



MAPUA UNIVERSITY

SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Lab 7: Design Patterns

CPE106L (Software Design Laboratory)

Hannah Mae Antaran

Darrel Tristan Virtusio

Kathleen Joy Tupas

Group: 01

Section: E01



PreLab

Readings, Insights and Reflection

Design Patterns and Refactoring. (n.d.). Retrieved August 07, 2020, from https://sourcemaking.com/design_patterns

The Catalog of Design Patterns. (n.d.). Retrieved August 07, 2020, from <https://refactoring.guru/designpatterns/catalog?fbclid=IwAR2GRRicJJIr6lkP1GJiV1oxKuh-WJb79xXyLODJ2EtHaHAT0edaJAWnEA>

Design Patterns

In software engineering, design patterns are repeatable solutions for common problems in software design. A design pattern is a template on how to solve a problem that are used in various situations. It speeds up the development process by providing development paradigms which can be reused to prevent further problems. It is characterized by a general solution with no specifics in solving software design problems. With design patterns, developers are able to communicate using naming software interactions and improves the patterns more, making it powerfully built. Design patterns have three different types which are elaborated below:

Creational Design Pattern

Creational Design Patterns deal with creation of classes or objects. They serve to abstract away the specifics of classes so that we'd be less dependent on their exact implementation, or so that we wouldn't have to deal with complex construction whenever we need them, or so we'd ensure some special instantiation properties. They're very useful for lowering levels of dependency and controlling how the user interacts with our classes.

Structural Design Pattern

Structural Design Patterns deal with assembling objects and classes into larger structures, while keeping those structures flexible and efficient. They tend to be really useful for improving readability and maintainability of the code, ensure functionalities are properly separated, encapsulated, and that there are effective minimal interfaces between interdependent things.

Behavioral Design Pattern

Behavioral Design Patterns deal with algorithms in general, and assignment of responsibility between interacting objects. For example, they're good practices in cases where you may be tempted to implement a naive solution, like busy waiting, or load your classes with unnecessary code for one specific purpose that isn't the core of their functionality.

- **Objectives**

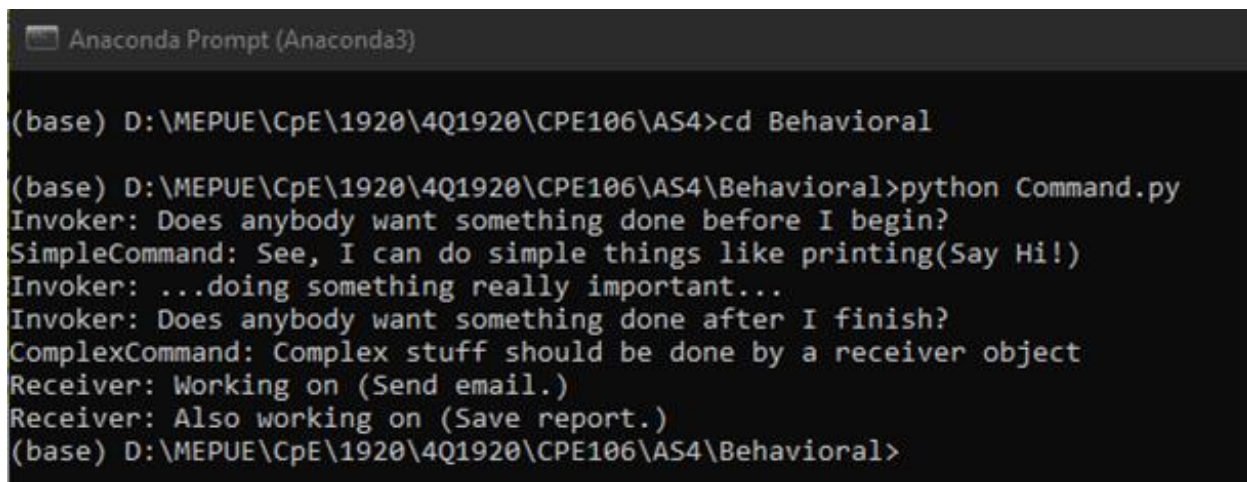
1. To understand about design patterns.
2. To differentiate the different types of design patterns.
3. To demonstrate the types of design patterns using Python and VS Code.

- **Tools Used**

1. Anaconda Prompt
2. Spyder

- **Procedure**

PART 1: Behavioral Patterns



```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4>cd Behavioral

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Command.py
Invoker: Does anybody want something done before I begin?
SimpleCommand: See, I can do simple things like printing(Say Hi!)
Invoker: ...doing something really important...
Invoker: Does anybody want something done after I finish?
ComplexCommand: Complex stuff should be done by a receiver object
Receiver: Working on (Send email.)
Receiver: Also working on (Save report.)
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>
```

Figure 1.1 Command Pattern.

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python CoR.py
Chain: Monkey > Squirrel > Dog

Client: Who wants a Nut?
  Squirrel: I'll eat the Nut
Client: Who wants a Banana?
  Monkey: I'll eat the Banana
Client: Who wants a Cup of coffee?
  Cup of coffee was left untouched.

Subchain: Squirrel > Dog

Client: Who wants a Nut?
  Squirrel: I'll eat the Nut
Client: Who wants a Banana?
  Banana was left untouched.
Client: Who wants a Cup of coffee?
  Cup of coffee was left untouched.
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>
```

Figure 1.2 Chain of Responsibility Pattern.

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Iterator.py
Straight traversal:
First
Second
Third

Reverse traversal:
Third
Second
First
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>
```

Figure 1.3 Iterator Pattern.

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Mediator.py
Client triggers operation A.
Component 1 does A.
Mediator reacts on A and triggers following operations:
Component 2 does C.

Client triggers operation D.
Component 2 does D.
Mediator reacts on D and triggers following operations:
Component 1 does B.
Component 2 does C.

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>
```

Figure 1.4 Mediator Pattern.

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Memento.py
Originator: My initial state is: Super-duper-super-puper-super.

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: BEgQvoypfGllUwChVIqbMeAxcstZzj

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: eisNpkDMJwXSPGICKLZIdEBTuxomqQ

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: sRag10tViSueCvhmDEAXzjGuBQZFyM

Caretaker: Here's the list of mementos:
2020-07-27 14:35:20 / (Super-dup...)
2020-07-27 14:35:20 / (BEgQvoypf...)
2020-07-27 14:35:20 / (eisNpkDMJ...)

Client: Now, let's rollback!

Caretaker: Restoring state to: 2020-07-27 14:35:20 / (eisNpkDMJ...)
Originator: My state has changed to: eisNpkDMJwXSPGICKLZIdEBTuxomqQ

Client: Once more!

Caretaker: Restoring state to: 2020-07-27 14:35:20 / (BEgQvoypf...)
Originator: My state has changed to: BEgQvoypfGllUwChVIqbMeAxcstZzj
```

Figure 1.5 Memento Pattern.

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.


```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Observer.py
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 4
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event

Subject: I'm doing something important.
Subject: My state has just changed to: 6
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event

Subject: I'm doing something important.
Subject: My state has just changed to: 5
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event
```

Figure 1.6 Observer Pattern.

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python State.py
Context: Transition to ConcreteStateA
ConcreteStateA handles request1.
ConcreteStateA wants to change the state of the context.
Context: Transition to ConcreteStateB
ConcreteStateB handles request2.
ConcreteStateB wants to change the state of the context.
Context: Transition to ConcreteStateA

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>
```

Figure 1.7 State Pattern.

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

```

Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Strategy.py
Client: Strategy is set to normal sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
a,b,c,d,e

Client: Strategy is set to reverse sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
e,d,c,b,a

```

Figure 1.8 Strategy Pattern.

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

```

Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>Python TemplateMethod.py
Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass1 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass1 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway

Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass2 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass2 says: Overridden Hook1
ConcreteClass2 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>

```

Figure 1.9 Template Method Pattern.

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but let's subclasses override specific steps of the algorithm without changing its structure.

```

Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>python Visitor.py
The client code works with all visitors via the base Visitor interface:
A + ConcreteVisitor1
B + ConcreteVisitor1
It allows the same client code to work with different types of visitors:
A + ConcreteVisitor2
B + ConcreteVisitor2

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Behavioral>

```

Figure 1.10 Visitor Pattern.

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

PART 2: Creational Patterns

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>python AbstractFactory.py
Client: Testing client code with the first factory type:
The result of the product B1.
The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type:
The result of the product B2.
The result of the B2 collaborating with the (The result of the product A2.)
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>
```

Figure 2.1 Abstract Factory Pattern.

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>python Builder.py
Standard basic product:
Product parts: PartA1

Standard full featured product:
Product parts: PartA1, PartB1, PartC1

Custom product:
Product parts: PartA1, PartB1
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>
```

Figure 2.2 Builder Pattern.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>python FactoryMethod.py
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct1}

App: Launched with the ConcreteCreator2.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct2}
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>
```

Figure 2.3 Factory Method Pattern.

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.


```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>python Prototype.py
Adding elements to 'shallow_copied_component's some_list_of_objects adds it to 'component's some_list_of_objects.
Changing objects in the 'component's some_list_of_objects changes that object in 'shallow_copied_component's some_list_of_objects.
Adding elements to 'deep_copied_component's some_list_of_objects doesn't add it to 'component's some_list_of_objects.
Changing objects in the 'component's some_list_of_objects doesn't change that object in 'deep_copied_component's some_list_of_objects.
id(deep_copied_component.some_circular_ref.parent): 2135541640712
id(deep_copied_component.some_circular_ref.parent.some_circular_ref.parent): 2135541640712
^^ This shows that deepcopied objects contain same reference, they are not cloned repeatedly.

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>
```

Figure 2.4 Prototype Pattern.

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>python Singleton.py
Singleton works, both variables contain the same instance.

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Creational>
```

Figure 2.5 Singleton Pattern.

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

PART 3: Structural Patterns

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Adapter.py
Client: I can work just fine with the Target objects:
Target: The default target's behavior.

Client: The Adaptee class has a weird interface. See, I don't understand it:
Adaptee: .eetpadA eht fo roivaheb laicepS

Client: But I can work with it via the Adapter:
Adapter: (TRANSLATED) Special behavior of the Adaptee.
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.1 Adapter Pattern.

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Bridge.py
Abstraction: Base operation with:
ConcreteImplementationA: Here's the result on the platform A.

ExtendedAbstraction: Extended operation with:
ConcreteImplementationB: Here's the result on the platform B.
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.2 Bridge Pattern.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Composite.py
Client: I've got a simple component:
RESULT: Leaf

Client: Now I've got a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.3 Composite Pattern.

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Decorator.py
Client: I've got a simple component:
RESULT: ConcreteComponent

Client: Now I've got a decorated component:
RESULT: ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.4 Decorator Pattern.

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Facade.py
Facade initializes subsystems:
Subsystem1: Ready!
Subsystem2: Get ready!
Facade orders subsystems to perform the action:
Subsystem1: Go!
Subsystem2: Fire!
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.5 Facade Pattern.

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Flyweight.py
FlyweightFactory: I have 5 flyweights:
Camaro2018_Chevrolet_pink
C300_Mercedes Benz_black
C500_Mercedes Benz_red
BMW_M5_red
BMW_X6_white

Client: Adding a car to database.
FlyweightFactory: Reusing existing flyweight.
Flyweight: Displaying shared (["BMW", "M5", "red"]) and unique (["CL234IR", "James Doe"]) state.

Client: Adding a car to database.
FlyweightFactory: Can't find a flyweight, creating new one.
Flyweight: Displaying shared (["BMW", "X1", "red"]) and unique (["CL234IR", "James Doe"]) state.

FlyweightFactory: I have 6 flyweights:
Camaro2018_Chevrolet_pink
C300_Mercedes Benz_black
C500_Mercedes Benz_red
BMW_M5_red
BMW_X6_white
BMW_X1_red
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.6 Flyweight Pattern.

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

```
Anaconda Prompt (Anaconda3)

(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>python Proxy.py
Client: Executing the client code with a real subject:
RealSubject: Handling request.

Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
RealSubject: Handling request.
Proxy: Logging the time of request.
(base) D:\MEPUE\CpE\1920\4Q1920\CPE106\AS4\Structural>
```

Figure 3.7 Proxy Pattern.

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

OneDrive:

Group 1: <https://bit.ly/2Pz2W78>

Github:

Darrel Virtusio: <https://bit.ly/2C6iWu9>

Hannah Antaran: <https://bit.ly/3fF6Cyy>

Kathleen Tupas: <https://bit.ly/2XE3CMS>