breathing code here  
  
    def eat(self):  
        # common eating code here  
  
    @abc.abstractmethod  
    def walk(self):  
        # common walking code  
  
    @abc.abstractmethod  
    def swim(self):  
        # common swimming code  
  
    @abc.abstractmethod  
    def fly(self):  
        # common flying code
```

# How would you solve this?

---

Thoughts?

# Interface Segregation Principle

---

Many client-specific interfaces are better than one general-purpose interface.

Or

The interface segregation principle states that no client should be forced to depend on methods it does not use.

Put more simply: Do not add additional functionality to an existing interface by adding new methods.

This is the Single Responsibility Principle for Interfaces/Abstractions!



# Back To Our Zoo!

```
class Walkable(abc.ABC):
```

```
    @abc.abstractmethod
    def walk(self):
        pass
```

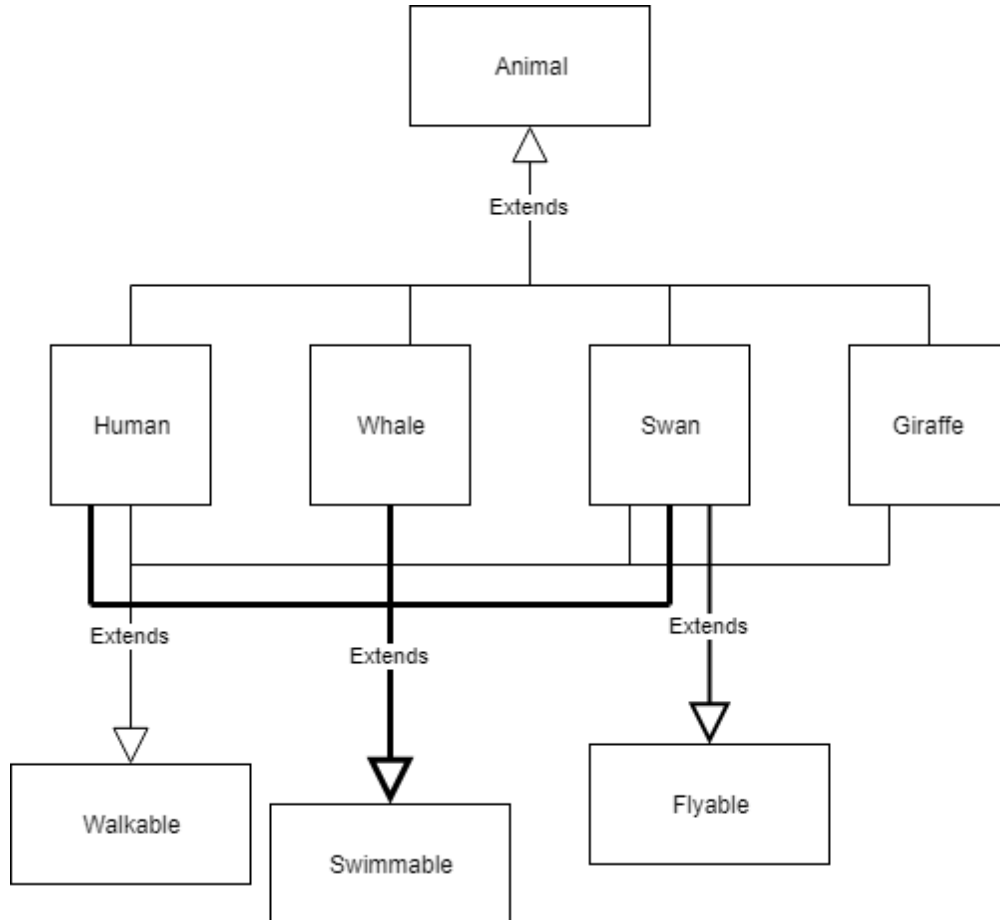
```
class Swimmable(abc.ABC):
```

```
    @abc.abstractmethod
    def swim(self):
        pass
```

```
class Human(Animal, Walkable, Swimmable):
```

```
    def walk(self):
        # overridden walking code
```

```
    def swim(self):
        # overridden swimming code
```



Instead of polluting the animal base class with additional functionality, we create separate interfaces (read: ABC) that handle different responsibilities.

This may seem difficult to maintain, but in fact it isn't.

# Dependency Inversion Principle

---

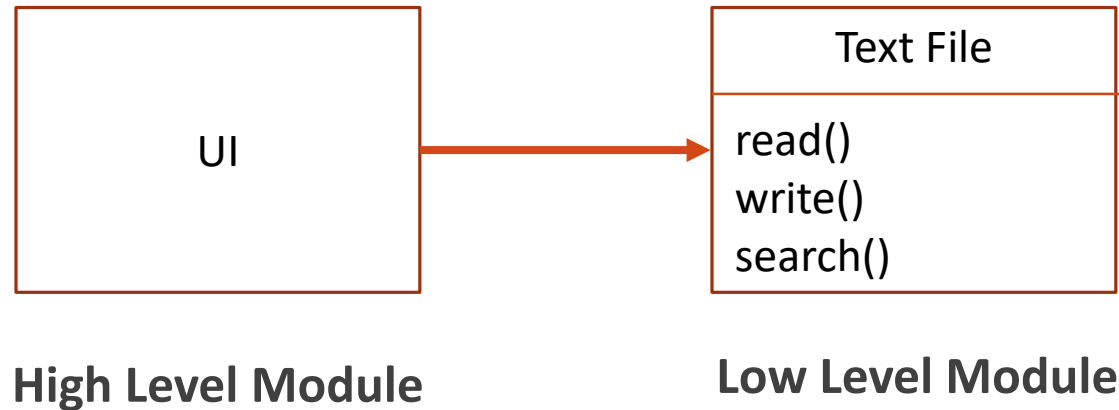
Dependency Inversion Principle can be thought as a combined effect of the previous principles.

Specifically, the Open Closed Principle and Liskov Substitution Principle

Before we dive into this, we need to look at a few concepts.

# High Level & Low Level Modules

---



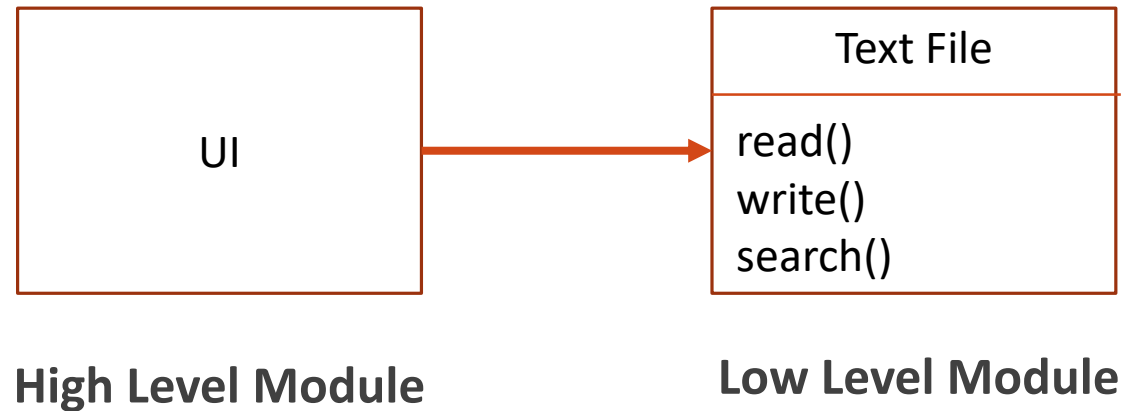
**High Level Modules** are the ones that contain complex logic made up of small 'chunks' of simpler code. For example, our UI class

A **Low Level Module** on the other hand, is a class that encapsulates some simple atomic behaviour. Such as writing and reading from a file, displaying an image, etc.

We generally compose high level modules with low level modules. That is, **high level modules are Coupled and Dependent on low level modules**

# High Level & Low Level Modules

---



**This is not very good design.**

We don't want this dependency. We want to be able to change or replace the lower level modules without tampering with the complex code in the higher level modules



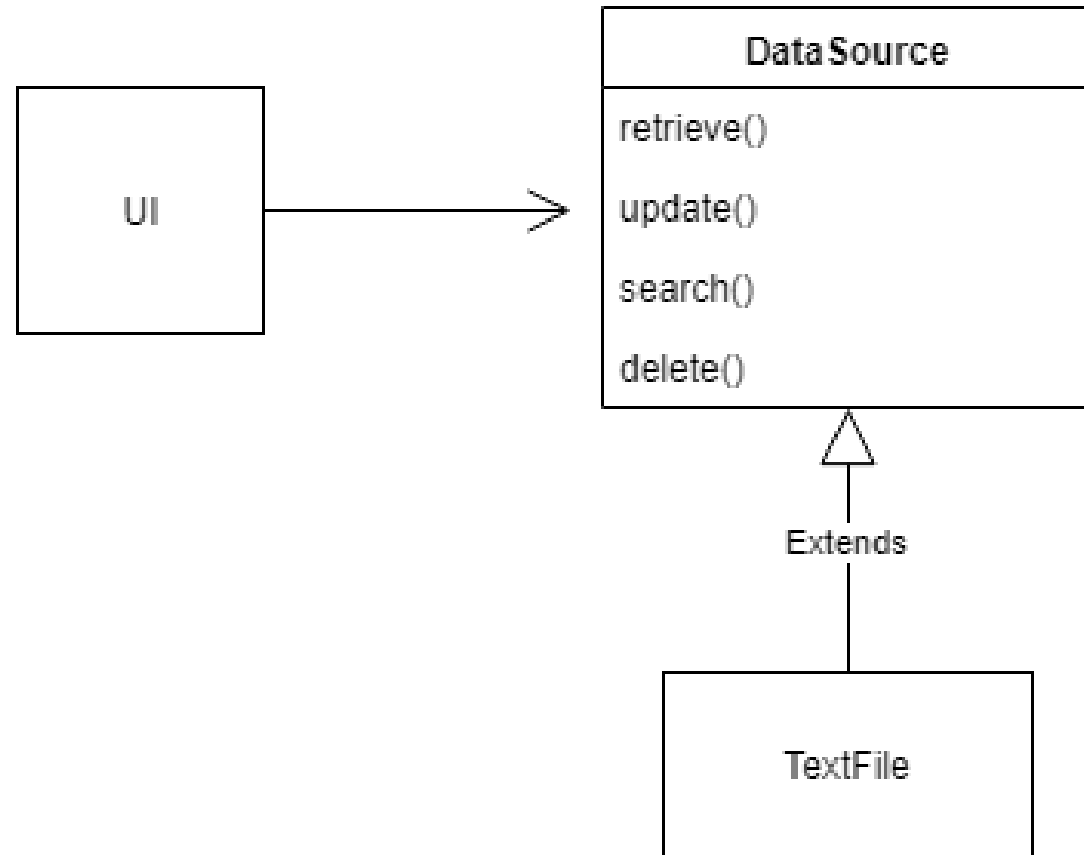
# We inversed the dependencies!

---

We introduced an abstract layer, a middle man that does not depend on anything.

We made both our high level and low level modules depend on the abstract instead.

Now we can switch out either side, the UI and the text file could be replaced with something completely different



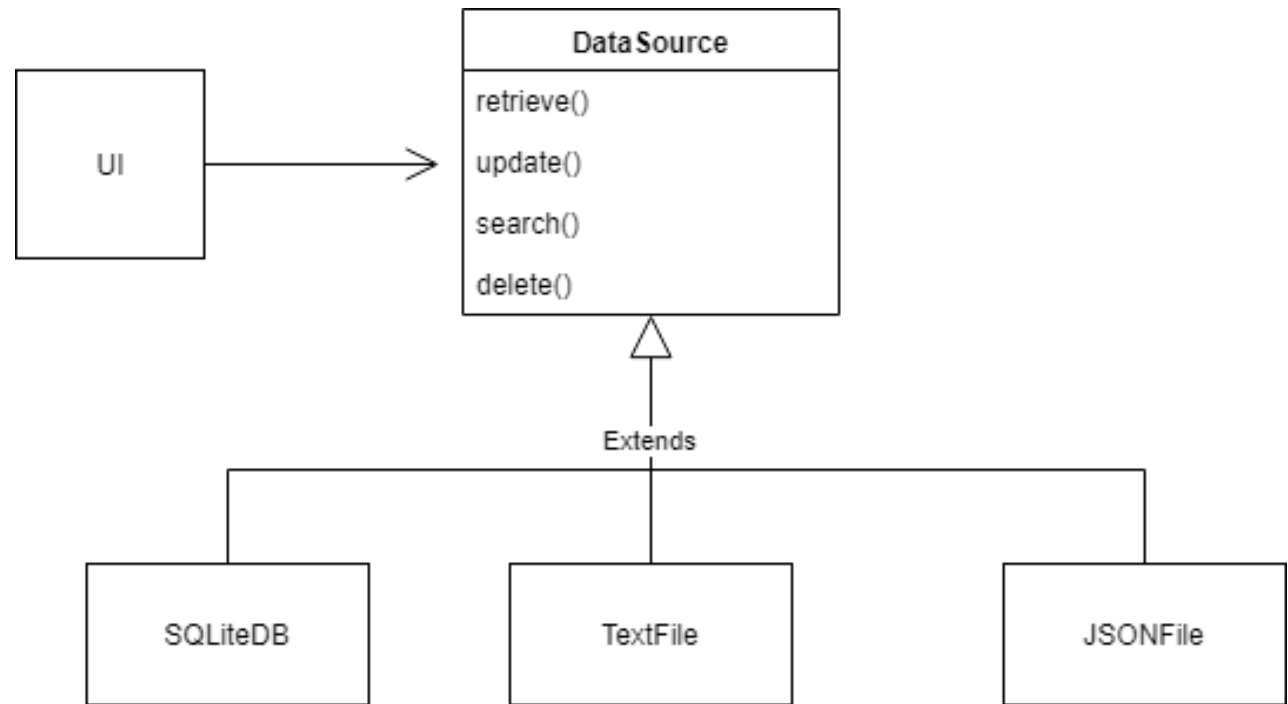
# We inversed the dependencies!

---

Everyone here has seen a diagram like this many times now.

We have been implementing this whole week.

We can even see the building blocks that are Liskov Substitution and Open Closed Principles.



# Dependency Inversion Principle

---

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

We call classes that implement details 'Concrete Classes'. In the example we just saw, the Data Source is our abstract class or Interface, and the UI and TextFile classes are concrete classes.

Abstract classes declare an interface.

Concrete classes actually contain the code that implements the details of the interface.

# Right, now your turn! Say we had a character creation UI in a game.

We have a Character Creation UI, where a player creates their character. When they click the create button, it gets saved to a save file.

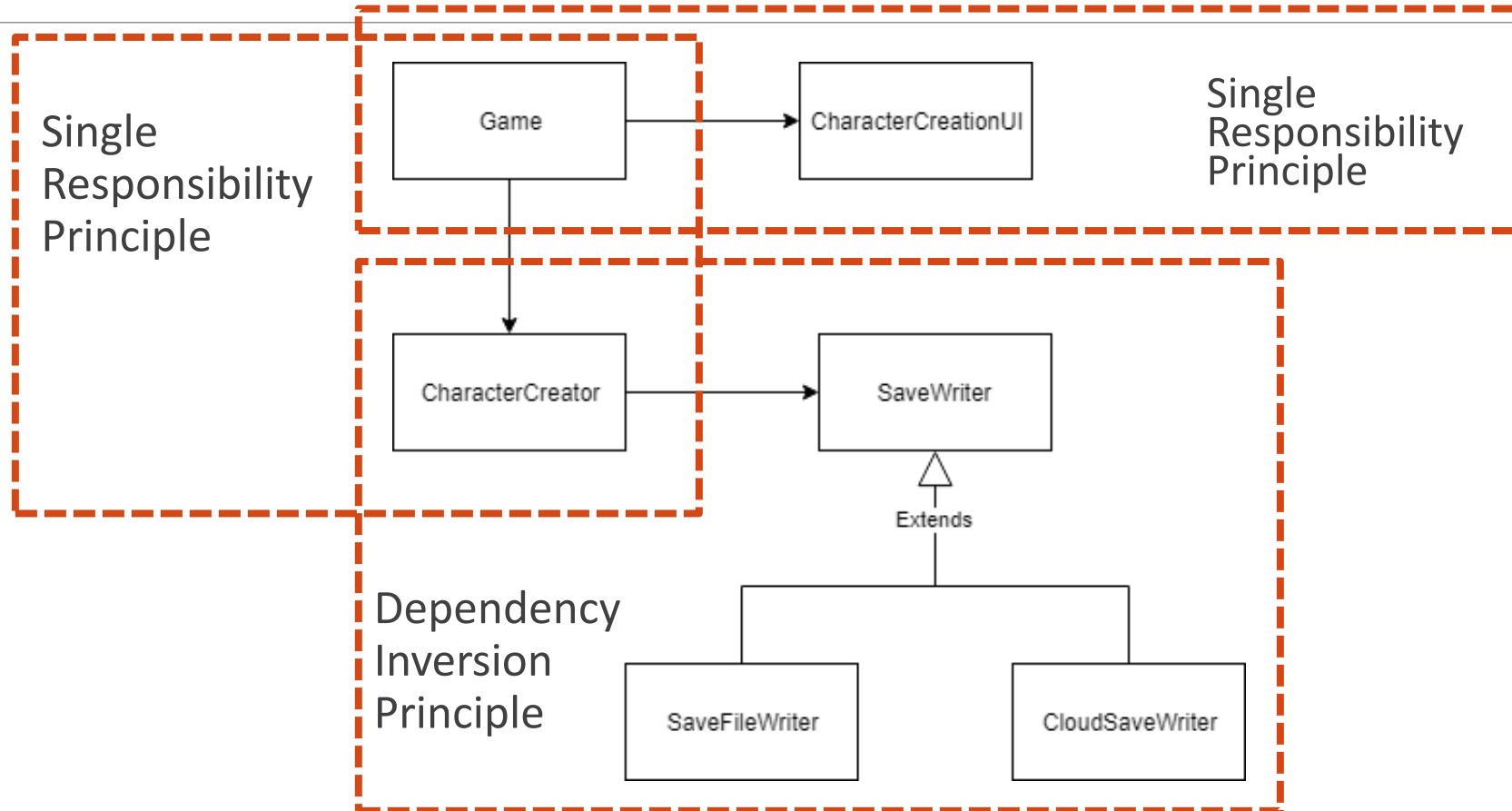
Apply everything you've learnt so far to "fix" this terrible design.



CharacterCreationUI

SaveFileWriter

# My Solution



# Law of Demeter

---

Real life - Wallet example

# Law of Demeter

---

“Each unit should have only limited knowledge about other units: only units “closely” related to the current unit. Each unit should only talk to its friends; don’t talk to strangers.”

Don’t have train wrecks like these:

```
obj.getX().getY().getZ().doSomething();
```

Or

```
MyCharacterUI.GetCharacterData().CreateCharacter().Save()
```

Poor Demeter is often forgotten, some people even call it “Suggestion of Demeter” since this is difficult to avoid

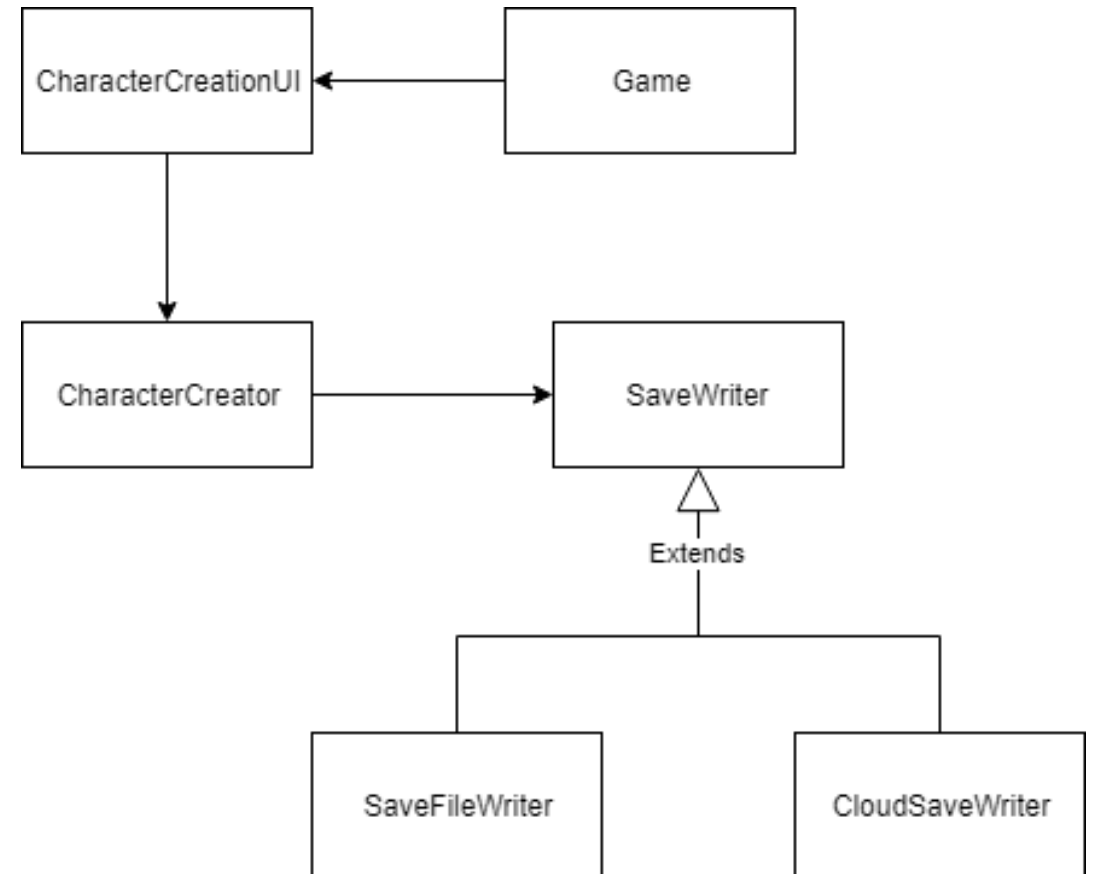
# Law of Demeter – Example Code

---

class game:

```
def create_character(self):  
    data = self.character_ui.get_character_creator().  
        save(self.character_ui  
            .get_character_creator()  
            .create_character()  
        )
```

This is obviously bad and hard to read code





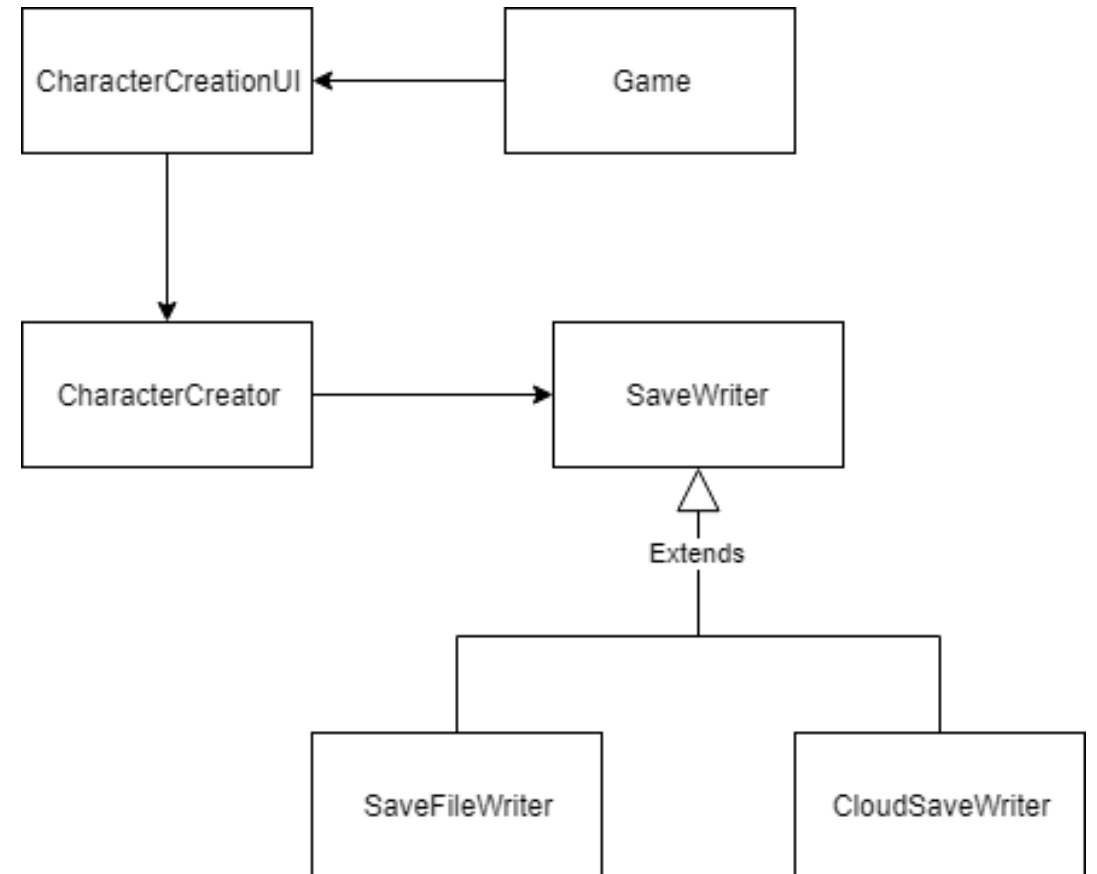
# Law of Demeter – Example Code

```
class game:
    def create_character(self):
        data = self.character_ui.get_character_creator().
            save(self.character_ui
                .get_character_creator()
                .create_character()
            )
```

This is obviously bad and hard to read code

I also had a hard time getting the lines to wrap in powerpoint. Bad for many reasons

Any change in the save method, which is all the way in SaveWriter, could have a ripple effect of changes all the way to our game class



# Law of Demeter – Example Code - Solution

```
class game:
```

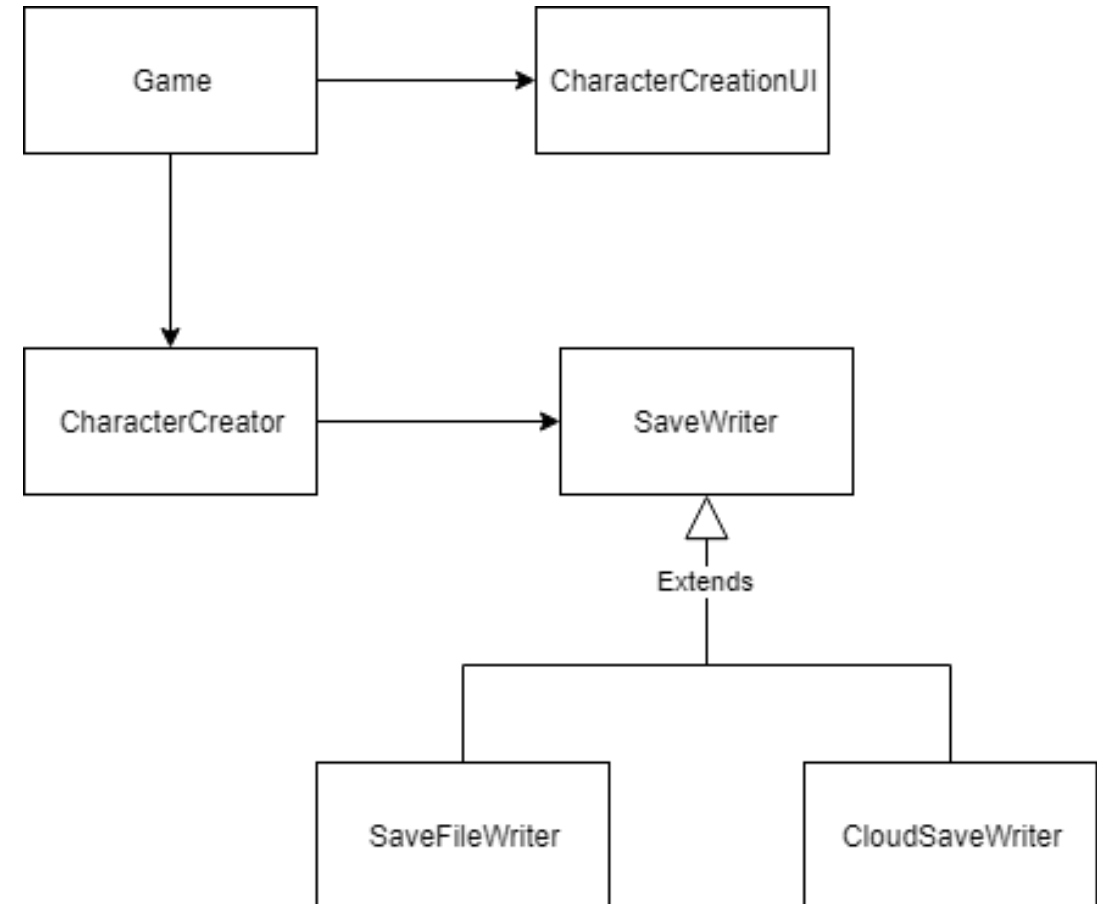
```
    def create_character(self):  
        data = self.character_ui.get_data()  
        self.character_creator  
        .initiate_character_creation(data)
```

```
class CharacterCreationUI:
```

```
    def get_data(self):  
        return self.character_data
```

```
class CharacterCreator:
```

```
    def initiate_character_creation(self, data):  
        my_char = self.create_character(data)  
        self.save_manager.save(my_char)
```



# Function parameters

---

PASS BY VALUE? PASS BY REFERENCE? OR SOMETHING ELSE???

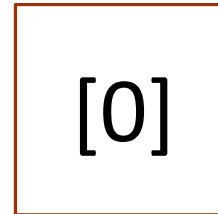
# Pass by value

---

Check out this code, and let's assume it's pass by value

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham

A new object is created in memory, and the variable ham is assigned to it

<http://stupidpythonideas.blogspot.com/2013/11/does-python-pass-by-value-or-by.html>

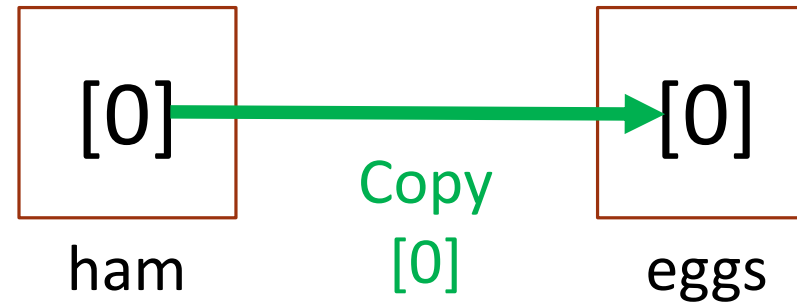
# Pass by value

---

Check out this code, and let's assume it's pass by value

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



eggs is a new parameter variable. A new piece of memory is created and a copy of [0] is placed into the memory of eggs

# Pass by value

---

Check out this code, and let's assume it's pass by value

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0]

ham

[0, 1]

eggs

eggs has the value 1 appended to it. This does not change the original ham

# Pass by value

---

Check out this code, and let's assume it's pass by value

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0]

ham

[2, 3]

eggs

eggs has now changed its value to [2,3]

# Pass by value

---

Check out this code, and let's assume it's pass by value

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0]

ham

[2, 3]

eggs

When we print ham, its value still remains as [0]



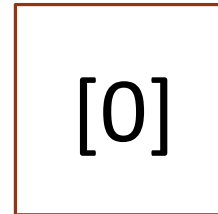
# Pass by reference

---

Check out this code, and let's assume it's pass by reference

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



A new object is created in memory, and the variable `ham` is assigned to it

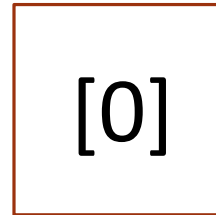
# Pass by reference

---

Check out this code, and let's assume it's pass by reference

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham  
eggs

eggs is a reference to ham. Eggs and ham are both referencing the same object in memory. There is NO COPYING

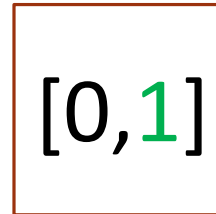
# Pass by reference

---

Check out this code, and let's assume it's pass by reference

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[0, 1]

ham  
eggs

eggs has the value 1 appended to it. This changes the original value ham is assigned to

# Pass by reference

---

Check out this code, and let's assume it's pass by reference

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham  
eggs

eggs has now changed its value to [2,3]. Again this changes the original value ham is assigned to

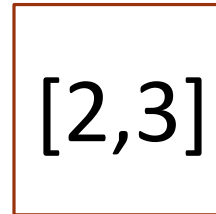
# Pass by reference

---

Check out this code, and let's assume it's pass by reference

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham  
eggs

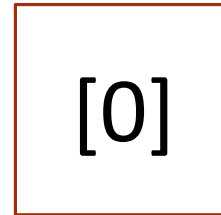
When we print ham, its value was changed within the function to [2,3]. These changes are reflected even outside the function

# Pass by ???

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham

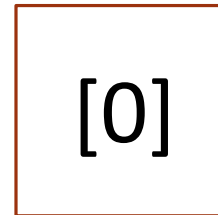
A new object is created in memory, and the variable `ham` is assigned to it

# Pass by ???

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham  
eggs

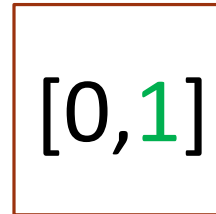
eggs is a new variable that points to the same value ham is pointing at. Eggs and ham are both pointing at the same object in memory. There is NO COPYING

# Pass by ???

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[0, 1]

ham  
eggs

eggs has the value 1 appended to it. This changes the original value ham is assigned to



# Pass by ???

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0,1]

ham

[2, 3]

eggs

The object [2,3] is created in memory and eggs has now changed to point to it

# Pass by ???

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0,1]

ham

[2, 3]

eggs

When we print ham, its value is [0,1]

# Pass by ???

---

Soooo, what is it called?

You might see some of the following ways to call it:

Call-by-Object

Call by Object Reference

Call by Sharing

Pass-by-object-reference

Passed by assignment

Some people don't bother giving it a name because it's an entirely different concept from pass by value or pass by reference

# That's it for Week 3

---

Remember Lab 2 is due Friday night

