DESIGN PATTERNS - CHAPITRE 25

Approfondissement : création des classes

Observer ce qui se passe à la définition d'une classe et à la création d'une instance.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Distinguer création de classe et création d'instance.
- Évaluer les limites des registres automatiques et des instances mémorisées.

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

Approfondissement : création des classes

Pourquoi agir au moment où une classe est définie ? Pour vérifier une déclaration ou enregistrer automatiquement un type. Une métaclasse est généralement une sous-classe de `type`. Son `__new__` reçoit le nom, les bases et l'espace de noms de la future classe, puis délègue sa création à `type`.

Ce chapitre est un approfondissement facultatif. Un dictionnaire explicite, un décorateur de classe ou `__init_subclass__` est souvent plus facile à lire. Ici, le validateur exige un attribut `name` de type chaîne dans chaque définition ; un attribut seulement hérité ne satisfait pas cette règle.

Python

```python
# example.py
class AttributeValidator(type):
    def __new__(mcs, name, bases, attrs):
        # Check if the required attribute exists and has the correct type
        required_attribute = "name" # defining the required attribute
        attribute_type = str # defining the attribute type

        if required_attribute not in attrs:
            raise ValueError(f"Class {name} must define attribute
'{required_attribute}'")

        if not isinstance(attrs[required_attribute], attribute_type):
            raise TypeError(f"Attribute '{required_attribute}' in class {name}
must be of type {attribute_type}")

        return super().__new__(mcs, name, bases, attrs)

class ValidatedClass(metaclass=AttributeValidator):
    name = "Example" # attribute validated by the metaclass

# This class will raise an error because 'name' is missing
# class InvalidClass(metaclass=AttributeValidator):
#    pass
```

Enregistrer une classe lors de sa définition

Le registre associe un nom public à une classe de plugin. La définition de `ImagePlugin` déclenche son enregistrement, sans créer d'instance. Cette variante accepte un nom hérité et remplace silencieusement un doublon : pour une extension réelle, décidez si ces deux comportements sont souhaitables.

Python

```python
# example.py
class PluginRegistry(type):
    def __init__(cls, name, bases, attrs):
        super().__init__(name, bases, attrs)
        if hasattr(cls, 'plugin_name'):
            PluginRegistry.register_plugin(cls.plugin_name, cls)

    _plugins = {}

    @classmethod
    def register_plugin(cls, name, plugin_class):
        cls._plugins[name] = plugin_class

    @classmethod
    def get_plugin(cls, name):
        return cls._plugins.get(name)

class BasePlugin(metaclass=PluginRegistry):
    pass

class ImagePlugin(BasePlugin):
    plugin_name = "image"

class TextPlugin(BasePlugin):
    plugin_name = "text"

# Accessing the registered plugins
image_plugin = PluginRegistry.get_plugin("image")
text_plugin = PluginRegistry.get_plugin("text")

print(PluginRegistry._plugins)  # Output: {'image': , 'text': }
```

Intercepter la création d'une instance

`SingletonMeta.__call__` est appelé quand la classe utilisatrice est appelée. Le dictionnaire est indexé par classe : chaque classe possède son instance mémorisée. Les arguments des appels suivants sont ignorés. Contrairement à un simple `__new__` qui retourne un objet existant, cette version n'appelle pas à nouveau `__init__`.

Aucun verrou ne protège la création simultanée et rien ne partage l'instance entre processus. Une métaclasse ajoute une contrainte de composition avec d'autres métaclasses : elle ne rend pas ce choix automatiquement plus clair ou plus sûr.

Python

```python
# example.py
class SingletonMeta(type):
    """
    A metaclass that ensures only one instance of a class exists.
    """
    _instances = {}  # Dictionary to store class instances

    def __call__(cls, *args, **kwargs):
        """
        Override the __call__ method to control instance creation.
        """
        if cls not in cls._instances:
            # If the class is not yet in the _instances dictionary, create a
new instance
            cls._instances[cls] = super().__call__(*args, **kwargs)
        # Return the instance stored in the dictionary
        return cls._instances[cls]


class Configuration(metaclass=SingletonMeta):
    """
    A sample class that uses the SingletonMeta metaclass.
    """
    def __init__(self, setting1="default_value", setting2=100):
        self.setting1 = setting1
        self.setting2 = setting2

    def display_settings(self):
        """
        Displays the current settings.
        """
        print(f"Setting 1: {self.setting1}, Setting 2: {self.setting2}")


# Example usage
config1 = Configuration(setting1="new_value", setting2=200)
config1.display_settings()
```

```
config2 = Configuration()
config2.display_settings()

# Verify that both variables point to the same object
print(config1 is config2)
```

Registre de constructeurs

Le premier type, `PaymentMethod`, reçoit un dictionnaire ; ses sous-classes y sont ajoutées sous leur nom. Les méthodes affichent des simulations. Ce registre ne distingue pas une sous-classe abstraite d'une classe instanciable et remplace les doublons : ces politiques devraient être définies pour un véritable système de plugins.

Python

```python
# example.py
class PaymentRegistry(type):
    """
    Metaclass that automatically registers payment methods.
    """
    def __init__(cls, name, bases, attrs):
        """
        Initializes the class. Called when a class is created using this metaclass.
        """
        super().__init__(name, bases, attrs)
        if not hasattr(cls, 'registry'):
            cls.registry = {}  # Initialize the registry on the base class
        else:
            cls.registry[name] = cls  # Register the concrete class

class PaymentMethod(metaclass=PaymentRegistry):
    """
    Base class for payment methods. All subclasses will be automatically registered.
    """
    pass

class CreditCardPayment(PaymentMethod):
    """
    Payment method using credit card.
    """
    def process_payment(self, amount, card_number, expiry_date, cvv):
        print(f"Processing credit card payment of {amount} using card {card_number}")

class PayPalPayment(PaymentMethod):
    """
    Payment method using PayPal.
    """
    def process_payment(self, amount, paypal_email):
```

```
        print(f"Processing PayPal payment of {amount} using email
{paypal_email}")
```

Suite du registre précédent : la fabrique recherche une classe puis la construit. Le nom inconnu doit lever `ValueError`. Le registre décide des classes disponibles ; la fabrique décide comment demander une instance.

Python

```python
# example.py
def payment_factory(payment_method_name):
    """
    Factory function to create payment method instances.
    """
    payment_class = PaymentMethod.registry.get(payment_method_name)
    if not payment_class:
        raise ValueError(f"Payment method '{payment_method_name}' not found")
    return payment_class()

# Example usage:
try:
    payment = payment_factory('CreditCardPayment')
    payment.process_payment(100, "1234-5678-9012-3456", "12/24", "123")
except ValueError as e:
    print(e)

try:
    payment = payment_factory('PayPalPayment')
    payment.process_payment(50, "test@paypal.com")
except ValueError as e:
    print(e)

try:
    payment = payment_factory('UnknownPayment')
    payment.process_payment(50, "test@paypal.com")
except ValueError as e:
    print(e)
```

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