Design Patterns in the PyNano Project

Teaching Assistant: Ashkan Marali
Guilan University — Advanced Programming Course
Course Code: 14032

May 20, 2025

Introduction

This documentation outlines and explains the software design patterns implemented in the PyNano project. Created and maintained as part of my responsibilities as a Teaching Assistant for the Advanced Programming course at Guilan University, the PyNano project is used as a hands-on educational tool for demonstrating real-world applications of object-oriented design.

In the sections that follow, we will explore key design patterns used in the project such as Command, Decorator, and Factory, along with additional recommended patterns like Singleton and Strategy. Each pattern is supported by real examples from the codebase or classic programming implementations to help clarify the concepts for students.

1 Command Pattern

The Command Pattern encapsulates actions as objects, allowing requests to be parameterized and executed dynamically. It also supports operations like undo/redo by maintaining command history.

Think of a remote control: each button is a command object that can be pressed (execute) or undone. You don't need to know how the TV works; you just press a button.

In `PyNano`, all commands inherit from a base class located at:

```python
# File: Commands/base.py
class BaseCommand:
    def execute(self): raise NotImplementedError
    def undo(self): raise NotImplementedError
```

Individual commands like `cd`, `ls`, and `nano` override these methods:

```python
# File: Commands/dir_commands.py
class CdCommand(BaseCommand):
    def execute(self): ...
    def undo(self): ...
```

This separation ensures modularity and simplifies the addition of new commands.

2 Decorator Pattern: Permission Handling

To manage user permissions cleanly, the project uses a decorator to check whether the current user has access to perform an action. This avoids repetitive 'if' checks throughout the code.

```
# File: Models/User/user_permissions.py
def has_permission(permission):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if not current_user.has_permission(permission):
                raise PermissionError("Insufficient rights")
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

Instead of doing:

```
if user.can_write:
    do_something()
```

We now just write:

```
@has_permission("write")
def save_file(): ...
```

This improves readability and enforces access control uniformly.

3 Custom Exception Handling

Custom exceptions are defined to cleanly handle specific error types. This improves debugging and isolates concerns related to errors.

```
# File: Core/Errors/exceptions.py
class PermissionError(Exception): pass
class FileTypeNotSupported(Exception): pass
```

These are used across the system to raise meaningful error messages, instead of generic 'Exception'.

Analogy: Rather than saying "something went wrong," custom exceptions are like specific traffic signs—"road closed" or "speed limit exceeded"—providing more clarity.

4 Plugin Architecture (Factory Pattern)

The system supports extensible file types using a registry system that mimics the Factory Pattern. New file types can be introduced with minimal code.

```python
# File: Models/File/file_registry.py
FILE_REGISTRY = {}

def register_filetype(extension):
    def decorator(cls):
        FILE_REGISTRY[extension] = cls
        return cls
    return decorator
```

```python
# File: Models/FileTypes/txt.py
@register_filetype(".txt")
class TxtFile(File):
    def read(self): ...
```

Classic Example: In GUI libraries like Tkinter or Qt, different button or label types are often registered behind the scenes and created dynamically. This matches the Factory Pattern's ability to decouple instantiation from usage.

5 Conclusion

The PyNano project is a strong example of modular object-oriented architecture using real-world design patterns. These patterns not only make the project maintainable and extensible but also offer excellent teaching opportunities for students to learn clean software design.

The current implementation of PyNano is modular and extendable. However, incorporating more design patterns could improve maintainability, performance, and feature capabilities. Below are five recommended patterns that could be added to enhance the system further.

5.1 Singleton Pattern

Use Case: `SessionManager` or a centralized `CommandRegistry`.

Ensures only one instance of critical classes (e.g., session manager) is created during runtime.

```python
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class SessionManager(metaclass=SingletonMeta):
    pass
```

Metaphor: Think of a country with only one president. No matter how many elections occur, there should be only one active leader at a time.

5.2 State Pattern

Use Case: Manage different user session states (e.g., logged in, editing, idle).

Encapsulates states as classes so that behavior changes depending on the current state.

```python
class SessionState:
    def handle(self): pass

class LoggedOutState(SessionState):
    def handle(self): print("User not logged in")

class EditingState(SessionState):
    def handle(self): print("User editing a file")
```

Metaphor: A vending machine behaves differently when it's out of order, in standby, or dispensing. Each state defines what the machine does when a button is pressed.

5.3 Strategy Pattern

Use Case: Implement different search or syntax highlighting algorithms.

Allows selecting the algorithm dynamically at runtime.

```python
class SearchStrategy:
    def search(self, text, query): raise NotImplementedError

class NaiveSearch(SearchStrategy):
    def search(self, text, query): return query in text

class RegexSearch(SearchStrategy):
    def search(self, text, query):
        import re
        return re.search(query, text) is not None
```

Real-World Example: Google Maps lets you choose the "fastest," "shortest," or "traffic-avoiding" route—each a different strategy for solving the same problem.

5.4 Memento Pattern

Use Case: Advanced undo/redo functionality.

Captures and restores an object's state without violating encapsulation.

```python
class Memento:
    def __init__(self, state): self.state = state

class Editor:
    def __init__(self): self._state = ""
    def save(self): return Memento(self._state)
    def restore(self, memento): self._state = memento.state
```

Metaphor: Like a time machine that lets you jump back to a previous version of reality, the memento pattern lets you jump back to earlier program states.

5.5 Observer Pattern

Use Case: Notify components (like UI or logging) when the editor content changes.

Defines a subscription mechanism to notify multiple objects about state changes.

```python
class Observable:
    def __init__(self): self._observers = []
    def register(self, observer): self._observers.append(observer)
    def notify(self):
        for observer in self._observers:
            observer.update()

class Editor(Observable):
    def change_content(self):
        # ... some change logic
        self.notify()

class Logger:
    def update(self): print("Editor changed")
```

Real-World Analogy: Like a YouTube notification system—subscribers get notified every time a channel uploads something new.

These additions provide extensibility and better abstraction in larger-scale collaborative development settings.

Final Thoughts

While design patterns are powerful tools that promote clean, modular, and maintainable code, it's important to apply them judiciously. Overusing design patterns—especially in simple or early-stage projects—can lead to unnecessary complexity, reduced readability, and maintenance overhead. This phenomenon is sometimes referred to as "design pattern overengineering."

Just because a pattern exists doesn't mean it should be used. A good developer understands not only how a pattern works, but also when to avoid it. Patterns are best used to solve recurring problems in a standardized way, not to show off technical knowledge.

According to the seminal work *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides (collectively known as the "Gang of Four"), there are 23 classic design patterns. These are divided into three categories:

- **Creational Patterns:** Deal with object creation (e.g., Singleton, Factory, Builder)
- **Structural Patterns:** Deal with object composition (e.g., Decorator, Adapter, Composite)
- **Behavioral Patterns:** Deal with communication between objects (e.g., Command, Observer, Strategy, State)

In real-world development, experienced engineers tend to use a subset of these patterns frequently—like Singleton, Strategy, Observer, and Decorator—while others may appear only in specific contexts.

As students continue to learn and grow, they should focus not only on mastering design patterns but also on recognizing when simplicity and clarity are more appropriate than abstraction and indirection.

Reference: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.