1. **Write a Python program which applies a factory design pattern to generate objects.**

Design patterns in software, like in buildings and towns, provide solutions to recurring problems. They are expressed in terms of objects and interfaces and are essential for leveraging the expertise of other skilled architects. A pattern has four elements: a name, a problem description, a solution description, and consequences. Understanding the effects helps evaluate the cost and benefit of using a pattern (Gamma et al. 1995).

Design patterns speed up development by providing proven paradigms. They prevent subtle issues and improve code readability. Patterns offer general solutions without specifics tied to a particular problem. They also use well-known names for software interactions and can be improved over time (Source Making, 2019).

Therefore, Kampffmeyer & Zschaler (N.D.) said that design patterns provide reusable solutions to recurring problems in software design. They originated in architecture and were adapted to object-oriented software design in the book Design Patterns: Elements of Reusable Object-Oriented Software. A typical pattern description includes pattern name and classification, intent, motivation, applicability, structure, participants, collaboration, consequences, implementation, sample code, known uses, and related patterns. Formalisations of design patterns usually focus on the solution's structure, but more is needed to define a design pattern fully.

A creational design pattern called the Factory allows subclasses to change the kind of objects that will be produced while still providing an interface for creating things (Refactoring Guru, 2014).

The basic principle of the Factory Pattern is to avoid directly constructing objects using constructors and instead to isolate object creation logic in a separate method or class.

This encourages loose coupling between the client code and the instantiated classes, making the codebase easier to extend and maintain (Wikipedia, 2023).

When it is necessary to build objects of various types dynamically, the factory design pattern is the right one to use. This is so that the factory method can choose the concrete class that will be used to generate the object at runtime. This increases the code's adaptability and flexibility (Blackler, 2023).

Using the factory design pattern, the following Python programme creates objects based on the concepts presented in (python-ood, N.D.):

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Square(Shape):
    def draw(self):
        print("Drawing a square")

class ShapeFactory:
    @staticmethod
    def create_shape(shape_type):
        if shape_type == "circle":
            return Circle()
        elif shape_type == "square":
            return Square()
        else:
            raise ValueError("Invalid shape type")

if __name__ == "__main__":
    factory = ShapeFactory()
    shape = factory.create_shape("circle")
    shape.draw()
```

This code includes an abstract class called Shape and two subclasses, Circle and Square. The ShapeFactory class has a static method that creates a Shape object

based on the specified shape type. The main function calls the ShapeFactory to create a Circle object, which is then drawn.

The factory design pattern is used in this code to create objects without specifying their concrete class. Instead, the creation is delegated to a factory method that returns an object of the appropriate concrete class based on specific criteria. In this case, the ShapeFactory class implements the factory design pattern with the create_shape() method as the factory method. This method takes a shape type as input and returns an object of the corresponding concrete class, such as a Circle object for the "circle" shape type.

2. **Discuss how multiple design patterns can be applied to support single code development.**

Design patterns are methods for resolving common issues that transcend programming languages. A design pattern thus represents an idea rather than a specific execution. Design patterns can make the code more adaptable, reusable, and maintained (GeeksforGeeks, 2015).

According to Rahman (2019), design patterns are design-level fixes for reoccurring issues that software engineers frequently run against. Using these patterns is considered best practice because the final code is more readable due to the solution's well-proven design. There are three design patterns: creational, structural, and behavioural, and they are frequently designed for and used by OOP Languages like Java.

Source Making (2019) also mentioned that patterns enable engineers to discuss software interactions using well-known, widely-accepted terms. Over time, common

design patterns can be strengthened, becoming more durable than ad hoc designs. Three different design patterns, for instance, can be used to help the development of a single code:

- Design patterns that focus on class instantiation are called creational design patterns. This pattern can be further broken down into patterns for creating classes and objects. Object-creation patterns use delegation efficiently to complete the task, whereas class-creation patterns successfully use inheritance in the instantiation process.
- Structural design patterns: The main focus of these design patterns is the composition of classes and objects. Structural class-creation patterns use inheritance to create interfaces. Structural object patterns specify how to do it when composing objects to provide new functionality.
- Behavioural design patterns: These are all about communication between classes of things. The behavioural patterns that are most focused explicitly on inter-object communication.

3. **Describe the design patterns which are more likely to be applied with one another in a single code package.**

Design patterns are often combined. The Strategy pattern selects behaviour at runtime and calls an implementation class. Using a Factory Method class is better than embedding code in the calling class, especially for multithreaded applications. It's recommended to implement the Strategy pattern with a Factory Method pattern (Oak, N.D.).

Here are some common design pattern pairings that can be found in a codebase:

1. A creational design pattern called Abstract Factory enables the creation of families of linked things without identifying their concrete classes. Many designs begin with the Factory Method, which is more straightforward and more customisable via subclasses, then progress to the Abstract Factory, Prototype, or Builder, which are more flexible but more complex (refactoring.guru, N.D.).
2. A creational design pattern called Singleton guarantees that a class has just one instance while giving it a global access point. Because a single facade object is frequently adequate, a Facade class can often be converted into a

Singleton. Builders, prototypes, and abstract factories can all be implemented as singletons (refactoring.guru, N.D.).

3. Using a structural design pattern called Composite, things are composed into tree structures and then used as independent objects. Because you can programme its construction processes to operate recursively, Builder is useful when creating complex Composite trees. Since both Composite and Decorator use recursive composition to manage an infinite number of things, their structure diagrams are comparable. Similar to a Composite, a Decorator has just one child component. Another critical distinction is that whereas Composite "sums up" the results of its offspring, Decorator adds additional obligations to the wrapped object. The patterns can, however, also work together. For instance, Decorator can be used to extend the behaviour of a particular object in the Composite tree (Refactoring.guru, 2014).
4. A Decorator is a structural design pattern that enables us to add new behaviours to objects by enclosing them in wrapper objects that already have the new behaviours attached. Adapter modifies an object's interface, whereas Decorator improves an object without altering its interface. The recursive composition is also supported by Decorator but not by Adapter. Since both Composite and Decorator use recursive composition to manage an infinite number of things, their structure diagrams are comparable. Similar to a Composite, a Decorator has just one child component. Another critical distinction is that although Composite "sums up" the outcomes of its children, Decorator adds additional responsibilities to the wrapped object (refactoring.guru, N.D.).

References:

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. [online] *Google Books*. Pearson Deutschland GmbH. Available at: https://www.google.co.uk/books/edition/Design_Patterns/tmNNfSkfTlcC?hl=en&gbpv=1&dq=Design+Patterns+Elements+of+Reusable+Object-Oriented+Software&pg=PR11&printsec=frontcover [Accessed 20 Jul. 2023].

Source Making (2019). *Design Patterns and Refactoring*. [online] Sourcemaking.com. Available at: https://sourcemaking.com/design_patterns.

Kampffmeyer, H. & Zschaler, S. (n.d.). *Finding the Pattern You Need: The Design Pattern Intent Ontology*. [online] Available at: http://blog.holger-kampffmeyer.de/files/models07_dpio.pdf [Accessed 20 Jul. 2023].

Refactoring Guru (2014). *Factory Method*. [online] Refactoring.guru. Available at: https://refactoring.guru/design-patterns/factory-method.

Wikipedia. (2023). *Factory method pattern*. [online] Available at: https://en.wikipedia.org/wiki/Factory_method_pattern#:~:text=In%20class-based%20programming%2C%20the%20factory%20method%20pattern%20is [Accessed 20 Jul. 2023].

Blackler, S. (2023). *Using The Factory Design Pattern in .NET*. [online] CodeWithStu's Blog. Available at: https://im5tu.io/article/2023/05/using-the-factory-design-pattern-in-.net/ [Accessed 20 Jul. 2023].

python-ood. (n.d.). *python-ood*. [online] Available at: https://vyahello.com/python-ood/ [Accessed 20 Jul. 2023].

GeeksforGeeks. (2015). *Introduction to Pattern Designing*. [online] Available at: https://www.geeksforgeeks.org/introduction-to-pattern-designing/.

Rahman, S. (2019). *The 3 Types of Design Patterns All Developers Should Know (with code examples of each)*. [online] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/.

Oak, M. (n.d.). *Design Pattern Combination – Strategy with Factory Method*. [online] blog.e-zest.com. Available at: https://blog.e-zest.com/design-pattern-combination-strategy-with-factory-method/#:~:text=When%20we%20implement%20certain%20design [Accessed 20 Jul. 2023].

refactoring.guru. (N.D.). *Abstract Factory*. [online] Available at: https://refactoring.guru/design-patterns/abstract-factory.

refactoring.guru. (N.D.). *Singleton*. [online] Available at: https://refactoring.guru/design-patterns/singleton#:~:text=Relations%20with%20Other%20Patterns [Accessed 20 Jul. 2023].

Refactoring.guru. (2014). *Composite*. [online] Available at: https://refactoring.guru/design-patterns/composite.

refactoring.guru. (N.D.). *Decorator*. [online] Available at: https://refactoring.guru/design-patterns/decorator.