Final Examination

BCPR301 Advance Programming

Jignesh Patel

2018

# **Self-Marking**

|  |  |  |
| --- | --- | --- |
|  | **Topic** | **Marks** |
| 1 | Finding appropriate objects | 3 |
| 2 | Determining object granularity | 3 |
| 3 | Specifying object interfaces | 3 |
| 4 | Specifying object implementation | 2.5 |
| 5 | Class versus Interface inheritance | 3 |
| 6 | Programming to an interface, not an implementation | 3 |
| 7 | Inheritance versus Composition | 2 |
| 8 | Delegation | 2 |
| 9 | Inheritance versus parameterized types | 2.5 |
| 10 | Relating run-time and compile-time structures | 2.5 |
| 11 | Designing for change | 3 |
| 12 | A common design vocabulary | 2.5 |
| 13 | A documentation and learning aid | 3 |
| 14 | An adjunct to existing methods | 2 |
| 15 | A target for refactoring | 2.5 |
| 16 | Single Responsibility Principle | 3 |
| 17 | Open Close Principle | 3 |
| 18 | Liskov Substitution Principle | 2.5 |
| 19 | Interface Segregation Principle | 3 |
| 20 | Dependency Inversion Principle | 3 |
|  | **Total** | 54/60 |

**Evaluate and discuss how effectively your solution of Assignment 3 uses design patterns to solve design problems existing in your solution of Assignment 2.**

# **Finding appropriate objects**

## Knowledge Point:

* “Object-oriented programs are made up of objects. The hard part about object-oriented design is decomposing a system into objects. Design patterns can help in this process by identifying less obvious abstractions and the objects that capture them.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* Three different way of instantiating the Interpreter class with different parameters supplied by the users.
* The instantiation process included switch statement and Duplicate code bad smell.
* The instantiation process was a series of conditional statements which made the code complex and inflexible.
* Every time a new requirement was introduced, a new conditional statement was added with the same complex logic.

## Solution:

* By using the Factory Method design pattern, I was able to extract common functionalities and put them into methods and put those common methods into an Abstract superclass called InterpreterCreator.
* The Factory method design forced me to create an abstract method called create\_interpreter which must be implemented by any subclass that inherits from the InterpreterCreator superclass.
* The factory method helped me in identifying the three subclasses DefaultInterpreterCreator, OutputPathInterpreterCreator and DatabaseInterpreterCreator. These three classes are the child classes of the InterpreterCreator class which mean they must provide an implementation for the create\_interpreter method.
* Each subclass mentioned above had different implementation for the create\_interpreter method thus, at runtime based on the arguments provided by the user the client code can decide which subclass to instantiate.
* From this, we can see that by applying the Factory Method pattern, the process of instantiating the Interpreter class went from monolithic conditional statement approach to a class and object-based approach which is more maintainable.
* From the explanation provided above, we can see that the Factory Method pattern has helped me in identifying an appropriate object to instantiate the Interpreter class.

## Effectiveness:

* Factory Method has made the process of instantiating the Interpreter class more maintainable by the encapsulating individual algorithms of instantiating the Interpreter class into different subclasses, which means that if a new requirement comes in, I can create a new subclass and override the creater\_interpreter method.
* Now my client code that uses the Factory Method does not care about how the Interpreter class is instantiated if all the subclass follows the InterpreterCreator interface.
* We can see that the Factory Method pattern has made my program more maintainable. However, the solution is not very useful regarding “Finding appropriate objects”. The more appropriate solution to the problem would be Strategy Pattern where I could have encapsulated the algorithm into concreate Strategies, and the client code would have been encapsulated into the Context class.

# **Determining object granularity**

## Knowledge Point:

* “Objects can vary tremendously in size and number. Design patterns can help to determine proper object granularity by describing specific ways of decomposing an object into smaller objects.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* If we look at the code from lines 176 – 201, that block of code has nested conditionals and multiple code duplications which make it complicated to understand, and hard to maintain.
* That block of code is written in a very naive way, which requires decomposition and abstraction to make the code more readable and maintainable.

## Solution:

* Firstly, I used refactoring technique ‘extract method’ to extract out different method from the monolithic block of code.
* Because of the refactoring, I ended up five methods, each method responsible for performing a single task.
* I used the Factory Method design pattern to put the common method into an Abstract superclass and other methods that used these common methods in the subclasses.
* All the five individual methods were encapsulated into classes, and because of this encapsulation, I ended up with three objects whose only responsibilities was to create the Interpreter object differently based on the input provided by the user.

## Effectiveness:

* After evaluating the solution, I think factory was appropriate for the problem mentioned above because, if the Factory Method pattern were not applied, I would have ended up with one large class with all the five methods in it.
* The result of applying the Factory Method pattern was an InterpreterCreator interface for creating the Interpreter object in the superclass but allowed the subclasses DefaultInterpreterCreator, OutputPathInterpreterCreator and DatabaseInterpreterCreator to alter how the interpreter object was created.
* We can clearly see that the Factory Method pattern has helped in defining object granularity regarding; I ended with objects that are only responsible for creating Interpreter object thus increasing the maintainability of the program.

# **Specifying object interfaces**

## Knowledge Point:

* “An object's interface says nothing about its implementation—different objects are free to implement requests differently.”

## Problem:

* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29
* I wanted to add additional functionality to my program by adding Pickle storage for object serialisation.
* The major problem for adding additional functionality to my program was that my program was tightly coupled with the Shelve storage.
* Adding Pickle storage would result in having a similar structure of Shelve Storage class with different implementations for Pickle Storage.

## Solution:

* I used the Template Method Design pattern to implement the solution.
* By using this pattern, I was able to convert the concrete Storage class into an abstract class.
* The Abstract class provided an interface for storing and retrieving the serialise objects.
* Now that I have an interface in the form Storage Abstract class, any class that inherited from this class would have to implement this class interface.
* I created two classes one called PickelStorage and another called ShelveStorage, both the class are subclasses of the Storage class, so they are forced to provide an implementation for the unimplemented methods in the superclass.
* Now both the classes PickelStorage and ShelveStorage can provide different implementations for the same unimplemented methods. Thus, we can say, “Different objects with the same interface are free to implement requests differently.”

## Effectiveness:

* After evaluating the solution, I think applying the Template Method design pattern was an effective solution because I aimed to have a fixed algorithm for storing and retrieving the serialise objects while having different implementation for Shelve and Pickle technologies.
* Now that I have a fixed algorithm for storing and retrieving the serialise objects, I can send the same request to PickleStorage and ShelveStorage object, and they do their job based on the implementation provided.
* From this, we can see that the Storage class interface says nothing about its implementation and as PickleStorage and ShelveStorage classes are implementing the interface provided by the Storage class, they are free to implement requests differently.

# **Specifying object implementation**

## Knowledge Point:

* “An object's implementation is defined by its class. New classes can be defined in terms of existing classes using class inheritance.”

## Problem:

* I am going to talk about the same problem mentioned above, where I want to add additional functionalities to my existing program by introducing additional Pickle technology with the existing Shelve technology.
* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29
* One solution would be by creating a class and let it inherit from concrete Storage class and override the all the method in the superclass to provide the functionalities for Pickle object sterilisation.
* While the solution mentioned above works, but it will make the make program hard to maintain, as any changes made to the superclass would affect the subclass.

## Solution:

* I used the Template Method Design pattern to implement the solution.
* I created the superclass Storage that had the structure of the algorithm, and I created two subclasses PickleStorage and ShelveStorage and made them inherit from the Storage superclass.
* As PickleStorage and ShelveStorage are inheriting from the Storage superclass, they both can use the same algorithm structure defined in the Storage Superclass.
* Now that I separated the Pickle and Shelve technology into subclasses, if a change is, made to one of the subclasses, that change would not affect the other subclass.

## Effectiveness:

* I think using templet was the right choice for solving the problem because Template method uses inheritance to alter the algorithm by extending its parts in PickleStorage and ShelveStorage classes.
* Template Method pattern was effective for the problem mentioned above because it made it easy to extract the common steps of the algorithm and put them into the Storage superclass.
* The code that differed between PickleStorage and ShelveStorage subclasses was put inside the individual subclasses.
* By using inheritance both PickleStorage and ShelveStorage were able to use the common step of the algorithm defined in the Storage superclass while providing different implementations for unimplemented abstract methods.

# **Class versus Interface inheritance**

## Knowledge Point:

* “An object can have many types, and objects of different classes can have the same type.”

## Problem:

* The problem of adding additional Pickle Serialisation will be used again.
* In my assignment two, I have Shelve object serialisation technology to store and retrieve the serialised objects, and I wanted to add additional Pickle Object serialisation technology.
* Currently, I have a class called Storage which uses Shelve technology, and the object that is created from the Storage class is type “Storage”.
* If I create another class say PickleStorage and make this class use the Pickle technology, the object that is created from this class would have the type of “PickleStorage”.
* Now I have two objects one is type “Storage”, and another is type “PickleStorage”. This means that these objects are not interchangeable.

## Solution:

* I used the Template Method design pattern to implement my solution.
* I created an abstract base class called Storage. This base class included common functionalities required by its subclass and includes abstract operations that any class inheriting from this class must implement.
* I created two classes one PickleStorage and another ShevelStorage. Both the classes are the subclasses of the base class Storage, and both classes provide an implementation for the abstract methods defined in the base class.
* By implementing my solution in this way, The PickleStorage and ShevelStorage have two types. One is their default type either “PickleStorage” or “ShevelStorage” and another type is “Storage”.
* Now both the objects are interchangeable as they both are the type of “Storage”.

## Effectiveness:

* By looking at the problem domain, I think by applying the Template Method I have a been able to resolve the problem. The main problem in the program was that I had two different type of object that means they were not interchangeable.
* As PickleStorage and ShelveStorage are subclasses of the Storage class which means both objects are the type of “Storage” and now can be interchanged.
* From this, we can prove “An object can have many types,” i.e. PickleStorage has the type of “PickleStorage” as well the type of “Storage”.
* We can also prove “Objects of different classes can have the same type.” I.e. PickleStorage and ShelveStorage are the object of two different class, but they both can have the same type “Storage”.

# **Programming to an interface, not an implementation**

## Knowledge Point:

* “Inheritance's ability to define families of objects with identical interfaces (usually by inheriting from an abstract class) is also important as polymorphism depends on it.”

## Problem:

* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29
* The Storage module in my 2nd assignment which is responsible for object serialisation is written regarding implementation.
* The main Interpreter class is tightly coupled with Storage class which means that the Interpreter class know the implementation details of the Storage class.
* This makes the program hard to maintain as adding additional functionalities require major rework to the Interpreter class as well as the Storage class.

## Solution:

* To solve this problem, I used the Template Method design pattern.
* I created an abstract base class called Storage. This base class provide an interface for storing and retrieving serialised objects.
* The next step was to create two new classes PickleStorage and ShelveStorage and make them inherit from the Storage base class.
* With both the subclasses inheriting from the base class, both the subclasses have an identical interface, which means the same request that is sent to the PickleStorage and can be sent to ShelveStorage.
* By doing this, the Interpreter class become loosely coupled with the Storage module. Now the Interpreter class knows about the Storage interface rather than the implementation of it.

## Effectiveness:

* I think the approach of using the Template Method design pattern was effective because by applying the pattern the Interpreter remain unaware of the specific types of objects (Either PickleStorage or ShelveStorage) is used, as long as both PickleStorage and ShelveStorage adhere to the interface of the Storage class.
* Interpreter class remain unaware of the PickleStorage and ShelveStorage classes that implement the storage class interface.
* Now the Interpreter class only know about the Storage abstract class defining the interface thus we can say that the Interpreter class is programmed to an interface, not an implementation.

# **Inheritance versus Composition**

## Knowledge Point:

* “Inheritance causes Implementation dependencies, one cure for this is to inherit only from abstract classes since they usually provide little or no implementation.”

## Problem:

* Adding additional Pickle technology to the existing Shelve technology.
* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29
* One solution would be “white-box” reuse in which I would create a class and let it inherit from concrete Storage class and override the all the method in the superclass to provide the functionalities for Pickle object sterilisation.
* As I have used “white-box” as my reuse mechanism, the internals of Storage base class are now visible to subclasses which will make the make program hard to maintain, as any changes made to the Storage base class would affect the subclass.

## Solution:

* I created an abstract base class Storage and declared the template methods get\_data and set\_data outline the algorithm structure inside template methods using abstract methods
* I created two subclasses PickleStorage and ShelveStorage and made them inherit from the Storage base class.
* As both the subclass inherit from the base class, they must provide the implementation for the abstract methods defined in the base class.
* Now the application use either PickleStorage or ShelveStorage for object serialisation.

## Effectiveness:

* I think using templet was the right choice for solving the problem because Template method uses inheritance to alter the algorithm by extending its parts in PickleStorage and ShelveStorage classes.
* We can see that by inheriting only from an abstract class, I tried to reduce the Implementation dependency between the base class Storage and the subclasses PickleStorage and shelvedStorage.
* However, the PickleStorage and ShelveStorage subclasses were still depended on the skeleton of an existing algorithm defined in the base class storage.

# **Delegation**

## Knowledge Point:

* “Delegation shows you can always replace inheritance with object composition as a mechanism for code reuse.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* This block of code is tightly coupled to the Interpreter class.
* Adding a new Interpreter class would require making changes to the entire codebase.
* Moreover, if I decide to add another type of Interpreter to the app, I will probably need to make all those changes again.
* The code is riddled with conditionals, which select behaviours depending on user-supplied parameters.

## Solution:

* I used the Factory Method Design pattern to Implement the solution.
* I created an Abstract base class called InterpreterCreator which included common functionalities required by the subclasses and an abstract method create\_interpreter.
* I created three classes DefaultInterpreterCreator, OutputPathInterpreterCreator and DatabaseInterpreter- Creator.
* These three classes are the child classes of the InterpreterCreator class which mean they had to implement the create\_interpreter method.
* Now at runtime based on the arguments provided by the user the client code can decide which subclass to instantiate.
* Once the client code identifies a subclass, it calls the create\_interpreter method declared in the interface of the InterpreterCreator.
* If all parameters supplied by the user are valid, the subclass returns an instance of the interpreter class.

## Effectiveness:

* From the solution provided in section 8.3, we can see that we have been able to solve the problem defined in section 8.2, however, to solve this problem we used the Factory Method Design pattern.
* Factory Method design pattern is a class-based creational pattern, which means it uses Inheritance as a mechanism for code reuse.
* The Knowledge point mentioned in 8.1 says that we can replace inheritance with object composition and that is true in this case.
* Instead of using the Factory Method pattern I could have used the Strategy pattern which is an object-based pattern and uses composition as a mechanism of reuse.
* By using the Strategy pattern, I could have encapsulated the different algorithm of instancing the interpreter class into concreate Strategies, and the client code would have been encapsulated into the Context class.

# **Inheritance versus parameterised types**

## Knowledge Point:

* “Object composition lets you change the behaviour being composed at run-time, but neither inheritance nor parameterised types can change at run-time.”

## Problem:

* In my 2nd assignment, the Storage class used for the serialising object was a static class, and the Interpreter class is using the static methods defined the Storage class.
* The storage class is using Shelve technology for object serialisation.
* This means that at run-time my Interpreter class is tightly coupled with Storage class and it cannot be changed during run-time.
* I wanted to add another class which used Pickle object serialisation technology, and the Interpreter class can choose which technology to use during runtime.
* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29

## Solution:

* To make my system more flexible during runtime, I used the Template Method pattern to implement the solution.
* By using the Template Method pattern, I was able to create one abstract base class Storage and two subclasses PickleStorage and ShelveStorage.
* Both PickleStorage and ShelveStorage yield a concrete object.
* Now the Interpreter can choose between the object of PickleStorage or ShelveStorage during run-time as both these objects have a similar interface.

## Effectiveness:

* I think using object composition is as powerful as using inheritance for reuse mechanism.
* From the solution mentioned above, we can see at the start the Interpreter class was tightly coupled with the Static Storage class and could not change during runtime.
* The result of applying the Template Method design pattern was, the Interpreter class could yield two different objects during run-time.
* From this we can how by using Object Composition over static class we can change the behaviour being composed at run-time.

# **Relating run-time and compile-time structures**

## Knowledge Point:

* “The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects.”

## Problem:

* If I take the same problem mentioned in section 9.2 about the static Storage class and the Interpreter class.
* As we know, the Interpreter class uses the static Storage class, and as we know at the compile time, we cannot change the definition of a class. Hence the Interpreter class is tightly coupled with the static methods defined in the Storage class.
* This means that during runtime the Interpreter class cannot be changed to other class for the same functionality which makes the system inflexible.

## Solution:

* By using the Template Method design pattern, I was able to encapsulate the properties and method of the Storage class into concrete class ShelveStorage.
* I also added another concrete class PickleStoage which has the same interface as ShelveStorage but provides different implementations.
* At the runtime, the interpreter class was able to choose between the object of PickleStorage or ShelveStorage class.
* This means that during runtime the Interpreter class no longer tightly bound with the static Storage class and can make use of either functionality defined in the PickleStorage or ShelveStorage class.

## Effectiveness:

* From the solution mentioned above, we can see that even though at compile time the code structure of Interpreter, PickleStorage and ShelveStorage class is frozen, during runtime the Interpreter class was able to choose which object to use.
* This means, that with such disparity between a program's run-time and compile-time structures it's clear that code won't reveal everything about how a system will work.
* And that is why by applying the Template Method design pattern the relationships between PickleStorage and ShelveStorage objects and their types are designed with great care because they determine how good or bad the run-time structure is.

# **Designing for change**

## Knowledge Point:

* “Design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* This block of code is tightly coupled to the Interpreter class.
* Adding a new Interpreter class would require making changes to the entire codebase.
* Moreover, if I decide to add another type of Interpreter to the app, I will probably need to make all those changes again.
* This block of code is riddled with conditionals, which select behaviours depending on user-supplied parameters.

## Solution:

* The solution is to decouple the client code that is used for instantiating the Interpreter class.
* I used the Factory Method design pattern to solve this problem.
* I created an Abstract base class InterpreterCreator which provides an interface for creating the Interpreter object.
* I created subclasses that inherit from this abstract base class and provide an implementation for the unimplemented method.
* Each subclass has different implementation for the abstract method thus, at runtime based on the arguments provided by the user the client code can decide which subclass to instantiate.
* Now the client code only knows about the interface provided by the abstract base class which decouple the client code and the Interpreter class.

## Effectiveness:

* I think the Factory Method has to help me in implementing the solution effectively because we know that the Factory Method is a creational pattern and Creational patterns ensure that our system is written regarding interfaces, not implementations.
* We can see that the client code responsible for instantiating the Interpreter class must commit to an interface defined by an abstract class InterpreterCreator, and not to an instance of a concrete class.
* This means that the client code remains unaware about the subclasses that inherit from the abstract class.
* And as the client does not know about any subclasses, which means that if a new requirement comes in I have to create a new subclass and override the creater\_interpreter method and my client will still work.

# **A common design vocabulary**

## Knowledge Point:

* “Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* If some looks at this block of code, they would have no idea what it does,
* And they would not be able to communicate with someone about this block of code.
* They would have to go line by line and understand what each line does to understand the block of code.
* This is a big problem for someone maintaining the code as adding new functionality or fixing a bug would require them to understand the chunk of the monolithic code before they can make any changes.

## Solution:

* I took the monolithic block of code and refactored it.
* By refactoring the code, I identified that there three different way on instantiating the interpreter class.
* Then I went through the “Intent” of all creational design pattern.
* This is where I found that the “Intent” of the Factory Method pattern matches with the existing problem.
* Then I understood how to implement the pattern and implemented a solution based on the Factory Method design pattern.

## Effectiveness:

* I think applying the design pattern was an appropriate solution because, when Dr Luofeng conducted the code review of my implemented solution, we were able to communicate in common vocabulary that we both understood.
* I was able to explain my solution to Dr Luofeng regarding Factory Method pattern as he understood the Factory Method pattern, he was able to suggest many alternative approaches I could have taken to solve the same problem.
* One the approach Dr Luofeng suggested was, I could have used the Strategy and encapsulate the different algorithm into individual Strategies.
* From this, we can see Dr Luofeng, and I was able to communicate and explore design alternatives by using the common vocabulary provided by the design patterns.

# **A documentation and learning aid**

## Knowledge Point:

* “Learning the design patterns will help you understand existing object-oriented systems.”

## Problem:

* I am going to use the same problem mentioned in section 12.2, but the main problem is about documenting the solution.
* If I do not use design pattern to fix the problem mentioned in the section 12.2, I would have to create lot more documentation for some maintaining the code to make them understand what the code is doing.

## Solution:

* I order to make my code self-documenting, I used design pattern name for my class name and method name.
* To fix the problem mentioned in section 12.2, I used the Factory Method pattern.
* So, I named my abstract base class InterpreterCreator.
* And the subclasses DefaultInterpreterCreator, OutputPathInterpreterCreator and DatabaseInterpreterCreator.
* By implementing my solution in this way someone who knows the Factory Method design pattern can understand the system quickly without reading any documentation.

## Effectiveness:

* I think applying design patterns to my solutions was very effective as anyone who understands the Factory Method can understand my code by reading the class names.
* By applying these design pattern, the person maintaining the program will understand the flow of control.
* By having a common vocabulary means I don't have to describe the whole design pattern; I can name it and expect the person maintaining to know it.
* If someone who doesn't know the patterns, will have to look them up at first, but that's still easier than reverse-engineering.

# **An adjunct to existing methods**

## Knowledge Point:

* “The design patterns show how to use primitive techniques such as objects, inheritance, and polymorphism.”

## Problem:

* One of the major problems in my assignment two was there were very few objects and many static classes. This means that my system was a class-based system instead of being an object-based system.
* Because it is a class-based system, all the submodules are tightly coupled together, which any change would require significant reworking.
* In my second assignment I want to add additional functionality for object serialisation, but as my Interpreter class was tightly coupled with the static Storage class, it was not possible to change the functionality at runtime.

## Solution:

* To fix this problem, I used the Template Method design pattern.
* By applying the Template Method pattern, I created abstract base class Storage and two more classes PickleStorage and ShelveStorage and made them inherit from the Storage base class.
* As PickleStorage and ShelveStorage class both inherit from the Storage base class, they have the same interface which means they can be interchangeable during runtime.
* By applying the Template method design pattern, the Interpreter class was decoupled from the concrete implementation of the Storage class.

## Effectiveness:

* From the solution, we can see the applying the design pattern was very effective. I converted my class-based system into an object-based system.
* Initially, there were no objects for the Storage class; now I have two different objects that provide different implementations for the same functionality.
* I correctly used inheritance to put the structure of the algorithm in the base class so that both PickleStorage and ShelveStorage class can use the structure defined in the Storage base class.
* I used polymorphism in the Interpreter class, where I could use object of PickleStorage or object of ShevleStorage for object serialisation as both PickleStorage and ShelveStorage class have the same interface.

# **A target for refactoring**

## Knowledge Point:

* “Even if you do not see how to apply a pattern until after you have built your system, the pattern can still show you how to change it. Design patterns thus provide targets for your refactoring’s.”

## Problem:

* The 2nd assignment is a result of rapid prototyping and incremental changes.
* Which means the program was brought to life without following any pattern or structure.
* As a result of this if we look at the code at:
  + Folder Name: PythonInterpreter.
  + File Name: main.py
  + Lines: 176 – 201
* This bock of code is dense, has many bad smells like, switch statement and code Duplication.
* This block of code was written to bring the application to life, which requires decomposition and abstraction to make the code readable and maintainable.

## Solution:

* Firstly, I used refactoring technique ‘extract method’ to extract out different method from the monolithic block of code.
* Because of the refactoring, I ended up five methods, each method responsible for performing a single task.
* I used the Factory Method design pattern to put the common method into an Abstract superclass and other methods that used these common methods in the subclasses.
* The individual methods are encapsulated into different classes; each method has a meaningful name which tells the reader about what it does with looking at its implementation.
* After implementing the Factory Method, the switch statement and duplicate code bad smell were removed.

## Effectiveness:

* Learning design pattern was very effective because, by knowing design pattern, I can now look at a block of code and identify if it structured incorrect format or does it require refactoring.
* Before learning the design pattern, all the code that I wrote worked but it was unstructured, and I can this now because design pattern has provided me with the knowledge of how to structure code.
* As we can see in my 2nd assignment, there was need of refactoring, but I could not understand how to it, however by learning the design patterns I learned how to reorganise a design which can reduce the amount of refactoring I need to do later.

# **Single Responsibility Principle**

## Knowledge Point:

* "Every software module should have only one reason to change."

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* If we look at this block of code, it has many responsibilities like:
  + How many parameters the user has provided
  + Are the parameters provided by the user valid?
  + If the parameters are valid, instantiate the Interpreter class
* From this we can see that this block of code has multiple reasons to change:
  + If we increase the number of parameters
  + If we change the validation process
  + If we have a different class to instantiate.

## Solution:

* Firstly, I used refactoring technique ‘extract method’ to extract out different method from the monolithic block of code.
* As a result of the refactoring, I ended up five methods, each method responsible for performing a single task.
* I used the Factory Method design pattern to put the common method into an Abstract base class and other methods that used these common methods in the subclasses.
* All the five individual methods were encapsulated into classes, and because of this encapsulation, I ended up with three objects whose only responsibilities was to create the Interpreter object differently based on the input provided by the user.

## Effectiveness:

* By applying the Factory Method design pattern, I have been able to achieve the Single Responsibility for my class and the client code that uses these classes.
* First, the client code is only responsible for selecting an Interpreter Factory based on the parameters provided by the user.
* Second, Each Individual Factory has only one responsibility, to create the Interpreter object based on the parameters provided by the user.

# **Open Close Principle**

## Knowledge Point:

* “Software modules should be closed for modifications but open for extensions.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* Every time a new requirement is introduced, I must add another if – else statement to accomplish the requirement.
* Violation of the Open Close Principle, as this block of code, is modified and not extended.

## Solution:

* Define an abstract class InterpreterCreator and define an abstract method create\_interpreter.
* Define three class DefaultInterpreterCreator, OutputPathInterpreterCreator and DatabaseInterpreterCreator that inherit from the InterpreterCreator class and override the create\_interperter method.
* Client code creates an instance of InterpreterCreator factory based on the parameters provided by the user.
* The client code calls the create\_interpreter method on the InterpreterCreator factory and passes the parameters provided by the user.
* If all the parameters are valid, the InterpreterCreator factory returns an instance of the Interpreter class.
* Client code uses the instance of the Interpreter class.

## Effectiveness:

* The factory method pattern follows the Open-Closed principle. The InterpreterCreator is closed for modification, but the class is extended by creating three concrete factories.
* If I want to add a new way of instantiating the Interpreter class, then I must subclass the InterpreterCreator class to instantiate the Interpreter class.
* If the factory method create\_interpreter needs to support multiple Interpreter types, then the factory method must accept a parameter to decide which concrete Interpreter object to create.

# **Liskov Substitution Principle**

## Knowledge Point:

• “Subclasses should be substitutable for base classes.”

## Problem:

* In my assignment two, I have Shelve object serialisation technology to store and retrieve the serialised objects, and I wanted to add additional Pickle Object serialisation technology.
* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29
* Currently, I have a class called Storage which uses Shelve technology, and the object that is created from the Storage class is type “Storage”.
* If I create another class say PickleStorage and make this class use the Pickle technology, the object that is created from this class would have the type of “PickleStorage”.
* Now I have two objects one is type “Storage”, and another is type “PickleStorage”. This means that these objects are not interchangeable thus, violation of the Liskov Substitution Principle.

## Solution:

* Define an abstract base class called Storage. This base class includes common functionalities required by its subclass and the abstract operations.
* Define two classes PickleStorage and ShevelStorage. Both the classes are the subclasses of the base class Storage, and both classes provide different implementations for the abstract methods defined in the base class.
* By making PickleStorage and ShevelStorage subclasses of the Storage class both PickleStorage and ShevelStorage have the type of Storage, thus making them Interchangeable at runtime.

## Effectiveness:

* The solution mentioned above was based on the Template Method design pattern.
* If we look at the knowledge point “Subclasses should be substitutable for base classes.” The Template Method pattern has been able to provide that functionality as PickleStorage and ShelveStorage are subclasses of the Storage class which means both objects are the type of “Storage” and now can be interchanged.
* However, the Template Method may violate the Liskov Substitution Principle while suppressing a default step implementation via the subclasses PickleStorage and ShelveStorage.

# **Interface Segregation Principle**

## Knowledge Point:

* “Clients should not be forced to implement interfaces they do not use.”

## Problem:

* I wanted to add additional functionality to my program by adding Pickle storage for object serialisation.
* Location of code:
  + Folder: PythonInterpreter/object\_store
  + File Name: Storage.py
  + Lines: 5 – 29
* One solution would be by creating a class and let it inherit from concrete Storage class and override the all the method in the superclass to provide the functionalities for Pickle object sterilisation.
* While the solution mentioned above works, as both the class have the same interface, but it will make the program hard to maintain, as any changes made to the super class would affect the subclass.

## Solution:

* I used the Template Method Design pattern to implement the solution.
* Define an abstract base class called Storage. This base class defines a common interface for its subclasses.
* The Interface of the base class include two public method get\_data and get\_data, one protected method \_close and three protected abstract method \_open, \_store, and \_get\_data
* Define two class PickleStorage and ShelveStorage, both classes inherit from Storage base class and provide a different implementation for the abstract method defined in the Storage base class.
* The Picklestorage class has additional methods \_append\_to\_existing, \_store\_new\_data and \_storage\_is\_empty to achieve the functionality required for storing and retrieving using Pickle technology.

## Effectiveness:

* From the above-implemented solution, we can see that by applying the Template Method pattern I was able to achieve the Interface Segregation Principle as both PickleStorage, and ShelveStorage subclass implemented the common interface from the Storage class which included all the method used by both the subclasses.
* If I had put the extra three defined in the PickleStorage class in the base class Storage, then ShelveStorage would also have access to the three-extra method.
* The point mentioned above violates the Interface Segregation Principle as ShelveStorage class will be forced to implement interfaces they do not use.

# **Dependency Inversion**

## Knowledge Point:

* “High-level modules should not depend upon low-level modules. Rather, both should depend upon abstractions.”

## Problem:

* A monolithic approach of instantiating the Interpreter class in the main method.
  + Location of code:
    - Folder Name: PythonInterpreter.
    - File Name: main.py
    - Lines: 176 – 201
* The client code directly instantiates the Interpreter class which create object dependencies as the client code is depending on concrete Interpreter class.
* By client code depending directly on the Interpreter, class violates the Dependency Inversion Principle as High-level modules (Client Code) is dependent on the low-level module (Interpreter) class.

## Solution:

* Define an abstract class InterpreterCreator and define an abstract method create\_interpreter.
* Define three class DefaultInterpreterCreator, OutputPathInterpreterCreator and DatabaseInterpreterCreator that inherit from the InterpreterCreator class and override the create\_interperter method.
* Client code creates an instance of InterpreterCreator factory based on the parameters provided by the user.
* The client code calls the create\_interpreter method on the InterpreterCreator factory and passes the parameters provided by the user.
* If all the parameters are valid, the InterpreterCreator factory returns an instance of the Interpreter class.
* Client code uses the instance of the Interpreter class.

## Effectiveness:

* By applying the Factory Method pattern, I have decoupled the client code from directly instantiating the Interpreter class. This is means that High-level modules (Client Code) no longer depends on the low-level module (Interpreter) class rather it depends on the abstraction. (Interface of the InterpreterCreator factory).
* By placing all my creation code into individual factories, I avoid duplication in my code and provide one place to perform maintenance. That also means clients depend only upon interfaces rather than the concrete classes required to instantiate objects.
* My client code is decoupled from the concert Interpreter class, but the factories are still depending on the concrete Interpreter class. Therefore a better solution would be to create an interface for the Interpreter class and bind the factories to that interface.