DESIGN PATTERNS - CHAPITRE 27

Refactoriser sans changer le comportement

Simplifier les dépendances sans perdre les cas limites du comportement initial.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Expliquer les arguments mutables et les erreurs masquées.
- Comparer les traitements avant et après refactorisation sur plusieurs tailles d'entrée.

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

Refactoriser sans changer le comportement

Argument mutable : constater l'état partagé

Un argument par défaut est évalué lors de la définition de la fonction. Cette liste est donc réutilisée, pas recréée à chaque appel. Exécutez ce contre-exemple et comparez l'identité de `list1` et `list2` : les anciennes références voient aussi les ajouts suivants.

Python

```python
# example.py
def append_to_list(item, my_list=[]):
    """
    Appends an item to a list.

    Args:
        item: The item to append.
        my_list: The list to append to (defaults to an empty list).
    """
    my_list.append(item)
    return my_list

# Example usage demonstrating the anti-pattern
list1 = append_to_list(1)
print(list1)  # Output: [1]

list2 = append_to_list(2)
print(list2)  # Output: [1, 2] - Not what we expected!  The list is shared
between calls.

list3 = append_to_list(3, [])
print(list3)  # Output: [3] - Correct, because a new list was passed.

list4 = append_to_list(4)
print(list4)  # Output: [1, 2, 4] - Still affected by previous calls!
```

La valeur `None` indique ici qu'aucune liste n'a été fournie. On crée alors une liste neuve. Si une liste est explicitement transmise, la fonction continue à la modifier : ce comportement fait partie de son contrat.

Python

```python
# example.py
def append_to_list_fixed(item, my_list=None):
    """
    Appends an item to a list (fixed version).

    Args:
        item: The item to append.
        my_list: The list to append to (defaults to None).
    """
    if my_list is None:
        my_list = []  # Create a new list if my_list is None
    my_list.append(item)
    return my_list

# Corrected Example usage
list1 = append_to_list_fixed(1)
print(list1) # Output: [1]

list2 = append_to_list_fixed(2)
print(list2) # Output: [2]

list3 = append_to_list_fixed(3, [])
print(list3) # Output: [3]

list4 = append_to_list_fixed(4)
print(list4) # Output: [4]
```

Exceptions : traiter seulement le cas attendu

Ce contre-exemple utilise `except:`, qui intercepte aussi les interruptions du programme. Il transforme toutes les erreurs en absence de résultat et peut masquer un défaut de programmation.

Python

```python
# example.py
def divide(x, y):
    """
    Divides two numbers.
    """
    try:
        result = x / y
    except:  # Avoid this! Catches all exceptions - BAD PRACTICE
        print("An error occurred!")
        return None
    return result
```

La correction choisit un contrat explicite : division par zéro donne `None`, les autres erreurs se propagent. D'autres choix sont possibles, dont laisser aussi `ZeroDivisionError` se propager ; l'appelant doit connaître celui retenu.

Python

```python
# example.py
def divide_fixed(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        return None

print(divide_fixed(6, 2))  # 3.0
print(divide_fixed(6, 0))  # None
# TypeError is deliberately allowed to propagate for invalid input types.
```

État global : repérer une dépendance cachée

Le résultat de cet appel dépend de l'historique du module. Un état global mutable n'est pas toujours interdit, mais son propriétaire et son cycle de vie doivent être explicites. Deux tests qui partagent ce module partageront aussi ce compteur.

Python

```python
# example.py
global_counter = 0  # Avoid this!

def increment_counter():
    """
    Increments a global counter.
    """
    global global_counter
    global_counter += 1

increment_counter()
print(global_counter)
```

Un objet permet ici de créer deux compteurs indépendants et de choisir lequel transmettre au code appelant. Une fonction qui reçoit et retourne un nombre serait encore plus simple si l'on n'a pas besoin d'un objet mutable.

Python

```python
# example.py
class Counter:
    """
    A simple counter class.
    """
    def __init__(self):
        self.count = 0

    def increment(self):
        """
        Increments the counter.
        """
        self.count += 1

# Example usage
my_counter = Counter()
my_counter.increment()
print(my_counter.count)
```

Fonctions ou classe : choisir selon le besoin

Les fonctions d'aire et de périmètre sont déjà cohésives et réutilisables. La classe est une alternative pour regrouper des dimensions avec leurs opérations ; le simple nombre de fonctions ne constitue pas un défaut. Dans les deux versions, les dimensions sont supposées non négatives et exprimées dans la même unité.

Python

```python
# example.py
def calculate_area_rectangle(length, width):
    """
    Calculates the area of a rectangle.
    """
    return length * width

def calculate_perimeter_rectangle(length, width):
    """
    Calculates the perimeter of a rectangle.
    """
    return 2 * (length + width)

# Alternative: an object groups dimensions and operations; functions remain valid.

class Rectangle:
    """
    Represents a rectangle.
    """
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        """
        Calculates the area of the rectangle.
        """
        return self.length * self.width

    def calculate_perimeter(self):
        """
        Calculates the perimeter of the rectangle.
        """
        return 2 * (self.length + self.width)
```

God Class : séparer les raisons de changer

Une classe devient difficile à maintenir lorsqu'elle concentre des responsabilités qui évoluent pour des raisons différentes. Ici validation, paiement et inventaire sont réunis. L'exemple simule ces opérations ; vérifier la présence de trois clés n'est pas une validation métier complète.

Python

```python
# example.py
class OrderProcessor:
    def __init__(self, order_data):
        self.order_data = order_data

    def validate_order(self):
        # Order validation logic
        if not all(key in self.order_data for key in ['customer_id', 'items',
'shipping_address']):
            raise ValueError("Invalid order data: Missing required fields.")
        # Additional complex validation rules...
        return True

    def process_payment(self):
        # Payment processing logic
        # Connect to payment gateway, handle transactions, etc.
        print("Payment processed successfully.")
        return True

    def update_inventory(self):
        # Inventory management logic
        # Update stock levels based on the order
        print("Inventory updated.")
        return True

    def process_order(self):
        if self.validate_order():
            if self.process_payment():
                self.update_inventory()
                print("Order processed successfully!")
            else:
                print("Payment failed.")
        else:
            print("Order validation failed.")

# Example usage
order_data = {'customer_id': 123, 'items': ['item1', 'item2'],
```

```
    'shipping_address': '123 Main St'}
processor = OrderProcessor(order_data)
processor.process_order()
```

La version séparée reçoit explicitement les trois collaborateurs. On peut alors remplacer un service de paiement pour observer si l'inventaire est mis à jour uniquement après succès. Cette séparation devient utile si ces responsabilités varient indépendamment ; un petit programme n'a pas systématiquement besoin de cinq classes.

Une application réelle doit aussi traiter les échecs intermédiaires et les transactions. Ce code n'assure ni atomicité ni compensation d'un paiement déjà effectué.

Python

```python
# example.py
class Order:
    def __init__(self, order_data):
        self.order_data = order_data

class OrderValidator:
    def validate_order(self, order: Order):
        if not all(key in order.order_data for key in ['customer_id', 'items',
'shipping_address']):
            raise ValueError("Invalid order data: Missing required fields.")
        return True

class PaymentService:
    def process_payment(self, order: Order):
        # Payment processing logic
        print("Payment processed successfully.")
        return True

class InventoryService:
    def update_inventory(self, order: Order):
        # Inventory management logic
        print("Inventory updated.")
        return True

class OrderService:
    def __init__(self, validator: OrderValidator, payment_service:
PaymentService, inventory_service: InventoryService):
        self.validator = validator
        self.payment_service = payment_service
        self.inventory_service = inventory_service

    def process_order(self, order: Order):
        if self.validator.validate_order(order):
            if self.payment_service.process_payment(order):
                self.inventory_service.update_inventory(order)
                print("Order processed successfully!")
            else:
```

```
                print("Payment failed.")
        else:
            print("Order validation failed.")

# Example Usage
order_data = {'customer_id': 123, 'items': ['item1', 'item2'],
'shipping_address': '123 Main St'}
order = Order(order_data)
validator = OrderValidator()
payment_service = PaymentService()
inventory_service = InventoryService()
order_service = OrderService(validator, payment_service, inventory_service)

order_service.process_order(order)
```

Refactoriser : conserver toutes les branches

Cette fonction applique une politique selon `flag`, puis ajoute un à chaque résultat si la liste contient plus de cinq éléments. Ce post-traitement fait partie du comportement à préserver, même si les premiers exemples utilisent seulement cinq valeurs.

Python

```python
# example.py
def process_data(data, flag):
    result = []
    if flag == 1:
        for item in data:
            if item > 0:
                result.append(item * 2)
            else:
                result.append(0)
    elif flag == 2:
        for item in data:
            if item < 0:
                result.append(abs(item))
            else:
                result.append(item)
    else:
        for item in data:
            result.append(item)

    if len(result) > 5:
        final_result = [x + 1 for x in result]
    else:
        final_result = result

    return final_result
```

Les stratégies isolent le premier traitement et le contexte conserve le post-traitement commun. Comparez les deux versions avec cinq puis six valeurs et les indicateurs `1`, `2` et un autre entier. Les résultats doivent être identiques, et la liste d'entrée doit rester intacte.

Python

```python
# example.py
class DataProcessor:
    def __init__(self, strategy):
        self.strategy = strategy

    def process(self, data):
        result = self.strategy.process(data)
        return [value + 1 for value in result] if len(result) > 5 else result

class Strategy:
    def process(self, data):
        raise NotImplementedError("Subclasses must implement this method")

class Strategy1(Strategy):
    def process(self, data):
        return [item * 2 if item > 0 else 0 for item in data]

class Strategy2(Strategy):
    def process(self, data):
        return [abs(item) if item < 0 else item for item in data]

class StrategyDefault(Strategy):
    def process(self, data):
        return data[:]  # Return a copy of the data

# Example usage
data = [1, -2, 3, -4, 5]

# Using Strategy 1
processor = DataProcessor(Strategy1())
result1 = processor.process(data)
print(f"Result with Strategy 1: {result1}")

# Using Strategy 2
processor = DataProcessor(Strategy2())
result2 = processor.process(data)
print(f"Result with Strategy 2: {result2}")
```

```
# Using Default Strategy
processor = DataProcessor(StrategyDefault())
result_default = processor.process(data)
print(f"Result with Default Strategy: {result_default}")
```

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