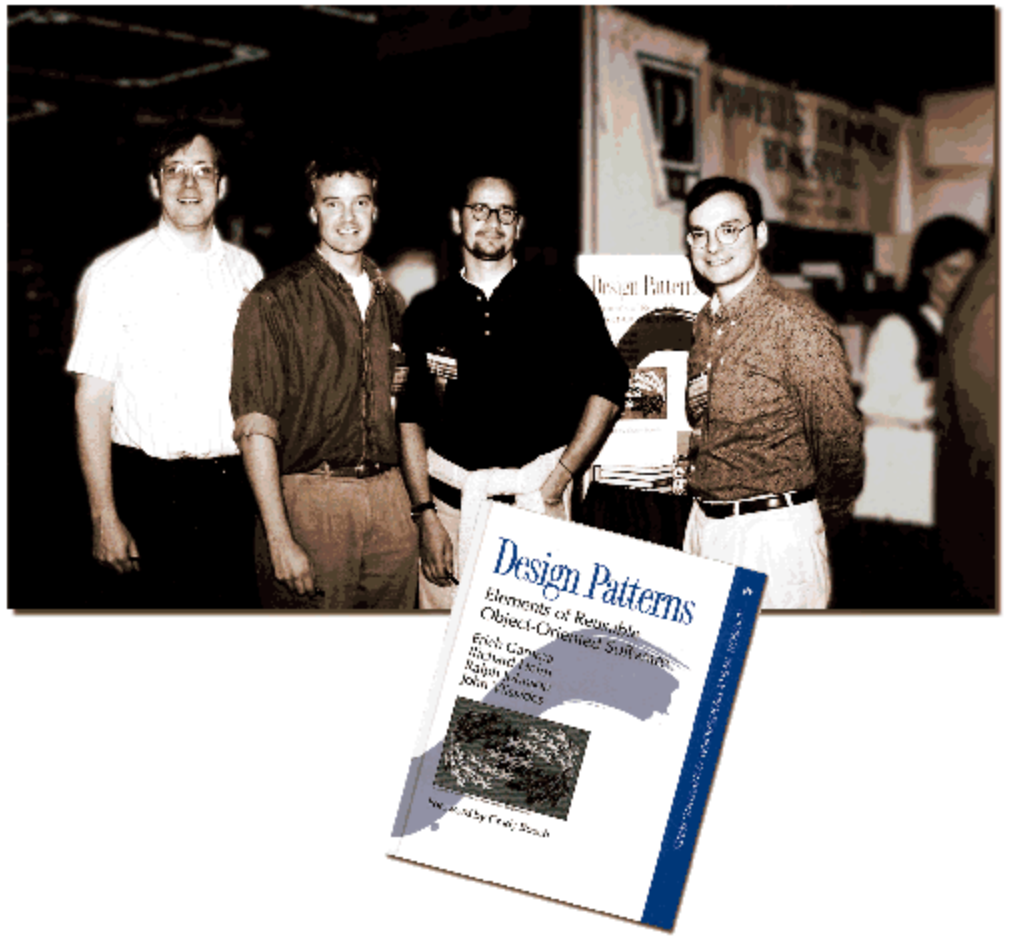
Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

Wykład 8: Wzorce projektowe



Geneza wzorców projektowych

Wzorce projektowe w informatyce wywodzą się z wzorców projektowych w architekturze (*most*, *fasada*, *budowniczy*), które zostały zaproponowane przez austriackiego architekta **Christophera Alexandra** i miały ułatwić konstruowanie mieszkań i pomieszczeń biurowych. Pomysł ten nie został jednak przyjęty.

Wzorce projektowe w informatyce

Termin wzorca projektowego został wprowadzony do inżynierii oprogramowania przez **Kenta Becka** oraz **Warda Cunninghama** w 1987 roku. Na konferencji **OOPSLA**, przedstawili oni wyniki swojego eksperymentu dotyczącego ich zastosowania w programowaniu. Został spopularyzowany w 1995 przez **Gang of Four** (Erich Gamma, Richard Helm, Ralph Johnson oraz John Vlissides) dzięki książce *Wzorce projektowe: Elementy oprogramowania obiektowego wielokrotnego użytku*.

Definicja

Wzorzec projektowy (*ang. design pattern*) – uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych.

Co nam daje stosowanie wzorców projektowych?

Pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz utrzymanie kodu źródłowego. Wzorce projektowe stosowane są w projektach wykorzystujących programowanie obiektowe.

Podział wzorców

- **Kreacyjne**: Opisują elastyczne sposoby tworzenia obiektów, uniezależniają system od sposobu tworzenia obiektu.
- **Strukturalne**: Opisują sposób konstrukcji struktur obiektowych, korzystają z dziedziczenia i delegacji.
- **Behawioralne**: Opisują algorytmy i przydział odpowiedzialności, charakteryzują sposób interakcji pomiędzy obiektami.

Wzorce kreacyjne

- **Abstract factory** (tworzy rodziny powiązanych obiektów bez konkretnych klas),
- Builder (buduje złożone obiekty krok po kroku),
- **Factory method** (pozwala podklasom wybrać klasę do instancjonowania),
- **Prototype** (tworzy obiekty przez kopiowanie prototypu),
- Singleton (zapewnia pojedynczą instancję klasy).

Wzorce strukturalne

- **Adapter** (konwertuje interfejs klasy na inny, oczekiwany przez klienta),
- Bridge (oddziela abstrakcję od implementacji, umożliwiając ich niezależny rozwój),
- Composite (traktuje obiekty pojedyncze i złożone w jednolity sposób),
- **Decorator** (dynamicznie dodaje nowe zachowania do obiektów),
- Facade (dostarcza uproszczony interfejs do złożonego systemu),
- Flyweight (minimalizuje zużycie pamięci przez współdzielenie obiektów),
- **Proxy** (dostarcza zastępczy obiekt, który kontroluje dostęp do innego obiektu).

Wzorce behawioralne

- Chain of responsibility (przekazuje żądanie przez łańcuch),
- **Command** (hermetyzuje żądanie jako obiekt),
- Interpreter (interpretuje język),
- Iterator (dostęp sekwencyjny do elementów),
- Mediator (hermetyzuje interakcje obiektów),
- Memento (przywraca stan obiektu).

Wzorce behawioralne

- **Observer** (powiadamia obiekty o zmianach stanu),
- State (zmienia zachowanie obiektu w zależności od stanu),
- **Strategy** (definiuje wymienne algorytmy),
- Template method (definiuje szkielet algorytmu),
- Visitor (oddziela algorytm od struktury obiektu).

Fabryka abstrakcyjna (1/5)

Jest kreacyjnym wzorcem projektowym, który pozwala tworzyć rodziny spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.

```
from abc import ABC, abstractmethod

# Abstract Products
class Button(ABC):
    @abstractmethod
    def paint(self):
        pass

class Checkbox(ABC):
    @abstractmethod
    def paint(self):
        pass
```

Fabryka abstrakcyjna (2/5)

```
# Concrete Products
class WindowsButton(Button):
    def paint(self):
        return "Windows Button"

class WindowsCheckbox(Checkbox):
    def paint(self):
        return "Windows Checkbox"

class MacButton(Button):
    def paint(self):
        return "Mac Button"

class MacCheckbox(Checkbox):
    def paint(self):
        return "Mac Checkbox"
```

Fabryka abstrakcyjna (3/5)

```
# Abstract Factory (*)
class GUIFactory(ABC):
    @abstractmethod
    def create_button(self):
        pass

    @abstractmethod
    def create_checkbox(self):
        pass
```

Fabryka abstrakcyjna (4/5)

```
# Concrete Factories
class WindowsFactory(GUIFactory):
    def create_button(self):
        return WindowsButton()

    def create_checkbox(self):
        return WindowsCheckbox()

class MacFactory(GUIFactory):
    def create_button(self):
        return MacButton()

    def create_checkbox(self):
        return MacCheckbox()
```

Fabryka abstrakcyjna (5/5)

```
# Client
def create_ui(factory: GUIFactory):
    button = factory.create_button()
    checkbox = factory.create_checkbox()
    return button.paint(), checkbox.paint()

# Usage
windows_ui = create_ui(WindowsFactory())
mac_ui = create_ui(MacFactory())

print(windows_ui)
print(mac_ui)
```

```
('Windows Button', 'Windows Checkbox')
('Mac Button', 'Mac Checkbox')
```

Metoda wytwórcza (1/4)

Kreacyjny wzorzec projektowy, którego celem jest dostarczenie interfejsu do tworzenia obiektów nieokreślonych jako powiązanych typów.

```
from abc import ABC, abstractmethod

# Product interface
class Document(ABC):
    @abstractmethod
    def open(self):
        pass
```

Metoda wytwórcza (2/4)

```
# Concrete Products
class PDFDocument(Document):
    def open(self):
        return "Opening PDF document"

class WordDocument(Document):
    def open(self):
        return "Opening Word document"
```

Metoda wytwórcza (3/4)

```
# Creator interface
class DocumentCreator(ABC):
    @abstractmethod
    def create_document(self):
        pass # (*)

    def open_document(self):
        document = self.create_document()
        return document.open()

# Concrete Creators
class PDFDocumentCreator(DocumentCreator):
    def create_document(self):
        return PDFDocument()

class WordDocumentCreator(DocumentCreator):
    def create_document(self):
        return WordDocument()
```

Metoda wytwórcza (4/4)

```
# Client
def open_my_document(creator: DocumentCreator):
    return creator.open_document()

# Usage
pdf_creator = PDFDocumentCreator()
word_creator = WordDocumentCreator()

print(open_my_document(pdf_creator))
print(open_my_document(word_creator))
```

```
Opening PDF document
Opening Word document
```

Prototyp (1/3)

Kreacyjny wzorzec projektowy, którego celem jest umożliwienie tworzenia obiektów danej klasy bądź klas z wykorzystaniem już istniejącego obiektu, zwanego prototypem.

```
from abc import ABC, abstractmethod
import copy

class Prototype(ABC):
    @abstractmethod
    def clone(self):
        pass
```

Prototyp (2/3)

```
class StudentPrototype(Prototype):
    def __init__(self, name, points):
        self.name = name
        self.points = points

    def clone(self):
        return copy.deepcopy(self)

    def __str__(self):
        return f"Student(name={self.name}, points={self.points})"
```

Prototyp (3/3)

```
empty = StudentPrototype("", 0)
mike = empty.clone()
mike.name = "Mike"
mike.points = 32

sue = empty.clone()
sue.name = "Sue"
sue.points = 40

print(empty)
print(mike)
print(sue)
```

```
Student(name=, points=0)
Student(name=Mike, points=32)
Student(name=Sue, points=40)
```

Adapter (1/3)

Strukturalny wzorzec projektowy pozwalający na współpracę niekompatybilnych obiektów ze sobą.

```
class OldPrinter:
    def print_text(self, text):
        return f"Old Printer: {text}"

class NewPrinter:
    def print_document(self, document):
        return f"New Printer: {document.content}"

class Document:
    def __init__(self, content):
        self.content = content
```

Adapter (2/3)

```
class PrinterAdapter(NewPrinter): # (*)
    def __init__(self, old_printer):
        self.old_printer = old_printer

    def print_document(self, document):
        adapted_text = self.old_printer.print_text(document.content)
        return f"Adapter: {adapted_text}"
```

Adapter (3/3)

```
# Usage
old_printer = OldPrinter()
new_printer = NewPrinter()
adapter = PrinterAdapter(old_printer)

document = Document("Hello, Adapter!")

print(new_printer.print_document(document))
print(adapter.print_document(document))
```

```
New Printer: Hello, Adapter!
Adapter: Old Printer: Hello, Adapter!
```

Dekorator (1/4)

To strukturalny wzorzec pozwalający na dodawanie obiektom nowych obowiązków w sposób dynamiczny — poprzez **opakowywanie** ich w specjalne obiekty posiadające potrzebną funkcjonalność.

```
from abc import ABC, abstractmethod

# Component
class Text(ABC):
    @abstractmethod
    def render(self):
        pass
```

Dekorator (2/4)

```
# Concrete Component
class SimpleText(Text):
    def __init__(self, content):
        self._content = content

    def render(self):
        return self._content

# Decorator (*)
class TextDecorator(Text):
    def __init__(self, text):
        self._text = text

    def render(self):
        return self._text.render()
```

Dekorator (3/4)

```
# Concrete Decorators
class BoldText(TextDecorator):
    def render(self):
        return "<b>" + super().render() + "</b>"

class ItalicText(TextDecorator):
    def render(self):
        return "<i>" + super().render() + "</i>"
```

Dekorator (4/4)

```
# Usage
simple_text = SimpleText("Hello, Decorator!")
bold_text = BoldText(simple_text)
italic_bold_text = ItalicText(bold_text)

print(simple_text.render())
print(bold_text.render())
print(italic_bold_text.render())
```

```
Hello, Decorator!
<b>Hello, Decorator!</b>
<i><b>Hello, Decorator!</b></i>
```

Proxy (1/4)

Strukturalny wzorzec projektowy, który pełni rolę reprezentanta innego obiektu w celu uzyskania nadzorowanego dostępu do obiektu, który przedstawia.

Proxy (2/4)

```
import time

class LargeFile:
    def __init__(self, filename):
        self.filename = filename
        self._content = None

    def load_content(self):
        print(f"Loading large file: {self.filename}")
        time.sleep(2)  # Symulacja długiego ładowania
        self._content = f"Content of {self.filename}"

    def get_content(self):
        if self._content is None:
            self.load_content()
        return self._content
```

Proxy (3/4)

```
class FileProxy:
    def __init__(self, filename):
        self._filename = filename
        self._real_file = None

    def get_content(self):
        if self._real_file is None:
            self._real_file = LargeFile(self._filename)
        return self._real_file.get_content()
```

Proxy (4/4)

```
# Client code
proxy = FileProxy("very_large_file.txt")

print("First access:")
content1 = proxy.get_content()
print(content1)

print("\nSecond access:")
content2 = proxy.get_content()
print(content2)
```

```
First access:
Loading large file: very_large_file.txt
Content of very_large_file.txt

Second access:
Content of very_large_file.txt
```

Strategia (1/3)

To behawioralny wzorzec projektowy pozwalający zdefiniować rodzinę algorytmów, umieścić je w osobnych klasach i uczynić obiekty tych klas wymienialnymi.

```
from abc import ABC, abstractmethod

# Strategy interface (*)
class CalculationStrategy(ABC):
    @abstractmethod
    def calculate(self, a, b):
        pass
```

Strategia (2/3)

```
# Concrete Strategies
class AdditionStrategy(CalculationStrategy):
    def calculate(self, a, b):
        return a + b

class SubtractionStrategy(CalculationStrategy):
    def calculate(self, a, b):
        return a - b

# Context (*)
class Calculator:
    def __init__(self, strategy: CalculationStrategy):
        self.strategy = strategy

    def perform_calculation(self, a, b):
        return self.strategy.calculate(a, b)
```

Strategia (3/3)

```
# Usage
addition = AdditionStrategy()
subtraction = SubtractionStrategy()

calculator = Calculator(addition)
print(calculator.perform_calculation(5, 3))

calculator = Calculator(subtraction)
print(calculator.perform_calculation(5, 3))
```

```
8
2
```

Polecenie (1/4)

Polecenie (*ang. Command*) to czynnościowy wzorzec projektowy, traktujący żądanie wykonania określonej czynności jako obiekt.

```
class Command:
    def execute(self):
        pass

class Light:
    def turn_on(self):
        print("Light is on")

    def turn_off(self):
        print("Light is off")
```

Polecenie (2/4)

```
class LightOnCommand(Command):
    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_off()
```

Polecenie (3/4)

```
class RemoteControl:
    def __init__(self):
        self._command = None

    def set_command(self, command):
        self._command = command

    def press_button(self):
        if self._command:
            self._command.execute()
```

Polecenie (4/4)

```
# Client code
light = Light()
light_on = LightOnCommand(light)
light_off = LightOffCommand(light)

remote = RemoteControl()

remote.set_command(light_on)
remote.press_button()

remote.set_command(light_off)
remote.press_button()
```

```
Light is on
Light is off
```

Obserwator (1/4)

To czynnościowy (behawioralny) wzorzec projektowy pozwalający zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach dziejących się w obserwowanym obiekcie.

Obserwator (2/4)

```
class Subject:
    def __init__(self):
        self._observers = []
        self._state = None

    def attach_observer(self, observer):
        self._observers.append(observer)

    def detach_observer(self, observer):
        self._observers.remove(observer)

    def set_state(self, state):
        self._state = state
        self.notify_observers()

    def notify_observers(self):
        for observer in self._observers:
            observer.update(self._state)
```

Obserwator (3/4)

```
class Observer: # (*)
    def update(self, state):
        pass

class ConcreteObserverA(Observer):
    def update(self, state):
        print(f"Observer A: Subject state changed to {state}")

class ConcreteObserverB(Observer):
    def update(self, state):
        print(f"Observer B: Subject state changed to {state}")
```

Observator (4/4)

```
# Usage
subject = Subject()

observer_a = ConcreteObserverA()
observer_b = ConcreteObserverB()

subject.attach_observer(observer_a)
subject.attach_observer(observer_b)

subject.set_state("New State")
```

```
Observer A: Subject state changed to New State
Observer B: Subject state changed to New State
```

Model-View-Controller

Wzorzec architektoniczny służący do organizowania struktury aplikacji posiadających graficzne interfejsy użytkownika.

- Model (Reprezentuje dane aplikacji i logikę biznesową. Nie wie nic o View ani Controller).
- View (Odpowiada za wyświetlanie danych użytkownikowi. Nie wie nic o Model ani Controller).
- Controller (Obsługuje interakcje użytkownika i aktualizuje Model i View. Nie zna ich wewnętrznej implementacji.).

MVC oddziela logikę biznesową (Model) od logiki prezentacji (View) i logiki sterowania (Controller).

Model-View-Controller (1/3)

```
# Model
class Model:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

# View
class View:
    def display(self, value):
        print(f"Current value: {value}")
```

Model-View-Controller (2/3)

```
# Controller
class Controller:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def increment_and_display(self):
        self.model.increment()
        self.view.display(self.model.value)
```

Model-View-Controller (3/3)

```
# Client code
model = Model()
view = View()
controller = Controller(model, view)

controller.increment_and_display()
controller.increment_and_display()
```

```
Current value: 1
Current value: 2
```

Koniec