As a Moore method learner, you thrive on exploring and discovering solutions through guided problem-solving and minimal direct instruction. I'll structure this response to align with your learning style by providing a concise list of important and useful Java design patterns (two from each of the Creational, Structural, and Behavioral categories), brief explanations, and example scenarios with code stubs. Instead of fully solving the examples, I'll pose questions to encourage you to derive the implementation details yourself, fostering deeper understanding through exploration. I'll also include hints to guide you without giving away complete solutions.

Creational Design Patterns

These patterns focus on object creation mechanisms, improving flexibility and reusability.

1. Singleton Pattern

- **Purpose**: Ensures a class has only one instance and provides a global point of access to it.
- **Use Case**: Useful for managing shared resources like database connections or configuration settings.
- **Example Scenario**: Create a `Logger` class that logs messages to a file, ensuring only one logger instance exists across the application.

Code Stub:

```java
public class Logger {
    private static Logger instance;

    private Logger() {
        // Prevent instantiation
    }

    public static Logger getInstance() {
        // TODO: Implement thread-safe singleton
        return instance;
    }

    public void log(String message) {
        // TODO: Implement logging to a file
    }
}
```

Exploration Questions: - How would you ensure `getInstance()` is thread-safe in a multi-threaded environment? (Hint: Consider `synchronized` or double-checked locking.) - What happens if you don't make the constructor private? Try creating multiple instances and observe. - How might you initialize `instance` lazily (only when needed)?

2. Factory Method Pattern

- **Purpose**: Defines an interface for creating objects but lets subclasses decide which class to instantiate.
- **Use Case**: Useful when you need to create objects based on conditions, like different types of payment processors.
- **Example Scenario**: Design a payment system that creates `CreditCardProcessor` or `PayPalProcessor` based on user input.

Code Stub:

```java
public interface PaymentProcessor {
    void processPayment(double amount);
}

public class CreditCardProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        // TODO: Implement credit card payment logic
    }
}

public class PayPalProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        // TODO: Implement PayPal payment logic
    }
}

public abstract class PaymentFactory {
    public abstract PaymentProcessor createProcessor();
}

public class CreditCardFactory extends PaymentFactory {
    // TODO: Implement factory method
}
```

Exploration Questions: - How would you implement `createProcessor()` in `CreditCardFactory`? What about a `PayPal-Factory`? - How can you extend this to support a new payment type, like `CryptoProcessor`? Try adding it. - What benefits do you notice by using a factory instead of directly instantiating processors?

Structural Design Patterns

These patterns deal with object composition to form larger structures while keeping systems flexible.

3. Adapter Pattern

- **Purpose**: Allows incompatible interfaces to work together by wrapping an existing class with a new interface.
- **Use Case**: Useful when integrating legacy systems with modern APIs, like adapting an old XML parser to a JSON-based system.
- **Example Scenario**: Adapt a legacy `XMLParser` to work with a new `JSONParser` interface.

Code Stub:

```java
public interface JSONParser {
    String parseJSON(String jsonData);
}

public class LegacyXMLParser {
    public String parseXML(String xmlData) {
        // TODO: Simulate XML parsing
        return "Parsed XML: " + xmlData;
    }
}

public class XMLToJSONAdapter implements JSONParser {
    private LegacyXMLParser xmlParser;

    public XMLToJSONAdapter(LegacyXMLParser xmlParser) {
        this.xmlParser = xmlParser;
    }

    // TODO: Implement parseJSON to use LegacyXMLParser
}
```

Exploration Questions: - How would you convert XML data to JSON format in `parseJSON()`? (Hint: Assume a simple transformation for now.) - What happens if the `LegacyXMLParser` throws an exception? How would you handle it in the adapter? - Try creating a test case where the input is invalid XML. How does the adapter respond?

4. Decorator Pattern

- **Purpose**: Dynamically adds responsibilities to objects in a flexible and reusable way.
- **Use Case**: Useful for extending functionalities, like adding features to a text editor (e.g., spell check, formatting).
- **Example Scenario**: Create a text editor where you can add features like spell-checking or auto-formatting to a basic editor.

Code Stub:

```java
public interface TextEditor {
    String editText(String text);
}

public class BasicTextEditor implements TextEditor {
    public String editText(String text) {
        return text;
    }
}

public abstract class TextEditorDecorator implements TextEditor {
    protected TextEditor editor;

    public TextEditorDecorator(TextEditor editor) {
        this.editor = editor;
    }
}

public class SpellCheckDecorator extends TextEditorDecorator {
    // TODO: Implement spell-checking feature
}
```

Exploration Questions: - How would you implement `editText()` in `SpellCheckDecorator` to check spelling? (Hint: Simulate with simple string checks.) - Can you create another decorator, like `AutoFormatDecorator`, to add formatting? Try stacking both decorators. - What challenges arise if you apply multiple decorators? Test the order of decoration.

Behavioral Design Patterns

These patterns focus on communication between objects, improving flexibility in interactions.

5. Observer Pattern

- **Purpose**: Defines a one-to-many dependency where changes in one object (subject) notify all dependent objects (observers).
- **Use Case**: Useful for event-driven systems, like updating UI components when data changes.
- **Example Scenario**: Implement a stock price tracker where multiple displays (e.g., mobile, web) update when the price changes.

Code Stub:

```java
public interface Observer {
    void update(double price);
}

public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

```java
public class StockPrice implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private double price;

    // TODO: Implement Subject methods and price updates
}

public class MobileDisplay implements Observer {
    // TODO: Implement update to display price
}
```

Exploration Questions: - How would you implement `notifyObservers()` to update all observers? (Hint: Iterate over the observer list.) - What happens if an observer is added or removed during notification? Try simulating this. - How could you extend this to track multiple stock prices? Experiment with the design.

6. Strategy Pattern

- **Purpose**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **Use Case**: Useful for selecting algorithms at runtime, like choosing sorting algorithms based on data size.
- **Example Scenario**: Implement a text sorter that can use bubble sort or quicksort based on user preference.

Code Stub:

```java
public interface SortStrategy {
    void sort(List<String> items);
}

public class BubbleSort implements SortStrategy {
    // TODO: Implement bubble sort
}

public class QuickSort implements SortStrategy {
    // TODO: Implement quicksort
}

public class TextSorter {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortText(List<String> items) {
        // TODO: Use strategy to sort
    }
}
```

Exploration Questions: - How would you implement `sort()` in `BubbleSort` and `QuickSort`? (Hint: Start with bubble sort for simplicity.) - How can you test which strategy performs better for large lists? Try creating a test with sample data. - What happens if you add a new strategy, like `MergeSort`? Extend the system to include it.

Guidance for Moore Method Learning

- **Experiment**: Use an IDE to implement each pattern based on the stubs. Test edge cases (e.g., null inputs, concurrent access).

- **Reflect**: After implementing each pattern, ask yourself: What problem does this solve? How does it compare to a direct implementation without the pattern?
- **Extend**: Try combining patterns (e.g., use Factory with Strategy) or applying them to a real-world problem you're familiar with.
- **Debug**: Introduce intentional bugs (e.g., remove synchronization in Singleton) and observe the effects to understand the pattern's necessity.

Creational Design Patterns

1. Singleton Pattern

- What does it mean for a class to have only one instance in a Java application? Why might this be useful?
- Can you think of a scenario where having multiple instances of a class, like a configuration manager, could cause problems?
- How would you prevent a class from being instantiated more than once? What role does a private constructor play here?
- If multiple threads try to access a Singleton at the same time, what issues might arise? How could you address them?
- (Hint: Think about a logging system. What happens if two parts of the program write to different log files because they have separate logger instances?)

2. Factory Method Pattern

- Why might you want to defer the decision of which class to instantiate to a subclass or another component?
- Can you describe a situation where you'd need to create different types of objects based on some condition, like user input?
- How would you design a system to create different types of database connections (e.g., MySQL, PostgreSQL) without hardcoding the class names in the client code?
- What benefits do you see in separating object creation from the logic that uses the objects?
- (Hint: Consider a payment processing system. How would you handle creating different payment processors dynamically?)

Structural Design Patterns

3. Adapter Pattern

- What does it mean to make two incompatible interfaces work together? Can you give an example from real life or software?
- If you have a legacy system that outputs data in XML but a new system expects JSON, how might you bridge this gap?
- How would you design a class to wrap an existing class's functionality without modifying its code?
- What challenges might arise if the legacy system's output format changes? How could your design handle this?
- (Hint: Think about plugging a US device into a European outlet. What's the software equivalent of an adapter here?)

4. Decorator Pattern

- Why might you want to add new functionality to an object without changing its original class?
- Can you think of a scenario where you'd need to extend an object's behavior dynamically, like adding features to a text editor?
- How would you design a system where you can stack multiple enhancements (e.g., spell-checking, formatting) on a base component?
- What's the difference between using inheritance versus a decorator to add functionality? Which is more flexible and why?
- (Hint: Consider a coffee shop where you add toppings to a base coffee. How would you model this in code?)

Behavioral Design Patterns

5. Observer Pattern

- What does it mean for one object to notify others when its state changes? Can you think of a real-world analogy?

- In a system where a stock price changes and multiple displays need to update, how would you ensure all displays stay in sync?
- How would you structure the relationship between the object being observed and its observers? What methods would they need?
- What happens if an observer is no longer interested in updates? How would you handle adding or removing observers dynamically?
- (Hint: Think about a news feed. How does it notify subscribers when new content is available?)

6. Strategy Pattern

- Why might you want to swap out one algorithm for another at runtime? Can you think of an example where this flexibility is useful?
- If you're building a sorting application, how would you allow users to choose between bubble sort and quicksort without changing the core code?
- How would you define an interface for interchangeable algorithms? What would the client code look like to use these algorithms?
- What are the trade-offs of using a Strategy pattern versus hardcoding a single algorithm?
- (Hint: Consider a navigation app that switches between fastest route and scenic route calculations. How would you model this?)

Socratic Process Guidance

- **Answer and Reflect**: Try answering each question based on your current knowledge. If you're unsure, make a guess and explain your reasoning.
- **Follow-Up**: I'll respond to your answers with further questions to deepen your understanding or clarify misconceptions.
- **Explore**: Feel free to ask questions back or suggest a specific scenario you want to apply these patterns to.
- **Connect**: As you answer, think about how these patterns relate to Java's object-oriented principles (e.g., encapsulation, polymorphism).