Współczesne techniki wytwarzania oprogramowania

Adam Kobus

Failures

Przyczyny

- nowe funkcjonalności
- ciągłe zmiany
- certyfikacja kodu

Skutki

- spaghetti code
- redundancja
- niespójność
- próg wejścia
- kod legacy
- zacofanie technologiczne
- obstrukcja technologiczna
- finanse...

Przykład (feature)

- resolution1 - Bad1
- if (resolution1) Bad1
  else (res2) Bad2
- if (goodversion) Good1 else if (reso...

Przykład (system)

- 3x Superman without time
- working system on time
- 10x Man without time
- multiple systems – 3xMan for each – almost on time
- 20x Supportman without time
- plenty of systems – delay after delay...

SOLID

UNCLE BOB – Robert C. Martin

Single Responsible Principle – Zasada pojedynczej odpowiedzialności

Rzeczy, które zmieniają się z tych samych powodów powinny być zgrupowane, a które zmieniają się z różnych powodów powinny być oddzielone

Obiekt powinien posiadać pojedynczą odpowiedzialność - zmieniamy klasę tylko z jednego powodu

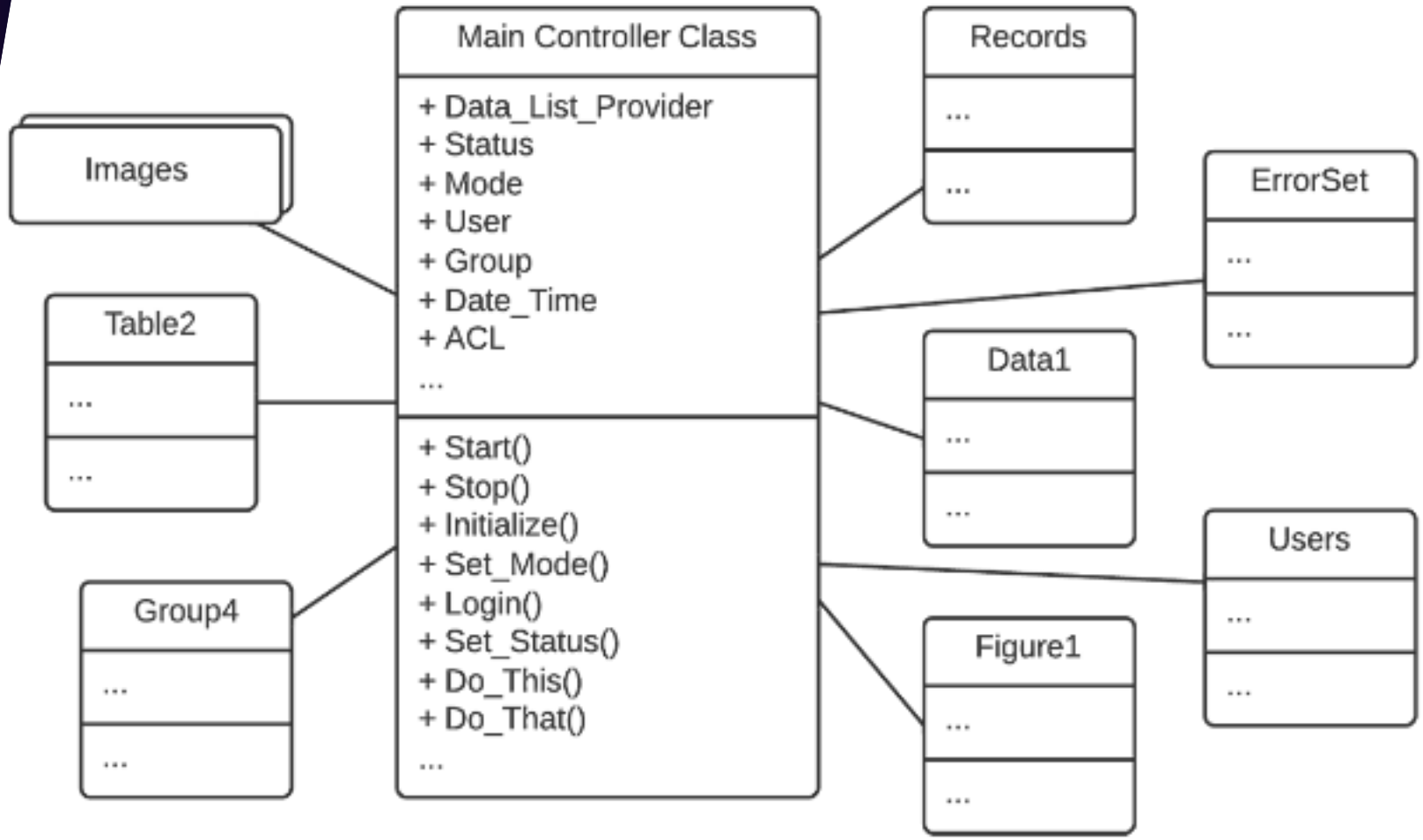
Odpowiedzialność ta powinna być enkapsulowana

Odpowiedzialność jest osią zmian - jeżeli są zmiany wymagań, to zmiany jednej odpowiedzialności mogą hamować lub zaburzać pozostałe funkcje klasy

Zasada SRP jest najprostszą i najtrudniejszą z zasad. Łączenie obowiązków jest czymś, co robimy w sposób naturalny.

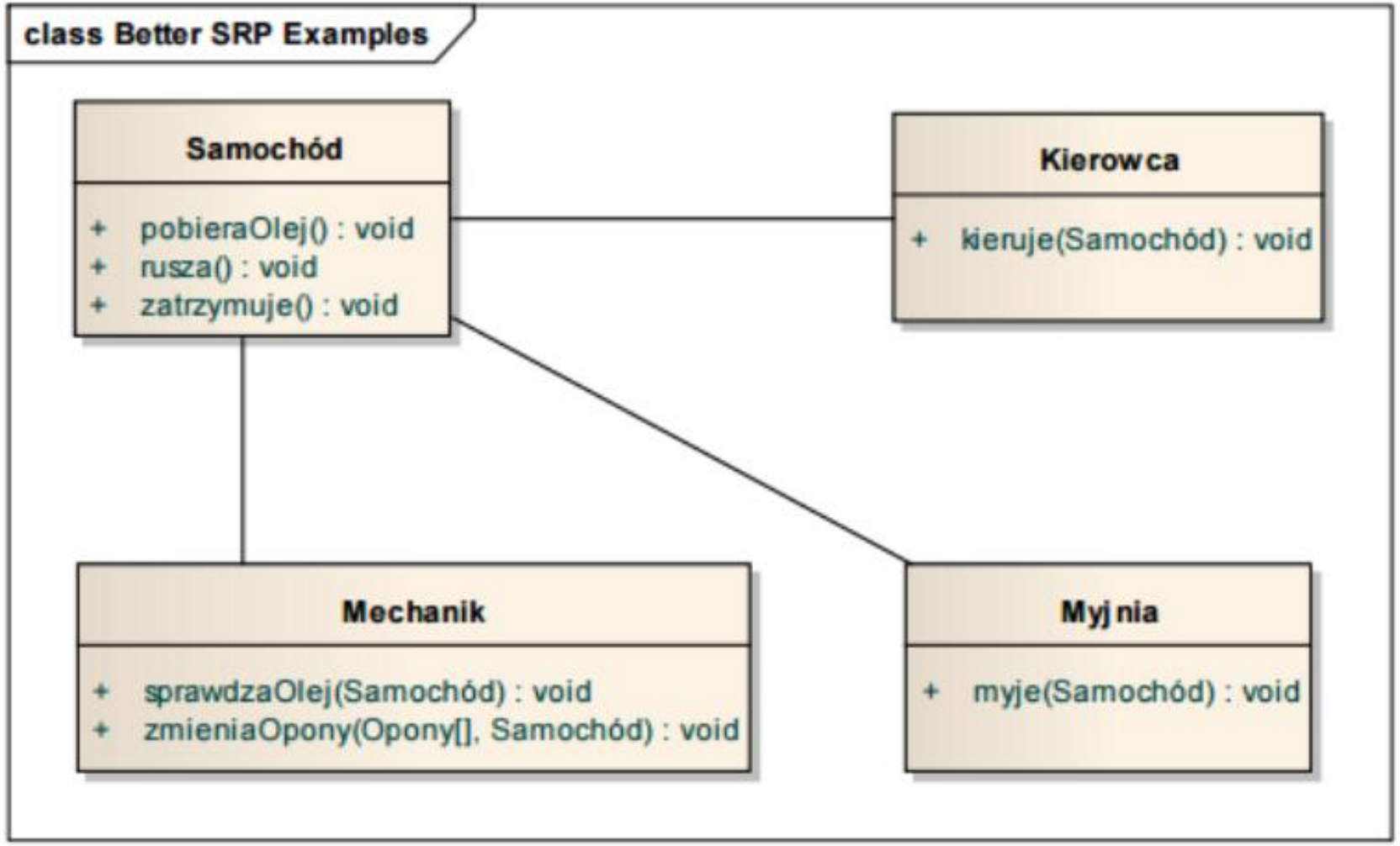
Z separacją obowiązków wiążą się również pozostałe zasady.

Przykład - boska klasa



Samochód: rusza, zatrzymuje się, myje się, wymienia opony, kieruje, sprawdza olej, pobiera olej

Przykład - rozdział odpowiedzialności



Open-closed principle (OCP) - zasada otwarte-zamknięte

Otwarte na rozszerzenia, zamknięte na modyfikacje.

Mamy switch – gdzie jeszcze będzie? Zdegradowany albo w ogóle go nie będzie.

Zamykanie na modyfikację – kod (core) zostaje niezmieniony, dopisujemy tylko nowy

Jednocześnie Otwieranie na rozszerzenia – możemy dodawać nowe zachowania klasie, dodając nowy kod

Jak połączyć te dwie „sprzeczności” – korzystając z abstrakcji

- Moduł jest zamknięty na modyfikację, ponieważ zależy od abstrakcji
- Można dodawać nowe zachowania tworząc klasy pochodne abstrakcji

Obiektowość pomaga – daje elastyczność, wymienność i łatwość konserwacji kodu.

Barokowość nie pomaga - rozbudowane abstrakcje do każdej części aplikacji. Abstrakcje powinny być stosowane tylko do tych części programu, które często się zmieniają i gdzie mają sens.

Przykład OCP

Obliczanie podatków:

- w zależności od kraju
- w zależności od typu podatku
- w zależności od płacącego

Złe rozwiązanie: kaskada ifów/switchów

Dobre rozwiązanie: różnego rodzaju strategie/polityki

Strategie/polityki

```
public class TaxCalculator
{
    public BigDecimal CalculateTax(BigDecimal amount, String country)
    {
        BigDecimal taxAmount = new BigDecimal( val: 0);
        switch(country)
        {
            case "USA":
                //calculate tax as per USA rules
                break;
            case "UK":
                //calculate tax as per UK rules
                break;
            case "IN":
                //calculate tax as per India rules
                break;
        }
        return taxAmount;
    }
    public BigDecimal CalculateTax(BigDecimal amount, TaxCalculatorBase strategy)
    {
        return strategy.CalculateTax().multiply(amount);
    }
}
```

```
public abstract class TaxCalculatorBase
{
    public BigDecimal TotalAmount;
    public abstract BigDecimal CalculateTax();
};
public class USATax extends TaxCalculatorBase
{
    public BigDecimal CalculateTax()
    {
        //calculate tax as per USA rules
        return new BigDecimal( val: 0);
    }
};
public class UKTax extends TaxCalculatorBase
{
    public BigDecimal CalculateTax()
    {
        //calculate tax as per UK rules
        return new BigDecimal( val: 0);
    }
}
public class IndiaTax extends TaxCalculatorBase
{
    public BigDecimal CalculateTax()
    {
        //calculate tax as per India rules
        return new BigDecimal( val: 0);
    }
}
```

Liskov substitution principle (LSP) - zasada podstawiania Liskov

BARBARA LISKOV – Musi być możliwość podstawiania typów pochodnych pod ich bazowe

Innymi słowy – klasa pochodna nie może zmieniać zasady działania klasy bazowej, czyli klasy pochodne nie mogą ‚usuwać' funkcjonalności klasy bazowej

Wspólne funkcjonalności raczej powinny być wyodrębniane do nowych klas abstrakcyjnych/interfejsów a następnie dziedziczone

Dzięki możliwości podstawiania podtypów moduł, który jest zaimplementowany w kontekście klasy bazowej, można rozszerzać bez modyfikowania. - calculateTax się nie zmienia

Definicja relacji IS-A jest zbyt szeroka, aby mogła służyć za określenie podtypu. Prawdziwa definicja podtypu to „możliwość podstawienia", gdzie podstawienie jest zdefiniowane za pomocą jawnego lub niejawnego kontraktu.

```
    this.listener = listener;
}
```

Przykład

```
public interface ILead
{
    void CreateSubTask();
    void AssignTask();
    void WorkOnTask();
};
public class TeamLead implements ILead {
    public void AssignTask() {
        //Code to assign a task.
    }

    public void CreateSubTask() {
        //Code to create a sub task
    }

    public void WorkOnTask() {
        //Code to implement perform assigned task.
    }
};
public class Manager implements ILead {
    public void AssignTask() {
        //Code to assign a task.
    }

    public void CreateSubTask() {
        //Code to create a sub task.
    }

    public void WorkOnTask() {
        throw new Exception("Manager can't work on Task");
    }
}
```

```
public interface IProgrammer  {
    void WorkOnTask();
}
public interface ILead  {
    void AssignTask();
    void CreateSubTask();
}
public class Programmer implements IProgrammer  {
    public void WorkOnTask() {
        //code to implement to work on the Task.
    }
}
public class Manager implements ILead  {
    public void AssignTask() {
        //Code to assign a Task
    }
    public void CreateSubTask() {
        //Code to create a sub taks from a task.
    }
}
public class TeamLead implements IProgrammer, ILead  {
    public void AssignTask() {
        //Code to assign a Task
    }
    public void CreateSubTask() {
        //Code to create a sub task from a task.
    }
    public void WorkOnTask() {
        //code to implement to work on the Task.
    }
}
```

Interface Segregation Principle (ISP) - zasada segregacji interfejsów

Klasy nie powinny być zmuszane do zależności od metod, których nie używają, ponieważ muszą dostosowywać się do zmian w tych metodach

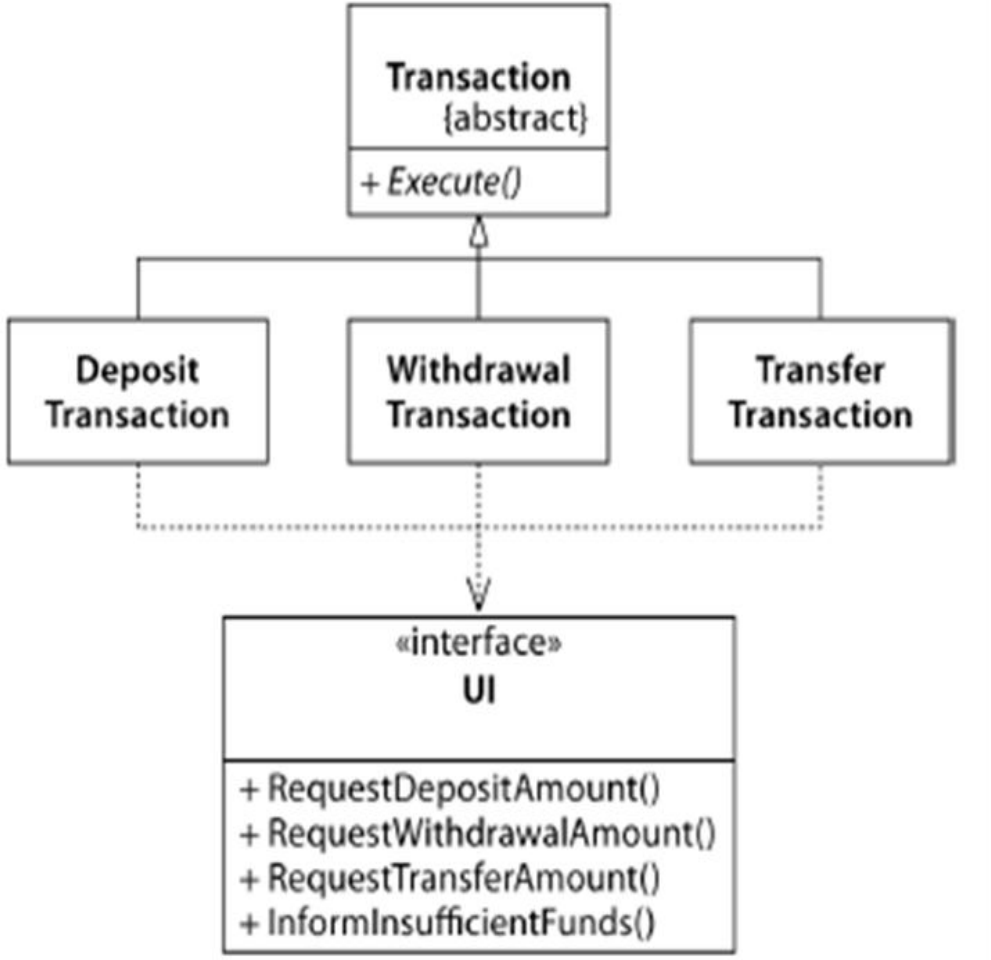
Klasa nie powinna musieć implementować interfejsu, którego nie używa - nie tworzymy grubych interfejsów

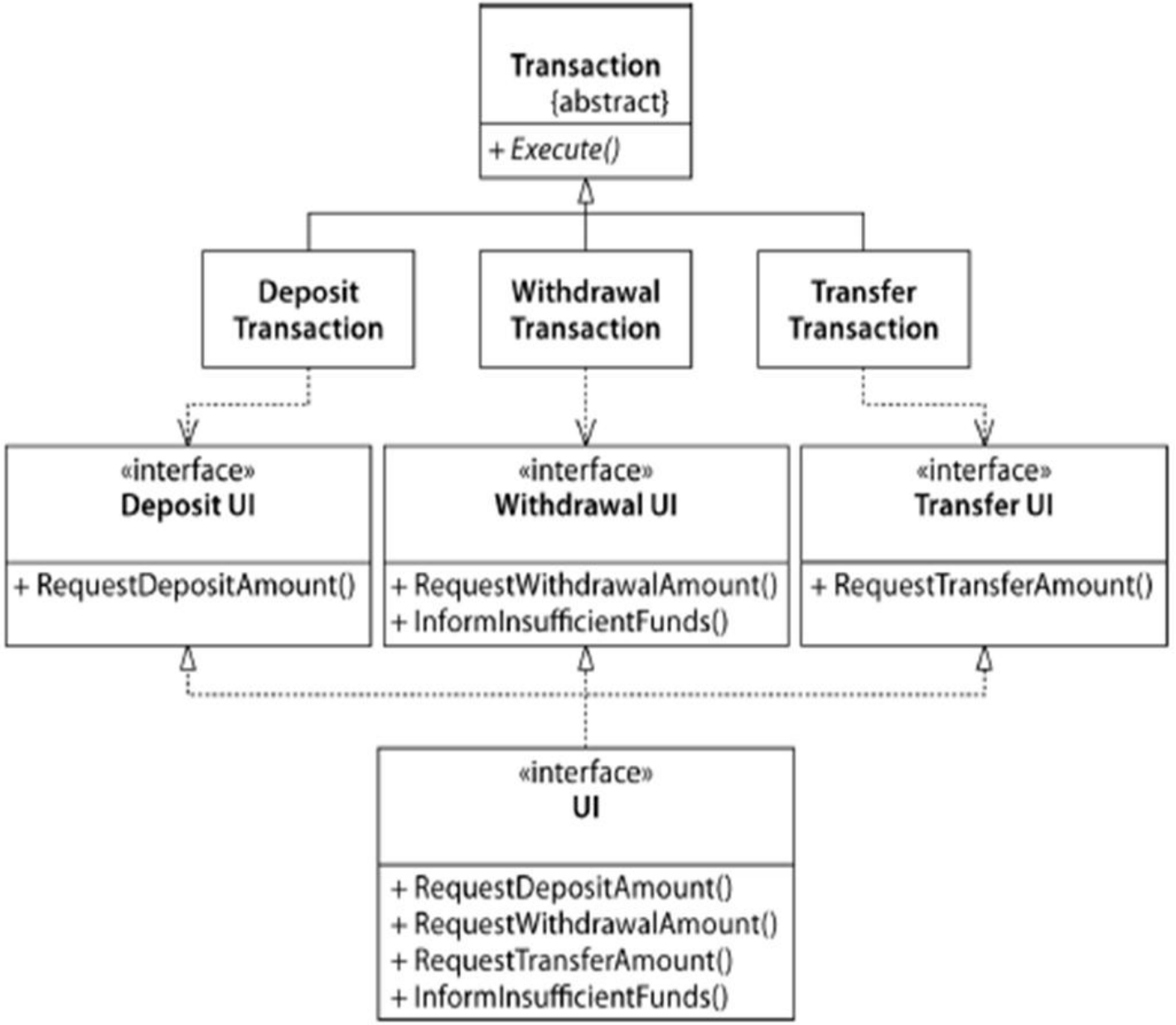
Jak SRP tylko dla interfejsów lub LSP dla klas.

„Grube” klasy powoduję dziwaczne i szkodliwe sprzężenia pomiędzy klasami klienckimi. Gdy jeden z klientów wymusza zmiany w grubej klasie, zmiany te mają wpływ na wszystkie pozostałe klasy klienckie. Z tego względu klienty powinny zależeć tylko od tych metod, które faktycznie wywołują.

Należy rozdzielić interfejs grubej klasy na wiele specyficznych dla poszczególnych klientów. Każdy interfejs dla klienta deklaruje tylko te funkcje, które wywołuje ten konkretny klient lub grupa klientów. W takiej sytuacji grube klasy mogą dziedziczyć i implementować wszystkie interfejsy specyficzne dla klientów. To eliminuje zależność klientów od metod, których one nie wywołują, i pozwala klientom zachować niezależność od siebie.

Przykład


Transaction
{abstract}
+ Execute()
Deposit Transaction
Withdrawal Transaction
Transfer Transaction
«interface»
UI
+ RequestDepositAmount()
+ RequestWithdrawalAmount()
+ RequestTransferAmount()
+ InformInsufficientFunds()


Transaction
{abstract}
+ Execute()
Deposit Transaction
Withdrawal Transaction
Transfer Transaction
«interface»
Deposit UI
+ RequestDepositAmount()
«interface»
Withdrawal UI
+ RequestWithdrawalAmount()
+ InformInsufficientFunds()
«interface»
Transfer UI
+ RequestTransferAmount()
«interface»
UI
+ RequestDepositAmount()
+ RequestWithdrawalAmount()
+ RequestTransferAmount()
+ InformInsufficientFunds()

Dependency inversion principle (DIP) - zasada odwracania zależności

Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. I jedne, i drugie powinny zależeć od abstrakcji.

Abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji.

Dlaczego odwracania? - w bardziej tradycyjnych metodykach wytwarzania oprogramowania, takich jak analiza i projektowanie strukturalne, obowiązuje tendencja do tworzenia struktur, w których moduły wysokopoziomowe zależą od modułów niskopoziomowych – a my odwracamy tą zależność

Przykład

```
public class Dependency {


    public class PDFReader {
        private PDFBook book;

        public void read() {
            book.read();
        }
    }

    public class PDFBook {
        void read() {

        }
    }

}
```

```
public interface EBook {
    void read();
}

public class DependencyInversion {

    public class EBookReader {
        private EBook book;

        public void read() {
            book.read();
        }
    }

    public class PDFBook implements EBook {
        public void read(){

        }
    }
}
```

Podsumowując

"S" - Single responsibility principle

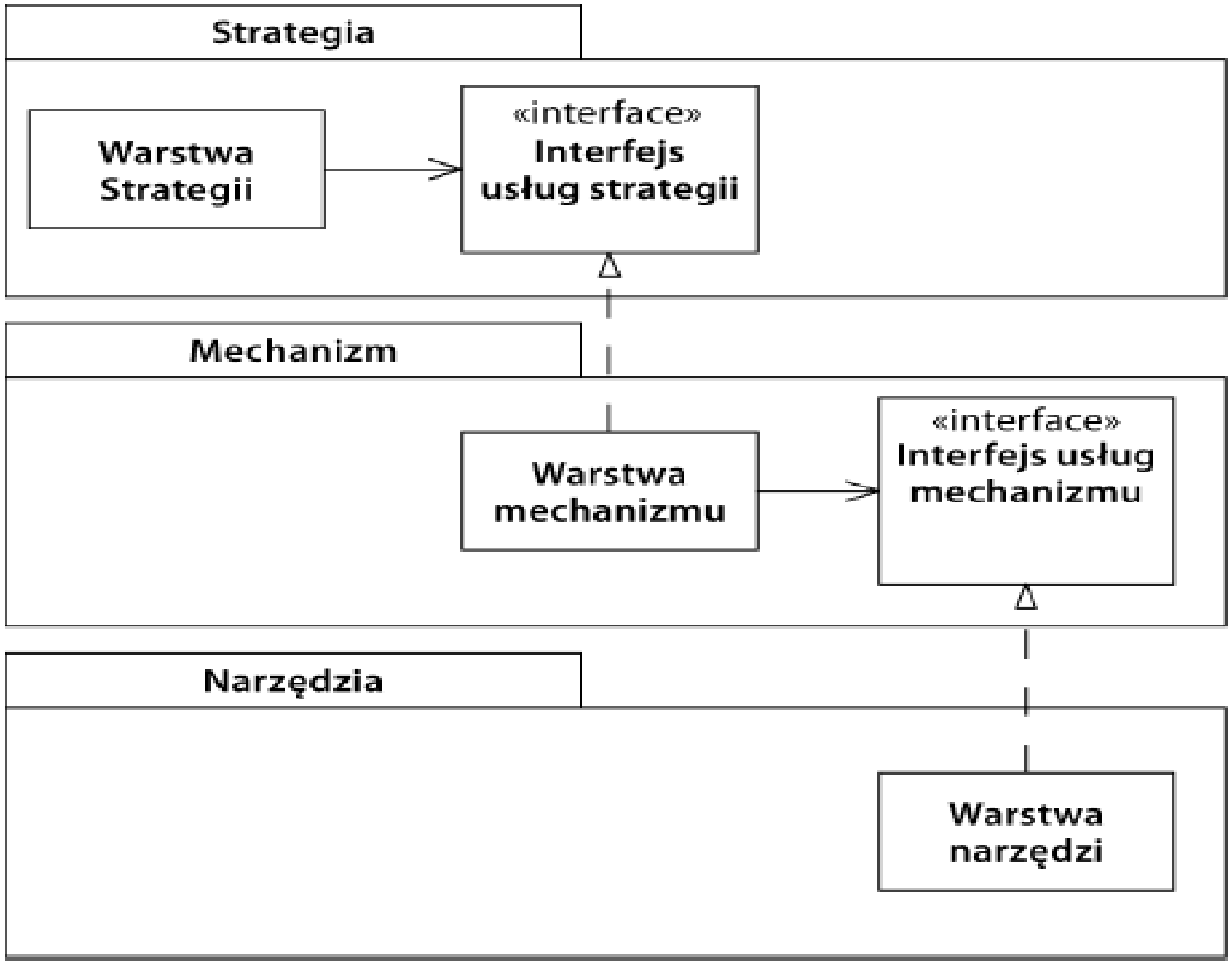
"O" - Open closed principle

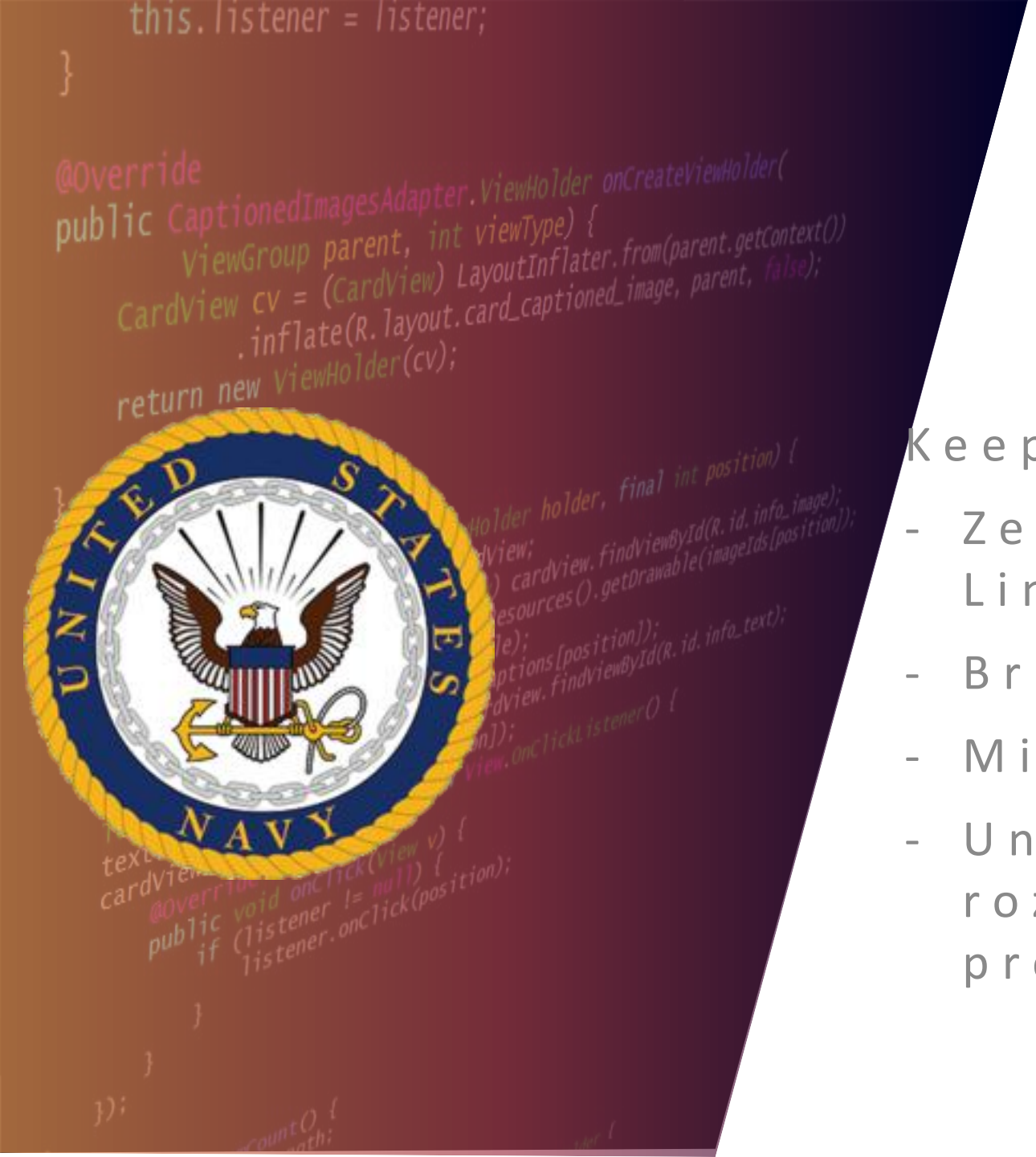
"L" - Liskov substitution principle

"I" - Interface Segregation principle

"D" - Dependency inversion principle

Architektura warstwowa

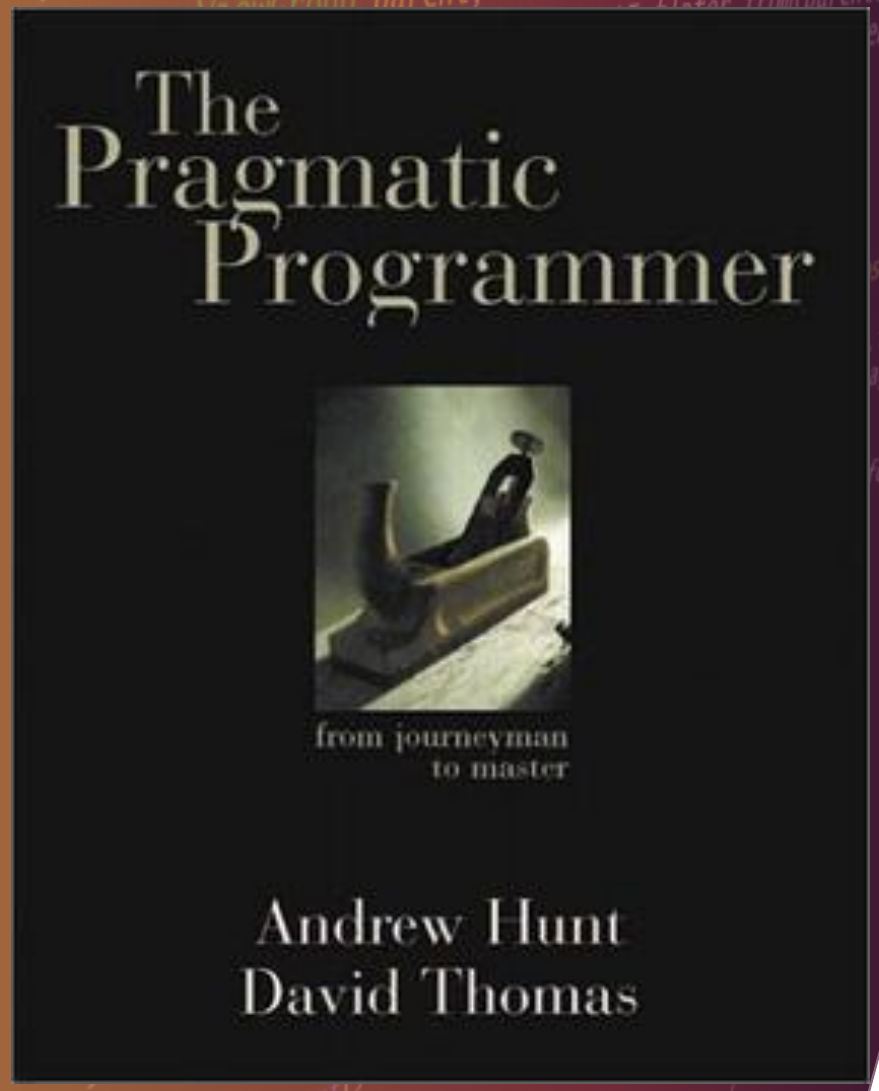




KISS

Keep It Simple, Stupid

- Zestaw narzędzi w systemie Linux
- Brzytwa Ockhama
- Minimalizm
- Unikanie popularnych rozwiązań dla KAŻDEGO problemu



DRY

Don't Repeat Yourself

- Każdy element wiedzy ma swoją klarowną i pewną reprezentację tylko w jednym miejscu
- W przeciwnym wypadku zaczną pojawiać się niespójności
- WET = write everything twice...

Patterns - GoF

Gang of Four (GoF)

Od lewej:

Ralph Johnson

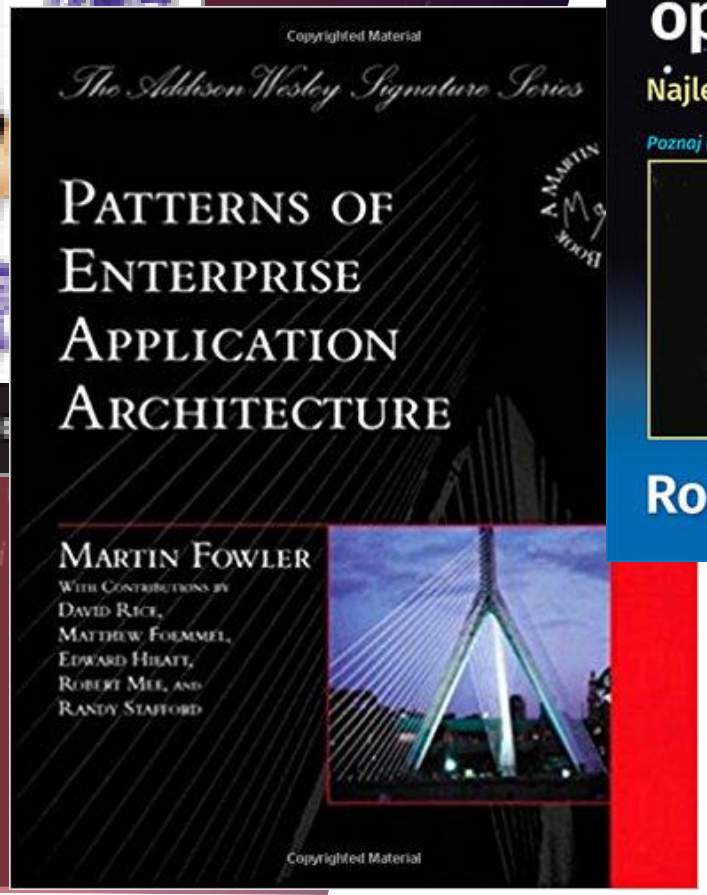
Richard Helm

Erich Gamma

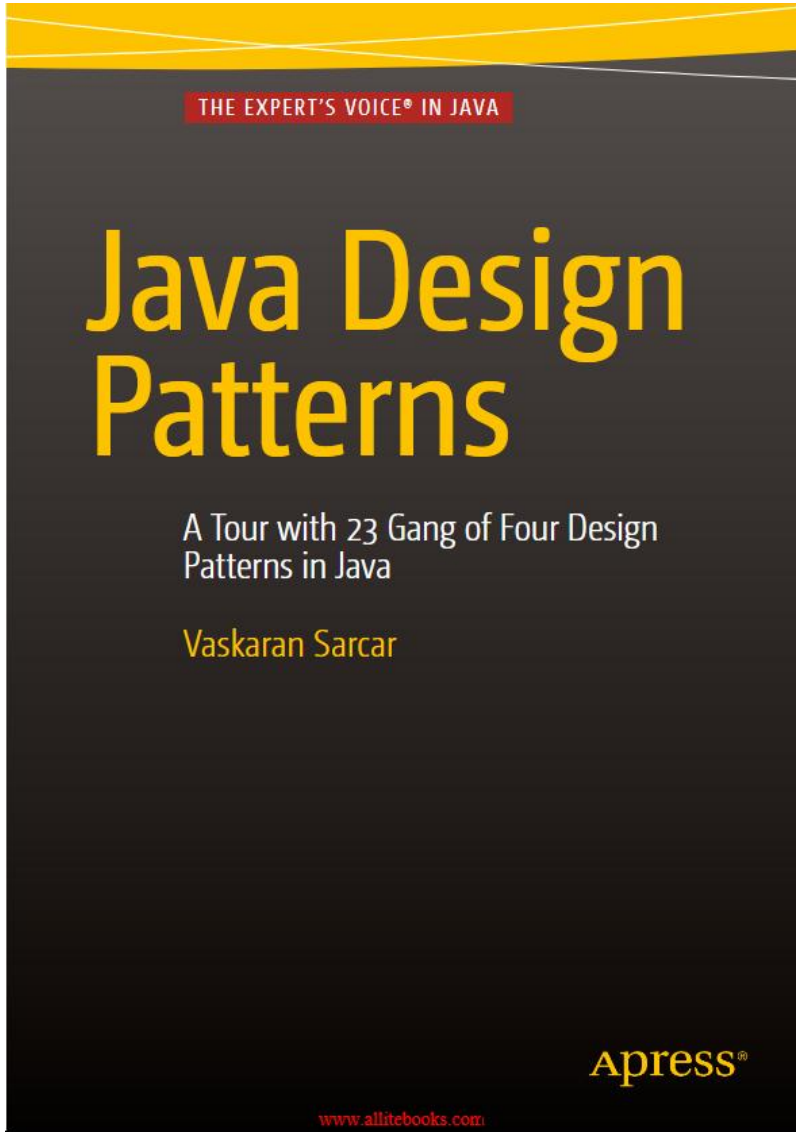
John Vlissides

Książki


O'REILLY
Head First
Design Patterns
A Brain-Friendly Guide
Eric Freeman & Elisabeth Robson


Copyrighted Material
The Addison Wesley Signature Series
PATTERNS OF
ENTERPRISE
APPLICATION
ARCHITECTURE
MARTIN FOWLER
WITH CONTRIBUTIONS BY
DAVID RICE,
MATTHEW FOEMMEL,
EDWARD HIEATT,
ROBERT MEE, AND
RANDY STAFFORD
Copyrighted Material


Helion
Zwinne wytwarzanie oprogramowania
Najlepsze zasady, wzorce i praktyki
Poznaj nowoczesne sposoby wytwarzania oprogramowania!
Robert C. Martin


THE EXPERT'S VOICE® IN JAVA
Java Design Patterns
A Tour with 23 Gang of Four Design Patterns in Java
Vaskaran Sarcar
Apress®
www.allitebooks.com

Sens stosowania wzorców

- otwartość/uodpornienie na zmiany
- przewidywanie możliwych zmian
- ułatwienie dostosowania do zmian uwarunkowań (nie 1000 klientów, a 10000)

Czym są wzorce?

- rozwiązaniem pewnej klasy zadań, które się sprawdza
- rozwiązuje lub ułatwia rozwiązywanie typowych problemów w pewnej klasie zadań
- jest to jedynie szablon - trzeba go umieć zastosować i dostosować
- ułatwiają komunikację/pomoc przy rozwiązaniach

Rodziny wzorców

- kreacyjne - opisujące proces tworzenia nowych obiektów: Abstract Factory, Builder, Factory, Prototype, Singleton

- strukturalne - opisujące struktury powiązanych ze sobą obiektów: Adapter, bridge, composite, decorator, facade, flyweight, proxy

- czynnościowe (behawioralne) - opisujące zachowanie i odpowiedzialność współpracujących ze sobą obiektów: Chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor, saga

STRATEGIA – matka wielu wzorców

- wzorzec projektowy pozwalający na implementację kilku różnych wersji algorytmu rozwiązującego dane zadanie.
- opiera się on na zestawie podobnych klas, z których każda jest implementacją innej wersji algorytmu. Pozwala to na wymienne wykorzystanie algorytmów podczas działania programu

Strategia - zastosowanie

- gdy logika (reguły) mogą się zmienić zarówno statycznie (raz na początku) jak i dynamicznie (at runtime)

- zamiast tworzyć rozbudowane wyrażenie warunkowe lepiej jest, zgodnie z Zasadą Jednej Odpowiedzialności wydzielić poszczególne algorytmy do osobnych klas, z których każda będzie odpowiadała za wykonanie zadania według innego algorytmu.

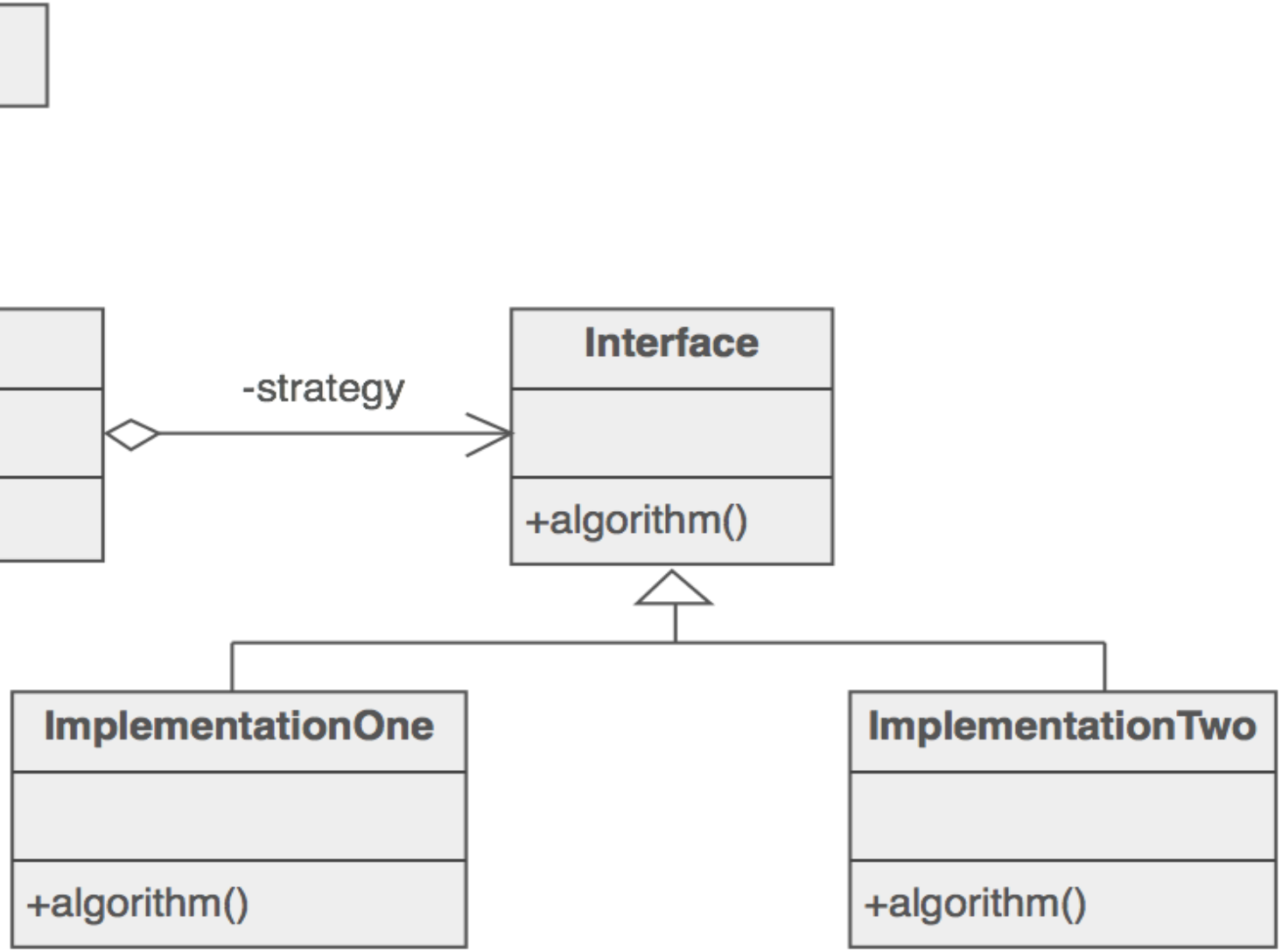
- jeśli widzimy już zaimplementowaną rozbudowaną „if-ologię”, która robi to samo na różne sposoby, warto zrefaktoryzować to do wzorca Strategia.

Strategie/Polityki

```
public class TaxCalculator
{
    public BigDecimal CalculateTax(BigDecimal amount, String country)
    {
        BigDecimal taxAmount = new BigDecimal( val: 0);
        switch(country)
        {
            case "USA":
                //calculate tax as per USA rules
                break;
            case "UK":
                //calculate tax as per UK rules
                break;
            case "IN":
                //calculate tax as per India rules
                break;
        }
        return taxAmount;
    }
    public BigDecimal CalculateTax(BigDecimal amount, TaxCalculatorBase strategy)
    {
        return strategy.CalculateTax().multiply(amount);
    }
}
```

```
public abstract class TaxCalculatorBase
{
    public BigDecimal TotalAmount;
    public abstract BigDecimal CalculateTax();
};
public class USATax extends TaxCalculatorBase
{
    public BigDecimal CalculateTax()
    {
        //calculate tax as per USA rules
        return new BigDecimal( val: 0);
    }
};
public class UKTax extends TaxCalculatorBase
{
    public BigDecimal CalculateTax()
    {
        //calculate tax as per UK rules
        return new BigDecimal( val: 0);
    }
}
public class IndiaTax extends TaxCalculatorBase
{
    public BigDecimal CalculateTax()
    {
        //calculate tax as per India rules
        return new BigDecimal( val: 0);
    }
}
```

Strategie/polityki - diagram



Fabryka

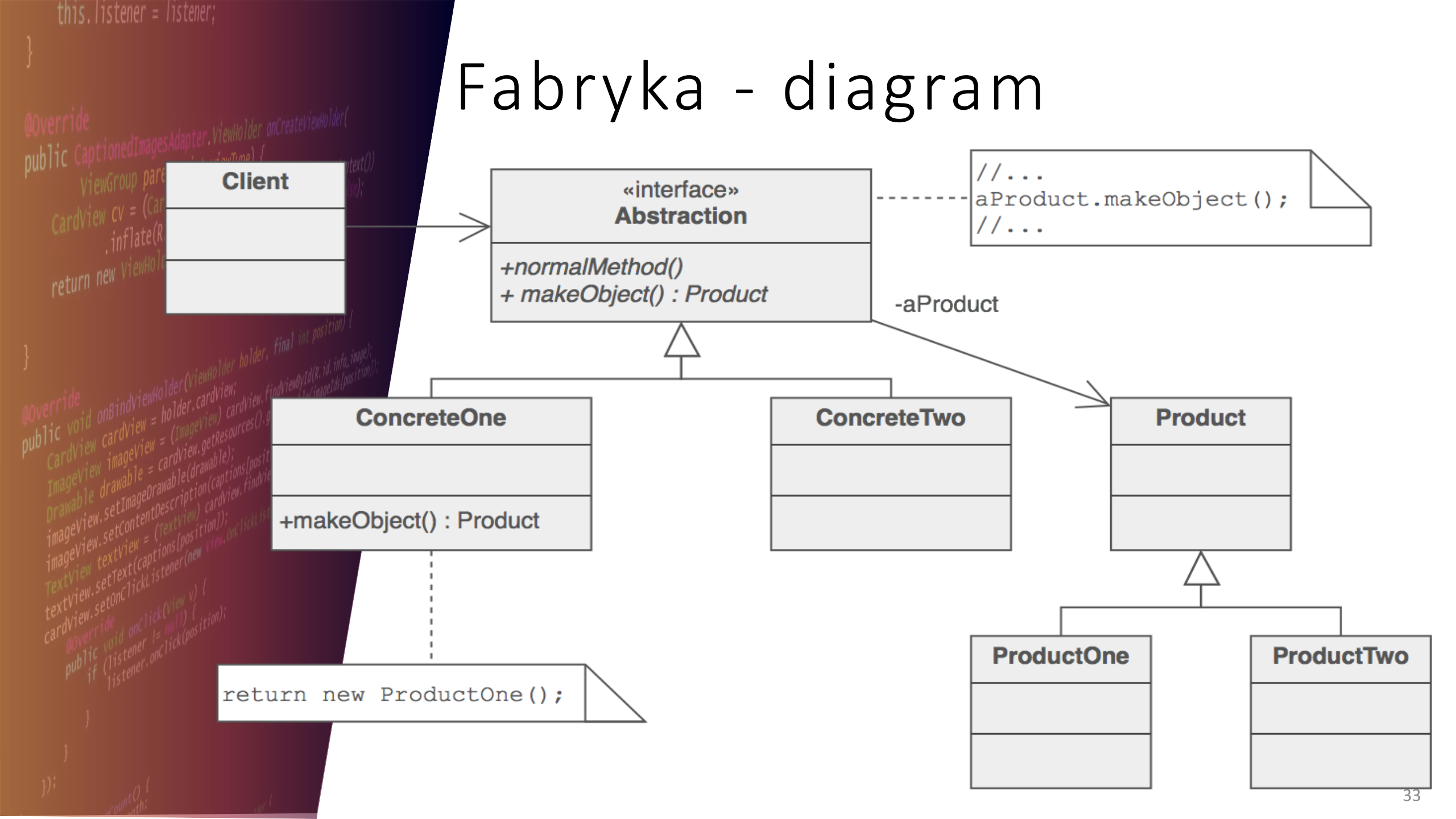
Mamy wiele klas w ramach strategii, to przydałoby się je prosto tworzyć. Fabryka:

- zwraca różne warianty klas dziedziczących po wspólnym interfejsie
- rodzaj tworzonej klasy jest wybierany w trakcie uruchomienia (at runtime)
- często metoda create() jest statyczna

Fabryka - zastosowanie

- gdy w danym kawałku kodu nie do końca wiemy która klasa będzie użyta
- gdy potencjalne klasy dziedziczą po wspólnym interfejsie
- aby ukryć rodzaje podklas
- aby ukryć (enkapsulacja) logikę (warunki) tworzenia klas

Fabryka - diagram



Adapter

Zmieniamy interfejs używając starych klas. Adapter:

- owija istniejącą funkcjonalność nowym interfejsem (na przykład dostosowanie starych komponentów do nowego systemu)

- konwersja interfejsu klasy do innego interfejsu, którego klienty oczekują. Adapter, dzięki uzupełnieniu brakujących funkcjonalności (niekompatybilnych funkcji), pozwala klasom wspólnie pracować.

Adapter - diagram