

Metody testowania oprogramowania

Style programowania

Ukrywanie implementacji



```
public class Point {  
    public double x;  
    public double y;  
}
```

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

```
public class Point {  
    private double x;  
    private double y;  
  
    public double getX() {  
        return x;  
    }  
  
    public void setX(double x) {  
        this.x = x;  
    }  
  
    public double getY() {  
        return y;  
    }  
  
    public void setY(double y) {  
        this.y = y;  
    }  
}
```

Ukrywanie implementacji



```
public interface Vehicle {  
    double getFuelTankCapacityInGallons();  
  
    double getGallonsOfGasoline();  
}
```

```
public interface Vehicle {  
    double getPercentFuelRemaining();  
}
```

Dane <-> Obiekty



- Obiekty – ukrywają dane za abstrakcją i udostępniają funkcje na nich operujące.
- Struktury danych udostępniają dane i nie posiadają znaczących funkcji.

Strukturalny/proceduralny kształt



```
class Square {  
    public Point topLeft;  
    public double side;  
}
```

Struktura danych

```
class Rectangle {  
    public Point topLeft;  
    public double height;  
    public double width;  
}
```

```
class Circle {  
    public Point center;  
    public double radius;  
}
```

```
class Geometry {  
    public final double PI = 3.141592653589793;  
  
    public double area(Object shape) throws NoSuchShapeException {  
        if (shape instanceof Square) {  
            Square s = (Square) shape;  
            return s.side * s.side;  
        } else if (shape instanceof Rectangle) {  
            Rectangle r = (Rectangle) shape;  
            return r.height * r.width;  
        } else if (shape instanceof Circle) {  
            Circle c = (Circle) shape;  
            return PI * c.radius * c.radius;  
        }  
        throw new NoSuchShapeException();  
    }  
}
```

Procedura

Obiektowy kształt



```
public interface Shape {  
    double area();  
}
```

```
class Square implements Shape {  
    private Point topleft;  
    private double side;  
  
    public double area() {  
        return side * side;  
    }  
}
```

```
class Rectangle implements Shape {  
    private Point topleft;  
    private double height;  
    private double width;  
  
    public double area() {  
        return height * width;  
    }  
}
```

```
class Circle implements Shape {  
    private Point center;  
    private double radius;  
    public final double PI = 3.141592653589793;  
  
    public double area() {  
        return PI * radius * radius;  
    }  
}
```

Dychotomia obiektów i struktur danych



- Kod proceduralny wykorzystujący struktury danych pozwala na dodawanie funkcji bez zmiany struktur danych
- Kod obiektowy pozwala na dodanie nowych klas obiektów bez potrzeby zmieniania istniejących funkcji.

... to prowadzi nas do wniosków:



1. „Everything is an object” – to mit
2. Korzystając ze składni obiektowej możemy definiować różne style
 - Struktury danych – styl strukturalny
 - Obiekty – styl zorientowany obiektowo
- Ale również
 - Procedury – styl proceduralny
 - Funkcje – styl funkcyjny
- Ale nie hybrydy!!
 - To prowadzi do zapachów (ang. Code smell)

Obiekty czyli abstrakcje



- Eksponują zachowanie, hermetyzują dane
- Zastosowanie
 - Przydzielanie odpowiedzialności za realizację zadań – interfejs (API)
 - Sposób implementacji podatny na zmiany – hermetyzacja struktur danych
- Cechy
 - Łatwość rozbudowy przy zachowaniu interfejsu
 - Wysoka spójność (ang. High cohesion)
 - Luźne powiązania (ang. Loose coupling)

Obiekty - projektowanie



```
public class Client {  
    private String name;  
    private Id id;  
  
    public ClientData generateSnapshot() {..  
  
    public boolean canAfford(Money amount) {..  
  
        public Payment charge(Money amount) {..  
    }  
}
```

1. Za co obiekt jest odpowiedzialny (operacje)
2. Według jakich reguł działają operacje
3. Jakie dane są potrzebne do wykonania operacji

Struktury danych



- Nie posiadają zachowania – odpowiedzialności, eksponują struktury danych
- Zastosowanie
 - Reprezentacja (stabilnych) struktur danych
 - Operacje na danych wykonywane są przez inne części aplikacji
 - Reprezentacja stanu (np. ValueObject)

Struktury danych



```
public class ClientData {  
    private Id id;  
    private String name;  
  
    public ClientData(Id id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    public Id getId() {  
        return id;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

- Mogą być modyfikowalne, niemodyfikowalne

Procedury



- Posiadają wiele zależności, są bezstanowe (to przecież tylko procedura), mają efekty uboczne.
- Zastosowanie
 - Projektowanie API „na styku”
 - Modułów, podsystemów
 - Projektowanie usług (np. Web Services)
 - Modelowanie scenariuszy (np. przypadków użycia)
- Cechy
 - Implementowane jako bezstanowe klasy implementujące interfejs (API usługi)
 - Mogą być transakcyjne

Procedury

```
public class SimpleOrderingService implements OrderingService {  
    private SystemContext systemContext;  
  
    private ClientRepository clientRepository;  
  
    private ReservationRepository reservationRepository;  
  
    private ReservationFactory reservationFactory;  
  
    private PurchaseFactory purchaseFactory;  
  
    private PurchaseRepository purchaseRepository;  
  
    private ProductRepository productRepository;  
  
    private PaymentRepository paymentRepository;  
  
    private DiscountFactory discountFactory;  
  
    private SuggestionService suggestionService;  
  
    public Id createOrder() {..  
  
    public void addProduct(Id orderId, Id productId, int quantity) {..  
  
    public Offer calculateOffer(Id orderId) {..  
  
    public void confirm(Id orderId, Offer seenOffer) {..  
}
```

- To nie obiekty – reguły obiektowe niekoniecznie powinny być stosowane



Funkcje – styl funkcyjny



- Nie mają zależności, nie powodują efektów ubocznych -> zawsze zwracają ten sam rezultat dla tych samych argumentów, mogą przyjmować inną funkcję jako argument (domknięcie - closure).
- Zastosowania
 - Strategia, polityka, która musi być uwzględniona w algorytmie, usługi

Funkcje – styl funkcyjny



```
public class BookKeeper {  
  
    private InvoiceFactory invoiceFactory;  
  
    public Invoice issuance(InvoiceRequest invoiceRequest, TaxPolicy taxPolicy) {  
        Invoice invoice = invoiceFactory.create(invoiceRequest.getClientData());  
  
        for (RequestItem item : invoiceRequest.getItems()) {  
            Money net = item.getTotalCost();  
            Tax tax = taxPolicy.calculateTax(item.getProductData().getType(),  
                                             net);  
  
            InvoiceLine invoiceLine = new InvoiceLine(item.getProductData(),  
                                                     item.getQuantity(), net, tax);  
            invoice.addItem(invoiceLine);  
        }  
  
        return invoice;  
    }  
}
```


Wracamy do obiektowości



- Zasady

1. Hermetyzacja
2. Abstrakcja
3. Hierarchizacja
4. Modularyzacja

Command-query separation



- Metoda obiektu (operacja, którą można wykonać na obiekcie) powinna:
 - zmieniać stan obiektu (Command)
- lub (!)
 - zwracać stan obiektu (Query)
- Nie powinna robić jednego i drugiego

Naruszenie zasady Command-query separation



```
public interface MessageBuffer {  
    public List<Message> addMsg(Message msg);  
}
```

```
public interface MessageBuffer {  
    public void addMsg(Message msg);  
    public List<Message> getContent();  
}
```

Command-query separation

```
public class Client {  
    private String name;  
    private Id id;  
  
    public ClientData generateSnapshot() {..  
  
    public boolean canAfford(Money amount) {..  
  
    public Payment charge(Money amount) {..  
}
```

Prawo Demeter



- Metoda danego obiektu może odwoływać się jedynie do metod należących do:
 - tego samego obiektu,
 - dowolnego parametru przekazanego do niej,
 - dowolnego obiektu przez nią stworzonego,
 - dowolnego składnika klasy do której należy dana metoda.
 - dostępnych globalnych obiektów

Prawo Demeter



```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

```
Options opts = ctxt.getOptions();  
File scratchDir = opts.getScratchDir();  
final String outputDir = scratchDir.getAbsolutePath();
```

Train wreck

Method chain

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

```
ctx.getScratchDirectoryOption().getAbsolutePath();
```

Prawo Demeter



```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();

String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

```
BufferedOutputStream bos = ctxt.createScratchFileStream(className);
```

Prawo Demeter



```
human.getDigestionSystem().  
    getPeritoneum().getStomach().  
        add(new Sausage(2));
```

```
human.eat(new Sausage(2));
```

```
public void eat(Food f){  
    if (! iLike(f))  
        throw new IDontLikeItException(f);  
    this.digestionSystem.swallow(f);  
}
```


Zasady SOLID



- SRP - Single Responsibility
 - OCP - Open/Closed
 - LSP - Liskov Substitution
 - ISP - Interface Segregation
 - DIP - Dependency Inversion
- } Principle

Zasady GRASP



General Responsibility Assignment Software Patterns

- Wzorce przypisywania odpowiedzialności klasom obiektów w projektowaniu obiektowym.
- Controller, Creator, High Cohesion, Indirection, Information Expert, Low Coupling, Polymorphism, Protected Variations, Pure Fabrication

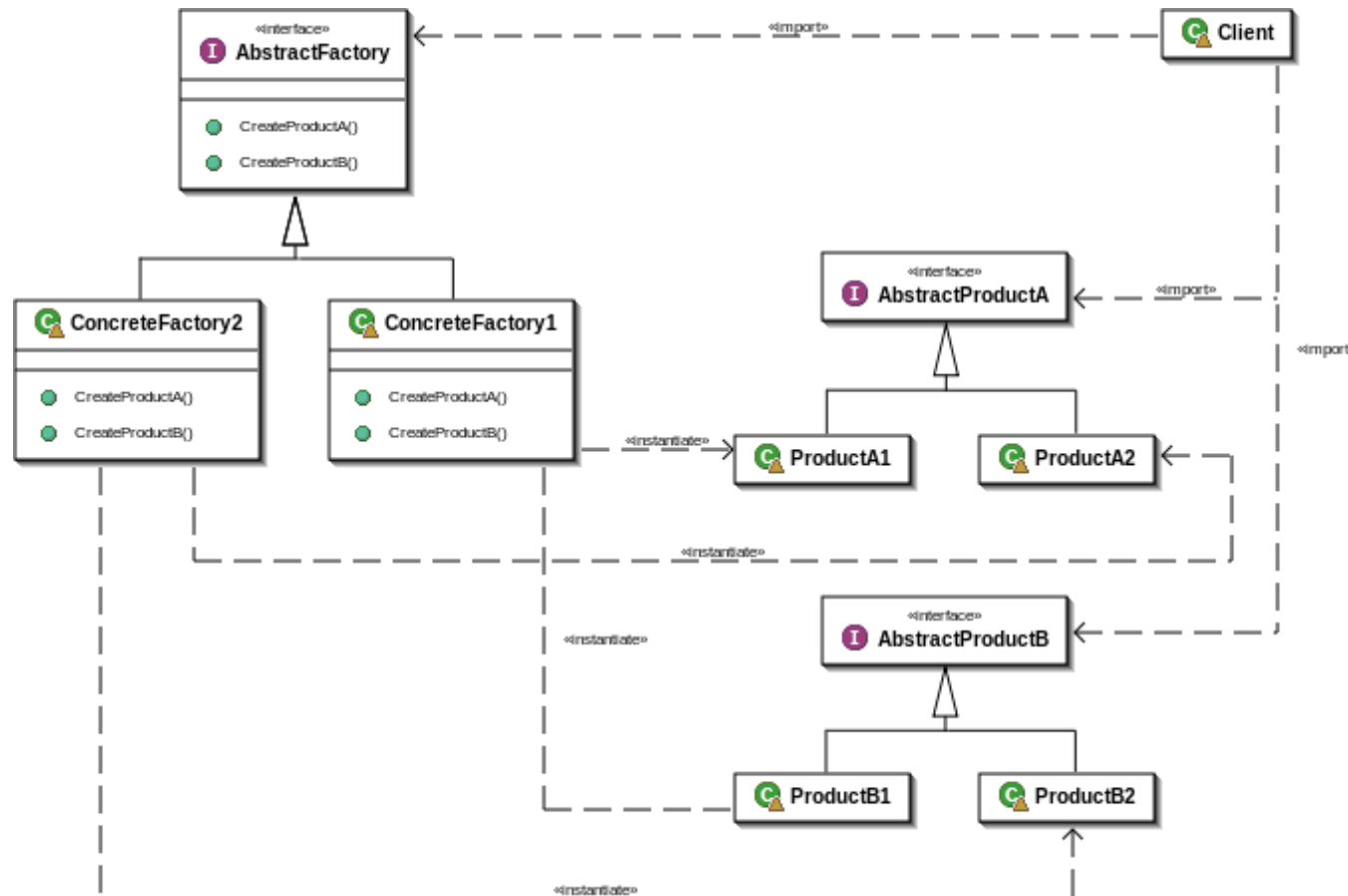
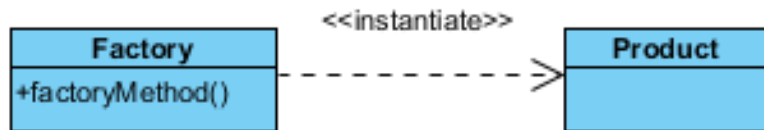
<http://www.cs.cmu.edu/~aldrich/214/slides/design-grasp.pdf>

Wzorce projektowe – wybrane i proste 😊



- Fabryki – Factories
 - Bezpośrednie wykorzystanie konstruktora w kodzie programu wymaga jawnego podania konkretnego typu. Wprowadza to silną zależność pomiędzy kodem klienta i kodem usługi
 - Fabryka jest najprostszym sposobem „wyciągnięcia przed nawias” procesu tworzenia konkretnego obiektu.
 - Izolujemy się tak od konkretnego typu jaki i mechanizmu zarządzania cyklem życia obiektu.
 - Do utworzenia obiektu bezpośrednio wykorzystywana jest tzw. metoda wytwarzająca (ang. Factory Method).

Fabryki w UML



Metoda wytwarzająca - przykład



```
public class PaymentFactory {  
    private DomainEventPublisher publisher;  
  
    public Payment createPayment(ClientData clientData, Money amount){  
  
        Id aggregateId = Id.generate();  
        publisher.publish(new ClientPaidEvent(aggregateId, clientData, amount));  
        return new Payment(aggregateId, clientData, amount);  
    }  
}
```

```
public Payment charge(Money amount) {  
    if (! canAfford(amount)){  
        domainError("Can not afford: " + amount);  
    }  
  
    return paymentFactory.createPayment(generateSnapshot(), amount);  
}
```

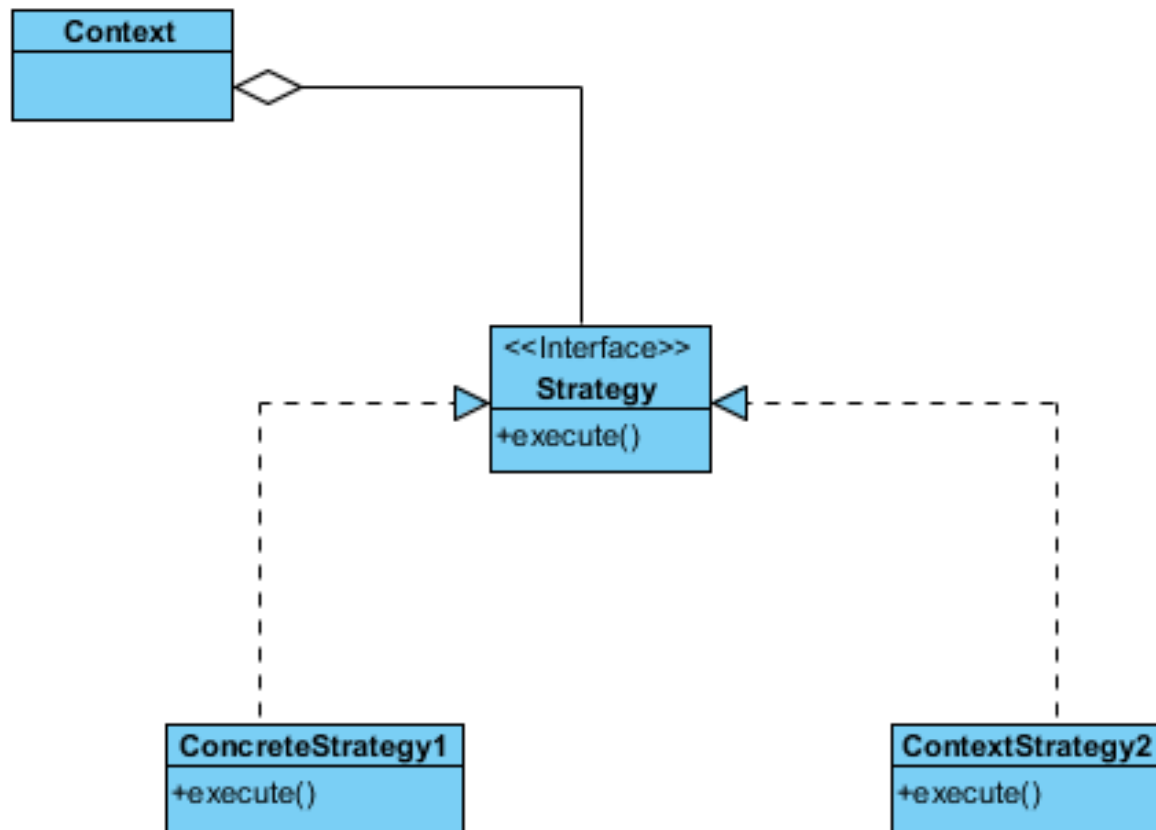
```
Payment payment = client.charge(purchase.getTotalCost());  
paymentRepository.save(payment);
```

Strategia



- Wsparcie dla zasady otwarte-zamknięte (SOLID).
- Izolacja zmienności algorytmu rozwiązania danego problemu.
 - Od sortowania
 - Po politykę przydzielania kredytu

Strategy w UML




Strategia – wykorzystanie w stylu funkcyjnym



```
public interface DiscountPolicy {  
  
    public Discount applyDiscount(Product product, int quantity, Money regularCost);  
}
```

```
public class QuantityDiscount implements DiscountPolicy {  
    private double rebateRatio;  
  
    private int minimalQuantity;  
  
    public QuantityDiscount(double rebate, int minimalQuantity) {  
        rebateRatio = rebate / 100;  
        this.minimalQuantity = minimalQuantity;  
    }  
  
    @Override  
    public Discount applyDiscount(Product product, int quantity,  
        Money regularCost) {  
        if (quantity >= minimalQuantity)  
            return new Discount("over: " + quantity,  
                regularCost.multiplyBy(rebateRatio));  
        return null;  
    }  
}
```

Domknięcie



```
public Offer calculateOffer(DiscountPolicy discountPolicy) {  
    List<OfferItem> availableItems = new ArrayList<OfferItem>();  
    List<OfferItem> unavailableItems = new ArrayList<OfferItem>();  
  
    for (ReservationItem item : items) {  
        if (item.getProduct().isAvailable()) {  
            Discount discount = discountPolicy.applyDiscount(item  
                .getProduct(), item.getQuantity(), item.getProduct()  
                .getPrice());  
            OfferItem offerItem = new OfferItem(item.getProduct()  
                .generateSnapshot(), item.getQuantity(), discount);  
  
            availableItems.add(offerItem);  
        } else {  
            OfferItem offerItem = new OfferItem(item.getProduct()  
                .generateSnapshot(), item.getQuantity());  
  
            unavailableItems.add(offerItem);  
        }  
    }  
  
    return new Offer(availableItems, unavailableItems);  
}
```


Deklaratywność języka



- Programowanie imperatywne
 - Określa algorytm rozwiązania
- Programowanie deklaratywne
 - Określa cel przetwarzania bez podania sposobu rozwiązania – algorytmu
- Styl deklaratywny
 - Sposób budowania konstrukcji językowych bazujący na wykorzystaniu konstrukcji specyficznych dla dziedziny (DSL), definiującego cel raczej niż sposób rozwiązania

Bibliografia



- Robert C. Martin: Czysty kod. Podręcznik dobrego programisty
- Sławomir Sobótka: [Przewodnik strukturyzacji architektury systemu. 10,5 klasycznych technik programistycznych leżących u podstaw nowoczesnej inżynierii oprogramowania](#)
- Sławomir Sobótka: [Receptury projektowe – niezbędnik początkującego architekta](#)
- <http://sourcemaking.com/refactoring/feature-envy>
- <http://www.martinfowler.com/bliki/AnemicDomainModel.html>