**Conceptual Understanding**

**What is the Memento design pattern, and why is it used?**  
The **Memento** pattern provides a way to capture and restore an object's state without exposing its internal details. It is commonly used for undo/redo functionality.

**2. Implementation Questions**

**How would you implement the Memento pattern in Python?**  
A simple implementation for saving and restoring the state of a TextEditor:

python

Copy code

# Memento

class TextEditorMemento:

def \_\_init\_\_(self, state):

self.state = state

# Originator

class TextEditor:

def \_\_init\_\_(self):

self.\_content = ""

def write(self, text):

self.\_content += text

def save(self):

return TextEditorMemento(self.\_content)

def restore(self, memento):

self.\_content = memento.state

def get\_content(self):

return self.\_content

# Caretaker

class History:

def \_\_init\_\_(self):

self.\_history = []

def push(self, memento):

self.\_history.append(memento)

def pop(self):

return self.\_history.pop() if self.\_history else None

# Usage

editor = TextEditor()

history = History()

editor.write("Hello, ")

history.push(editor.save())

editor.write("World!")

print(editor.get\_content()) # Output: Hello, World!

editor.restore(history.pop())

print(editor.get\_content()) # Output: Hello,

**What are the key components of the Memento pattern?**

1. **Memento**: Stores the internal state of the originator object.
2. **Originator**: The object whose state needs to be saved and restored.
3. **Caretaker**: Manages memento objects, ensuring the originator's state can be restored when needed.

**What are the responsibilities of each component in the Memento pattern?**

* **Memento**: Holds the state without exposing its structure.
* **Originator**: Creates and restores mementos.
* **Caretaker**: Requests mementos from the originator and stores them.

**3. Real-World Scenarios**

**What are common use cases for the Memento pattern?**

1. **Undo/Redo**: Text editors, drawing applications, or IDEs.
2. **Game Development**: Saving and loading game states.
3. **Database Transactions**: Storing rollback points in case of failure.
4. **Configuration Management**: Saving snapshots of settings before applying changes.

**How would you use the Memento pattern in a game?**  
Store the state of the game (e.g., level, player health, inventory) as a memento. When the player chooses to load a previous state, restore from the corresponding memento.

**4. Behavioral Questions**

**What are the advantages and disadvantages of the Memento pattern?**

* **Advantages**:
  1. Provides a clean way to implement undo/redo functionality.
  2. Avoids exposing the internal state of the originator to other objects.
  3. Supports rollback or checkpointing mechanisms.
* **Disadvantages**:
  1. Can increase memory usage if many mementos are stored.
  2. May involve deep copying for complex objects, which can be slow.
  3. The caretaker has to manage the lifecycle of mementos, which can complicate implementation.

**What design principles does the Memento pattern follow?**

* **Single Responsibility Principle**: The originator handles its state, and the caretaker manages the memento's lifecycle.
* **Encapsulation**: The memento hides the originator's internal state from other objects.

**How would you handle large states in the Memento pattern?**

* Use incremental snapshots (only save the differences between states).
* Serialize the memento to disk to reduce memory usage.
* Use version control systems for complex configurations.

**5. Code Debugging Questions**

**Given a Memento implementation, identify issues or suggest improvements.**

* **Problem**: Caretaker holds too many unnecessary mementos.
  + **Solution**: Limit the number of mementos stored (e.g., a fixed-size stack).
* **Problem**: Mementos expose the internal structure of the originator.
  + **Solution**: Ensure the memento only stores state in a minimal and encapsulated manner.

**How would you test a Memento implementation?**

1. Test saving and restoring the state of the originator.
2. Test undo and redo functionality with multiple mementos.
3. Verify that restoring a memento does not affect unrelated objects.
4. Test edge cases like restoring when no memento is available.

**6. Advanced Topics**

**How would you handle versioning in the Memento pattern?**  
Add a version or timestamp to each memento:

python

Copy code

class Memento:

def \_\_init\_\_(self, state, version):

self.state = state

self.version = version

**Can the Memento pattern be combined with other patterns?**

1. **Command**: Use commands to execute actions and store mementos for undo/redo functionality.
2. **State**: Save and restore states of a finite state machine using mementos.
3. **Observer**: Notify other objects when a state is restored.

**7. Comparison with Other Patterns**

**How does the Memento pattern differ from the Prototype pattern?**

* **Memento**: Saves the internal state of an object for later restoration.
* **Prototype**: Creates a copy of an object to reuse or clone.

**How does the Memento pattern differ from the Command pattern?**

* **Memento**: Focuses on saving and restoring object state.
* **Command**: Encapsulates an action or operation as an object and may also store state for undo functionality.

**8. Real-Life Questions**

**Describe a real-world scenario where you used the Memento pattern.**  
Example: Implementing an **undo/redo** feature in a text editor. Each time a user makes an edit, the editor saves the state as a memento. When the user clicks "Undo," the editor restores the last saved state.

**Can the Memento pattern be used in a multithreaded environment? How?**  
Yes, but ensure thread safety:

1. Use locks when accessing the memento stack.
2. Use thread-safe collections for storing mementos.
3. Serialize mementos to avoid shared memory conflicts.