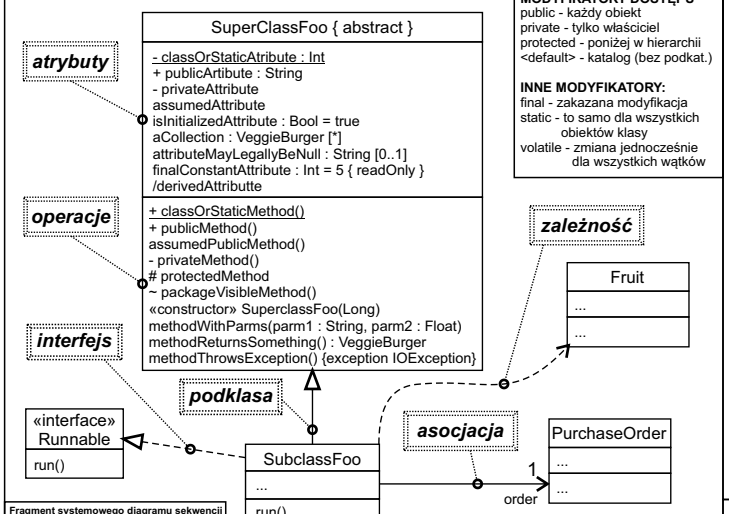
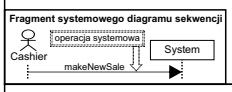


MODYFIKATORY DOSTĘPU
public - każdy obiekt
private - tylko właściciel
protected - poniżej w hierarchii
<default> - katalog (bez podkat.)

INNE MODYFIKATORY:
final - zakazana modyfikacja
static - to samo dla wszystkich obiektów klasy
volatile - zmiana jednocześnie dla wszystkich wątków

Fragment systemowego diagramu sekwencji



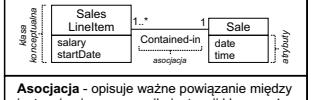
Parallelism - zadania wykonywane równolegle
Concurrency (współbieżnie) - dwa albo więcej zadań robi postępy, nawet jeśli nie są wykonywane równolegle

Walka ze złożonością dziedziny
- programowanie obiektowe
- OOA/D (Object Oriented Analysis and Design)

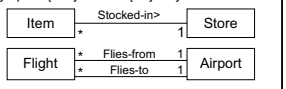
Walka ze złożonością projektu
- wzorce projektowe

Walka ze złożonością systemu
(np. naprawienie błędu powoduje inne)
- testowanie na bieżąco (JUnit)
- systemy kontroli wersji (SVN, Git)
- metodyki zwinne (np. RUP, Scrum)

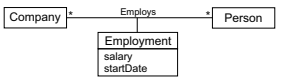
Model dziedziny - reprezentacja obiektów materialnych (klas konceptualnych), powiązań między nimi (asocjacji) i własności (atrybutów). Nie jest to diagram klas, tylko inspiracja.



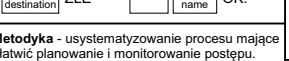
Asocjacja - opisuje ważne powiązanie między instancjami; wyznacza ile instancji klasy może być powiązanych z inną w jednym momencie.



Klasa asocjacyjna - klasa z atrybutami dla relacji wiele-do-wielu,gdzie każda asocjacja wymaga przechowywania osobnych informacji



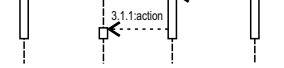
Atrybut - krótki tekst lub liczba, z którym nie są powiązane dodatkowe informacje. Częsty błąd:



Metodyka - usystematyzowanie procesu mające ułatwić planowanie i monitorowanie postępu.

Agile Manifesto (ang. zwinny)
ludzie i interakcje > procesy i narzędzia
działający program > obszerna dokumentacja
współpraca z klientem > formalne ustalenia
reagowanie na zmiany > podążanie za planem

Diagram sekwencji wzorca Command



$$\frac{1}{(1-P)+\frac{P}{N}}$$

Prawo Amdahla
P - ułamek podlegający zrównoległeniu, od 0 do 1
N - liczba procesorów

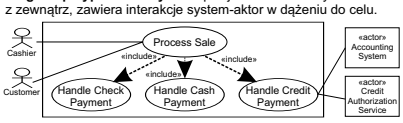
DOC - Depended On Components

Fake − gdy DOC niedostępny albo nie nadaje sie do testów tworzymy „lekką" klasę zastępczą
Dummy Object − dla testu bez znaczenia jaki DOC

```
LinkedList mockList = mock(List.class);
when(mockList.get(0)).thenReturn("first");
when(mockList.get(1)).thenThrow(new Exception());
when(mockList.get(anyInt())).thenReturn("X");
mockList.add("A")
verify(mockList).add("A")
```

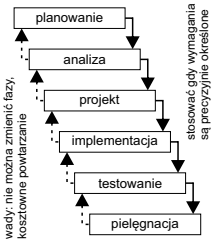
MOCKITO

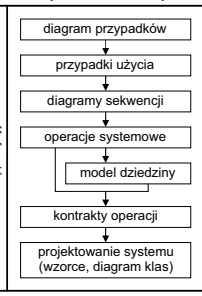
Diagram przypadków użycia - opisuje zachowanie systemu z zewnątrz, zawiera interakcje system-aktor w dążeniu do celu.



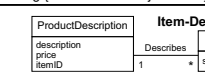
Akka - model asynchroniczny sterowany zdarzeniami. Klasy opakowane w aktorów. Aktor to kontener: State (pola pryw.), Behavior (metody pryw. i onRecive), Mailbox, Childrens, Supervisor Strategy. Komunikacja do mailbox odbiorcy, referencje ActorRef. Zmienne nie są współdzielone. System hierarchiczny.

Model kaskadowy - jeśli odkryto błędy, cofamy fazę.

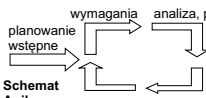




Najważniejsze cechy Agile:
- zwinność, adaptacja do zmian wymagań, współpraca z klientem
- zauważalne postępy, krótkie iteracje, istotna funkcjonalność
- brak prototypowania, szybko tworzone programy wysokiej jakości
- małe zespoły, bezpośredni kontakt, bez dokładnej dokumentacji
- uwzględnianie informacji zwrotnej od deweloperów



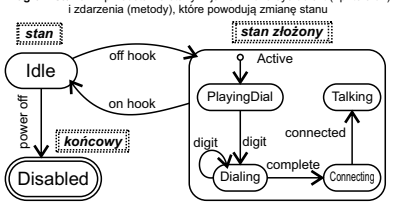
XP, Extreme Programming
- niewielkie projekty wysokiego ryzyka
- parami
- ciągłe modyfikacje
- test driven development



Schemat Agile

Rational Unified Process	*zrozumieć problem*	*zrozumieć rozwiązanie*	*stworzyć rozwiązanie*	*postawić system*
	Inception	Elaboration	Construction	Transition
business modeling *słownik, model dziedziny*				
requirements *przypadki użycia*				
design *wzorce projektowe*				
implementation *java, frameworki*				
test *JUnit, Mockito, JMeter*				

Diagram stanów - przedstawia stany w jakich może być obiekt (np. telefon) i zdarzenia (metody), które powodują zmianę stanu



Różnica między Abstract Factory a Builder
AF Produkty mają wspólny interfejs.
BU Produkty tworzone przez różnych budowniczych moga byc różne, wiec nie ma powodu by ustalać dla nich wspólny interfejs.
AF Akcent jest na tworzenie rodziny powiązanych produktów, kazdy produkt w jednym kroku.
BU Akcent jest na tworzenie złożonego produktu krok po kroku. Kroki mozemy omijać/modyfikować co daje elastyczność.
AF Klient używa metod Abstract Factory do stworzenia grupy produktów, ale czesto nie wie z jakiej fabryki korzysta.
BU Klient wybiera budowniczego, od którego zależy postać produktu i (poprzez nadzorcę) poleca ten produkt zbudować.

Klasy znaków

.	dowolny znak
[ab]	*a* lub *b*
[^ab]	dowolny oprócz *a*,*b*
[a-t]	od *a* do *t*
[a-zA-z]	suma zbiorów
[1-9&&3-5]	część wspólna zbiorów
[1-9&&[^47]]	różnica zbiorów

Kotwice

^	początek linii
$	koniec linii
\b	na granicy słowa
\B	nie na granicy
\A	początek wejścia
\G	koniec poprz. dopas.
\Z	koniec wejścia

Kwantyfikatory

?	0 lub 1
*	0 lub więcej
+	1 lub więcej
{n}	dokładnie n
{n,}	co najmniej n
{n,m}	od n do m

Flagi

	off case-sensitive	on case-sensitive	(?i) plus Unicode
	(?i)	(?-i)	(?i)(?u)

Lista specjalnych
\ ^ $. | ? + () [] {}

Klasy predefiniowane

\d	[0-9]
\D	[^\d]
\s	[\t\n\f\r]
\S	[^\s]
\w	[a-zA-Z0-9]
\W	[^\w]
\p{ASCII}	znak ASCII
\p{Lower}	małe Unicode
\p{Upper}	duże Unicode
\p{Punct}	interpunkcyjne

Dodatkowe operatory

\	znak wyjścia
|	logiczna alternatywa
( )	grupowanie
\n	odwołanie n-tej grupy

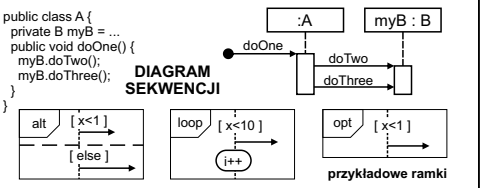
wzorzec .*+XXX nigdy nie będzie dopasowany (sprzeczność)

Tryby kwantyfikatorów

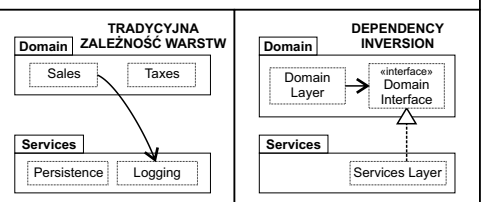
	domyślny zachłanny (max)
?	wstrzemięźliwy (min)
+	zaborczy (wszystko/nic)

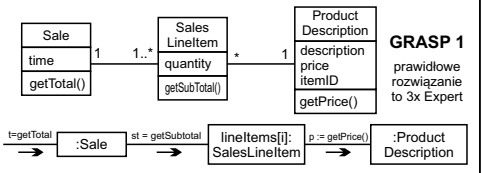
```
public class A {
  private B myB = ...
  public void doOne() {
    myB.doTwo();
    myB.doThree();
  }
}
```

DIAGRAM SEKWENCJI

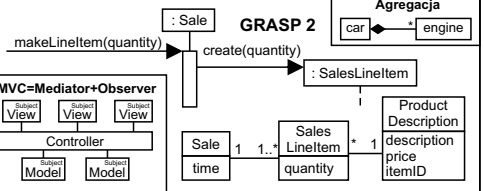


przykładowe ramki





GRASP 1
prawidłowe rozwiązanie to 3x Expert



GRASP 2

Agregacja

MVC=Mediator+Observer

zależność obiekty jednej klasy działają wykorzystując przelotnie obiekty innej

asocjacja obiekty jednej klasy działają wykorzystując obiekty innej dłuższy czas

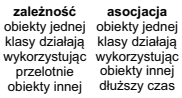
agreg.częśc. jedna klasa zawiera, ale jednocześnie współdzieli odwołanie do obiektów innej

agreg.całk. jedna klasa zawiera obiekty innej

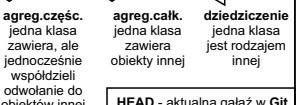
dziedziczenie jedna klasa jest rodzajem innej

HEAD - aktualna gałąź w **Git**

Wysokie sprzężenie (źle)
Niska spójność (źle)

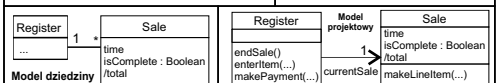


GRASP 3,4



GRASP 3,4

Niskie sprzężenie (dobrze)
Wysoka spójność (dobrze)




```
<!ELEMENT studenci (student)+>
<!ELEMENT student (imie,koledzy)>
<!ELEMENT koledzy (kolega)+>
<!ELEMENT kolega (student|null)>
<!ELEMENT null EMPTY>
<!ELEMENT imie (#PCDATA)>
<!ATTLIST student p (K|M) #REQUIRED>
```

- W **Factory Method** mamy jeden rodzaj produktów, w **Abstract Factory** rodzine powiazanych produktów, wiele interfejsów
- Konkretne fabryki dziedziczace po Abstract Factory moga wykorzystywac metode szablonowa, ale nie muszą.

```
<studenci>
  <student p="K">
  <imie>Ola</imie>
    <koledzy>
      <kolega>
        <null></null>
      </kolega>
    </koledzy>
  </student>
</studenci>
```

parsowalna zawartość tekstowa
<!ELEMENT tytul (#PCDATA)>
tytuł lub autor
<!ELEMENT publikacja (tytul | autor)>
tytuł i nieobowiązkowy autor
<!ELEMENT publikacja (tytul, autor?)>
dodatnia liczba tytułów, dowolna autorów
<!ELEMENT publikacja (tytul+, autor*)>
zawartość dowolna
<!ELEMENT publikacja ANY>

XML NameSpace

```
<pub:autorzy>
  <prac:pracownik>
    <prac:nazwisko>
      Kowalski
    </prac:nazwisko>
  </prac:pracownik>
</pub:autorzy>
```

Polecenie: Przy zdaniach, których prawdziwość wynika z diagramu napisz „TAK". Przy zdaniach których prawdziwości nie można wywnioskować napisz „NIE".
Tłumaczenie na polski: prawda ⇨ TAK, nieprawda ⇨ NIE, brak info ⇨ NIE.

GRASP - zasady przydziału metod do klas, rozkład odpowiedzialności w dążeniu do celu
1 **Expert** - klasa mająca informacje konieczne do realizacji celu; przy dużej złożoności systemu może być konieczna współpraca ekspertów; wspiera hermetyzację i równomierny podział. Każda metoda powinna zwracać wartość i nie zmieniać stanu lub zmieniać stan, ale nie zwracać wartości. Ułatwia zrozumienie, pozwala na bezpieczne wnioskowanie
2 **Creator** - nową instancję A powinna tworzyć klasa B, która: zawiera, pamięta, bezpośrednio używa lub posiada dane inicjalizacyjne A. Gdy wiele klas spełnia warunki, wybierz B zawierające A. Mniejsza liczba powiązań, kod łatwy w utrzymaniu.
3 **Low Coupling** (zasada ewaluacyjna, niskie sprzężenie) - klasa ma być jak najmniej zależna od innych (atrybuty, argumenty metod, wywołania metod, dziedziczenie). Ogranicza zasięg zmian, umożliwia ponowne wykorzystanie kodu. Sprzężenie z java.util to nie problem. Pewnien stopień powiązania musi być też zawsze, żeby umożliwić komunikację.
4 **High Cohesion** (zasada ewaluacyjna, wysoka spójność) - mała liczba podobnych zobowiązań w ustalonym obszarze funkcjonalnym (np. grupa klas realizuje obsługę bazy danych). Dzięki temu kod jest łatwy w utrzymaniu i zrozumiały.
5 **Polymorphism** - nadanie tej samej nazwy metodom różnych klas. Nadpisywanie (overriding) - wspólny interfejs lub nadklasa, metody o tej samej strukturze. Przeciążanie (overloading) - ta sama nazwa, ale inny kod w zależności od sygnatury.
6 **Indirection** - Przy powiązaniach z elementami niestabilnymi należy wykorzystać obiekt pośredniczący. Problemy projektowe można często rozwiązać dodatkowym pośrednikiem. Za to usuwanie pośredników wpływa pozytywnie na wydajność.
7 **Pure Fabrication** - ograniczenie się w modelu projektowym do obiektów z modelu dziedziny może prowadzić do niskiej spójności i wysokiego sprzężenia (np. każdy obiekt zapisuje sam do bazy). Należy wymyślić klasę pomocniczą, spoza modelu dziedziny i przydzielić jej odpowiednie zobowiązania (np. klasa obsługi bazy).
8 **Protected Variations** - system otwarty na rozszerzenia i zakmnięty na modyfikacje (Open-Close Principle). Każda metoda może wysłać komunikat do obiektu/atrybutu this, do parametru metody oraz do obiektu wewnątrz metody (Law of Demeter). Zmienny element jest wyprowadzany na zewnątrz systemu i są z niego wczytywane potrzebne dane (data driven programming), przykładem jest programowanie refleksyjne (modykacje w czasie wykonania).

SOLID
- **Single responsibility**
- **Open-closed principle** - [opis ↑]
- **Liskov substitution** - metody, które używają referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych (polymorphism, design by contract)
- **Interface segregation** - podziel duże interfejsy na mniejsze i dokładniejsze, tak by żaden klient nie był zależny od metod których nie używa
- **Dependency inversion** - uzależniać klasy od abstrakcji, a nie od konkretnych klas; decyzję o wiązaniu wyprowadzamy na zewnątrz. Warstwa to grupa pakietów o podobnym zakresie odpowiedzialności. Klasycznie warstwy układa się tak, aby wyższe wywoływały usługi niższych (najlepiej sąsiednich), a niższe nie zależały od wyższych. Dependency inversion odwraca: wyższe warstwy są wtedy zależne od interfejsów a nie od warstw niższych - odporniejsze na zmiany, przenośne.

KISS (Keep It Simple, Stupid) - pisz prosty kod dla innych ludzi, nie dla komputera.

YAGNI (You Ain't Gonna Need It) - nie pisz kodu tylko dlatego, bo możesz go potrzebować w przyszłości. W tym czasie może się coś zmienić. Twórz kod gdy go potrzebujesz.

Test Driven Development - pisz tyle, by przejść testy.

DRY (Don't Repeat Yourself) - nie powtarzaj kodu w wielu miejscach.

LEMIESZ NA SZCZĘŚCIE

Miejsce na Twoje imię i nazwisko

Z **adnotacjami** jest jak z budyniem. Jest smaczny, ale jak starasz się z niego zrobić sanki, to wychodzi z tego coś śmiesznego. /Szymon Wróbel, A.D. 2018

(CC) Mikołaj Pietrek, semestr zimowy 2017/2018

creativecommons.org/licenses/by-nc-sa/4.0/
BY NC SA

W tworzeniu ściągi pomogli Jakub Gogola i Max Telepchuk - wielkie dzięki!

ADAPTER (S)

przekształca interfejs jednej z klas na interfejs drugiej klasy

```
interface Adapter{
    void draw(int x1, int y1, int x2, int y2);
}
class Adaptee{
    public void draw(int x, int y, int w, int h) {
    //rysuje z jednego wierzchołka i wymiarów
    }
}
class ConcreteAdapter implements Adapter{
    private Adaptee adaptee = new Adaptee();
    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw( x1, y1, (x2 - x1), (y2 - y1) );
    }
}
public class Demo{
    public static void main(String[] args) {
        Adapter adapter = new ConcreteAdapter();
        //user podaje dwa wierzchołki
        adapter.draw(x1,y1,x2,y2);
    }
}
```

PROXY (S)

zastępuje inny obiekt, np. dla kontroli dostępu

```
interface Subject {
    public abstract void DriveCar();
}
public class Car implements Subject {
    public void DriveCar() {
        System.out.println("Car started");
    }
}
public class ProxyCar implements Subject {
    private Subject realCar;
    private int driverAge;

    public ProxyCar(int driverAge){
        this.driverAge = driverAge;
    }
    @Override
    void DriveCar() {
        if (driverAge > 18)
            realCar.DriveCar();
        else
            System.out.println("Forbidden");
    }
}
public class Demo {
    public static void main(String[] args) {
        Subject car = new ProxyCar(19);
        car.DriveCar();
    }
}
```

ABSTRACT FACTORY (C)

dostarcza interfejs do tworzenia różnych obiektów jednego typu

```
abstract class AbstractFactory {
    public abstract AbstractProduct  createButton();
}
class WinFactory extends AbstractFactory {
    public AbstractProduct createButton() {
        return new WinButton();
    }
}
class OSXFactory extends AbstractFactory {
    public AbstractProduct createButton() {
        return new OSXButton();
    }
}
abstract class AbstractProduct {
    public abstract void paint();
}
class WinButton extends AbstractProduct {
    public void paint() {
        System.out.println("Przycisk WinButton");
    }
}
class OSXButton extends AbstractProduct {
    public void paint() {
        System.out.println("Przycisk OSXButton");
    }
}
public class Demo {
    public static void main(String[] args) {
        AbstractFactory factory = new WinFactory();
        AbstractProduct button = factory.createButton();
        button.paint();
    }
}
```

MEDIATOR (B)

rozsyła dane do odpowiednich obiektów

```
public interface Mediator {
    void send(String message, Colleague colleague);
}
public abstract Colleague{
    protected Mediator mediator;
    public Colleague(Mediator m) {mediator = m;}
    public void send(String message, this);
    public abstract void receive(String message);
}
public class ApplicationMediator implements Mediator {
    private Array<List>Colleague> colleagues;
    public ApplicationMediator() {
        colleagues = new Array<List>Colleague>();
    }
    public void addColleague(Colleague colleague) {
        colleagues.add(colleague);
    }
    public void send(String message, Colleague originator) {
        colleague.receive(message);
    }
}
public class ConcreteColleague extends Colleague {
    public void receive(String message) {
        System.out.println("Concrete Received: " + message);
    }
}
public class Demo {
    public static void main(String[] args) {
        ApplicationMediator mediator = new ApplicationMediator();
        Colleague desktop = new ConcreteColleague(mediator);
        mediator.addColleague(desktop);
        desktop.send("Hello World");
    }
}
```

SINGLETON (C)

ogranicza ilość instancji do jednej, przy zapewnieniu globalnego dostępu

```
TRADYCYJNY
public final class Singleton {
    private final static Singleton inst = new Singleton();
    private static Singleton getInstance() {
        return inst;
    }
    int count = 0;
    int getNext(int count){
        count = count + 1;
        return count;
    }
    //zeby nie było konstruktora public
    private Singleton() {
    }
}

DOUBLE-CHECKED LOCKING
public final class Singleton {
    //uwaga, volatile!
    private static volatile Singleton instance = null;

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
    //zeby nie było konstruktora public
    private Singleton() {
    }
}

ZA POMOCĄ WYLICZENIOWEGO
public enum Singleton { INSTANCE }
```

COMPOSITE (S)

Composite powinien mieć metodę wypisującą listę, ale zabrakło miejsca

```
public interface Component {
    public void add(Component emp);
    public void remove(Component emp);
    public Component getChild(int i);
    public String getName();
}
public class Composite implements Component{
    private String name;
    public Composite(String name)
    {
        this.name = name;
        List<Component> = new ArrayList<Component>();
    }
    public void add(Component emp) {
        (as.add(emp);)
    }
    public void remove(Component emp) {
        (as.remove(emp);)
    }
    public Component getChild(int i)
    {
        (this.as.get(i);)
    }
    public String getName()
    {
        (return name;)
    }
}
public class Leaf implements Component{
    private String name;
    public Leaf(String name){this.name = name;}
    public void add(Component emp){}
    public void remove(Component emp){}
    public Component getChild(int i) {return null;}
    public String getName() {return name;}
}
public class Demo {
    public static void main(String[] args){
        Component leaf = new Leaf("John");
        Component manager = new Composite("Daniel");
        manager1.add(emp1);
    }
}
```

BUILDER (C)

rozdziela tworzenie od reprezentacji, dzieli tworzenie na kilka etapów

```
class Pizza {
    private String dough = "";
    public void setDough(String dough) {this.dough = dough;}
}
abstract class AbstractBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {return pizza;}
    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }
    public abstract void buildDough();
}
class ConcreteBuilder extends AbstractBuilder {
    public void buildDough() {pizza.setDough("cross");}
}
class Director {
    private AbstractBuilder pizzaBuilder;
    public void setBuilder(AbstractBuilder pb) {
        pizzaBuilder = pb;
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza() {
        pizzaBuilder.createNewPizza();
        pizzaBuilder.buildDough();
    }
}
public class Demo {
    public static void main(String[] args) {
        Director director = new Director();
        director.setAbstractBuilder (concreteBuilder );
        director.constructPizza();
        Pizza pizza = director.getPizza();
    }
}
```

Cytaty zwykłych kodu: ROBIE TE *** PIZZE Z SAMEGO *** CIASTA,
A I TAK *** COREL MI TO *** POZA, *** MARGINES :)

DECORATOR (S)

dynamicznie dodaje nowe funkcje podczas działania

```
public interface Component{
    public String getCost();
}
public class ConcreteComponent
    implements Component{
    public String getCost() {
        return 10;
    }
    @Override
    public String getCost();{
    }
}
public class ConcreteDecorator
    implements Component{
    Component original;
    public ConcreteDecorator(Component org){
        original = org;
    }
    @Override
    public String getCost();{
        oldCost = original.getCost();
        return (oldCost + 10);
    }
}
public class Demo{
    public void Main() {
        Decorator c = new ConcreteComponent();
        c = new ConcreteDecorator(c);
    }
}
```

FACTORY METHOD (C)

egzemplarze są tworzone przez podklasy

dostarcza interfejs do tworzenia obiektów nieokreślonych jako powiązanych typów

```
public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();
    public MazeGame() {
        Room room = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }
    abstract protected Room makeRoom();
}
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
public class Demo {
    public static void main(String[] args) {
        MazeGame ordinaryGame = new OrdinaryMazeGame();
        MazeGame magicGame = new MagicMazeGame();
    }
}
```

OBSERVER (B)

powiadamia zainteresowane obiekty o zmianie stanu obserwowanego

```
public class Subject{
    private Vector observers = new Vector();
    protected void addObserver(Observer o) {
        if (observers.contains(o, observer) == false )
            observers.addElement(o, observer );
    }
    public boolean detach(Observer observer X;
        return observers.removeElement(o, observer );
    }
    public void notifyObservers() {
        for (int i = observers.size() - 1; i >= 0; i--) {
            Observer observer = (Observer) observers.elementAt(i);
            observer.update();
        }
    }
}
public interface Observer {
    void update();
}
public class ConcreteSubject extends Subject {
    public int getState()
    {
        return 1;
    }
}
public class ConcreteObserver implements Observer {
    public void update()
    {
        ( System.out.println("Update!") catch* ! ; )
    }
}
public class Test{
    public static void main(String[] args) {
        Subject subject = new ConcreteSubject();
        subject.attach(new ConcreteObserver());
        subject.notify();
    }
}
```

FACADE (S)

upraszcza dostęp do złożonego systemu

```
class PaperTray{
    private int sheets;
    public PaperTray(int sheets){
        this.sheets = sheets;
    }
    public boolean checkPaper() {
        if (i>0) return true;
        else return false;
    }
    public void getPaper() {
        sheets = sheets - 1;
    }
}
class PrintingModule{
    public void printText(String text){
        System.out.println(text);
    }
}
class PrinterFacade {
    private PaperTray paperTray;
    private PrintingModule printingModule;
    public ComputerFacade() {
        this.paperTray = new PaperTray(100);
        this.printingModule = new PrintingModule();
    }
    public void print() {
        if(checkPaper(String text){
            paperTray.getPaper();
            printingModule.printText(text);
        }
    }
}
public class Demo {
    public static void main(String[] args) {
        Facade printer = new Facade();
        printer.print("Dziwaszujesz środowisko!");
    }
}
```

STATE (B)

reprezentuje obiektowy stan, zmienia funkcje w zależności od stanu

```
public class Context {
    private State state;
    public Context(State state) {
        this.state = state;
    }
    public void play() {
        state.presPlayThis();
    }
    protected void setState(State state) {
        this.state = state;
    }
    public State getState() {
        return state;
    }
}
private interface State {
    void presPlay(Context context);
    public void presPlay(Context context);
}
public class StandbyState implements State {
    public void presPlay(Context context) {
        context.setState(new PlayingState());
    }
}
public class PlayingState implements State {
    public void presPlay(Context context) {
        context.setState(new StandbyState());
    }
}
public class Demo {
    public static void main(String[] args) {
        Context mp3 = new Context(new StandbyState);
        mp3.play();
    }
}
```

FLYWEIGHT (S)

oddziela powtarzalne dane, żeby zmniejszyć zużycie pamięci

```
public interface Shape { void draw();}
public class Circle implements Shape {
    private int x, y;
    public int r;
    public Circle(String color){this.color = color }
    public void draw() {
    System.out.println(color + " " + x + " " + y + " " + radius);
    }
}
public class Factory {
    public static final HashMap<String, Shape> map = new HashMap();
    public static Shape getCircle(String color) {
        Circle c = (Circle)factory.get(color);
        if(c == null) {
            c = new Circle(color);
            factory.put(color, c);
        }
        return circle;
    }
}
public class FlyweightDemo { // "Red", "Green", "Blue";
    private static String getRandomColor() {
    public static void main(String[] args) {
        for(int i = 0; i < 20; ++i) {
            Circle circle = (Circle)Factory.getCircle(getRandomColor());
            circle.setX(i);
            circle.setY(i);
            circle.setRadius(100);
            circle.draw();
        }
    }
}
```

TEMPLATE METHOD (B)

wyznacza niezmienną część algorytmu, a kroki mogą być definiowane w dziedziczących

```
public abstract class AbstractClass{
    protected abstract void initGame(int count);
    protected abstract void makeMove(int player);
    protected abstract void playGame(int count) {
    int count = 0;
    while (true) {
        makeMove();
        j = (j + 1) % count;
    }
    }
}
public class Szachy extends AbstractClass{
    @Override
    protected void initGame(int count){
        //inicjalizacja szachów
    }
    @Override
    protected void makeMove(int player){
    //ruch szachowy
    }
    @Override
    protected void initGame(int count){
        //inicjalizacja warcabów
    }
    @Override
    protected void makeMove(int player){
    //ruch w warcabach
    }
}
public class Demo {
    public static void main(String[] args) {
        AbstractClass szachy = new Szachy();
        szachy.playGame;
    }
}
```

zasada Hollywood
nie dzwoń,
my zadzwonimy
(dzwoni „góra")

STRATEGY (B)

definiuje wymienne algorytmy w postaci klas

```
public interface Strategy {
    public void compressFiles(ArrayList<File> files);
}
public class ZipStrategy implements Strategy {
    public void compressFiles(ArrayList<File> files) {
    //using ZIP
    }
}
public class RarStrategy implements Strategy {
    public void compressFiles(ArrayList<File> files) {
    //using RAR
    }
}
public class Context {
    private Strategy strategy;
    private void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }
    public void createArchive(ArrayList<File> files) {
        strategy.compressFiles(files);
    }
}
public class Client {
    public static void main(String[] args) {
        Context cx = new Context();
        cx.setStrategy(new ZipStrategy());
        cx.createArchive(new ArrayList);
    }
}
```

COMMAND (B)

traktuje żądanie wykonania czynności jako obiekt

```
public interface Command{
    public void execute();
}
public class LightOnCommand implements Command{
    public LightOnCommand(Light light){this.light = light;}
    public void execute(){light.switchOn();}
}
public class LightOffCommand implements Command{
    Light light;
    public LightOffCommand(Light light)
    {this.light = light;}
    public void execute()
    {light.switchOff();}
}
public class Light {
    private boolean on;
    public void switchOn(){on = true;}
    public void switchOff(){on = false;}
}
public class RemoteControl {
    public void setCommand(Command command)
    {command = command;}
    public void pressButton()
    {command.execute();}
}
public class Client{
    public static void main(String[] args) {
        RemoteControl control = new RemoteControl();
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);
        Command lightOff = new LightOffCommand(light);
        control.setCommand(lightOn);
        control.pressButton();
        control.setCommand(lightOff);
        control.pressButton();
    }
}
```


LEMIESZ NA SZCZĘŚCIE

Miejsce na Twoje imię i nazwisko

Z **adnotacjami** jest jak z budyniem. Jest smaczny, ale jak starasz się z niego zrobić sanki, to wychodzi z tego coś śmiesznego. /Szymon Wróbel, A.D. 2018

creativecommons.org/licenses/by-nc-sa/4.0/

(CC) Mikołaj Pietrek, semestr zimowy 2017/2018