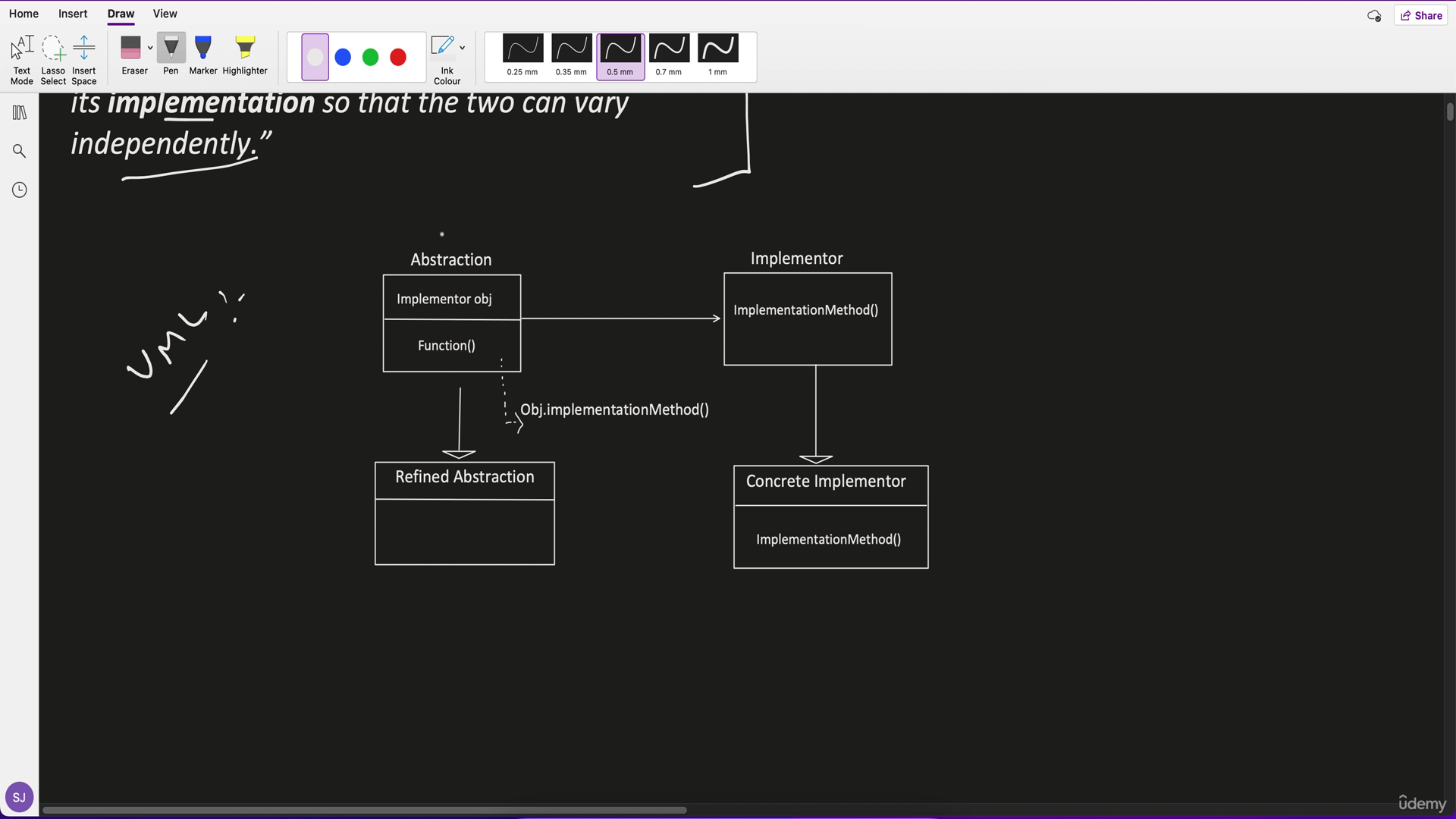
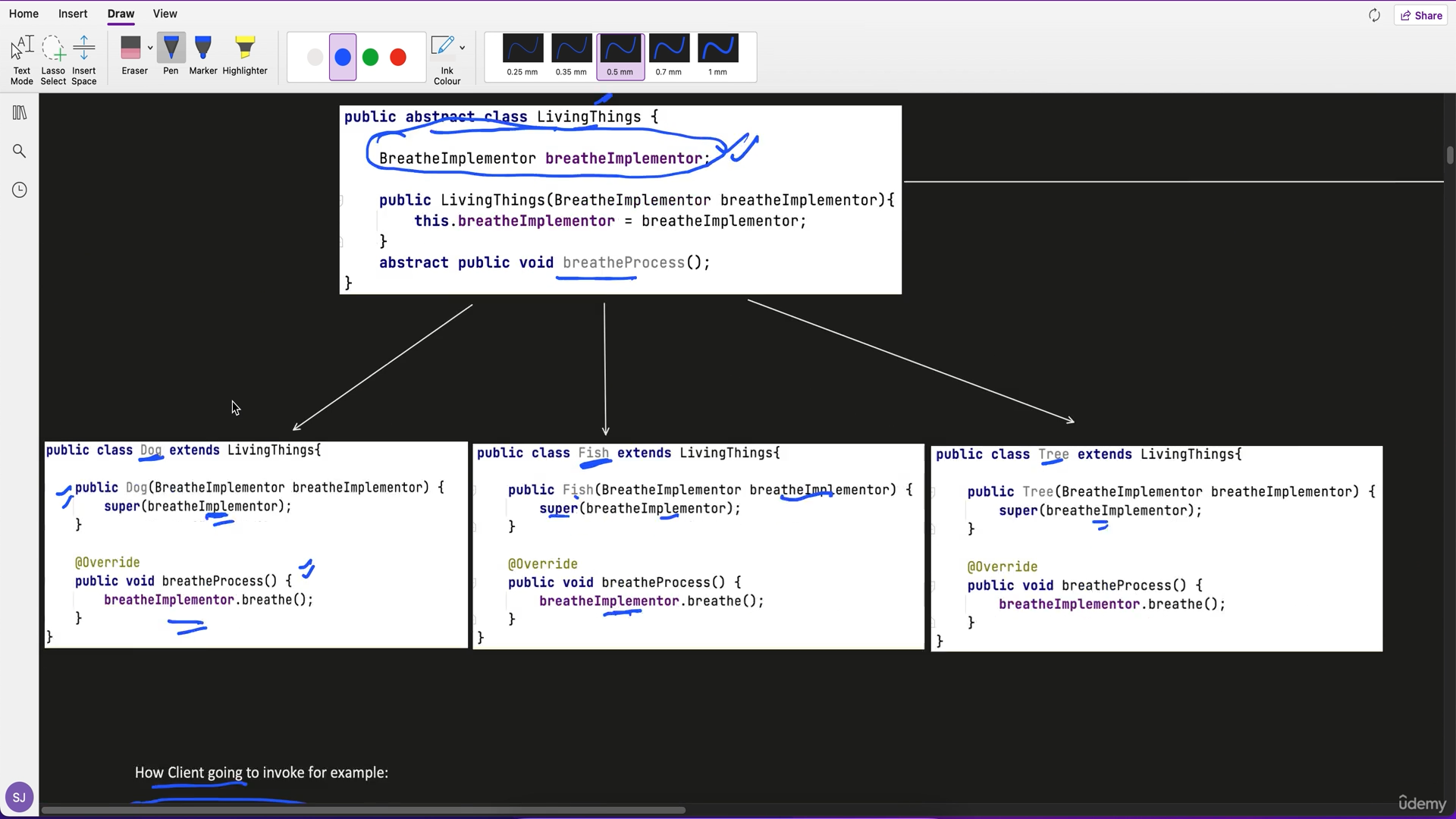
**Bridge Design Pattern**

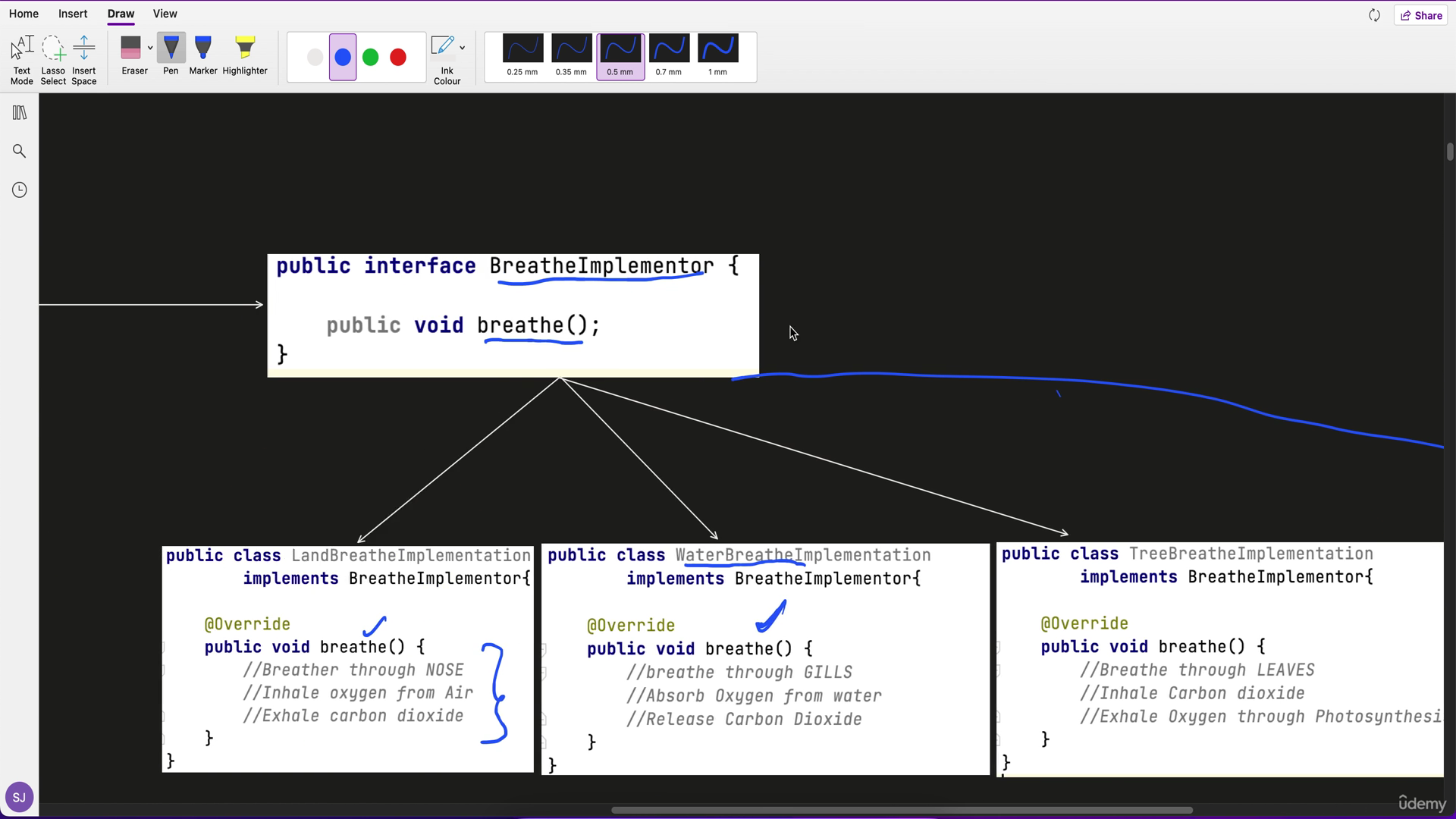
The Bridge design pattern is a structural pattern that separates an object's abstraction from its implementation, allowing the two to vary independently. This pattern helps in decoupling the interface and implementation, enabling them to be developed and evolved separately.

**Definition**

The Bridge pattern allows you to decouple an abstraction from its implementation so that the two can vary independently. It is useful when you need to avoid a permanent binding between an abstraction and its implementation and when both the abstractions and implementations should be extensible by subclassing. UML:-







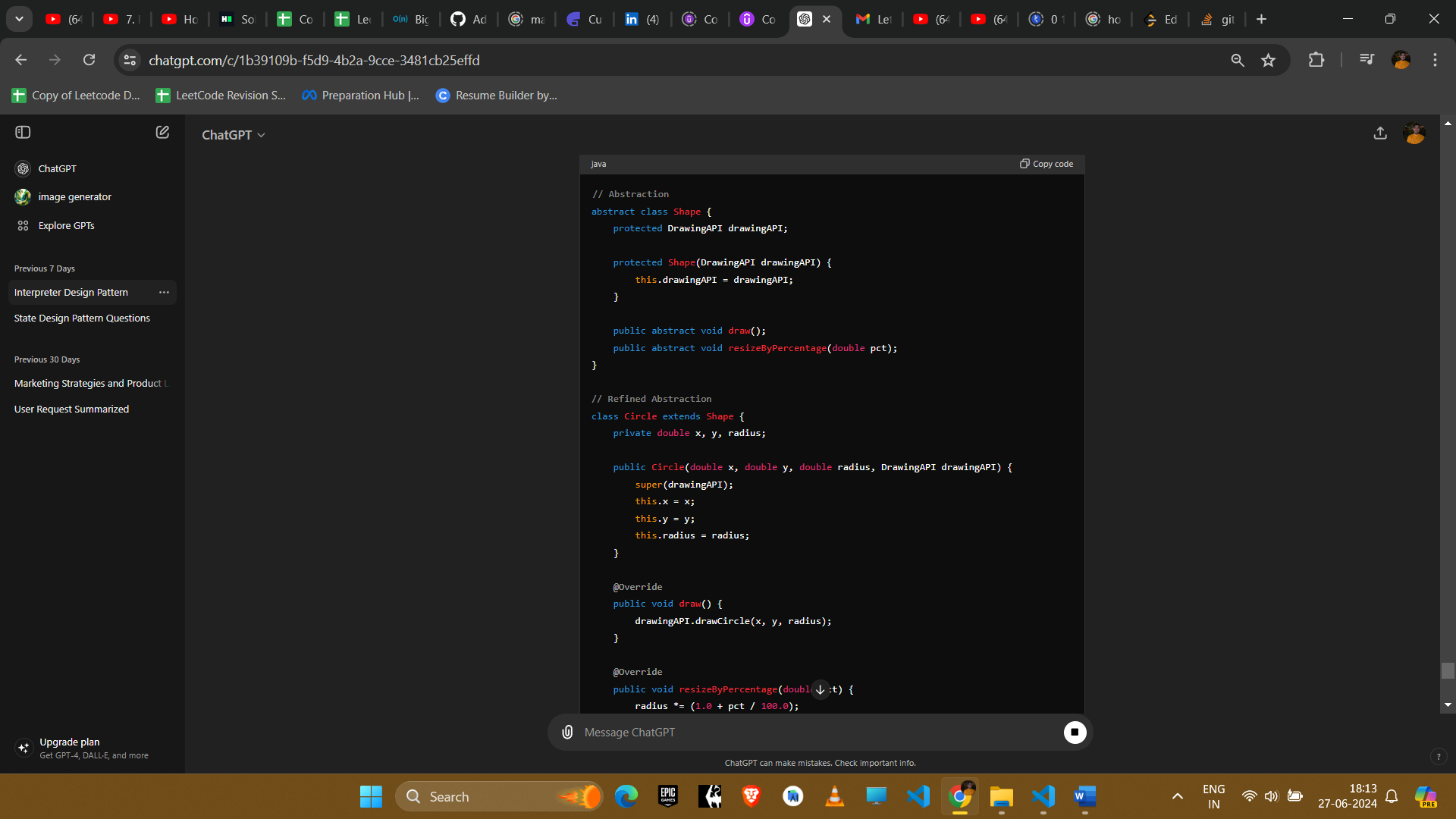
How client will invoke this:-

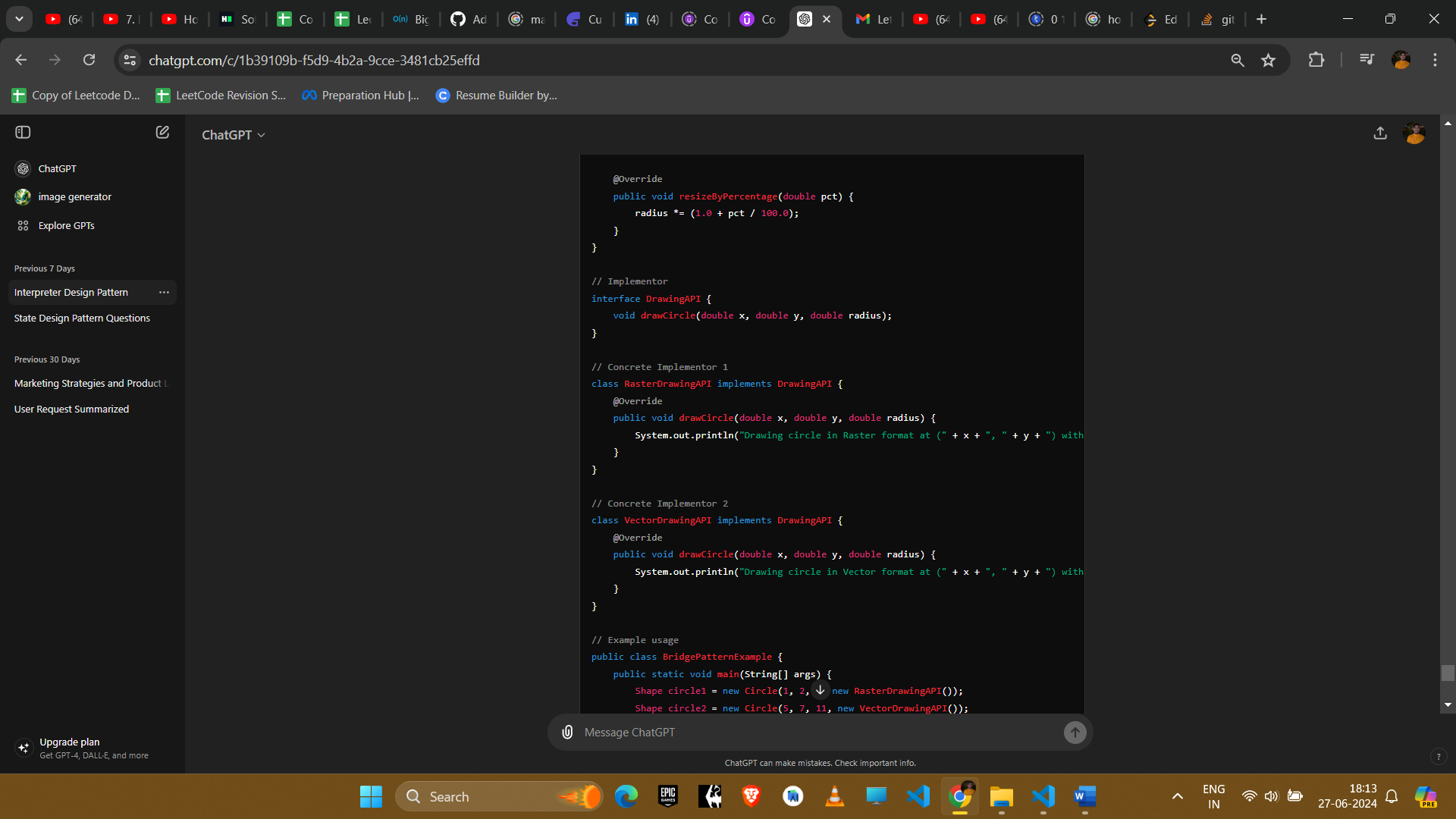
LivingThings fishObj = new Fish(new WaterBreatheImplementor());

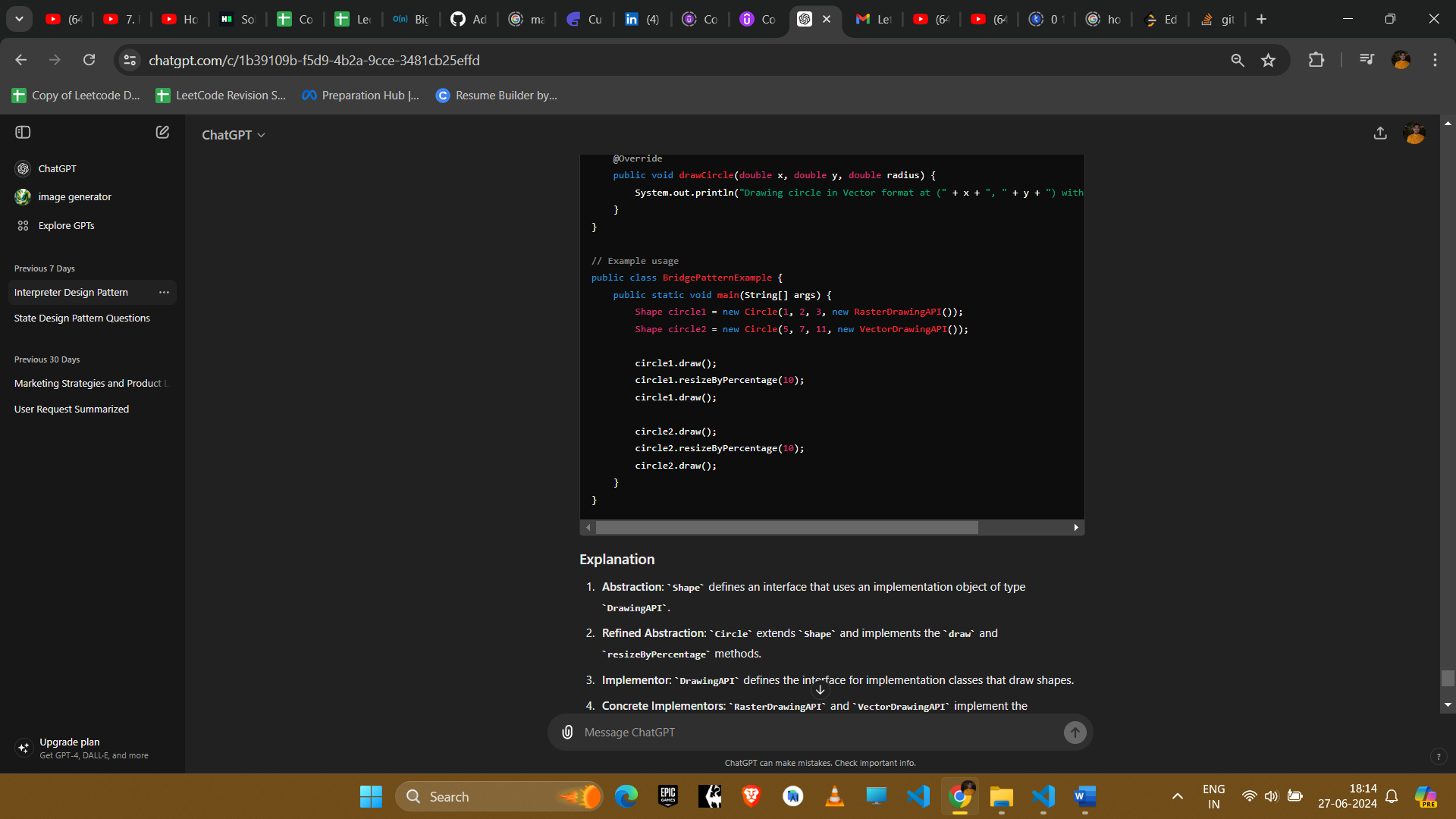
fishObj.breatheProcess();

**Example**

Consider a scenario where we have different shapes that can be drawn with different drawing APIs (like a raster or vector drawing API).







**Explanation**

1. **Abstraction**: Shape defines an interface that uses an implementation object of type DrawingAPI.
2. **Refined Abstraction**: Circle extends Shape and implements the draw and resizeByPercentage methods.
3. **Implementor**: DrawingAPI defines the interface for implementation classes that draw shapes.
4. **Concrete Implementors**: RasterDrawingAPI and VectorDrawingAPI implement the DrawingAPI interface, providing concrete implementations for drawing shapes in different formats.

**Example Uses in Amazon Interviews**

1. **Device Control Systems**
   * **Scenario**: Controlling various devices (like printers, scanners) using different APIs.
   * **Implementation**: Use the Bridge pattern to decouple the device control interface from the specific device control implementations, allowing for flexible addition of new devices and control methods.
2. **UI Themes**
   * **Scenario**: Developing a user interface that can support multiple themes.
   * **Implementation**: Use the Bridge pattern to separate the UI controls from their theme-specific rendering, allowing easy addition of new themes without modifying the controls.
3. **Database Persistence**
   * **Scenario**: Supporting multiple database backends (like SQL and NoSQL databases).
   * **Implementation**: Use the Bridge pattern to separate the data access logic from the specific database implementation, enabling the application to switch databases with minimal changes.
4. **Logging Frameworks**
   * **Scenario**: Integrating multiple logging frameworks in an application.
   * **Implementation**: Use the Bridge pattern to decouple the logging interface from the specific logging framework, allowing the application to switch or extend logging frameworks easily.

**Conclusion**

The Bridge pattern is beneficial for decoupling an abstraction from its implementation, allowing them to evolve independently. It promotes flexibility and scalability in the system design, making it a valuable tool in various software development scenarios where you need to vary both the abstraction and implementation independently.