Polymorphism

"Each class in a well-designed program denotes a distinct concept with its own set of responsibilities. Nevertheless, it is possible for two (or more) classes to share some common functionality. For example, the Java classes HashMap and TreeMap are distinct implementations of the concept of a "map," and both support the methods get, put, keySet, and so on. The ability of a program to take advantage of this commonality is known as polymorphism"

- *Sciore E. (2019) Polymorphism. In: Java Program Design. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-4143-1_2*

Poly-morph *(many-forms)*, is the concept of objects of the same class exhibiting different characteristics. Objects belonging to two different subclass that share the same parent, will inherit the same methods. Of course, due to the differences the subclass brings, the methods will change according to the nuances of the subclass.

The benefit of this concept is that we can build a program around an abstract superclass, and we can later pass different objects that extends the superclass into the program. These objects can belong to varying subclasses and be different from each other, but the implementation remains agnostic to these object-classes.

In my opinion, this is especially useful when:

1. The details of the subclass are not clear at the start of implementation
2. There are differences between subclasses, but they share some common functionality
3. You need a collection of different objects to use a related method (which can be inherited)

Head First Design Patterns explains this rather prolixly:

"The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code. And we could rephrase "program to a supertype" as "the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types!"

- *Welcome to Design Patterns: Intro to Design Patterns; Head First Design Patterns, 2nd Edition by Eric Freeman, Elisabeth Robson, Kathy Sierra, and Bert Bates. O' Reilly & Associates, Inc*

For example :

```
class BankAccount() :


 def __init__(self) :
  self.balance = 0.0
  self.customer_name = ""
  self.rate = 0.0

 def greet_customer(self) :
  print("Hello {} your balance is USD{}".format(self.customer_name, self.balance))

 def debit(self, amount) :
  self.balance -= amount

 def credit(self, amount) :
  self.balance += amount

 def add_interest(self) :
  self.balance += self.rate*self.balance
```

In this case, BankAccount is an abstract superclass. All accounts in this bank will have these basic functions.

```
class SavingsAccount(BankAccount) :


 def __init__(self) :
  super().__init__()
  self.num_transactions = 6
  self.rate = 0.042

 def greet_customer(self) :
  print("Hello Savings Account Customer {}, \
```

```
    your balance is USD{}. You have {} transactions \
    left this month.".format(self.customer_name, self.balance, self.num_transactions))

  def debit(self, amount) :
    if self.num_transactions==0 :
      print("You have used up all your transactions for this month. You cannot make anymore debits to this
account.")
    elif self.balance==0 or amount>self.balance :
      print("You have insufficient balance to perform this transaction")
    else :
      super().debit(amount)
      self.num_transactions -= 1

class InvestmentAccount(BankAccount) :

  def __init__(self) :
    super().__init__()
    self.leverage = 0.0
    self.rate = 0.42

  def greet_customer(self) :
    print("Hello Investment Customer {}, your current balance is USD{} and leverage is USD{}"
          .format(self.customer_name, self.balance, self.leverage))

  def credit(self, amount, is_loan=False) :
    super().credit(amount)
    if is_loan :
      self.leverage += amount

  def add_interest(self) :
    self.balance -= self.leverage*self.rate
```

In this case, both SavingsAccount and InvestmentAccount are from the same class BankAccount, but their implementation of the same credit(), debit() and add_interest() methods are very different. Let us consider 2 cases:

1. Adding interest at the end of every month.
2. Crediting a government incentive amount to each account.

```
SB = SavingsAccount()
IB = InvestmentAccount()

gov_subsidy = 1000.00

SB.credit(1000.00)
IB.credit(1000.00, True)

while True:

  if datetime.date.tomorrow("%D")== "01" :  #if tomorrow is the first of a month, i.e. today is the last day
    SB.add_interest()
    IB.add_interest()
```

In this case, both the accounts may get credited with the same government subsidy - but when the interest is calculated, the InvestmentAccount will actually lose value, and the SavingsAccount will gain in value.

- *This example is mine, but the idea was inspired from Sciore E. (2019) Polymorphism. In: Java Program Design. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-4143-1_2*

Method Overriding

In this case of polymorphism, a subclass can change the way a function behaves by re-defining a method from the superclass.

Method Overloading

A function can be re-defined in the same or different class, where these definitions differ in input parameters. In such a case, the program will interpret which method is being called by the parameters provided.

Duck Typing or Dynamic Typing

Languages like Python are not very strict about specifying the type of input parameters to a function. So, the program will run dynamically - and try to perform the operations in the method regardless of input. An error will only be raised when the operations in the method cannot be performed on the input object. Depending on their stance, one could argue that this is a special case of Polymorphism as well.

Coupling

"Coupling refers to how tightly different classes are connected to one another. Tightly coupled classes contain a high number of interactions and dependencies. Loosely coupled classes are the opposite in that their dependencies on one another are kept to a minimum and instead rely on the well-defined public interfaces of each other…

Legos, the toys that SNAP together would be considered loosely coupled because you can just snap the pieces together and build whatever system you want to. However, a jigsaw puzzle has pieces that are TIGHTLY coupled. You can't take a piece from one jigsaw puzzle (system) and snap it into a different puzzle, because the system (puzzle) is very dependent on the very specific pieces that were built specific to that particular "design". The legos are built in a more generic fashion so that they can be used in your Lego House, or in my Lego Alien Man."

- *Seth Long's Blog, https://megocode3.wordpress.com/2008/02/14/coupling-and-cohesion/*

A tightly coupled program means different objects in the program share resources and dependencies with each other. It is very difficult to make changes to a class in such cases, as dependent objects (that share resources) will be affected by this change.

Now any piece of software will have some degree of coupling because they need to interact and call each other's methods for exchanging data. But as a designer, we can follow intelligent designs to ensure that it is kept to a minimum.

One such design idea is interfaces, that allows classes to remain independent of each other. In such a case, the classes using the same interface will be able to communicate with each other, regardless of how it is implemented in code.

Rather than change a dependent object due to a change request to another object, you can implement the CR to adopt the interface demands.

- *This StackOverflow answer inspired this example*

Let us look at a highly coupled example of a Librarian class and Student class.

```
class Book() :

  '''Class Definition'''



class Librarian() :

  def __init__(self, book_listing) :
    self.collection = book_listing

  def issue(self, book) :
    return self.collection.pop(book)



  def take_in(self, book) :
    self.collection.append(book)



class Student() :

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

  def borrow_book(self, book) :
    self.librarian.issue(book)

  def return_book(self, book) :
    self.librarian.take_in(book)
```

Now, if there is a change to the Librarian.issue() method, there may be unexpected consequences for the Student.borrow_book(). Also the latter cannot even be instantiated without having a librarian object.

But we can try to decouple this, by adding a BookShelf class, where the Librarian can place books and Student can take the book!

```
class Book() :
  '''Class Definition'''

class BookShelf() :

  shelf = []

  def place_book(self, book) :
    self.shelf.append(book)

  def take_book(shelf, book) :
    return self.shelf.pop(book)


bookshelf = BookShelf()

class Librarian() :

  def __init__(self, book_listing) :
    self.collection = book_listing

  def stock(self, book) :
    self.collection.pop(book)
```

```
    bookshelf.place_book(book)


class Student() :

  def __init__(self, librarian) :
    self.librarian = librarian
 self.book = None

  def borrow_book(self, book) :
    self.book = bookshelf.take_book(book)

  def return_book(self, book) :
    self.book = None
    bookshelf.place_book(book)
```

In this example, the Student and the Librarian had access to the same BookShelf class. If the Librarian class was changed to do something else - for instance if the class changed its responsibility completely - to an extent where it does not even have a Librarian.stock method, the Student class will not be affected by it at all!