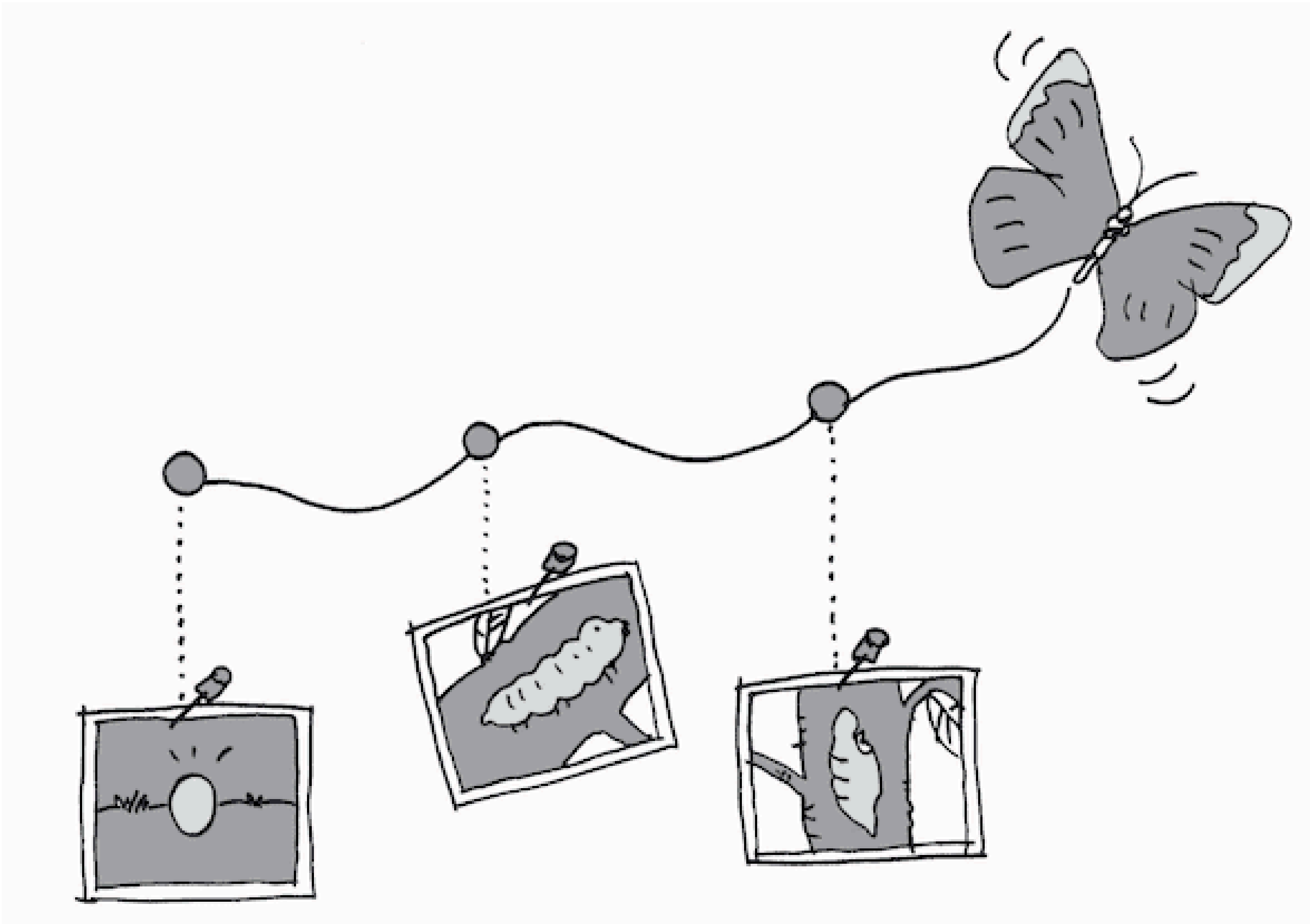
Memento Pattern

Save and Restore Object State Without Violating Encapsulation



Memento Pattern

Think of a **video game save system** - you can save your progress and later restore to that exact state:

- **Video game**: Save progress at a checkpoint, restore if you lose

We have other examples:

- **Text editor**: Undo/Redo functionality preserves previous states
- **Database**: Transaction rollback restores the previous state

The Problem

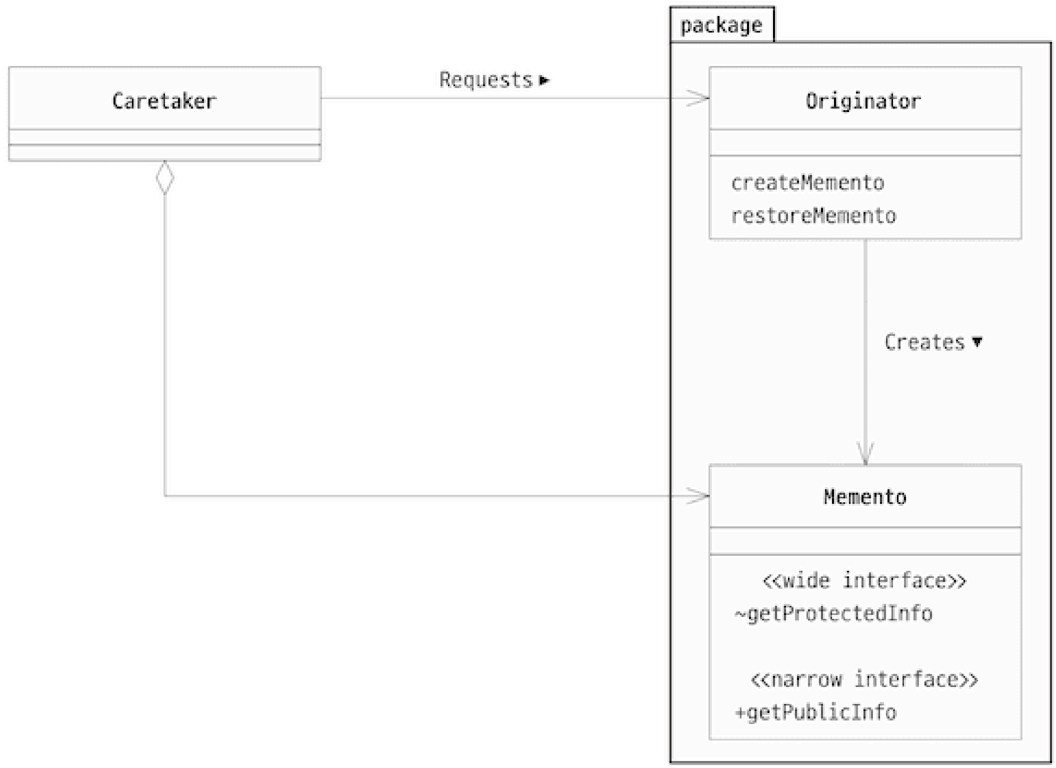
- We have a **gaming system** where players can bet and win/lose money.
- We want to provide **save/restore** functionality like game checkpoints.
- We cannot expose internal state directly - this would break **encapsulation**.

The challenge: how to save the state of a complex object **externally** while keeping its implementation details **hidden** from the user?

The *Memento* as the Solution

- We create a **memento object** that stores the state snapshot.
- Only the **originator** (original object) can create and restore from a memento.
- **External objects** can hold a memento but cannot examine its contents.

The Solution (Design)



Step 1: Understand the Players

In this design, we have three key players:

- **Originator** (Gamer): The object whose state we want to save
- **Memento**: Stores the internal state of the originator

We have the Catertaker (Main) that manages mementos but doesn't examine their contents.

- **Caretaker** (Main)

Step 2: Encapsulation is preserved

- Only the **originator** can access the memento's internal state.
- **Caretakers** treat memento as an **opaque object**.

Step 3: Understand the Originator

- **Originator** creates a memento containing a snapshot of the current state.
- **Originator** can restore its state from a given memento.
- In our example: **Gamer** can save its money and fruit state.

Step 4: Understand the Memento

- **Memento** stores state information from the originator.
- **Memento** provides **narrow interface** to caretaker (limited access).
- **Memento** provides **wide interface** to originator (full access).

Step 5: Understand the Caretaker

- **Caretaker** requests a memento from the originator and stores it.
- **Caretaker** gives the memento back to the originator when restoration is needed.
- **Caretaker** never examines or modifies memento contents.

Code

- Main Method (Caretaker)
- Originator Class (Gamer)
- Memento Class

Main Method (Game rules & Strategy)

Starting the game with initial money: 100

Game rules:

- Roll 1: Money increases by 100
- Roll 2: Money is halved
- Roll 6: Get a fruit
- Other: Nothing happens

Strategy:

- Save state when money is more memento
- Restore state when money drops to less than half 1/2 amount of memento

```python
from gamer import Gamer

def main():
    gamer = Gamer(100)  # Start with $100
    memento = gamer.create_memento()  # Save initial state

    for i in range(30):
        gamer.bet()  # Play the game

        if gamer.get_money() > memento.get_money():
            # Save state when money increases
            memento = gamer.create_memento()
        elif gamer.get_money() < memento.get_money() // 2:
            # Restore when money drops too much
            gamer.restore_memento(memento)

        if gamer.get_money() <= 0:
            break
```

Step 1: Create originator and save initial state

```
gamer = Gamer(100)  # Start with $100
memento = gamer.create_memento()  # Save initial state
```

- **Gamer** is our originator who manages money and fruits.
- We immediately save the initial state as our first **checkpoint**.

Step 2: Modify object state

```
gamer.bet()  # Play the game
```

- The **gamer** bets and the state changes (money increases/decreases, fruits gained).
- **Caretaker** doesn't know the internal details of how betting works.

Step 3: Conditionally save/restore state

```python
if gamer.get_money() > memento.get_money():
    memento = gamer.create_memento()  # Save good state
elif gamer.get_money() < memento.get_money() // 2:
    gamer.restore_memento(memento)   # Restore previous state
```

- **Smart strategy**: Save when we're doing well, restore when losing too much.
- **Caretaker** makes decisions but doesn't access memento contents directly.

Originator Class (Gamer)

```
class Gamer:
    def __init__(self, money):
        self.money = money
        self.fruits = []

    def create_memento(self):
        memento = Memento(self.money)
        for fruit in self.fruits:
            memento.add_fruit(fruit)
        return memento

    def restore_memento(self, memento):
        self.money = memento.get_money()
        self.fruits = memento.get_fruits()

    def bet(self):
        # Game logic that modifies state
        dice = random.randint(1, 6)
        if dice == 1: self.money += 100
        elif dice == 2: self.money //= 2
        elif dice == 6: self.fruits.append(self._get_fruit())
```

Key Points: Originator

1. Creates memento: `create_memento()` but never stores the memento.
2. Restores from memento: `restore_memento()` rebuilds state
3. Controls access: Only the originator can set/get memento data
4. Maintains encapsulation: Internal logic remains hidden

Memento Class

Memento manages Gamer's money and fruits.

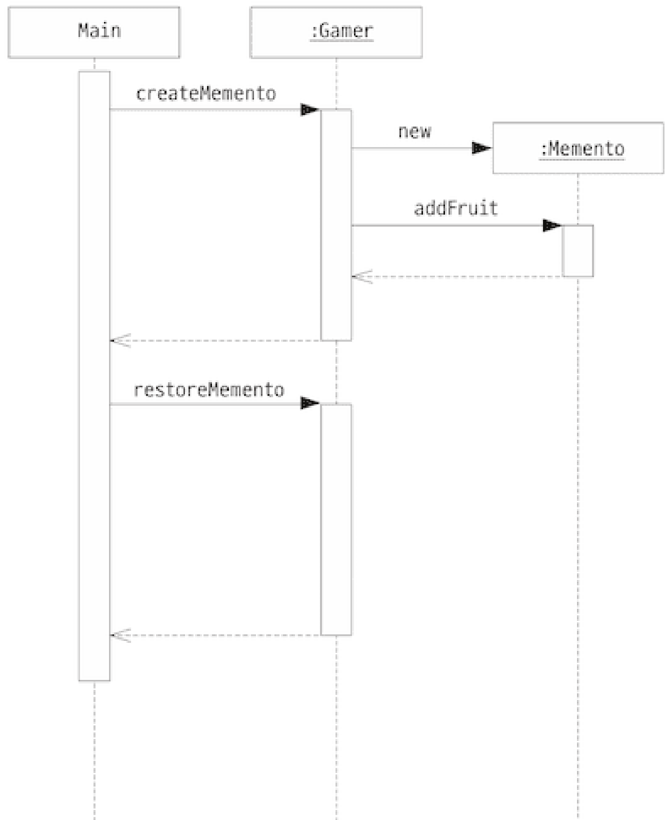
```
class Memento:
    def __init__(self, money):
        self._money = money         # Private to preserve encapsulation
        self._fruits = []           # Private to preserve encapsulation

    def get_money(self):            # Narrow interface for caretaker
        return self._money

    def add_fruit(self, fruit):     # Wide interface for originator
        self._fruits.append(fruit)

    def get_fruits(self):           # Wide interface for originator
        return self._fruits.copy()
```

Key Points: Memento

1. **Encapsulation**: Internal state marked private
2. **Immutable-like**: State not modified after creation
3. **State snapshot**: Complete copy of originator's relevant state

Discussion

Sequence of Operations



1. **Caretaker** asks **Originator** to create memento
2. **Originator** creates **Memento** with current state
3. **Caretaker** stores memento reference
4. Later, **Caretaker** gives memento back to **Originator**
5. **Originator** restores state from **Memento**

Two Interfaces

Caretaker can only do this:

```
memento = gamer.create_memento()  # Get memento
gamer.restore_memento(memento)    # Give memento back
# Caretaker CANNOT peek inside the memento or modify it
```

Originator can do everything:

```
class Memento:
    def get_money(self):           # Wide interface - internal access
        return self._money

    def get_fruits(self):          # Wide interface - internal access
        return self._fruits
```

Key Benefits

1. **Encapsulation**: Object's internal state remains private
2. **Externalized state**: State management separated from business logic
3. **Multiple snapshots**: Can store multiple states simultaneously
4. **Undo functionality**: Easy to implement rollback operations

When to Use Memento

- When you need to save snapshots of an object's state
- When a direct interface to the state would expose implementation details
- When you want to provide undo/rollback functionality
- When the state needs to be restored to its previous conditions

Cautions

1. **Memory usage**: Storing many mementos can be expensive
2. **State consistency**: Ensure memento represents consistent state
3. **Version compatibility**: Changes to the originator may invalidate old mementos
4. **Interface evolution**: Maintain narrow/wide interface distinction

Related Patterns

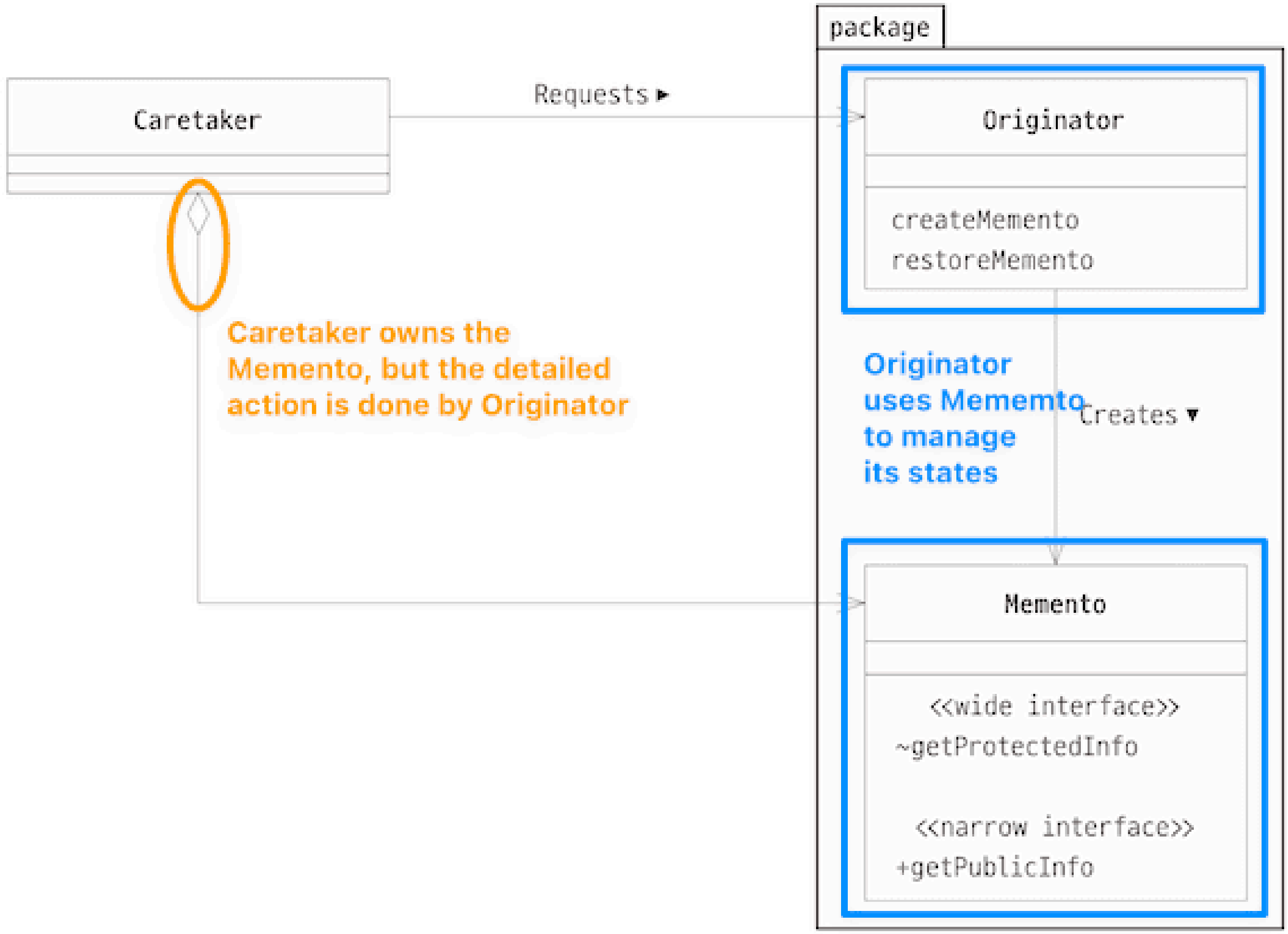
- **Prototype**: Memento saves state (object), Prototype clones entire objects
- **Iterator**: Both provide access without exposing internal structure

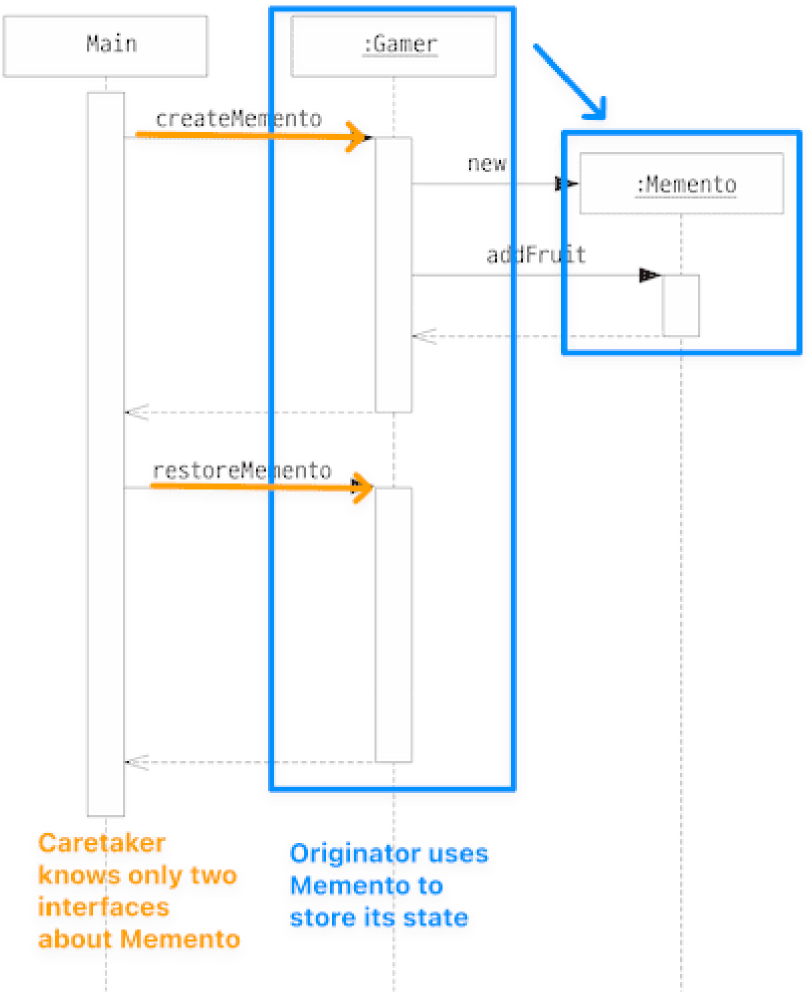
Memento vs Other Patterns

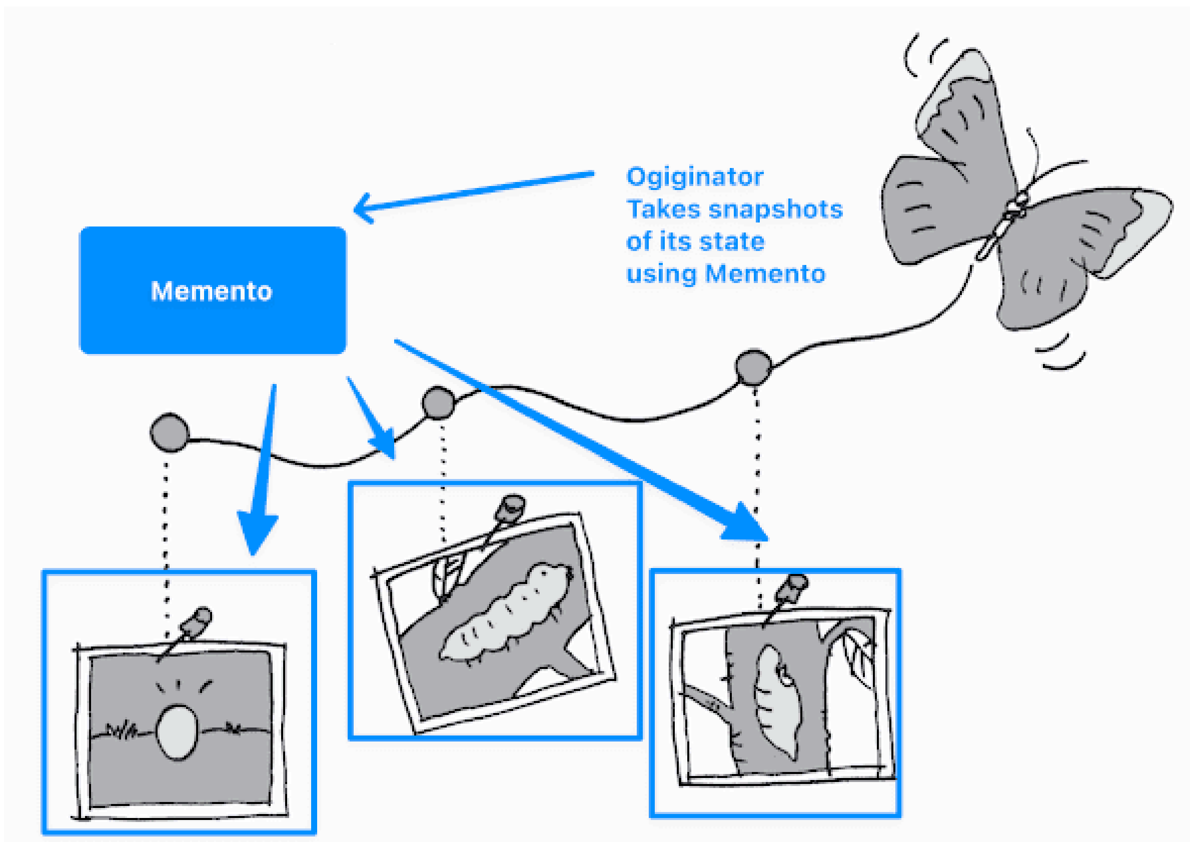
Prototype Pattern:

- **Prototype**: Creates new objects by cloning
- **Memento**: Saves/restores the state of existing objects

UML


package
Caretaker
Requests ▸
Originator
createMemento
restoreMemento
Caretaker owns the Memento, but the detailed action is done by Originator
Originator uses Mememto to manage its states
Creates ▾
Memento
<<wide interface>>
~getProtectedInfo
<<narrow interface>>
+getPublicInfo


Main
:Gamer
:Memento
createMemento
new
addFruit
restoreMemento
Caretaker knows only two interfaces about Memento
Originator uses Memento to store its state


Ogiginator
Takes snapshots
of its state
using Memento
Memento