Coffee Shop Project

Programming Methodologies

Alberto Charabati

7031859

2023

About the Project

The Coffee Shop Management System is designed to facilitate the management of beverages and meals in a coffee shop environment. Its purpose is to provide a structured and organised approach to handling different types of beverages, such as cappuccinos, americanos, and matcha, as well as meals like bagels and doughnuts. The system enables the creation, customisation, and preparation of these items, while maintaining consistency and efficiency in the coffee shop operations.

The main idea behind the project is to leverage object-oriented programming principles and design patterns to build a flexible and extensible system. The system allows the coffee shop to offer a variety of beverages and meals while providing the ability to add toppings to the base items. This enhances the customer experience by allowing customisation and personalisation of their orders.

The project implements the Decorator pattern, where beverages serve as the base component, and decorators (toppings) can be added dynamically to enhance their flavour. This approach enables the coffee shop to offer a wide range of beverage combinations without the need to create multiple subclasses for each variation.

Additionally, the system utilises the Template pattern to define a common structure for meal preparation. The **Meal** class provides a template method (**Assemble()**), creating a skeleton specifying the sequence of steps required to assemble and serve a meal. This ensures consistency in the preparation process while allowing specific meal types, such as bagels or doughnuts, to implement their unique variations of the steps, defining the overall structure and steps for meal preparation which can then be customised by the different meal types.

Furthermore, the project incorporates the Abstract Factory pattern to encapsulate the creation of meal objects. The **PlainBagelFactory**, **LaxBagelFactory**, **ChocolateDoughnutFactory**, and **GlazedDoughnutFactory** classes provide dedicated factories responsible for creating specific types of meals, implementing the **BagelFactory** and **DoughnutFactory** interfaces respectively. This separation of responsibilities simplifies the creation process and allows for easy extension by introducing new meal factories.

The CoffeeShop project is built using various classes that follow SOLID principles and object-oriented programming concepts. The project utilises interfaces, abstract classes, inheritance, and polymorphism to achieve modularity, extensibility, and code reuse. The **IBeverage** and **BeverageDecorator** interfaces along with the **BeverageToppingsManager** class implement the Decorator pattern, allowing dynamic addition of toppings to beverages.

The project also follows SOLID principles, such as the Single Responsibility Principle by separating responsibilities into different classes, the Open-Closed Principle by allowing easy extension through subclassing, and the Dependency Inversion Principle by depending on abstractions rather than concrete implementations.

By adhering to these design patterns and SOLID principles, the Coffee Shop Management System achieves maintainability, flexibility, and scalability, making it a robust solution for managing beverages and meals in a coffee shop environment.

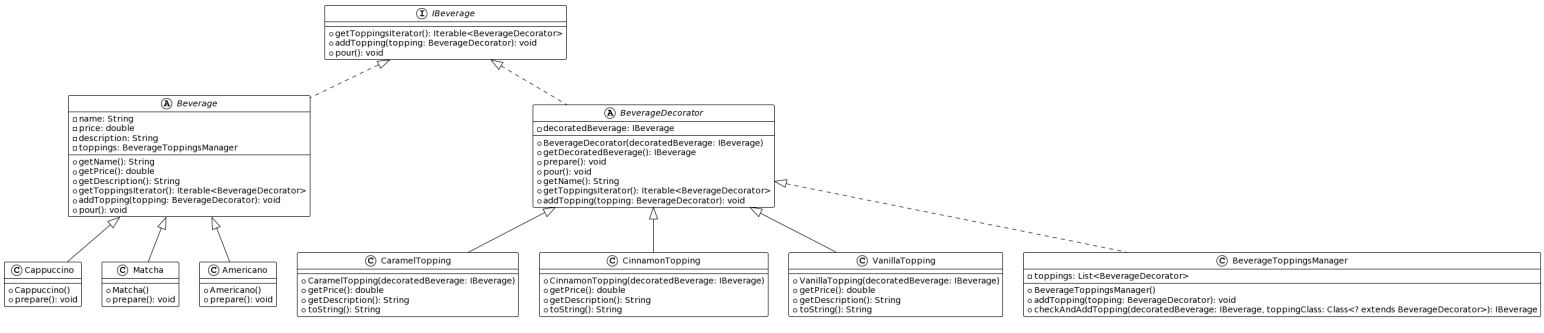
Design Patterns

The project made implements the Decorator, Template, and Abstract Factory design patterns explicitly, as well as the application of the Iterator Java library (pattern **not** implemented, Java Iterator used).

DECORATOR DESIGN PATTERN:

The Decorator pattern is a structural pattern that allows adding new behaviours or responsibilities to an object dynamically, without altering its structure. It involves creating a decorator class that wraps the original object and provides additional functionality by maintaining a reference to the decorated object. This pattern provides an alternative to subclassing for extending the functionality of an object.

- The BeverageDecorator class acts as the base decorator, implementing the IBeverage interface.
- Concrete decorators (CaramelTopping, CinnamonTopping, VanillaTopping) extend the BeverageDecorator class.
- Each decorator wraps an instance of the IBeverage interface and provides additional behaviour by modifying or extending its methods.
- The decorators can be stacked and chained, allowing for the dynamic addition of toppings to a beverage.

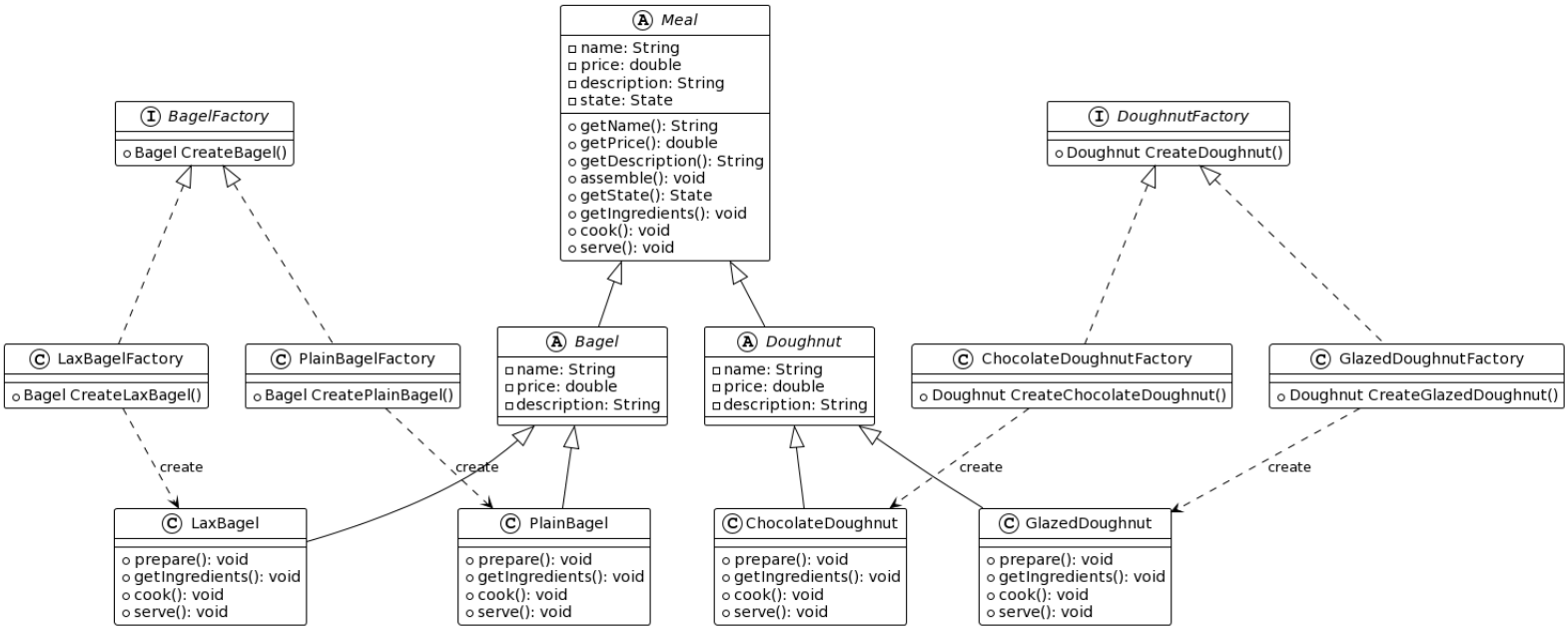


UML of classes involving the Decorator design pattern.

ABSTRACT FACTORY DESIGN PATTERN:

The Abstract Factory pattern is a creational pattern that provides an interface for creating objects without specifying their concrete classes. It encapsulates the object creation logic in a separate class, known as the factory, which is responsible for creating instances of objects based on certain conditions or parameters. This pattern promotes loose coupling by allowing the client code to work with the factory interface instead of concrete implementations.

- The LaxBagelFactory, PlainBagelFactory, GlazedDoughnutFactory, and ChocolateDoughnutFactory classes serve as concrete factories for creating Bagel and Doughnut instances, respectively.
- Each factory provides a method (e.g., createBagel(), createDoughnut()) that returns a concrete instance of the corresponding product (Bagel or Doughnut).
- The factories encapsulate the creation logic and allow the client code to create products without knowing the specific implementation details.

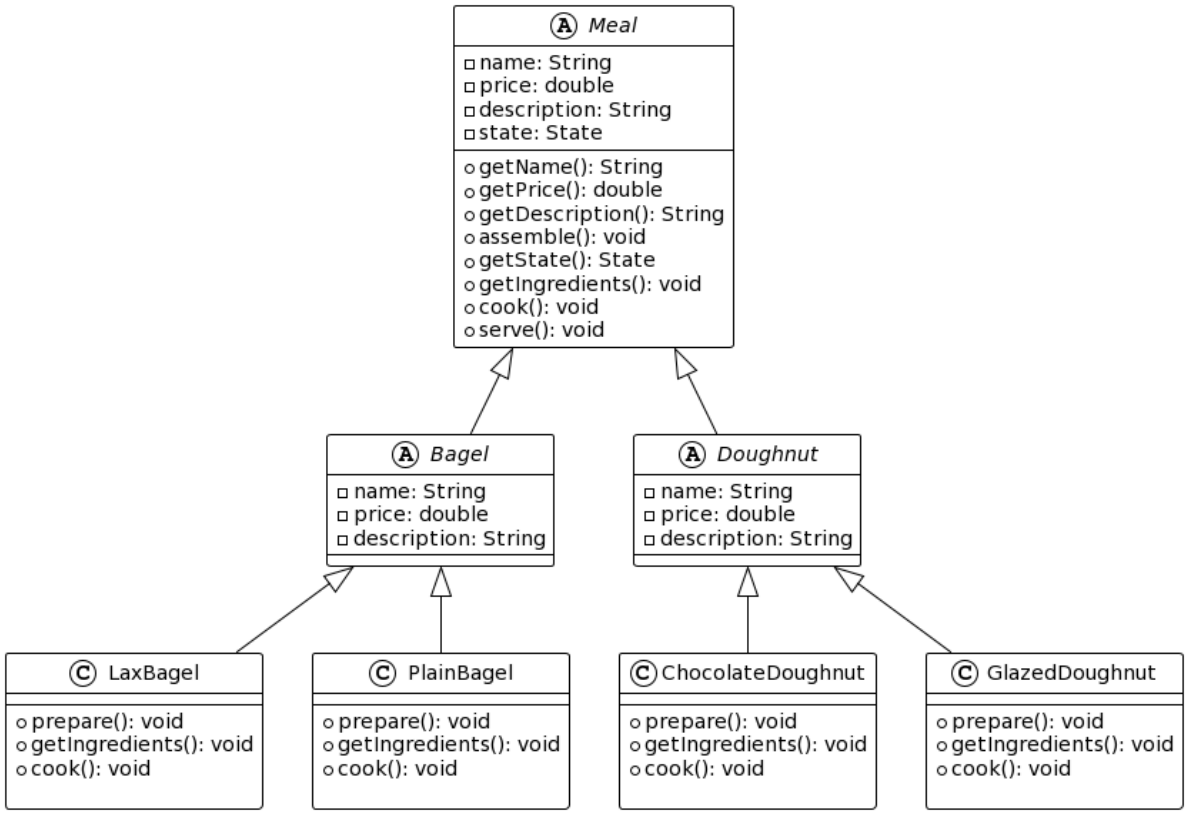


UML of classes involving the Abstract Factory design pattern.

TEMPLATE DESIGN PATTERN:

The Template pattern is a behavioural pattern that defines the skeleton of an algorithm in a base class, allowing subclasses to override specific steps of the algorithm without changing its structure. It encapsulates the common steps of an algorithm in a template method, which can be customised by subclasses to provide their own implementations for specific steps. This pattern promotes code reuse and provides a way to define a family of algorithms with a common structure.

- The Meal class serves as the base template for creating different types of meals with the Assemble() method.
- Subclasses (PlainBagel, LaxBagel, ChocolateDoughnut, GlazedDoughnut) extend the Meal class and provide their own implementations for the abstract methods (getIngredients(), cook()).
- The assemble() method in the Meal class defines the common algorithm for preparing a meal by calling the abstract methods implemented by subclasses.
- Subclasses can override specific steps of the algorithm while keeping the overall structure intact.



UML of classes involving the Template design pattern.

ITERATOR DESIGN PATTERN:

The Iterator pattern is a behavioural pattern that provides a way to access the elements of an aggregate object sequentially without exposing its internal representation. It defines an interface (Iterable) and an associated iterator class that encapsulates the traversal logic. The iterator class provides methods for accessing the elements of the aggregate in a controlled manner, such as moving to the next element and checking if there are more elements.

- The BeverageToppingsManager class implements the Iterable<BeverageDecorator> interface, enabling iteration over the toppings of a beverage.
- It provides an iterator() method that returns an iterator object capable of traversing the toppings list.
- The iterator object allows the client code to iterate over the toppings of a beverage using standard iteration constructs, such as loops.
- The project **does not** explicitly implement the Iterator Design Pattern, but rather uses the implementation offered by Java.

The combination of these design patterns enables the project to achieve modularity, extensibility, and flexibility in creating and modifying objects, adding behaviours to beverages, defining a common algorithm for meal preparation, and iterating over beverage toppings. The patterns promote code reuse, separation of concerns, and loose coupling, making the project more maintainable and adaptable.

SOLID

1. Single Responsibility Principle (SRP):
    - Each class in the project has a clear and single responsibility. For example, the **Beverage** class represents a beverage and provides methods related to its properties and behaviour, while the **BeverageDecorator** class adds toppings to a beverage. This separation of responsibilities improves maintainability and allows for easier modification of specific functionalities.
2. Open-Closed Principle (OCP):
    - The project follows the OCP by allowing the addition of new functionalities without modifying existing code. For example, new beverages can be added by creating subclasses of the **Beverage** class, and new toppings can be added by creating subclasses of the **BeverageDecorator** class. This promotes extensibility and reduces the risk of introducing bugs in existing code.
3. Liskov Substitution Principle (LSP):
    - The LSP is achieved through inheritance and polymorphism. Subclasses such as **Cappuccino**, **Matcha**, **Americano**, **ChocolateDoughnut**, and **GlazedDoughnut** can be used interchangeably with their base classes (**Beverage** and **Meal**). This allows objects of the base class to be replaced with objects of their subclasses without affecting the correctness of the program.
4. Interface Segregation Principle (ISP):
    - The project uses interfaces to define specific contracts for different functionalities. For example, the **Preparable** and **Product** interfaces define methods related to preparation and product information, respectively. This allows clients to depend on specific interfaces rather than on large, monolithic interfaces, promoting loose coupling and improved maintainability.
5. Dependency Inversion Principle (DIP):
    - The project adheres to the DIP by relying on abstractions and decoupling dependencies. The **IBeverage**, **Preparable**, and **Product** interfaces serve as abstractions that define the contracts for different components. Concrete classes depend on these interfaces, promoting loose coupling and enabling the use of different implementations at runtime. This enhances flexibility, testability, and maintainability.

OOP

1. Encapsulation:
    - The project encapsulates internal state and behaviour within classes, exposing only necessary methods and properties to the outside world. Classes such as **Beverage**, **BeverageDecorator**, **Meal**, and others encapsulate their respective functionalities, ensuring that internal details are hidden and protected from unauthorised access.
2. Inheritance:
    - The project utilises inheritance to model relationships between classes. Subclasses such as **Cappuccino**, **Matcha**, **Americano**, **ChocolateDoughnut**, and **GlazedDoughnut** inherit common behaviours and properties from their respective base classes (**Beverage** and **Meal**). This promotes code reuse, enables polymorphism, and simplifies the addition of new subclasses in the future.
3. Polymorphism:
    - The project demonstrates polymorphism through method overriding and dynamic dispatch. Subclasses override methods from their base classes to provide specific implementations. For example, **Cappuccino** overrides the **prepare()** method to prepare a cappuccino, while **Americano** overrides the **prepare()** method to prepare an Americano. This allows objects of different classes to be treated uniformly through their common base class interface.
4. Interface Contracts:

    Interface contracts play a vital role in information hiding by defining a set of public methods that implementing classes must adhere to. Interfaces specify the behaviour expected from concrete implementations, allowing client code to interact with objects through a standardised interface while hiding the underlying implementation details. For example, the **Preparable** interface defines a single method, **prepare()**, which is implemented by classes like **Americano** and **Bagel**. This enforces a contract that any class implementing the **Preparable** interface must provide the necessary functionality. By programming to interfaces rather than concrete implementations, the project achieves loose coupling and shields client code from specific implementation details.