DESIGN PATTERNS - CHAPITRE 28

Combiner des patrons dans un cas concret

Combiner les collaborations et préciser la validité des données partagées.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Définir les données transmises aux observateurs.
- Tester le nombre d'appels d'un service enveloppé par un cache.

Prérequis. Avoir étudié les patrons de création, de structure et de comportement.

Combiner des patrons dans un cas concret

Observer une configuration

Le sujet transmet un événement contenant la nouvelle configuration à deux observateurs. Les affichages simulent journal et alerte. Les données sont volontairement non sensibles ; transmettre un dictionnaire mutable partagé permettrait à un observateur de modifier les données vues par le suivant. Choisissez une copie ou une représentation immuable si cette isolation est nécessaire.

Python

```python
# example.py
class Event:
    """Represents an event."""
    def __init__(self, data):
        self.data = data

class Observer:
    """Abstract class for observers."""
    def update(self, event: Event):
        """Called when the subject's state changes."""
        raise NotImplementedError("Subclasses must implement update method")

class Subject:
    """Abstract class for subjects."""
    def __init__(self):
        self._observers = []

    def attach(self, observer: Observer):
        """Attaches an observer to the subject."""
        self._observers.append(observer)

    def detach(self, observer: Observer):
        """Detaches an observer from the subject."""
        self._observers.remove(observer)

    def notify(self, event: Event):
        """Notifies all observers about the event."""
        for observer in self._observers:
            observer.update(event)

class ConfigurationSubject(Subject):
    """Concrete subject that manages configuration changes."""
    def __init__(self):
        super().__init__()
        self._config = {}

    def set_config(self, config: dict):
```

```
        """Sets the configuration and notifies observers."""
        self._config = config
        self.notify(Event(config))

class LoggingObserver(Observer):
    """Concrete observer that logs configuration changes."""
    def update(self, event: Event):
        """Logs the configuration data."""
        print(f"Logging configuration change: {event.data}")

class AlertingObserver(Observer):
    """Concrete observer that sends alerts on configuration changes."""
    def update(self, event: Event):
        """Sends an alert about the configuration change."""
        print(f"Sending alert: Configuration changed to {event.data}")

# Example usage
config_subject = ConfigurationSubject()
logging_observer = LoggingObserver()
alerting_observer = AlertingObserver()

config_subject.attach(logging_observer)
config_subject.attach(alerting_observer)

config_subject.set_config({"theme": "dark", "timeout": 60})
```

Décorer un service avec un cache

Le service de base retourne toujours la même chaîne. L'enveloppe de journalisation transforme le résultat ; elle n'écrit aucun journal externe. Le cache mémorise ce résultat au premier appel. Comparez les deux appels : le second doit éviter d'appeler à nouveau le service enveloppé.

Ce cache convient uniquement à une opération sans arguments dont le résultat reste valide. Il est non borné dans le temps, sans invalidation ni synchronisation ; changer la donnée source exigerait une autre politique.

Python

```python
# example.py
from abc import ABC, abstractmethod

class BaseService(ABC):
    """Abstract base class for services."""
    @abstractmethod
    def execute(self):
        """Executes the service."""
        pass

class ConcreteService(BaseService):
    """Concrete implementation of the base service."""
    def execute(self):
        """Executes the core service functionality."""
        return "Core Service"

class ServiceDecorator(BaseService):
    """Abstract decorator class."""
    def __init__(self, service: BaseService):
        self._service = service

    def execute(self):
        """Delegates execution to the decorated service."""
        return self._service.execute()

class LoggingDecorator(ServiceDecorator):
    """Concrete decorator that adds logging functionality."""
    def execute(self):
        """Logs the execution and then executes the service."""
        result = super().execute()
        return f"Logged: {result}"

class CachingDecorator(ServiceDecorator):
    """Concrete decorator that adds caching functionality."""
    def __init__(self, service: BaseService):
        super().__init__(service)
        self._cache = None
```

```
    def execute(self):
        """Checks the cache before executing the service."""
        if self._cache is None:
            self._cache = super().execute()
        return f"Cached: {self._cache}"

# Example usage
service = ConcreteService()
logged_service = LoggingDecorator(service)
cached_logged_service = CachingDecorator(logged_service)

print(service.execute())
print(logged_service.execute())
print(cached_logged_service.execute()) # First execution - caches the result
print(cached_logged_service.execute()) # Second execution - retrieves from
cache
```

Version web : https://www.clearcode.fr/courses/programming/design-patterns/cas-pratiques