DESIGN PATTERNS - CHAPITRE 7

Adapter en Python

Traduire une interface existante vers les appels et les données attendus par un client.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Traduire une interface existante vers les appels et les données attendus par un client.
- Vérifier que la conversion conserve le sens du contrat.

Prérequis. Savoir utiliser classes, héritage, composition, listes, dictionnaires et exceptions Python. Les mécanismes spécifiques sont expliqués avant leur emploi.

Adapter : intention et expérience

Adapter : traduire une interface

Un **Adapter** reçoit un objet existant dont les opérations ne conviennent pas au client. Il traduit l'appel attendu vers l'appel disponible. Ici, `request()` délègue à `specific_request()` et adapte son résultat. Il faut préserver le sens des données, pas seulement renommer une méthode.

Python

```python
# example.py
# Target interface
class Target:
    def request(self):
        return "Target: The default target's behavior."

# Adaptee class
class Adaptee:
    def specific_request(self):
        return ".eetpadA eht fo roivaheb laicepseht"

# Adapter class
class Adapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return f"Adapter: (TRANSLATED) {self.adaptee.specific_request()[::-1]}"

# Client code
def client_code(target):
    print(target.request())

if __name__ == "__main__":
    adaptee = Adaptee()
    adapter = Adapter(adaptee)
    client_code(adapter)
```

Adapter : rendre la conversion explicite

Le nouveau format fournit `get_data()`, l'ancien processeur attend `process_image(data)`. L'adaptateur fait le lien. Les conversions sont volontairement simulées par des chaînes ; un vrai format d'image impose de définir et vérifier son encodage.

Python

```python
# example.py
# Assume this is an external library with an incompatible interface
class LegacyImageProcessor:
    def __init__(self):
        pass

    def process_image(self, image_data):
        # Legacy image processing logic
        # Expects data in a specific old format
        print(f"Legacy Image Processor: Processing image data in legacy format")
        return "Processed Image (Legacy)"

# New system that expects a different interface
class NewImageFormat:
    def __init__(self, data):
        self.data = data

    def get_data(self):
        return self.data

# Adapter class to make LegacyImageProcessor compatible with NewImageFormat
class ImageAdapter:
    def __init__(self, image_format, processor):
        self.image_format = image_format
        self.processor = processor

    def process(self):
        # Convert the new image format to the legacy format expected by the processor
        legacy_data = self.convert_to_legacy_format(self.image_format.get_data())
        return self.processor.process_image(legacy_data)

    def convert_to_legacy_format(self, new_data):
        # This is where the adaptation logic resides
        # Convert new_data to the format expected by LegacyImageProcessor
```

```
        print("Adapter: Converting new image format to legacy format")
        return f"Legacy Format: {new_data}"

# Usage
legacy_processor = LegacyImageProcessor()
new_image = NewImageFormat("New Image Data")
adapter = ImageAdapter(new_image, legacy_processor)
result = adapter.process()
print(f"Result: {result}")
```

Une fonction peut suffire pour adapter un appel isolé. Ajouter une classe serait surtout utile pour conserver un collaborateur ou une configuration entre plusieurs appels.

Python

```
# example.py
def legacy_function(data):
    # Legacy function that expects a certain format
    print(f"Legacy Function: Processing data in legacy format: {data}")
    return f"Legacy processed: {data}"

def adapter_function(new_data):
    # Adapter function to convert new data to the format expected by
legacy_function
    adapted_data = f"Adapted: {new_data}"
    print(f"Adapter Function: Converting new data to legacy format")
    return legacy_function(adapted_data)

# Usage
new_data = "New Data Format"
result = adapter_function(new_data)
print(f"Result: {result}")
```

Vérifier et interpréter

Remplacez le processeur par un faux objet qui enregistre les données reçues. Vérifiez la traduction produite par l'adaptateur, pas seulement le texte affiché. L'adaptateur localise la compatibilité, mais ne peut pas recréer une information absente ni rendre équivalentes deux opérations de sens différent.

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