Design Pattern Decision Document

This document describes why and how I implemented each design pattern

Assignment 2 code  
Kieran Abelen

Contents

[Beginning Class Diagram 2](#_Toc23678311)

[Factory pattern 3](#_Toc23678312)

[References 5](#_Toc23678313)

Design Pattern Implementation 1

# Beginning Class Diagram



# Factory pattern

The issue with this program is that two drawers and they cannot both be initialized at the same time since they will override each other.

c = config.read().splitlines()  
**if** c[0] == **'DrawerKieran'**:  
 **from** DrawerKieran **import** Drawer  
**elif** c[0] == **'DrawerTurtleJack'**:  
 **from** DrawerTurtleJack **import** Drawer  
config.close()

main = Interface(Parser(Drawer()))  
main.go()

1. To fix this we need a drawer and an abstract product for both real drawers.
2. The Main can initialize with the fake or basic drawers and the interface when trying to use drawer methods will select between the two real drawers.

This aligns perfectly with the design pattern book

“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclass. “ (Erich Gamma, 1994) pg121

1. It will select the drawer and the creator will build it.
2. The parent creator will have most of the functionality like a template method and the child class will just return the product

Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate. (Erich Gamma, 1994) pg122

1. The bonus the creator or factory can produce the desired outcome when it is needed.
2. After the real drawer is produced the interface need to re-initialize SourceReader to make sure the changes can be drawn to. This is done by returning the SourceReader with the new product.

This could also be applied to extra readers or parsers.

## Pro

1. Multiple drawers, parsers, readers/interfaces could be used, even while the program was running.
2. It will not bloat the font end with lots of different python references/library

## Con

1. The whole source reader needs to be restarted since any front-end references won’t work until it has been re-instanced.
2. If the Source Reader has a complex set up then the creator will have to retain that.

# After Factory Method Class Diagram



Design Pattern Implementation 2

# Beginning Class Diagram (same diagram)



# Strategy

The issue is still the two drawers, and this is another strategy to manage them

c = config.read().splitlines()  
**if** c[0] == **'DrawerKieran'**:  
 **from** DrawerKieran **import** Drawer  
**elif** c[0] == **'DrawerTurtleJack'**:  
 **from** DrawerTurtleJack **import** Drawer  
config.close()

main = Interface(Parser(Drawer()))  
main.go()

1. To fix the drawer problem this time we will reference the two drawers directly in the front end.
2. This is very different since nothing will be built instead any extra drawers will be added and unitized inside the method.
3. The Context class that will set the strategy will be used to re-initialize the source reader that the front end uses.
4. This is much simpler approach to the two-subclass issue by just changing the behaviour.

Use the Strategy pattern when

* Many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours. (Erich Gamma, 1994) pg 350

## Pro

1. Very little code is being added or changed
2. Any complex reinitialization of the program is handed with the context class.

## Con

1. If there are lots of subclasses this will bloat the method used to select the strategy
2. Can change any referenced subclasses once the program is launched.

# After Factory Method Class Diagram



# References

Erich Gamma, J. V. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* United States: Addison-Wesley. Retrieved from www.uml.org.cn: http://www.uml.org.cn/c++/pdf/designpatterns.pdf