11. Zasady SOLID w OOP (Java i Python)

Cel wykładu

Poznasz 5 zasad SOLID, zobaczysz dobre i złe przykłady (Java i Python), a także proste diagramy UML (Mermaid) ułatwiające zrozumienie. Każdą literę ilustrują co najmniej dwa przykłady (anty-przykład i poprawa, czasem więcej wariantów).

Spis treści:

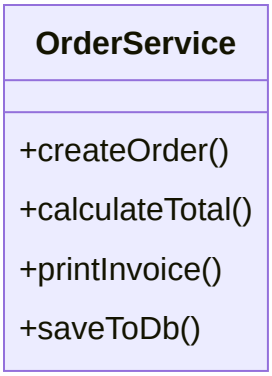
S — Single Responsibility Principle

„Klasa/moduł powinien mieć tylko jedną przyczynę do zmiany.”

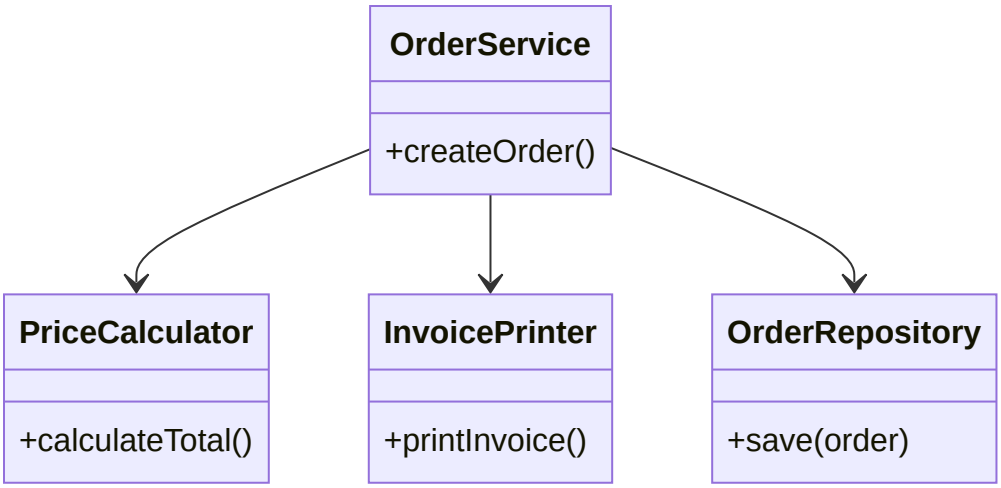
Dlaczego:

- mniejsza złożoność, łatwiejsze testy, mniej sprzężeń, mniej konfliktów przy zmianach.

UML (naruszenie SRP — klasa robi „wszystko”):



UML (zgodnie z SRP — rozdzielenie odpowiedzialności):



Zły przykład 1 (Java): jedna klasa od logiki, IO i formatowania

```
class ReportService {
    public void generateAndSaveReport(DataSource ds, String path) {
        // 1) zbieranie danych
        var data = ds.fetch();
        // 2) przetwarzanie/logika
        var total = 0.0;
        for (var d : data) total += d.value();
        // 3) formatowanie
        String content = "TOTAL: " + total + "\n";
        // 4) zapis do pliku
        try (var fw = new java.io.FileWriter(path)) {
            fw.write(content);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Lepszy przykład 1 (Java): separacja odpowiedzialności

```
class ReportGenerator {
    private final Calculator calculator;
    private final Formatter formatter;
    private final Saver saver;

    ReportGenerator(Calculator c, Formatter f, Saver s) {
        this.calculator = c; this.formatter = f; this.saver = s;
    }

    public void generate(DataSource ds, String path) {
        var result = calculator.calculate(ds.fetch());
        var content = formatter.format(result);
        saver.save(content, path);
    }
```

```
}

interface Calculator { double calculate(Iterable<Item> items); }
interface Formatter { String format(double value); }
interface Saver { void save(String content, String path); }
```

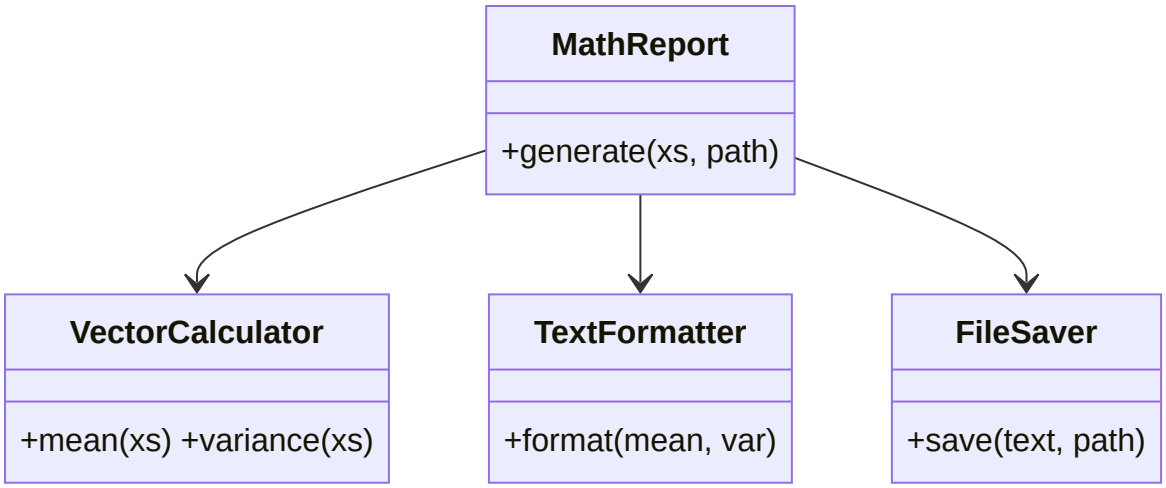
Zły przykład 2 (Python): klasa łączy walidację, zapis i wysyłkę e-mail

```python
class UserRegistration:
    def register(self, user):
        if not user.email or "@" not in user.email:
            raise ValueError("Invalid email")  # walidacja
        with open("users.txt", "a") as f:    # zapis
            f.write(user.email + "\n")
        print(f"Sending welcome email to {user.email}")  # wysyłka
```

Lepszy przykład 2 (Python):

```python
class UserValidator:
    def validate(self, user):
        if not user.email or "@" not in user.email:
            raise ValueError("Invalid email")

class UserRepository:
    def save(self, user):
        with open("users.txt", "a") as f:
            f.write(user.email + "\n")

class Mailer:
    def send_welcome(self, user):
        print(f"Sending welcome email to {user.email}")

class RegistrationService:
    def __init__(self, validator, repo, mailer):
        self.v = validator; self.r = repo; self.m = mailer
    def register(self, user):
        self.v.validate(user)
        self.r.save(user)
        self.m.send_welcome(user)
```

Przykład matematyczny (SRP) — rozdzielenie: obliczenia, formatowanie, zapis



Java (anty-przykład): jedna klasa liczy, formatuje i zapisuje

```
class MathReportBad {
    public void generate(double[] xs, String path) {
        // obliczenia
        double sum = 0; for (double x : xs) sum += x;
        double mean = xs.length == 0 ? 0 : sum / xs.length;
        double var = 0; for (double x : xs) var += (x - mean) * (x - mean);
        var = xs.length == 0 ? 0 : var / xs.length;
        // formatowanie
        String text = "mean=" + mean + ", var=" + var + "\n";
        // zapis
        try (java.io.FileWriter fw = new java.io.FileWriter(path)) {
            fw.write(text);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Java (poprawa): separacja odpowiedzialności

```
class VectorCalculator {
    double mean(double[] xs){ double s=0; for(double x:xs) s+=x; return xs.length==0?0:s/xs.le
    double variance(double[] xs){ double m=mean(xs), v=0; for(double x:xs) v+=(x-m)*(x-m); ret
}
class TextFormatter { String format(double mean, double var){ return "mean="+mean+", var="+var
class FileSaver { void save(String text, String path){ try(var fw=new java.io.FileWriter(path)

class MathReport {
    private final VectorCalculator calc; private final TextFormatter fmt; private final FileSa
    MathReport(VectorCalculator c, TextFormatter f, FileSaver s){ this.calc=c; this.fmt=f; thi
    public void generate(double[] xs, String path){ double m=calc.mean(xs); double v=calc.vari
}
```

Python (poprawa):

```python
class VectorCalculator:
    def mean(self, xs: list[float]) -> float:
        return 0.0 if not xs else sum(xs) / len(xs)
    def variance(self, xs: list[float]) -> float:
        m = self.mean(xs)
        return 0.0 if not xs else sum((x - m) ** 2 for x in xs) / len(xs)

class TextFormatter:
    def format(self, mean: float, var: float) -> str:
        return f"mean={mean}, var={var}\n"

class FileSaver:
    def save(self, text: str, path: str) -> None:
        with open(path, "w") as f:
            f.write(text)

class MathReport:
    def __init__(self, calc: VectorCalculator, fmt: TextFormatter, saver: FileSaver):
        self.c, self.f, self.s = calc, fmt, saver
    def generate(self, xs: list[float], path: str) -> None:
        m = self.c.mean(xs); v = self.c.variance(xs)
        self.s.save(self.f.format(m, v), path)
```

Checklist SRP:

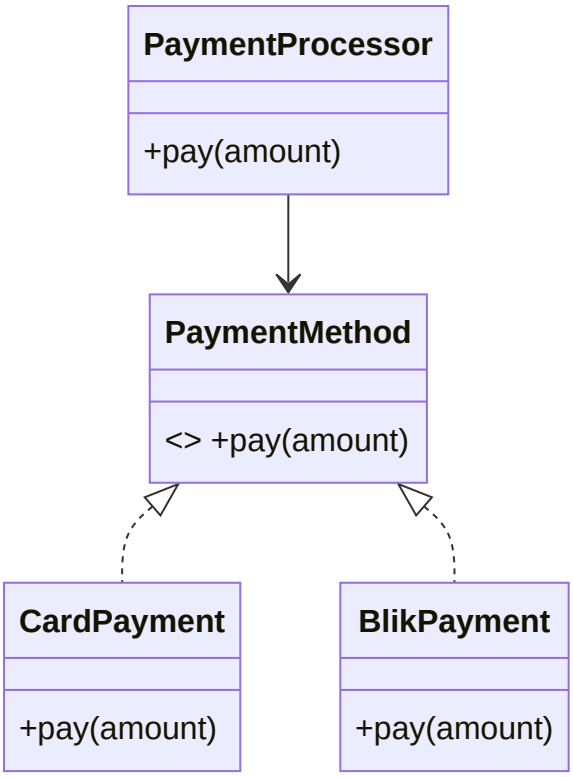
- Czy klasa ma jedną odpowiedzialność biznesową? Czy nazwa jest spójna z zawartością?
- Czy test zmiany formatu danych wymaga edycji logiki lub zapisu? Jeśli tak — naruszenie SRP.

O — Open/Closed Principle

„Być otwartym na rozszerzenia, zamkniętym na modyfikacje.”

Idea: dodajemy nowe zachowania (strategia, dziedziczenie, kompozycja), bez ruszania stabilnego, przetestowanego kodu.

UML (strategia zamiast if-ów):



Zły przykład 1 (Java): kaskada if/else w starej klasie

```
class PaymentProcessor {
    public void pay(String type, double amount) {
        if (type.equals("CARD")) {
            // płatność kartą
        } else if (type.equals("BLIK")) {
            // płatność BLIK
        } else if (type.equals("CRYPTO")) {
            // nowy typ => trzeba modyfikować klasę
        }
    }
}
```

Lepszy przykład 1 (Java): rozszerzalność przez interfejs

```
interface PaymentMethod { void pay(double amount); }
class CardPayment implements PaymentMethod { public void pay(double a){ /*...*/ } }
class BlikPayment implements PaymentMethod { public void pay(double a){ /*...*/ } }

class PaymentProcessor {
    private final PaymentMethod method;
    PaymentProcessor(PaymentMethod m) { this.method = m; }
    public void pay(double amount) { method.pay(amount); }
}
```

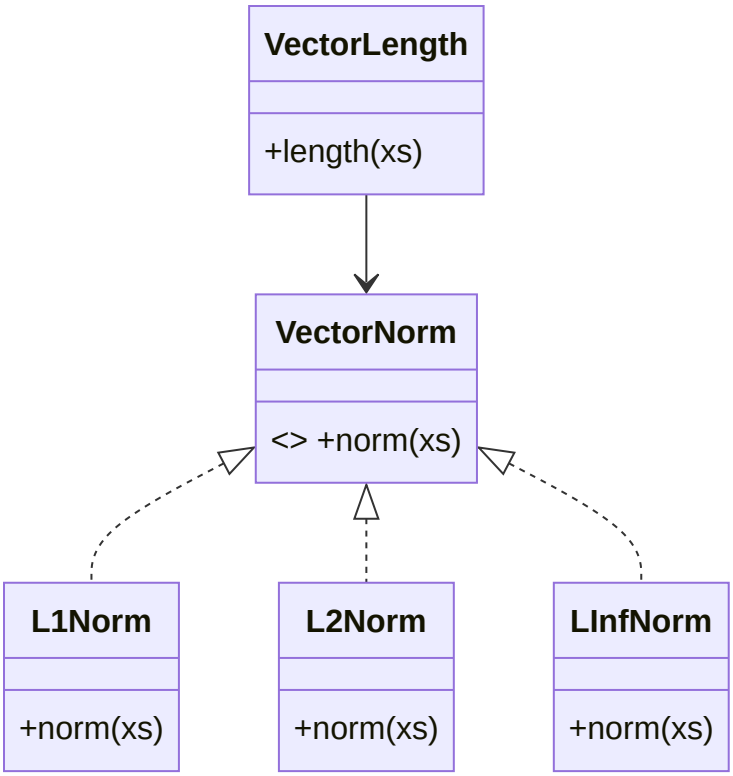
Zły przykład 2 (Python): modyfikacja funkcji za każdym razem

```python
def discount(price, type):
    if type == "student":
        return price * 0.8
    elif type == "vip":
        return price * 0.7
    elif type == "black_friday":
        return price * 0.5
    # każdy nowy typ => edycja funkcji
```

Lepszy przykład 2 (Python): rejestr strategii

```python
class Discount:
    def apply(self, price):
        raise NotImplementedError

class StudentDiscount(Discount):
    def apply(self, price):
        return price * 0.8

class VipDiscount(Discount):
    def apply(self, price):
        return price * 0.7

DISCOUNTS = {
    "student": StudentDiscount(),
    "vip": VipDiscount(),
}

def discount(price, kind):
    return DISCOUNTS[kind].apply(price)
```

Przykład matematyczny (OCP) — strategie liczenia normy wektora



Java: bez kaskady if-ów, łatwo dodać kolejną normę

```
interface VectorNorm { double norm(double[] xs); }
final class L1Norm implements VectorNorm { public double norm(double[] xs){ double s=0; for(do
final class L2Norm implements VectorNorm { public double norm(double[] xs){ double s=0; for(do
final class LInfNorm implements VectorNorm { public double norm(double[] xs){ double m=0; for(

final class VectorLength {
    private final VectorNorm norm;
    VectorLength(VectorNorm n){ this.norm = n; }
    public double length(double[] xs){ return norm.norm(xs); }
}
```

Python: rejestr strategii

```
class VectorNorm:
    def norm(self, xs: list[float]) -> float: raise NotImplementedError

class L1Norm(VectorNorm):
    def norm(self, xs: list[float]) -> float: return sum(abs(x) for x in xs)

class L2Norm(VectorNorm):
    def norm(self, xs: list[float]) -> float: return (sum(x*x for x in xs)) ** 0.5

class LInfNorm(VectorNorm):
    def norm(self, xs: list[float]) -> float: return max((abs(x) for x in xs), default=0.0)
```

```
NORMS = {"l1": L1Norm(), "l2": L2Norm(), "linf": LInfNorm()}

def vector_length(xs: list[float], kind: str = "l2") -> float:
    return NORMS[kind].norm(xs)
```

Checklist OCP:

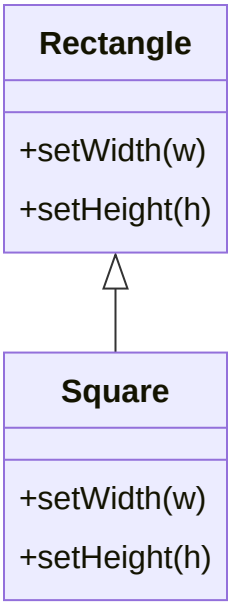
- Czy dodanie nowego wariantu wymaga modyfikacji istniejących klas, czy tylko dodania nowej implementacji/interpretera/strategii?

L — Liskov Substitution Principle

„Obiekty podklas powinny dać się użyć zamiast obiektów klas bazowych bez zmiany poprawności.”

Oznaki naruszenia: wymuszanie dodatkowych pre/postwarunków, rzucanie nieoczekiwanych wyjątków, „puste”/zablokowane metody.

UML (klasycznie błędny przykład: Kwadrat jako Prostokąt):



Zły przykład 1 (Java): `Square` łamie oczekiwania `Rectangle`

```
class Rectangle {
    protected int w, h;
    public void setWidth(int w){ this.w = w; }
    public void setHeight(int h){ this.h = h; }
    public int area(){ return w * h; }
}
class Square extends Rectangle {
    @Override public void setWidth(int w){ this.w = this.h = w; }
    @Override public void setHeight(int h){ this.w = this.h = h; }
```

```
}
```

Lepszy przykład 1 (Java): rozdziel interfejsy/abstrakcje

```
interface Shape { int area(); }
final class Rectangle implements Shape {
    private final int w, h;
    Rectangle(int w, int h){ this.w=w; this.h=h; }
    public int area(){ return w*h; }
}
final class Square implements Shape {
    private final int a;
    Square(int a){ this.a=a; }
    public int area(){ return a*a; }
}
```

Zły przykład 2 (Python): podklasa zawęża kontrakt

```
class FileWriter:
    def write(self, text: str) -> None:
        print(text)

class SafeFileWriter(FileWriter):
    def write(self, text: str) -> None:
        if len(text) > 100:
            raise ValueError("Too long")  # nowe ograniczenie łamie LSP
        super().write(text)
```

Przykład matematyczny (LSP) — „NonNegativeVector” jako podklasa, która zaostrza kontrakt

```
class Vector {
    protected final double[] xs;
    Vector(double[] xs){ this.xs = xs; }
    public double at(int i){ return xs[i]; }
    public void set(int i, double v){ xs[i] = v; }
}

// Anty-przykład: podklasa wprowadza dodatkowe ograniczenie (tylko >= 0)
class NonNegativeVector extends Vector {
    NonNegativeVector(double[] xs){ super(xs); }
    @Override public void set(int i, double v){
        if (v < 0) throw new IllegalArgumentException("negatives not allowed"); // zawężenie k
        super.set(i, v);
    }
}
```

Lepsze podejście: kompozycja i walidacja jako osobna odpowiedzialność

```
final class NonNegativeConstraint {
    void check(double v){ if (v < 0) throw new IllegalArgumentException(); }
}
final class ConstrainedVector {
    private final Vector inner; private final NonNegativeConstraint c;
    ConstrainedVector(Vector inner, NonNegativeConstraint c){ this.inner=inner; this.c=c; }
    public void set(int i, double v){ c.check(v); inner.set(i, v); }
    public double at(int i){ return inner.at(i); }
}
```

Python (alternatywa z kompozycją):

```
class Vector:
    def __init__(self, xs: list[float]):
        self.xs = xs
    def at(self, i: int) -> float:
        return self.xs[i]
    def set(self, i: int, v: float) -> None:
        self.xs[i] = v

class NonNegativeConstraint:
    def check(self, v: float) -> None:
        if v < 0:
            raise ValueError("negatives not allowed")

class ConstrainedVector:
    def __init__(self, inner: Vector, constraint: NonNegativeConstraint):
        self.inner, self.c = inner, constraint
    def set(self, i: int, v: float) -> None:
        self.c.check(v); self.inner.set(i, v)
    def at(self, i: int) -> float:
        return self.inner.at(i)
```

Lepszy przykład 2 (Python): wprowadź nowy typ/kontrakt

```
class Writer:
    def write(self, text: str) -> None:
        raise NotImplementedError

class ConsoleWriter(Writer):
    def write(self, text: str) -> None:
        print(text)

class BoundedWriter(Writer):
    def __init__(self, limit: int, inner: Writer):
        self.limit = limit; self.inner = inner
    def write(self, text: str) -> None:
        if len(text) > self.limit:
            text = text[:self.limit]
        self.inner.write(text)
```

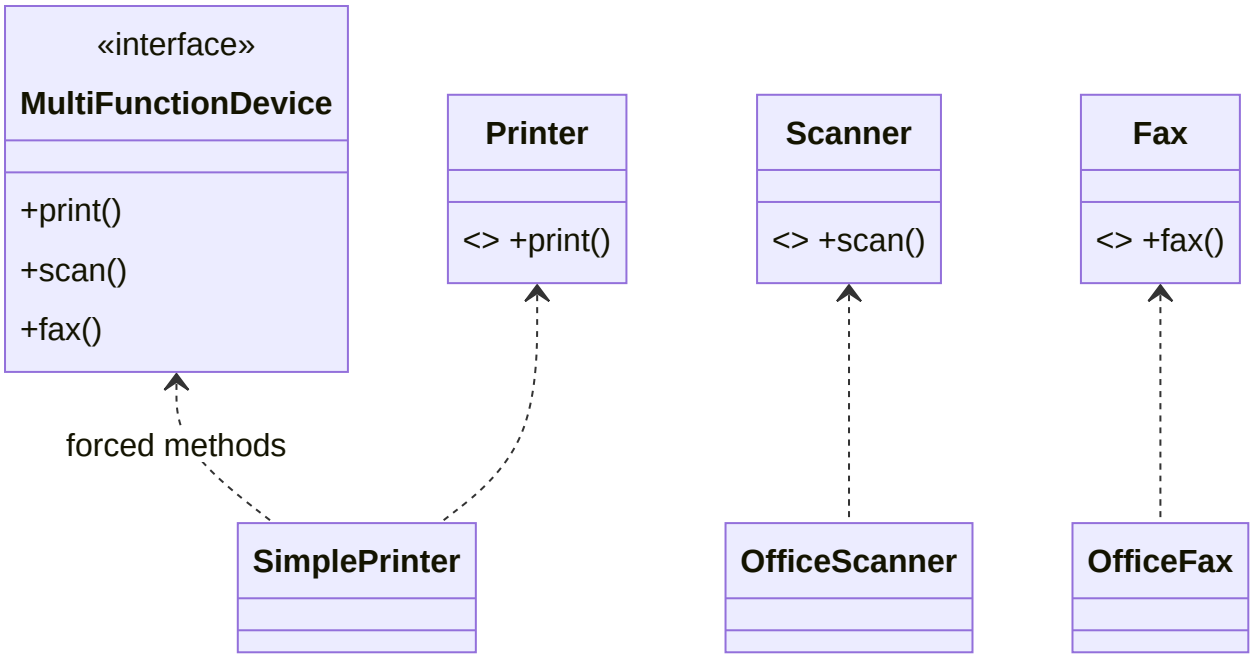
Checklist LSP:

- Czy podklasa nie wprowadza dodatkowych ograniczeń względem klasy bazowej?
- Czy kod używający bazowego interfejsu działa poprawnie z każdą implementacją?

I — Interface Segregation Principle

„Wiele specjalistycznych interfejsów jest lepsze niż jeden ogólny, przeładowany.”

UML (zbyt gruby interfejs vs. segregacja):



Zły przykład 1 (Java): jeden interfejs do wszystkiego

```
interface MultiFunctionDevice {
    void print();
    void scan();
    void fax();
}
class SimplePrinter implements MultiFunctionDevice {
    public void print() { /* ok */ }
    public void scan() { throw new UnsupportedOperationException(); }
    public void fax() { throw new UnsupportedOperationException(); }
}
```

Lepszy przykład 1 (Java): rozdziel interfejsy

```
interface Printer { void print(); }
interface Scanner { void scan(); }
```

```
class SimplePrinter implements Printer {
    public void print() { /* ... */ }
}
```

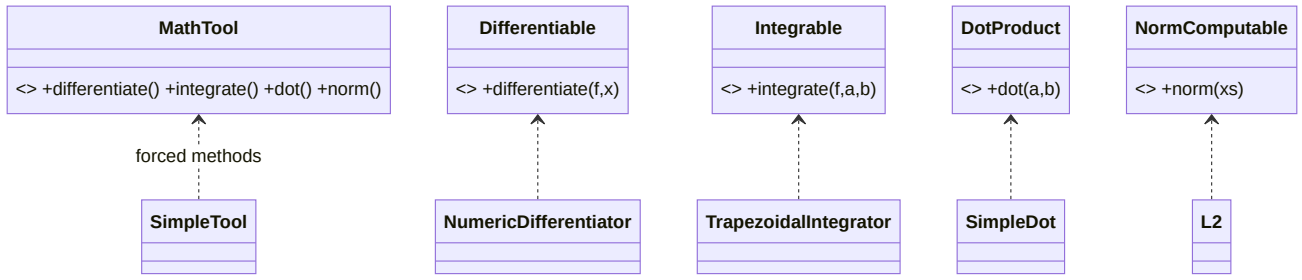
Zły przykład 2 (Python): „tłusty” protokół

```python
class Storage:
    def save(self, data): ...
    def load(self, key): ...
    def purge_cache(self): ...

class FileStorage(Storage):
    def save(self, data): ...
    def load(self, key): ...
    def purge_cache(self):
        raise NotImplementedError  # nie dotyczy
```

Lepszy przykład 2 (Python): mniejsze protokoły

```python
class Saver:
    def save(self, data): ...

class Loader:
    def load(self, key): ...

class FileStorage(Saver, Loader):
    def save(self, data): ...
    def load(self, key): ...
```

Przykład matematyczny (ISP) — unikamy „tłustego” interfejsu narzędzia



Java (anty-przykład):

```
interface MathTool {
    double differentiate(java.util.function.DoubleUnaryOperator f, double x);
    double integrate(java.util.function.DoubleUnaryOperator f, double a, double b);
    double dot(double[] a, double[] b);
    double norm(double[] xs);
}
```

```
class SimpleTool implements MathTool {
    public double differentiate(java.util.function.DoubleUnaryOperator f, double x){ throw new
    public double integrate(java.util.function.DoubleUnaryOperator f, double a, double b){ thr
    public double dot(double[] a, double[] b){ double s=0; for(int i=0;i<a.length;i++) s+=a[i]
    public double norm(double[] xs){ double s=0; for(double x:xs) s+=x*x; return Math.sqrt(s);
}
```

Java (poprawa): małe, wyspecjalizowane interfejsy

```
interface DotProduct { double dot(double[] a, double[] b); }
interface NormComputable { double norm(double[] xs); }

final class SimpleDot implements DotProduct { public double dot(double[] a, double[] b){ doubl
final class L2 implements NormComputable { public double norm(double[] xs){ double s=0; for(do
```

Python (poprawa z protokołami):

```
from typing import Protocol

class DotProduct(Protocol):
    def dot(self, a: list[float], b: list[float]) -> float: ...

class NormComputable(Protocol):
    def norm(self, xs: list[float]) -> float: ...

class SimpleDot:
    def dot(self, a: list[float], b: list[float]) -> float:
        return sum(x*y for x, y in zip(a, b))

class L2:
    def norm(self, xs: list[float]) -> float:
        return (sum(x*x for x in xs)) ** 0.5
```

Checklist ISP:

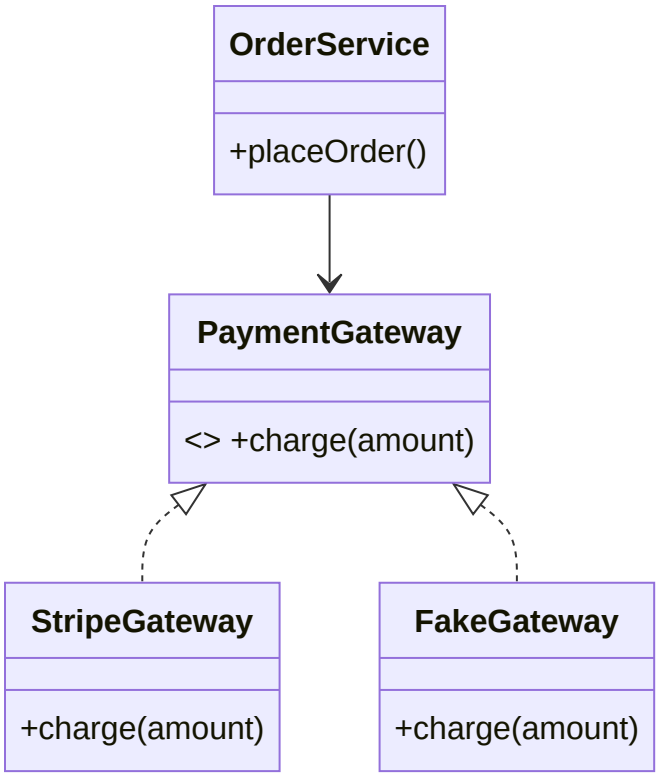
- Czy implementacje nie muszą rzucać `UnsupportedOperation` dla metod z interfejsu?
- Czy interfejsy odzwierciedlają rzeczywiste role/zdolności?

D — Dependency Inversion Principle

„Moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu. Oba powinny zależeć od abstrakcji.”

Konsekwencje: wstrzykiwanie zależności, testowalność (mocki/stubby), luźne powiązania.

UML (wysoki poziom zależy od interfejsu):



Zły przykład 1 (Java): twarda zależność od implementacji

```
class OrderService {
    private final StripeGateway gateway = new StripeGateway();
    public void placeOrder(double amount) { gateway.charge(amount); }
}
```

Lepszy przykład 1 (Java): zależność od abstrakcji + DI

```
interface PaymentGateway { void charge(double amount); }
class StripeGateway implements PaymentGateway { public void charge(double a){ /*...*/ } }
class OrderService {
    private final PaymentGateway gateway;
    OrderService(PaymentGateway g){ this.gateway = g; }
    public void placeOrder(double amount){ gateway.charge(amount); }
}
```

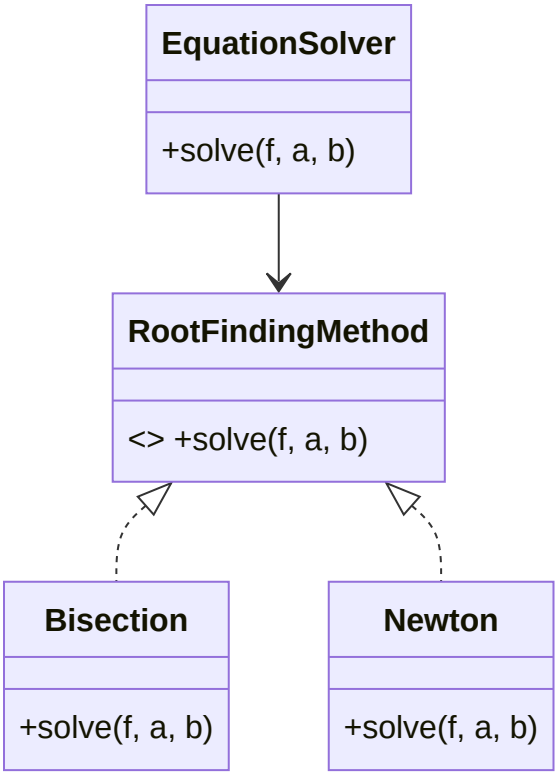
Zły przykład 2 (Python): tworzenie zależności w środku

```
class ReportService:
    def export(self, data):
        import json
        return json.dumps(data)  # trudno podmienić w testach/innym formacie
```

Lepszy przykład 2 (Python): podaj strategię z zewnątrz

```python
class Serializer:
    def dumps(self, data):
        raise NotImplementedError

class JsonSerializer(Serializer):
    def dumps(self, data):
        import json
        return json.dumps(data)

class ReportService:
    def __init__(self, serializer: Serializer):
        self.s = serializer
    def export(self, data):
        return self.s.dumps(data)
```

Przykład matematyczny (DIP) — solver równań zależny od abstrakcji metody



Java:

```java
import java.util.function.DoubleUnaryOperator;

interface RootFindingMethod { double solve(DoubleUnaryOperator f, double a, double b); }
final class Bisection implements RootFindingMethod {
    public double solve(DoubleUnaryOperator f, double a, double b){
        for(int i=0;i<50;i++){ double m=(a+b)/2, fm=f.applyAsDouble(m); if(f.applyAsDouble(a)*
```

```
        return (a+b)/2;
    }
}
final class Newton implements RootFindingMethod {
    public double solve(DoubleUnaryOperator f, double a, double b){
        double x=(a+b)/2; for(int i=0;i<20;i++){ double fx=f.applyAsDouble(x); double d=(f.app
        return x;
    }
}

final class EquationSolver {
    private final RootFindingMethod method;
    EquationSolver(RootFindingMethod m){ this.method = m; }
    public double solve(DoubleUnaryOperator f, double a, double b){ return method.solve(f, a,
}
```

Python:

```
from typing import Protocol, Callable

class RootFindingMethod(Protocol):
    def solve(self, f: Callable[[float], float], a: float, b: float) -> float: ...

class Bisection:
    def solve(self, f: Callable[[float], float], a: float, b: float) -> float:
        for _ in range(50):
            m = (a + b) / 2
            if f(a) * f(m) <= 0:
                b = m
            else:
                a = m
        return (a + b) / 2

class Newton:
    def solve(self, f: Callable[[float], float], a: float, b: float) -> float:
        x = (a + b) / 2
        for _ in range(20):
            fx = f(x)
            d = (f(x + 1e-6) - fx) / 1e-6
            x = x - fx / d
        return x

class EquationSolver:
    def __init__(self, method: RootFindingMethod):
        self.m = method
    def solve(self, f: Callable[[float], float], a: float, b: float) -> float:
        return self.m.solve(f, a, b)
```

Checklist DIP:

- Czy klasy wysokopoziomowe znają tylko interfejsy?
- Czy zależności są dostarczane z zewnątrz (konstruktor, setter, parametr metody)?

Wskazówki praktyczne i anty-wzorce

- Unikaj „Boga-klasy” (god object). SRP i ISP pomagają ją rozdrobnić.
- Zamiast instrukcji `switch / if-else` po typie — rozważ polimorfizm (OCP).
- Nie nadużywaj dziedziczenia — preferuj kompozycję (często pomaga w LSP i OCP).
- Wprowadzaj kontrakty/abstrakcje tam, gdzie zależą od nich moduły wyżej (DIP).
- Zadbaj o spójny, minimalny interfejs publiczny (ISP + SRP).

Szybkie mini-zadania (dla studentów)

1. SRP: Podziel klasę, która jednocześnie liczy, formatuje i zapisuje dane.
2. OCP: Usuń `if(type) z` procesora płatności, wprowadź strategię.
3. LSP: Przerób hierarchię tak, by zamienić bazę na interfejs `Shape` bez setterów.
4. ISP: Rozbij „gruby” interfejs repozytorium na mniejsze role.
5. DIP: Wstrzyknij interfejs logowania do serwisu zamiast używać `System.out / print` bezpośrednio.

Podsumowanie

- SOLID to zestaw heurystyk projektowych, nie dogmat. Stosuj, gdy upraszcza kod i ułatwia zmianę.
- Każdą zasadę możesz wdrożyć stopniowo. Refaktoryzuj, gdy pojawia się ból zmiany/testowania.

Dodatkowa teoria: pogłębienie SOLID

Skąd wzięło się SOLID (kontekst historyczny)

- Akronim SOLID spopularyzował Robert C. Martin (Uncle Bob) około 2000–2003 r., porządkując wcześniejsze prace m.in. B. Meyer (Design by Contract), M. Fowler (Refactoring), K. Beck (XP/TDD).
- Celem było dostarczenie zestawu praktycznych heurystyk ułatwiających zmienność i testowalność oprogramowania w środowisku zwinnego rozwoju.
- Zasady nie są prawami matematycznymi — to skróty myślowe, które pomagają podejmować decyzje projektowe.

Definicje i interpretacje praktyczne

- SRP: Jedna odpowiedzialność = jedna przyczyna zmiany. W praktyce: wysoka kohezja klasy/modułu, spójna nazwa, minimalny interfejs publiczny. Granice często wynikają z języka domeny (DDD: kontekst ograniczony).
- OCP: Zmieniaj zachowanie przez rozszerzanie (nowe implementacje, strategie, plug-iny), a nie przez edycję sprawdzonego kodu. W praktyce: stabilne API, punkty rozszerzeń, wyraźna separacja modułów.
- LSP: Zamienność typów. W praktyce: nie dodawaj wymagań w podklasie, których nie ma klasa bazowa; nie łam kontraktów (pre/postwarunki, niezmienniki). Preferuj kompozycję, gdy dziedziczenie kusi, ale łamie kontrakt.
- ISP: Małe, wyspecjalizowane interfejsy zamiast jednego „tłustego”. W praktyce: interfejsy zorientowane na role (role-based), zależności wyrażone tak, aby konsumenci nie byli zmuszani do metod, których nie

użyją.
- DIP: Kierunek zależności ku abstrakcjom. W praktyce: moduły wysokopoziomowe zależą od interfejsów/protokołów; konfiguracja i integracja dostarczają implementacje (DI), co ułatwia testy i wymienialność.

Jakie problemy (pain points) rozwiązuje SOLID

- Zmiany kaskadowe (shotgun surgery) — SRP, ISP, DIP.
- Krucha architektura i ryzyko regresji — OCP, DIP.
- Niezgodność kontraktów, zaskakujące wyjątki — LSP.
- Skomplikowane zależności i trudne testy — DIP, ISP.

Trade-offy i anty-dogmatyzm

- Nadmierna abstrakcja (overengineering): zbyt wiele interfejsów/warstw przy małej skali (narusza YAGNI). Objawy: dużo klas „przekaźników", trudna nawigacja.
- Przedwczesna generalizacja pod OCP potrafi utrudnić proste zmiany. Najpierw prosta implementacja, dopiero potem wyodrębnianie strategii, gdy pojawią się realne warianty.
- SRP a ergonomia: zbyt drobna granulacja może utrudnić zrozumienie i zwiększyć koszty orkiestracji.

Code smells powiązane z SOLID i taktyki refaktoryzacji

- SRP: God Object, Divergent Change, Shotgun Surgery, Feature Envy.
  - Taktyki: Extract Class/Module, Extract Function, Facade, rozdzielenie warstw IO/formatowania/logiki.
- OCP: Switch/if-else po typie, Rigid Design, Primitive Obsession.
  - Taktyki: Strategy/Policy, Registry/Plugin, Template Method, State, Polimorfizm zamiast instrukcji warunkowych.
- LSP: Refused Bequest, Broken Hierarchy, Throwing Unexpected Exceptions.
  - Taktyki: Prefer composition over inheritance, final/immutable types, wydzielenie wspólnego interfejsu zamiast dziedziczenia impl.
- ISP: Fat Interface, Interface Bloat, Unused Parameters/Methods.
  - Taktyki: Segregate interfaces (role-based), Adapter, Event/Observer do odsprzęglenia sygnałów.
- DIP: Concrete Dependency, Hidden Dependency (Service Locator), Hard-wired new.
  - Taktyki: Dependency Injection (konstruktor, setter, parametr), wprowadzenie interfejsów/protokołów, Inversion of Control (kontener DI).

Interakcje i potencjalne konflikty zasad

- SRP vs OCP: zbyt drobne SRP może utrudnić rozszerzalność bez orkiestracji. Równoważ: stabilne punkty rozszerzeń w modułach o spójnej odpowiedzialności.
- ISP vs LSP: segmentując interfejsy nie wprowadzaj podinterfejsów z węższymi kontraktami, które nie są zamienne w miejscach oczekujących bazowego kontraktu.
- DIP wspiera OCP: zależność od abstrakcji pozwala wstrzyknąć nowe warianty bez modyfikacji.

Kontekst architektoniczny: gdzie SOLID pasuje

- Clean/Hexagonal Architecture: DIP na granicy aplikacja–infrastruktura (porty i adaptery). SRP i ISP pomagają określić porty, OCP ułatwia dodawanie adapterów.
- DDD (Domain-Driven Design): SRP przy wyznaczaniu agregatów i serwisów domenowych; OCP w politykach domenowych; LSP przy kontraktach domenowych; DIP w warstwie aplikacyjnej.

Specyfika językowa (Java i Python)

- Java: bogate wsparcie interfejsów i frameworków DI (Spring, CDI). Wymusza czytelne zależności (konstruktor/setter) i ułatwia testy przez mocki. `final` pomaga w utrzymaniu kontraktów LSP.
- Python: duck typing i `typing.Protocol` zamiast interfejsów. Abstrakcje przez `abc.ABC` lub protokoły; DI zwykle ręcznie (przekazywanie zależności) lub lekkie kontenery.

Przykład (Python `Protocol` dla DIP/ISP):

```
from typing import Protocol

class PaymentGateway(Protocol):
    def charge(self, amount: float) -> None: ...

class StripeGateway:
    def charge(self, amount: float) -> None:
        print("stripe", amount)

class OrderService:
    def __init__(self, gateway: PaymentGateway):
        self.gateway = gateway
    def place_order(self, amount: float) -> None:
        self.gateway.charge(amount)
```

Przykład (Java — konstruktorowe DI bez frameworka):

```
interface Clock { java.time.Instant now(); }
class SystemClock implements Clock { public java.time.Instant now(){ return java.time.Instant.
class ReportService {
    private final Clock clock;
    ReportService(Clock clock){ this.clock = clock; }
    public String header(){ return "Generated at " + clock.now(); }
}
```

Najczęstsze nieporozumienia i mity

- „SRP = jedna funkcja w klasie” — nie. Chodzi o jedną przyczynę zmiany, nie o liczbę metod.
- „OCP = wszędzie interfejsy” — nie. Wprowadzaj abstrakcje tam, gdzie realnie oczekujesz wariantów.
- „LSP dotyczy tylko dziedziczenia” — dotyczy każdego kontraktu typu; także interfejsów/protokołów.
- „ISP zwiększa liczbę interfejsów = gorsza złożoność” — więcej, ale mniejszych i trafniejszych interfejsów upraszcza zmiany.
- „DIP wymaga kontenera DI” — nie, wystarczy przekazywanie zależności w konstruktorze.

- „SOLID zawsze skraca kod” — czasem zwiększa liczbę klas, ale zmniejsza sprzężenia i koszt zmiany.

Mini case studies (skrócone)

1. Moduł płatności: zastąpienie `if(type)` strategią (OCP) + wstrzyknięcie interfejsu bramki (DIP) skróciło testy integracyjne o 50% i umożliwiło testy z `FakeGateway` bez sieci.
2. Raportowanie: rozdzielenie zbierania danych, kalkulacji i zapisu (SRP) obniżyło konflikty mergowe między zespołami pracującymi nad formatem a logiką o 70%.
3. Hierarchia figur: usunięcie dziedziczenia `Square extends Rectangle` na rzecz wspólnego `Shape` (LSP) uprościło kontrakty i testy brzegowe.

Pytania kontrolne / quiz (z krótkimi odpowiedziami)

1. Co oznacza „jedna przyczyna zmiany” w SRP? — Zmiana jednego aspektu domeny powinna dotykać jednej klasy/modułu.
2. Jak OCP wpływa na regresje? — Zmiany realizujesz przez dodawanie nowych implementacji, nie ruszając starego, przetestowanego kodu.
3. Podaj przykład naruszenia LSP. — Podklasa zaostrza prewarunek (np. rzuca wyjątek dla danych akceptowanych przez bazę).
4. Dlaczego ISP pomaga w testach? — Mniejsze interfejsy łatwiej zamockować i izolować w testach.
5. Jak DIP wspiera wymienialność? — Moduł zależy od abstrakcji, implementacje można podmieniać bez zmian klienta.
6. Kiedy unikać nadmiernego OCP? — Gdy nie ma realnych wariantów; stosuj YAGNI.
7. Jaki zapach kodu sugeruje naruszenie SRP? — God Object, Divergent Change.
8. Jaką taktykę wybrać zamiast kaskady if-ów po typie? — Strategy/Registry + polimorfizm.
9. Co w Javie pomaga w egzekwowaniu LSP? — Testy kontraktów, klasy `final`, niemodyfikowalność stanu, jawne interfejsy.
10. Jak w Pythonie wyrazić kontrakt bez interfejsów? — `typing.Protocol` lub ABC + testy zachowania.

Źródła i dalsza lektura

- R. C. Martin: Agile Software Development, Principles, Patterns, and Practices; Clean Architecture.
- M. Fowler: Refactoring, Patterns of Enterprise Application Architecture.
- B. Meyer: Object-Oriented Software Construction (Design by Contract).
- Sandi Metz: Practical Object-Oriented Design (POODR).
- Artykuły: „The Principles of OOD” (Uncle Bob), katalog refaktoryzacji (refactoring.guru), dokumentacja `typing.Protocol` (Python), Spring Framework Reference (DI).

Krótka oś czasu i geneza (timeline)

- 1988–1997 — Bertrand Meyer formułuje Design by Contract (pre/postwarunki, niezmienniki) i promuje silne kontrakty typów — fundament LSP i myślenia o poprawności.
- 1994 — GoF „Design Patterns”: polimorfizm, kompozycja ponad dziedziczenie — paliwo dla OCP/ISP/DIP.
- 1999–2003 — Ekstremalne Programowanie (Kent Beck), Refactoring (Martin Fowler), testy automatyczne stają się normą — SRP/DIP zyskują praktyczny sens (testowalność).

- 2000–2003 — Robert C. Martin systematyzuje „zasady OOD”, akronim „SOLID” popularyzuje się w społeczności.
- 2010+ — Clean Architecture, Hexagonal/Onion, DDD w mainstreamie — DIP jako główna oś separacji domeny i infrastruktury.

Krótka historia i intuicja każdej zasady

- SRP: intuicja „jedna odpowiedzialność = jedna przyczyna zmiany” wywodzi się z dążenia do wysokiej kohezji modułu. Gdy różne decyzje biznesowe dotykają jednego pliku — cierpi przewidywalność zmian.
- OCP: obserwacja, że stabilne API + polimorfizm pozwalają dodawać nowe warianty bez ruszania starego kodu, co ogranicza regresje.
- LSP: ukonkretnienie myśli Meyera o kontraktach — jeśli typ B jest A, to musi respektować jego kontrakty, aby kod kliencki był nieświadomy różnicy.
- ISP: wynik doświadczeń z „tłustymi” interfejsami — konsumenci nie powinni być zmuszani do metod, których nie użyją; interfejs ma odzwierciedlać rolę, nie wszystko, co „możliwe”.
- DIP: odwrócenie zależności stabilizuje architekturę — moduły wyżej nie zależą od szczegółów technologii niżej; to umożliwia testy i wymienialność.

Sens stosowania i kiedy nie stosować (praktyka)

- SRP
  - Stosuj, gdy: zmiana jednego aspektu domeny wymusza dotykanie wielu miejsc w tej samej klasie (Divergent Change), gdy klasa łączy IO, formatowanie i logikę.
  - Uważaj, gdy: za drobna granulacja utrudnia ogarnięcie przepływu (zbyt wiele małych klas); łącz związane operacje w moduł o jednej odpowiedzialności.
  - Checklista: Czy istnieje jedna dominująca rola? Czy zmiana formatu danych wymaga zmiany logiki? Czy nazwa klasy precyzyjnie opisuje zawartość?
- OCP
  - Stosuj, gdy: często dodajesz warianty (np. strategie, adaptery, polityki), a stary kod powinien pozostać nietknięty.
  - Uważaj, gdy: nie ma realnych wariantów (YAGNI). Wczesna generalizacja utrudnia proste zmiany.
  - Checklista: Czy nowe zachowanie możesz dostarczyć przez nową implementację interfejsu? Czy unikasz edycji stabilnych klas?
- LSP
  - Stosuj, gdy: masz hierarchie typów i chcesz gwarantować zamienność (testy kontraktów, brak zaostrzania prewarunków).
  - Uważaj, gdy: dziedziczenie wymusza zbyt silne związki — wybierz kompozycję.
  - Checklista: Czy podklasa nie rzuca nowych wyjątków dla danych akceptowanych przez bazę? Czy nie zawęża zwracanych wartości/efektów ubocznych?
- ISP
  - Stosuj, gdy: konsumenci używają rozłącznych podzbiorów metod; mockowanie jest trudne przez „grube” interfejsy.
  - Uważaj, gdy: rozdzielając, tworzysz zbyt wiele podobnych interfejsów — grupuj naturalne role.
  - Checklista: Czy klient implementacji potrzebuje wszystkich metod? Czy istnieją naturalne role (Saver/Loader itd.)?

- DIP
  - Stosuj, gdy: warstwy wyższego poziomu zależą od technologii (HTTP/DB/filesystem); chcesz testować w izolacji.
  - Uważaj, gdy: prosta aplikacja o małej skali — nadmiar DI/abstrakcji może spowolnić rozwój.
  - Checklista: Czy moduł zna tylko interfejs? Czy konkret dostarczany jest z zewnątrz (konstruktor/parametr)?

SOLID a GRASP (krótko)

- GRASP (General Responsibility Assignment Software Patterns) to zbiór wzorców przydziału odpowiedzialności (m.in. Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations).
- Relacje:
  - SRP ≈ High Cohesion (GRASP) — jedna odpowiedzialność sprzyja spójności.
  - OCP ≈ Protected Variations — izolujemy zmienne elementy za stabilnym interfejsem.
  - DIP ≈ Indirection — wprowadzamy pośrednictwo (abstrakcje/porty), by odsprzęgnąć warstwy.
  - ISP wspiera Low Coupling — konsumenci widzą tylko potrzebne role.
  - LSP wzmacnia Polymorphism — zamienność bez łamania kontraktu.

Metryki i wskaźniki wspierające decyzje

- Kohezja/coupling: staraj się o wysoką kohezję (SRP/ISP) i niskie sprzężenie (DIP/OCP).
- Testowalność: możliwość podmiany zależności (DIP) i mockowania małych interfejsów (ISP) to szybkie testy jednostkowe.
- Stabilność pakietów (Robert C. Martin): zależności powinny kierować się ku stabilniejszym modułom (abstrakcje częściej są stabilne).

Krótki przewodnik decyzyjny (cheatsheet)

1. Czy ból zmiany wynika z duplikacji? — DRY (wyodrębnij, ujednolić).
2. Czy ból zmiany wynika z „instrukcji po typie”? — OCP (Strategy/State/Registry).
3. Czy ból zmiany wynika z niezamiennych podklas? — LSP (kompozycja, kontrakty, testy).
4. Czy implementacje są zmuszane do „pustych metod”? — ISP (segregacja interfejsów).
5. Czy warstwa aplikacji zna szczegóły infrastruktury? — DIP (interfejsy/porty + wstrzyknięcie).
6. Czy rozwiązanie jest za ciężkie jak na problem? — KISS/YAGNI (uprość; odłóż generalizację).

Zasady uzupełniające: DRY, KISS, YAGNI

Choć SOLID dotyczy głównie projektowania obiektowego, w praktyce codziennej warto stosować uzupełniające heurystyki: DRY, KISS i YAGNI. Pomagają one utrzymać kod krótki, czytelny i odporny na zmiany, a także chronią przed nadinżynierią. Te trzy zasady dobrze uzupełniają SRP/OCP/LSP/ISP/DIP i pomagają podejmować pragmatyczne decyzje w zespole.

DRY — Don't Repeat Yourself

„Nie powtarzaj się” — wiedza (logika biznesowa, reguły, formaty) powinna mieć jedno źródło prawdy. DRY dotyczy nie tylko duplikacji kodu, ale również duplikacji wiedzy w dokumentacji, testach i konfiguracji.

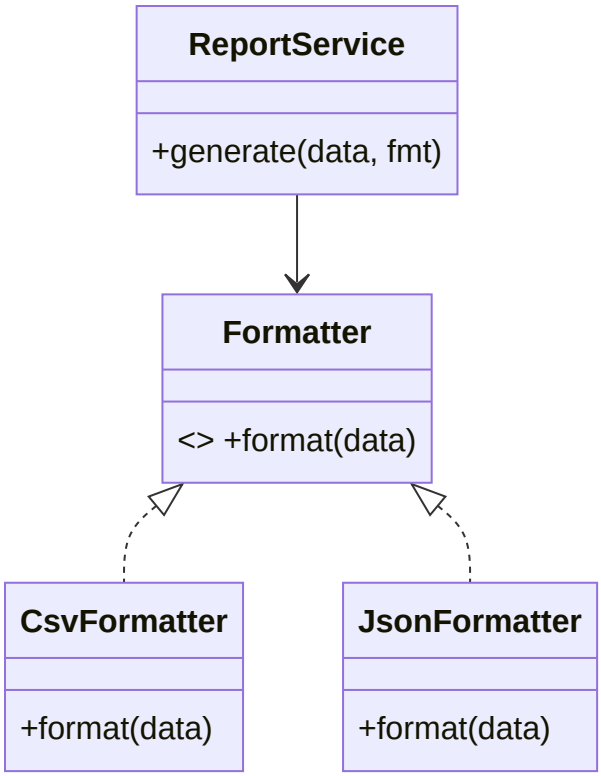
Kiedy łamiemy DRY:

- kopiuj-wklej podobnych fragmentów (copypasta),
- powielone stałe, formaty, walidacje w wielu miejscach,
- duplikacja SQL/JSON/konfiguracji w wielu plikach bez wspólnego modułu/wzorca,
- duplikacja testów pokrywających ten sam przypadek inną drogą.

Korzyści DRY:

- mniej miejsc do zmiany przy nowym wymaganiu (mniejszy koszt utrzymania),
- spójność zachowania, mniej rozjazdów i regresji,
- krótszy, bardziej modułowy kod (często współgra z SRP i OCP).

UML (refaktoryzacja pod DRY: Strategia zamiast duplikacji algorytmu):



Zły przykład 1 (Java): powtórzona logika formatowania w dwóch metodach

```
class InvoicePrinter {
    public String printCsv(List<Invoice> invoices) {
        StringBuilder sb = new StringBuilder();
        sb.append("id,amount\n");
        for (Invoice i : invoices) {
            // LOGIKA FORMATOWANIA #1 (duplikacja)
```

```
            sb.append(i.getId()).append(',').append(i.getAmount());
            sb.append('\n');
        }
        return sb.toString();
    }

    public String printCsvPaidOnly(List<Invoice> invoices) {
        StringBuilder sb = new StringBuilder();
        sb.append("id,amount\n");
        for (Invoice i : invoices) {
            if (!i.isPaid()) continue;
            // LOGIKA FORMATOWANIA #2 (druga kopia)
            sb.append(i.getId()).append(',').append(i.getAmount());
            sb.append('\n');
        }
        return sb.toString();
    }
}
```

Lepszy przykład 1 (Java): ekstrakcja wspólnej funkcji/strategii

```
class InvoiceCsvFormatter {
    public void appendRow(StringBuilder sb, Invoice i) {
        sb.append(i.getId()).append(',').append(i.getAmount()).append('\n');
    }
}

class InvoicePrinter {
    private final InvoiceCsvFormatter fmt = new InvoiceCsvFormatter();

    public String printCsv(List<Invoice> invoices) {
        StringBuilder sb = new StringBuilder();
        sb.append("id,amount\n");
        for (Invoice i : invoices) fmt.appendRow(sb, i);
        return sb.toString();
    }

    public String printCsvPaidOnly(List<Invoice> invoices) {
        StringBuilder sb = new StringBuilder();
        sb.append("id,amount\n");
        for (Invoice i : invoices) if (i.isPaid()) fmt.appendRow(sb, i);
        return sb.toString();
    }
}
```

Zły przykład 2 (Python): zduplikowane reguły walidacji w kilku miejscach

```
def register_user(user):
    if not user.email or "@" not in user.email:
        raise ValueError("Invalid email")  # duplikacja
    # ... zapis
```

```python
def send_newsletter(user):
    if not user.email or "@" not in user.email:
        raise ValueError("Invalid email")  # duplikacja
    # ... wysyłka
```

Lepszy przykład 2 (Python): pojedyncze źródło walidacji

```python
class UserValidator:
    def validate_email(self, email: str) -> None:
        if not email or "@" not in email:
            raise ValueError("Invalid email")

validator = UserValidator()

def register_user(user):
    validator.validate_email(user.email)
    # ... zapis

def send_newsletter(user):
    validator.validate_email(user.email)
    # ... wysyłka
```

Zły przykład 3 (Java): zduplikowane stałe i regexy

```java
class UserForm {
    private static final String EMAIL_REGEX = ".+@.+\\..+"; // kopia #1
    // ... użycie
}
class NewsletterForm {
    private static final String EMAIL_REGEX = ".+@.+\\..+"; // kopia #2
    // ... użycie
}
```

Lepszy przykład 3 (Java): współdzielone stałe w jednym miejscu

```java
final class Regexes {
    private Regexes() {}
    public static final String EMAIL = ".+@.+\\..+";
}

class UserForm { /* używa Regexes.EMAIL */ }
class NewsletterForm { /* używa Regexes.EMAIL */ }
```

Checklist DRY:

- Czy konkretną regułę/format/algorytm zmieniasz w jednym miejscu?
- Czy testy nie powielają tego samego przypadku logicznego tylko inną ścieżką?
- Czy konfiguracja (np. endpointy, klucze) nie jest duplikowana w wielu modułach?

Mini-zadania DRY:

- Wyodrębnij wspólny kod budowania CSV z kilku metod do jednej klasy formatującej.
- Zastąp powtarzające się walidacje użytkownika wywołaniem `UserValidator`.

KISS — Keep It Simple, Stupid

„Utrzymuj prostotę.” Preferuj proste rozwiązania działające dziś, zamiast skomplikowanych i ogólnych na hipotetyczne jutro. Prostota zmniejsza koszt poznawczy i ryzyko błędów.

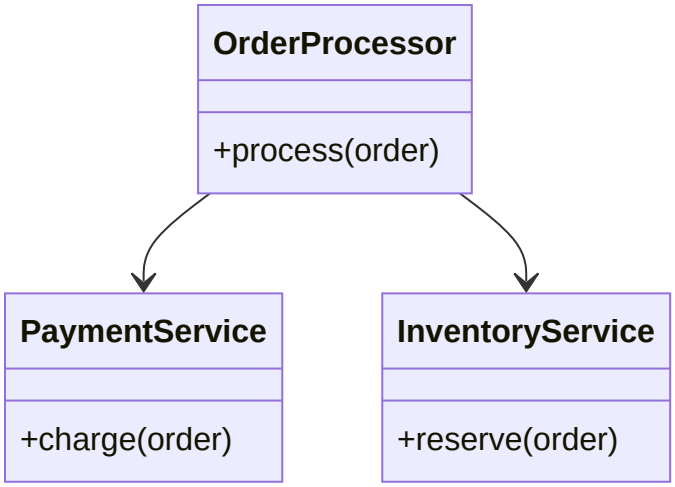
Złożoność niezbędna vs. przypadkowa:

- niezbędna — wynika z domeny; nie usuniesz jej bez utraty funkcji,
- przypadkowa — wynika z projektu/implementacji; możesz ją zredukować.

Heurystyki KISS:

- małe, spójne funkcje i klasy; preferuj kompozycję nad rozbudowane dziedziczenie,
- unikaj nadmiarowych wzorców, metaprogramowania i refleksji, gdy zwykły `if` wystarczy,
- pisz jawny kod zamiast „magii”; dodaj komentarz, nie framework,
- optymalizuj po pomiarze, nie przed (premature optimization szkodzi).

UML (upraszczanie — redukcja zbędnej hierarchii na rzecz kompozycji):



Zły przykład 1 (Java): nadmierna ogólność i dziedziczenie

```
abstract class AbstractProcessor<T> {
    protected abstract void before(T t);
    protected abstract void doProcess(T t);
    protected abstract void after(T t);
}

class OrderProcessor extends AbstractProcessor<Order> {
```

```
    protected void before(Order o) { /* ... */ }
    protected void doProcess(Order o) { /* ... */ }
    protected void after(Order o) { /* ... */ }
}
```

Lepszy przykład 1 (Java): bez zbędnej hierarchii, prosta kompozycja

```
class OrderProcessor {
    private final PaymentService payment;
    private final InventoryService inventory;

    OrderProcessor(PaymentService p, InventoryService i) {
        this.payment = p; this.inventory = i;
    }

    public void process(Order o) {
        inventory.reserve(o);
        payment.charge(o);
    }
}
```

Zły przykład 2 (Python): skomplikowane meta-fabryki bez potrzeby

```
class ProcessorFactoryRegistry:
    _registry = {}
    @classmethod
    def register(cls, name, factory):
        cls._registry[name] = factory
    @classmethod
    def create(cls, name, **kwargs):
        return cls._registry[name](**kwargs)

def process_order(order):
    proc = ProcessorFactoryRegistry.create("order", config={"modes": ["x","y"]})
    proc.process(order)
```

Lepszy przykład 2 (Python): prosty, jawny kod

```
class OrderProcessor:
    def __init__(self, payment, inventory):
        self.payment = payment
        self.inventory = inventory

    def process(self, order):
        self.inventory.reserve(order)
        self.payment.charge(order)

def process_order(order, payment, inventory):
    OrderProcessor(payment, inventory).process(order)
```

Zły przykład 3 (Java): „sprytna” optymalizacja przedwcześnie

```
class Stats {
    // skomplikowane cache + wielowątkowość, choć dane mają 100 rekordów
    private final java.util.concurrent.ConcurrentMap<String, Double> cache = new java.util.con
    public double avg(List<Double> xs) {
        return xs.stream().mapToDouble(Double::doubleValue).average().orElse(0.0);
    }
}
```

Lepszy przykład 3 (Java): najprostsze rozwiązanie wystarczające dziś

```
class Stats {
    public double avg(List<Double> xs) {
        double sum = 0.0; for (double x : xs) sum += x; return xs.isEmpty() ? 0.0 : sum / xs.s
    }
}
```

Checklist KISS:

- Czy rozumiesz funkcję/klasę w 60 sekund? Jeśli nie — rozbij lub uprość.
- Czy istnieje prostsza struktura danych/algorytm dający ten sam efekt?
- Czy nie wdrożyłeś wzorca tylko po to, by „brzmiało dobrze” w kodzie?

Mini-zadania KISS:

- Usuń zbędne klasy bazowe i interfejsy, zastępując je prostą kompozycją.
- Zamień rejestr meta-fabryk na jawne wywołania konstruktorów.

YAGNI — You Aren't Gonna Need It

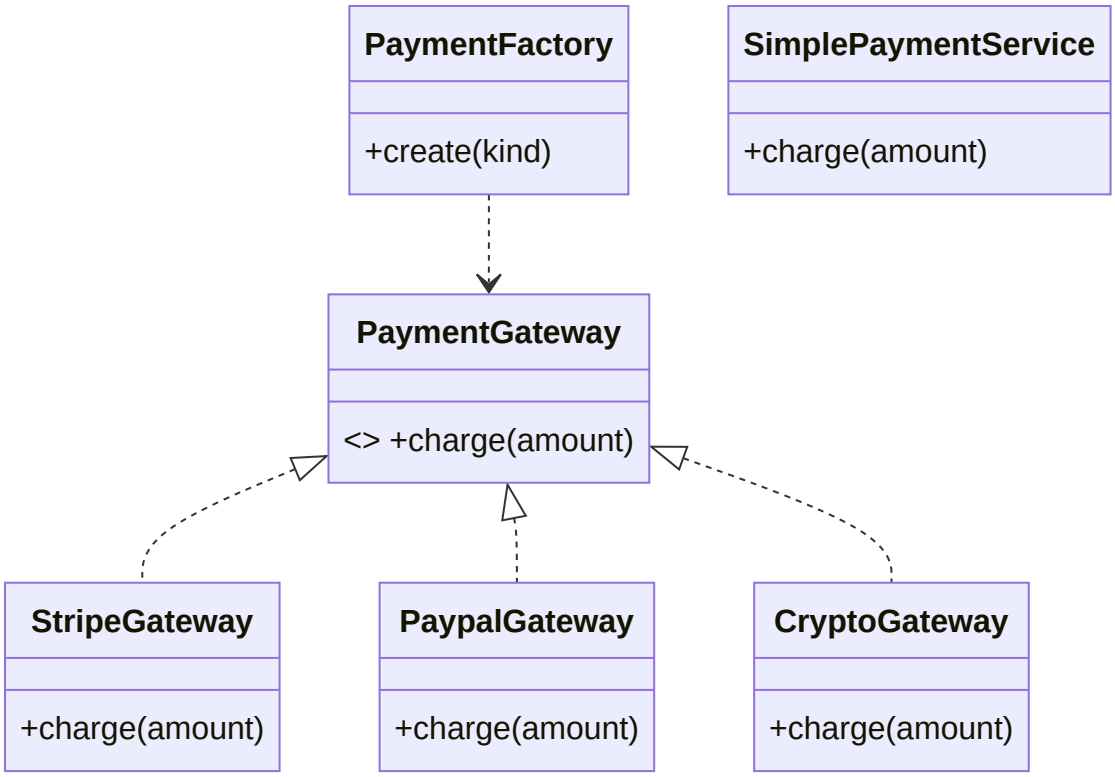
„Nie będziesz tego potrzebować.” Nie implementuj funkcji, rozszerzalności ani konfiguracji, dopóki nie ma realnej potrzeby (wymagania/klienta/testu).

Związek z SOLID:

- YAGNI równoważy OCP: nie twórz ogólnych punktów rozszerzeń bez sygnału z domeny,
- pomaga trzymać KISS: mniej warstw/abstrakcji, dopóki nie są konieczne,
- wspiera SRP: każda dodatkowa odpowiedzialność to potencjalny powód zmiany.

Kiedy ignorować YAGNI: gdy koszt późniejszej zmiany jest ekstremalny (np. wybór niezmiennego formatu danych publicznych API) — wtedy projektuj ostrożnie, ale wciąż minimalnie.

UML (przed YAGNI: nadmiar abstrakcji) vs (po YAGNI: tylko potrzebne elementy):



Zły przykład 1 (Java): fabryki i pluginy bez konsumentów

```
interface Exporter { void export(Object data); }
class PdfExporter implements Exporter { public void export(Object d){ /* ... */ } }
class CsvExporter implements Exporter { public void export(Object d){ /* ... */ } }
class ExporterFactory { Exporter create(String kind){ /* if-else */ return new PdfExporter();

class ReportService {
    public void generate(Object data){
        // zawsze PDF, ale mamy nieużywaną fabrykę i CSV
        new PdfExporter().export(data);
    }
}
```

Lepszy przykład 1 (Java): tylko to, co potrzebne teraz

```
class ReportService {
    public void generate(Object data){
        new PdfExporter().export(data); // dodamy CSV, gdy zajdzie potrzeba
    }
}
```

Zły przykład 2 (Python): rozbudowana konfiguracja feature-flag bez użycia

```
FEATURES = {"dark_mode": False, "beta_checkout": False, "ml_recommendations": False}

def render_home():
    if FEATURES["ml_recommendations"]:
        pass  # brak implementacji, gałąź martwa
    return "Home"
```

Lepszy przykład 2 (Python): usuń martwy kod, dodaj gdy realnie potrzebny

```
def render_home():
    return "Home"  # dodamy rekomendacje, gdy pojawi się wymaganie
```

Zły przykład 3 (Java): nadmiar parametrów konfiguracyjnych bez sensu

```
class Cache {
    Cache(int shards, boolean enableJmx, boolean enableMetrics, boolean allowDiskSpill, boolea
}
```

Lepszy przykład 3 (Java): minimalny konstruktor, reszta później

```
class Cache {
    Cache(int capacity) { /* ... */ }
}
```

Checklist YAGNI:

- Czy ta abstrakcja/funkcja ma konkretnego użytkownika dziś? Jeśli nie — odłóż.
- Czy powód dodania parametru/warstwy jest mierzalny (wydajność, bezpieczeństwo) i udowodniony?
- Czy możesz dostarczyć prostszą wersję i rozszerzyć ją iteracyjnie?

Mini-zadania YAGNI:

- Usuń nieużywane parametry/feature-flagi i martwe gałęzie kodu.
- Zastąp fabrykę jednego rodzaju prostą instancją; przy drugim rodzaju wprowadź dopiero OCP.

Podsumowanie DRY/KISS/YAGNI

- DRY redukuje koszt zmiany przez jedno źródło prawdy.
- KISS minimalizuje złożoność przypadkową — wybieraj proste konstrukcje.
- YAGNI broni przed nadinżynierią — implementuj tylko to, co potrzebne.

Te zasady wzajemnie się wzmacniają i równoważą SOLID. Stosuj je pragmatycznie: najpierw prostota i potrzeba, potem ewentualne uogólnienia i punkty rozszerzeń.