# Name: Julian Siddiqui

# General Problems In the before code:

**Singleton** **pattern** was not implemented

A problem in the **before**.**py** was it does not have a base class and no inheritance.

# Finding Appropriate Objects

## Evaluate

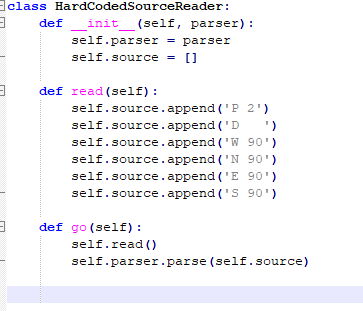
**After.py**

Graphical user interface

Description automatically generated with low confidence

* Adapter pattern helped FIND APPROPRIATE OBJECTS in the **Drawer** by looking at the specific request from the client code, checking which object is implementing that specific request and then looking at the object that is not implementing that request. But in the after code, it is applying It inherits from multiple classes, so it inherits attributes from both its parents and can perform their methods as well. This means less code is needed to be written and it also has extended capabilities since it is inherited from many parents. In the after.py in the reader file it is importing from ABC import and abstract method, it also importing the guidoSingleton import Singleton and class in the after.py then it added the base class which is abstractSourceReader which inherits the ABC it is also using use \_\_init\_\_ function (self, parser) which is the base class then after that the abstract method applied with comments which are the read(self) method and the hooks method raise NOT IMPLEMENTED which is for the **Raise NotImplementedError** solved the requirement derived classes to override all the methods from the AbstractDrawer parent class. then it implemented the template method which for Applying the Template pattern on class AbstractSourceReader made the code more reusable. And then it The method **read** in the **HardCodedSourceReader** class, append function is not good for reusability because when you want to draw more lines means more appends. HardCodedSourceReader(AbstractSourceReader, Singleton) which is for the **Singleton** Pattern solved the control of object creation in the class HardCodedSourceReader.

**Before.py**

* 
* The main problem with before.py code in Drawers is using the if and elif which makes the code bigger and more complex and there is no base class in drawers there is nothing about the pattern, for example, abstract pattern adopter pattern.
* **Factory method (Creational Pattern).TIGr:**
  + can be extended into the abstract factory since the method since assessed from within the object

there is no importing ABC, and abstractMethod there is no importing singleton guidoSingleton in the readers.py there is no base class in the readers.py there is no template method implemented in the reader.py there is no inheritance in readers.py hook method even not implemented in readers.py. there is no adapter class in the readers.py. the def go(self) method is missing the template method.

# Determining Object Granularity

## Evaluate

Graphical user interface, text, application

Description automatically generated**<After.py>**

When the **decorator** pattern was applied in the drawers.py the **OBJECT** **GRANULARITY** became bigger because three new classes had been added. The granularity on the client side becomes shorter. In the new classes that have been added, the granularity of the method with decorator pattern made the objects in printing Tigr which makes proper objects with make\_drawer or make\_parser, make reader then it inheritance the TurtleTigr class which is the printing TIGr and return is the TKDrawer()

* It is also importing the ABC method and abstract method even in the same file it is importing the PrintingDrawer, TKDrawer, and TurtleDrawer classes added which inherit the methods from AbstractDrawer and made the code ROBUST and FLEXIBLE. The Abstract Factory pattern helped control the classes of objects Abstract Factory Pattern solved the design problem in class PrintingTIGR is the base class then it inherited the Turtle Tihgr class and TK tigr class with decorator pattern It inherits from multiple classes, so it inherits attributes from both its parents, and can perform their methods as well. This means less code is needed to be written and it also has extended capabilities since it is inherited from many parents. can do the **Factory method (Creational Pattern).TIGr:**
  + can be extended into the abstract factory since the method since assessed from within the object

## Evaluate

**<Before.py>**

There is no **decorator** pattern was applied in the drawers.py the **OBJECT** **GRANULARITY** became bigger because three new classes had been added. The granularity on the client side becomes shorter. In the new classes that have been added, the granularity of the method with decorator pattern made the objects in printing Tigr which makes proper objects with make\_drawer or make\_parser, make reader then it inheritance the TurtleTigr class which is the printing TIGr and return is the TKDrawer()

* It is also not importing the ABC method and abstract method even in the same file it is also not importing the PrintingDrawer, TKDrawer, and TurtleDrawer classes which inherit the methods from AbstractDrawer and made the code ROBUST and FLEXIBLE. The Abstract Factory pattern helped control the classes of objects Abstract Factory Pattern solved the design problem in class PrintingTIGR is the base class then it inherited the Turtle tigr class and TK tigr class with decorator pattern It inherits from multiple classes, so it inherits attributes from both its parents, and can perform their methods as well. This means more code is written and it is also in the client code Parser could be refactored using the template method or extended using the decorator.
* The **OBJECT** **GRANULARITY** for the client code is less complex and smaller.
* The problem in the **TIGr** was ABC was imported but never used.
* The method **read** in the **HardCodedSourceReader** class, append function is not good for reusability because when you want to draw more lines means more appends.
* **Factory method (Creational Pattern).TIGr:**

The **object** **granularity**. for the client is more complex and bigger. The client code granularity of client code is too big. because it is not implementing

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

# Specifying Object Interface

## Evaluate

**After.py**

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generatedThe object interface that was required is the drawing. **Singleton** Pattern solved the control of object creation in the class HardCodedSourceReader. And It inherits from multiple classes, so it inherits attributes from both its parents and can perform their methods as well. This means less code is needed to be written and it also has extended capabilities since it is inherited from many parents. **Abstract Factory Pattern**

* The adapter and Abstract Factory pattern were done excellently on **drawers.py**. because PrintingDrawer, TKDrawer, and TurtleDrawer classes had been added which inherit the methods from AbstractDrawer and made the code ROBUST and FLEXIBLE.
* The Abstract Factory pattern helped control the classes of objects
* Abstract Factory Pattern solved the design problem in the class HardCodedSourceReader by

**Adapter Pattern**

* When the **Adapter** pattern was applied in the drawers.py the **OBJECT** **GRANULARITY** became bigger because three new classes had been added.
* The adapter pattern on **drawers**.py solved the problem by enabling it to work with multiple programs.
* The adapter pattern helped FIND APPROPRIATE OBJECTS in **Drawer** by looking at the specific request from the client code, checking which object is implementing that specific request and then looking at the object that is not implementing that request and finding a way how it can adapt to the specific request.
* The adapter pattern helped implementation by converting the interface into another interface that the client expects.
* The adapter pattern helped implementation by implementing the target interface.
* It inherits multiple classes, so it inherits attributes from both its parents and can perform their methods as well. This means less code is needed to be written and it also has extended capabilities since it is inherited from many parents.

## Evaluate

**Before**.**py**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. The **HardCodedSourceReader** class is hard coded which is not flexible and reusable.
2. A problem in the **before**.**py** was it doesn’t have a base class and no inheritance.
3. No inline comments in the **before**.py which makes the code harder to read. Both developer and client-wise.
4. The **OBJECT** **GRANULARITY** for the client code is more complex and bigger.
5. **Singleton** Pattern was not implemented.
6. The **Drawer** class on lines 8 and 20 is using if operation to check the object type.
7. The **Drawer** methods such as select\_pen, pen\_down, etc. use elif operation to check the object type. there are no comments at the end of the line
8. The specific request from the client does not match the interface name from the drawing object The interface name in the s and the interface name in the drawing are incompatible, but it has a similar purpose. The drawing is not implementing the adapter pattern and there is not flexible and cannot be extended because the method implementation is incompatible. The method in the drawline is broken down into 2 different if and elif function. In the before drawer, there is no class implemented which can inherit the abstractdrawer interface even if it is not importing the ABC.

# Specifying Object Implementation

## Evaluate

Graphical user interface, text, application

Description automatically generated

**<AFTER.PY>**

* The adapter pattern helped implementation by implementing the target interface. Separated different implementations into different objects which follow the same interface as the printing Drawer. Abstract Factory Pattern solved the design problem in class HardCodedSourceReader by (interfaces Abstract Source reader, Singleton).
* **Singleton creational pattern: Source Reader and Parser.**
  + No increase in flexibility but likely no need for multiple instances for the parser currently since re-instantiating it does change the specified the while otherwise using the same instance. Source reader uses singleton but only for the **hard-coded source reader** implementation.
* The adapter pattern helped implementation by converting the interface into another interface that the client expects.
* The adapter pattern on **drawers**.py solved the problem by enabling it to work with multiple programs.
* **Raise NotImplementedError** solved the requirement-derived classes to override all the methods from the AbstractDrawer parent class.
* The drawers.py code became more **ROBUST** because it uses inheritance which inherits the attributes/methods from its parent. Which means it can perform their methods as well.

## Adapter Structural Pattern: Drawer

## Evaluate

**<Before.py>**

**The problem in the before code**

**Hooks** are methods that should not need to be implemented. This is not how they have been implemented in the source reader, as read is commented as a hook.

**How to design pattern solved design problem**

The Adapter pattern solved the design problem in the **drawers** by letting the different adapters (PrintingDrawer, TKDrawer, and TurtleDrawer) work with many Adaptees.

**Evaluation**

* Regarding the hook method, Primitive methods are those which must be overridden in implementations, hooks are those that can be overridden but are not required to be done as such. This is partially a labelling issue since read can be considered a primitive method.
* The adapter pattern was done well on the drawers because 3 different types of drawer classes had been added which inherits its methods from the AbstractDrawer base class and made the code more ROBUST and FLEXIBLE.
* The Drawer class in the before code was using IF/ELIF statements to check the object type. This was solved by the adapter pattern by having their interfaces and inheriting the methods from the AbstractDrawer base class.

# Putting Reuse Mechanisms to Work

## Evaluate

## <After.py>

**Template** **Pattern**

* Applying the Template pattern to the class AbstractSourceReader made the code more reusable.

**How to design pattern solved design problem**

* The adapter and Factory pattern were implemented to the after code to solve the inheritance and reusability design problem. Adding responsibilities to objects make the code more flexible too. In addition, creating objects inside a class with a factory method is always more flexible than creating an object directly.
* Applying the Template pattern in the AbstractSourceReader class go method made the code more reusable.

**Evaluation**

The after code has a base class and inheritance implemented which now enables the classes to be reused, extended and/or modified. Although inheritance requires creating a new class for each additional responsibility and making the code more complex and bigger.







## Evaluate

**Before.py**

**The problem in the before code**

The before code does not have base classes to inherit from. This is a problem because inheritance enables you to create new classes that reuse, extend, and modify the behaviour defined in other classes. Therefore, this makes the code both not flexible and reusable.







Graphical user interface, text

Description automatically generated with medium confidence

# Designing for change

**The problem in the before code**

Text

Description automatically generated with low confidence

Source reader does not have a base class separated from the **HardCodedSourceReader** class. This is a problem because the class cannot be reused.

* The poor naming convention of variables in **TIGr.py** on lines 9 – 12. Could have named it as makeDrawer, makeParser, and make the reader.
* The parser could be refactored using the template method or extended using a decorator.
* The **OBJECT** **GRANULARITY** for the client code is less complex and smaller.
* The problem in the **TIGr** was ABC was imported but never used.
* The method **read** in the **HardCodedSourceReader** class, append function is not good for reusability because when you want to draw more lines means more appends.

Text

Description automatically generated**How to design pattern solved design problem**

* The template pattern solved the design problem in the **AbstractSourceReader** class. It made the code more extendable than before since it can be expanded with new implementations of the template and hook methods.
* **Raise NotImplementedError** solved the requirement-derived classes to override all the methods from the AbstractDrawer parent class.
* The drawers.py code became more **ROBUST** because it uses inheritance which inherits the attributes/methods from its parent. Which means it can perform their methods as well.

Factory pattern solved the design problem in the **TIGr** by providing a flexible alternative to subclassing and extending its functionality. In addition, the code became flexible by adding responsibilities to the objects.

A picture containing graphical user interface

Description automatically generated

**Evaluation**

The source reader is more extendable than before since it can be expanded with the new implementation of the template and hook methods

The **Drawer** **class** on lines 8 and 20 is using the if operation to check the object type.

The **Drawer** methods such as select\_pen, pen\_down, etc. use elif operation to check the object type.

Graphical user interface, text, application

Description automatically generated

# A Common Design Vocabulary

## < after.py >

**The problem in the before code**

The **naming** **convention** on the **tigr** was done poorly. This could have been **TIGrFactory** instead of PrintingTIGr. So that it contains the right information about where the Factory pattern was applied which will make it easier to distinguish what it does on the code.

1. The **HardCodedSourceReader** class is hard coded which is not flexible and reusable.
2. A problem in the **before**.**py** was it doesn’t have a base class and no inheritance.
3. No inline comments in the **before**.py which makes the code harder to read. Both developer and client-wise.
4. The **OBJECT** **GRANULARITY** for the client code is more complex and bigger.
5. **Singleton** Pattern was not implemented.
6. The **Drawer** class on lines 8 and 20 is using if operation to check the object type.
7. The **Drawer** methods such as select\_pen, pen\_down, etc. use elif operation to check the object type. there are no comments at the end of the line

The specific request from the client does not match the interface name from the drawline object The interface name in the s and the interface name in the drawline are incompatible, but it has a similar purpose. The drawline is not implementing the adapter pattern and there is not flexible and cannot be extended because the method implementation is incompatible. The method in the drawline is broken down into 2 different if and elif function. In the before drawer, there is no class implemented which can inherit the abstractdrawer interface even if it is not importing the ABC.

Text, letter

Description automatically generated

Poor variable names in the client code. This is a problem because the code is harder to read and understand. The variable names could be the\_drawer, the\_parser and such, instead of using just a letter only to make it easier to understand. Which follows the PEP8 naming convention as well.

Text

Description automatically generated

**How do design patterns solve design problems**

**Evaluation**

Using the actual name of the pattern in the class name was done excellently and made it easier to distinguish its purpose.

Text

Description automatically generated

Text

Description automatically generated

.In the after.py in the reader file it is importing from ABC import and abstract method

The **select\_pen** method name can be changed to **pen\_colour** instead to name it according to its purpose and information to the code. And also improve readability.

Text

Description automatically generated

**the pens** object name can also probably be changed to **colour** because it contains the colour of the pen. Therefore, it is named according to its content.



The method and class names were done excellently because this improved readability and follow the PEP8 style guide for python coding as well.

The object interface that was required is the drawline. **Singleton** Pattern solved the control of object creation in the class HardCodedSourceReader. And It inherits from multiple classes, so it inherits attributes from both its parents and can perform their methods as well. This means less code is needed to be written and it also has extended capabilities since it is inherited from many parents. **Abstract Factory Pattern**

* The adapter and Abstract Factory pattern were done excellently on **drawers.py**. because PrintingDrawer, TKDrawer, and TurtleDrawer classes had been added which inherit the methods from AbstractDrawer and made the code ROBUST and FLEXIBLE.
* The Abstract Factory pattern helped control the classes of objects
* Abstract Factory Pattern solved the design problem in the class HardCodedSourceReader by

**Adapter Pattern**

* When the **Adapter** pattern was applied in the drawers.py the **OBJECT** **GRANULARITY** became bigger because three new classes had been added.
* The adapter pattern on **drawers**.py solved the problem by enabling it to work with multiple programs.

The adapter pattern helped FIND APPROPRIATE OBJECTS in **Drawer** by looking at the specific request from the client code, checking which object is implementing that specific request and then looking at the object that is not implementing that request and finding a way how it can adapt to the specific request

A picture containing text

Description automatically generated

# A Documentation and Learning Aid

**The problem in the before code**

**<Before.py>**

No **inline** **comments** in all the before code. This is a problem because it makes the code harder to read and understand.

There is no **decorator** pattern was applied in the drawers.py the **OBJECT** **GRANULARITY** became bigger because three new classes had been added. The granularity on the client side becomes shorter. In the new classes that have been added, the granularity of the method with decorator pattern made the objects in printing Tigr which makes proper objects with make\_drawer or make\_parser, make reader then it inheritance the TurtleTigr class which is the printing TIGr and return is the TKDrawer()

* It is also not importing the ABC method and abstract method even in the same file it is also not importing the PrintingDrawer, TKDrawer, and TurtleDrawer classes which inherit the methods from AbstractDrawer and made the code ROBUST and FLEXIBLE. The Abstract Factory pattern helped control the classes of objects Abstract Factory Pattern solved the design problem in class PrintingTIGR is the base class then it inherited the Turtle Tihgr class and TK tigr class with decorator pattern It inherits from multiple classes, so it inherits attributes from both its parents, and can perform their methods as well. This means more code is written and it is also in the client code Parser could be refactored using the template method or extended using the decorator.
* The **OBJECT** **GRANULARITY** for the client code is less complex and smaller.
* The problem in the **TIGr** was ABC was imported but never used.
* The method **read** in the **HardCodedSourceReader** class, append function is not good for reusability because when you want to draw more lines means more appends.
* **Factory method (Creational Pattern).TIGr:**

The **object** **granularity**. for the client is more complex and bigger. The client code granularity of client code is too big. because it is not implementing

**How to design pattern solved design problem**

**Evaluation**

**After.py**

Inline comments were added in the after code which improve the code readability both developer and client-wise.

In the client code, 3 different drawers are written but only one is implemented with the help of comments. As result, this made the client code more flexible, improved readability and made it easier to understand what is going on.

When the decorator pattern was applied in the drawers.py the OBJECT GRANULARITY became bigger because three new classes had been added. The granularity on the client side becomes shorter. In the new classes that have been added, the granularity of the method with decorator pattern made the objects in printing Tigr which makes proper objects with make\_drawer or make\_parser, make reader then it inheritance the TurtleTigr class which is the printing TIGr and return is the TKDrawer()

• It is also importing the ABC method and abstract method even in the same file it is importing the PrintingDrawer, TKDrawer, and TurtleDrawer classes added which inherit the methods from AbstractDrawer and made the code ROBUST and FLEXIBLE. The Abstract Factory pattern helped control the classes of objects Abstract Factory Pattern solved the design problem in class PrintingTIGR is the base class then it inherited the Turtle Tihgr class and TK tigr class with decorator pattern It inherits from multiple classes, so it inherits attributes from both its parents, and can perform their methods as well. This means less code is needed to be written and it also has extended capabilities since it is inherited from many parents. can do the Factory method (Creational Pattern).TIGr:

o can be extended into the abstract factory since the method since assessed from within the object

A screenshot of a computer

Description automatically generated with low confidence

Docstrings were also added at the bottom of the code. Which makes it easier to understand the capabilities of the class or a function.

The docstring below states how to improve the reusability and extendibility of the code which serves as a learning aid to the code above it.

Logo

Description automatically generated with medium confidence

The docstring below states how to improve the robustness of the code which serves as a learning aid to the code above it.

A picture containing timeline

Description automatically generated

In the after code, there are missing comments in the **tigr** where the Factory method was applied. It could have the comments which are shown in the screenshot below.

Text

Description automatically generated with low confidence

The comments were done well in the **pygame** **drawer** after the code where the Adapter pattern was applied because there are inline comments. This improved the readability of the code. Although some comments are missing such as the request, specific request, and implementation which are shown in the screenshots below.

From **PrintintDrawer**:



From **TurtleDrawer**:

Text

Description automatically generated

From **TKDrawer:**

A picture containing text

Description automatically generated

Text

Description automatically generated

# An Adjunct to Existing Methods

**The problem in the before code**

These objects in the **Drawer** class are hard coded and were done poorly. This is a problem because it is not flexible and not good for reusability.



**How to design pattern solved design problem**

**Evaluation**

In the **AbstractDrawer** class, the abstract methods such as select\_pen, pen\_up, etc. have **Raise** **NotImplementedError** which was done well because it will throw an exception when you try to run it until you do so. This removes a lot of silent errors in the after code.

# A Target for Refactoring

**The problem in the before code**

The **IF** **statements** in the before code is still in the after code which can be a target for **refactoring**. This is a problem because having many IFs can potentially expose bugs in the future.

Text

Description automatically generated

The methods and classes in the before code uses **IF/ELIF statements** to check the object type which can also be a target for **refactoring**. This is a problem because if we have more types then it will continue to grow and potentially introduce bugs or errors.

Text

Description automatically generated

**How to design pattern solved design problem**

Singleton pattern solved the design problem by improving the flexibility of the code because the **HardCodedSourceReader** class has the flexibility to change the instantiation process.

**Evaluation**

* The **IF** **statements** in the **Parser** class can be refactored using **Switch** **Case** **statements** to allow a value to change the control flow of program execution which will improve the efficiency of the code.

Graphical user interface, text, application

Description automatically generated

* In the **TIGr**, the **Factory** method could be expanded into the **Abstract** **Factory** method and pass any concrete factory to the client class when initializing. The abstract factory could provide default implementations for the current source readers and parsers.
* **The parser** is now a Singleton but cannot be easily overridden because no hooks or template methods have been implemented. The parser could potentially be refactored to use the template method or extended using a decorator.
* Singleton may not account for different implementation classes. Primarily an issue with Source Reader since **HardCodedSourceReader** is a Singleton, but possibly with the parser if is subclassed or a structural pattern is added.
* **Singleton** does not increase **flexibility**, but likely no need for multiple instances for the parser currently since re-instantiating it, does change the specified drawer while otherwise using the same instance.