Wzorce projektowe

Adrian Nicoś

O mnie

- Adrian Nicoś
- od 10 lat programista / lider / architekt
- Języki: Java / Python / Groovy / JavaScript
- GitHub: https://github.com/anicos
- E-mail: adrian.nicos@gmail.com
- Zainteresowania
  - architektura oprogramowania
  - biegi na orientacje
  - kolarstwo górskie

Design patterns

Geneza wzorców

„Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy”

Christopher Alexander

"*A pattern language*"

1977

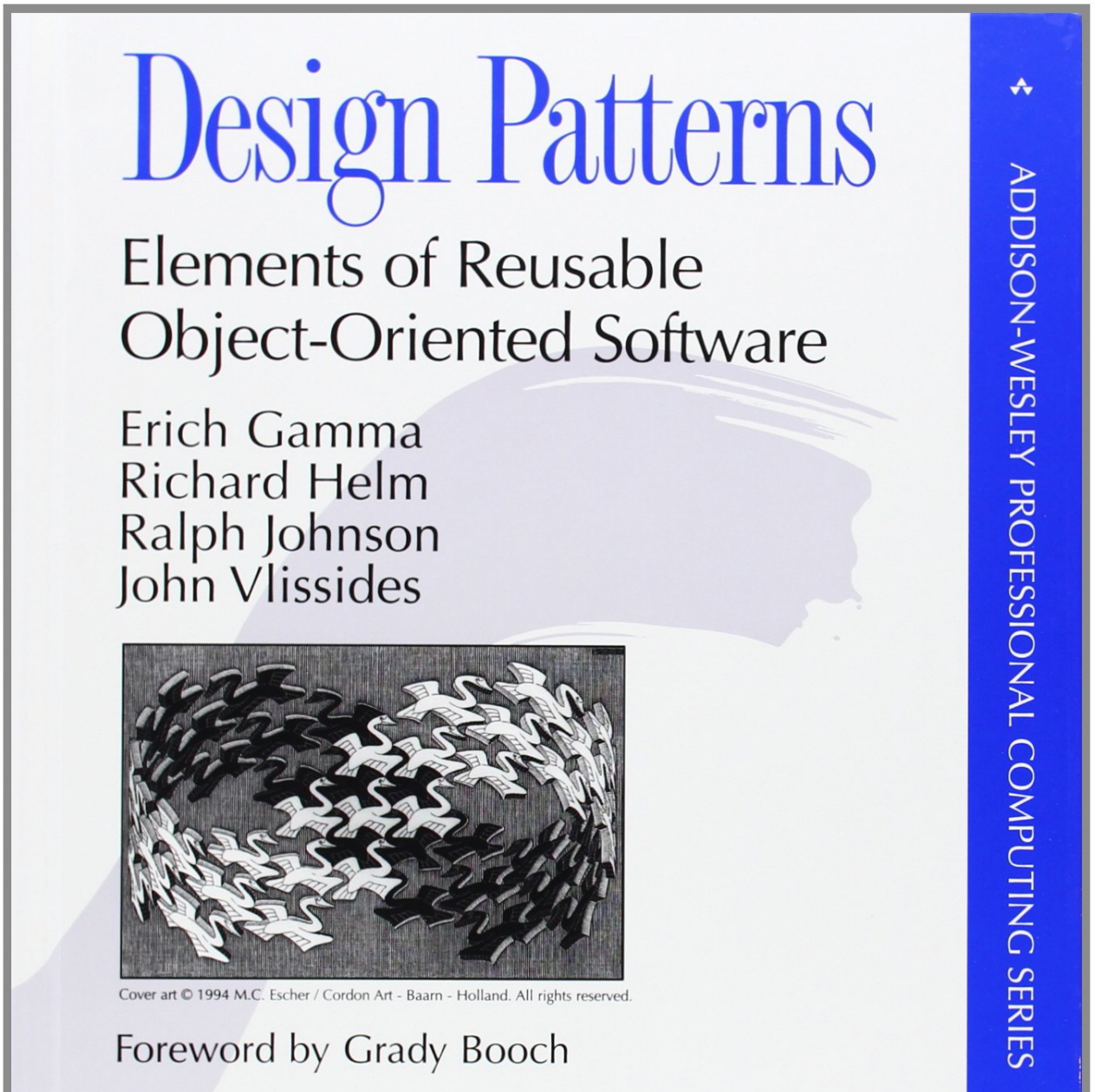
Historia

- wzorce projektowe w informatyce wywodzą się z wzorców projektowych w architekturze
- zaproponowane przez amerykańskiego architekta Christophera Alexandra
- miały ułatwić konstruowanie mieszkań i pomieszczeń biurowych
- pomysł ten nie został jednak przyjęty

Historia

- termin wzorca projektowego został wprowadzony przez Kenta Becka oraz Warda Cunninghama w 1987 roku, na konferencji OOPSLA
- został spopularyzowany w 1995 przez Bandę Czworga

Gang of Four


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

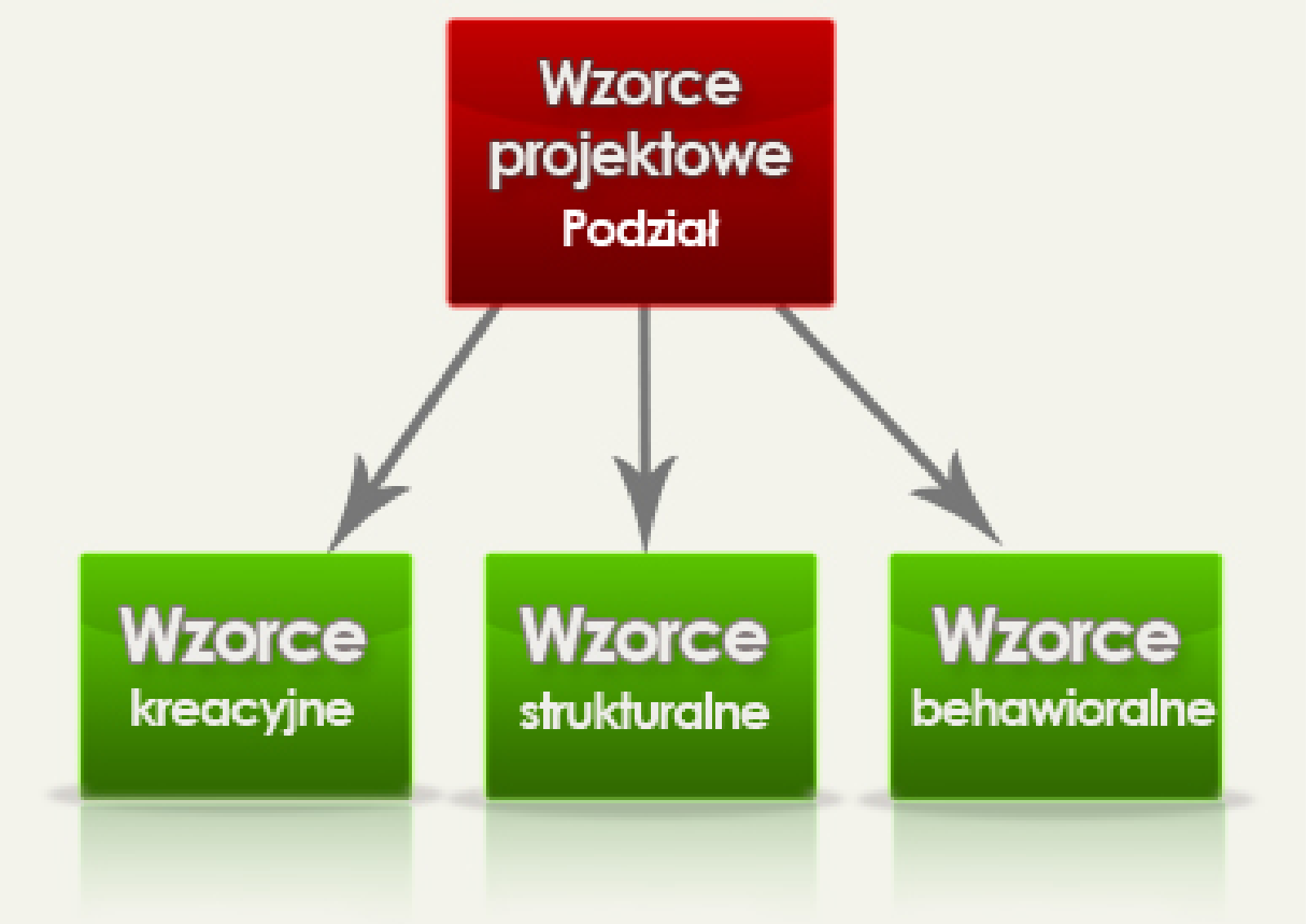
Definicja

- uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych
- pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz pielęgnację kodu źródłowego
- jest opisem rozwiązania, a nie jego implementacją

Zalety stosowania wzorców

- opisują rozwiązania często powtarzających się problemów
- pozwalają uniknąć typowych błędów
- wyjaśniają rozwiązanie problemu
- rozwiązują problemy spotykane w rzeczywistym świecie, w codziennej pracy przy systemach IT
- bazują na doświadczeniach nabytych w przeszłości
- ułatwiają komunikację w pracy

Podział wzorców projektowych



Podział ze względu na zakres

- klasowe
  - opisujące statyczne związki pomiędzy klasami;
- obiektowe
  - opisujące dynamiczne związki pomiędzy obiektami.

Wzorce kreacyjne(ang. *Creational pattern*)

- Abstract factory
- Builder pattern
- Factory method
- Prototype
- Singleton

Singleton

Singleton

Singleton
- singleton : Singleton
- Singleton() + getInstance() : Singleton

Singleton

- Zapewnia dokładnie jedną instancję klasy w systemie

Singleton – kiedy używamy?

- gdy chcemy mieć jedną instancję klasy w całym systemie
- gdy chcemy kontrolować dostęp do danej instancji

Singleton klasyczna implementacja

```
package pl.anicos.patterns.exercises.singleton.simple;
public class ClassicSingleton {
  private static ClassicSingleton instance = null;
  private ClassicSingleton() {
    // Exists only to defeat instantiation.
  }
  public void getSomeThing() {
    // do something here
    System.out.println("I am here....");
  }
  public static ClassicSingleton getInstance() {
    if (instance == null) {
      instance = new ClassicSingleton();
    }
    return instance;
  }
}
```

Czy to zawsze zadziała?

Eager Singleton

```
package pl.anicos.patterns.exercises.singleton.eager;
import pl.anicos.patterns.exercises.singleton.ThreadWrapper;
public class EagerInitializedSingleton {
  private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();
  //private constructor to avoid client applications to use constructor
  private EagerInitializedSingleton() {
    ThreadWrapper.sleep();
  }
  public void getSomeThing() {
    // do something here
    System.out.println("I am here....");
  }
  public static EagerInitializedSingleton getInstance() {
    return instance;
  }
}
```

Thread Safe Singleton

```
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance;
    private ThreadSafeSingleton() {
    }
    public static synchronized ThreadSafeSingleton getInstance() {
        if (instance == null) {
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }
    public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {
        if (instance == null) {
            synchronized (ThreadSafeSingleton.class) {
                if (instance == null) {
                    instance = new ThreadSafeSingleton();
                }
            }
        }
        return instance;
    }
}
```

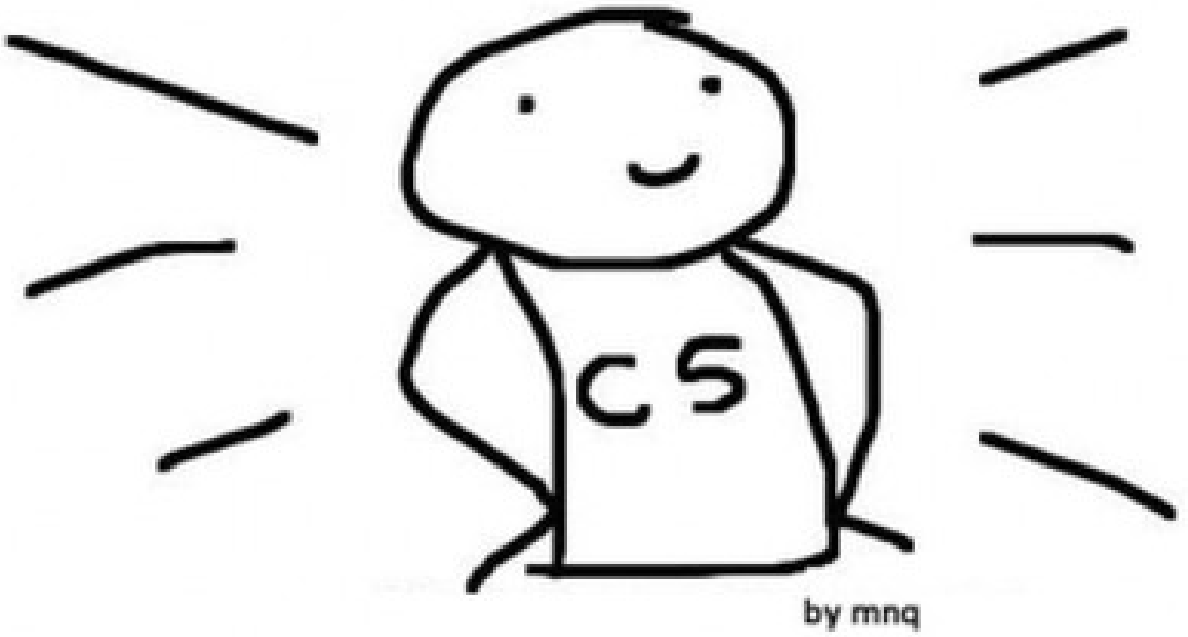
Enum Singleton

```
public enum EnumSingleton {
  INSTANCE;

  public static void doSomething() {
    //do something
  }
}
```

Bill Pugh Singleton

```
public class BillPughSingleton {
  private BillPughSingleton() {
    ThreadWrapper.sleep();
  }
  private static class SingletonHelper {
    private static final BillPughSingleton INSTANCE = new
BillPughSingleton();
  }
  public static BillPughSingleton getInstance() {
    return SingletonHelper.INSTANCE;
  }
}
```

Jak zepsuć singleton?



Jak zepsuć singleton?

- Refleksja
  - *ReflectionSingletonTest*
- *Serializacja*
  - *SerializedSingletonTest*

SerializedSingleton – czego implementacja jest OK?

```
public class SerializedSingleton implements Serializable {

  private static final long serialVersionUID =
-7604766932017737115L;

  private SerializedSingleton() {

  }

  private static class SingletonHelper {

    private static final SerializedSingleton instance = new
SerializedSingleton();

  }

  public static SerializedSingleton getInstance() {

    return SingletonHelper.instance;

  }

}
```

SerializedSingleton

```
public class SerializedSingleton implements Serializable {
  private static final long serialVersionUID = -7604766932017737115L;
  private SerializedSingleton() {
  }
  private static class SingletonHelper {
    private static final SerializedSingleton instance = new
SerializedSingleton();
  }
  public static SerializedSingleton getInstance() {
    return SingletonHelper.instance;
  }
  protected Object readResolve() {
    return getInstance();
  }
}
```

Przykłady w JDK

- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSecurityManager()

Builder

- celem wzorca jest rozdzielenie sposobu tworzenia obiektów od ich reprezentacji

Builder

- dzieli proces tworzenia obiektu na kilka mniejszych etapów, a każdy z nich może być implementowany na wiele sposobów.
- dzięki takiemu rozwiązaniu możliwe jest tworzenie różnych reprezentacji obiektów w tym samym procesie konstrukcyjnym

```
Computer testObj = new Computer.ComputerBuilder("500 GB", "2 GB")
    .setBluetoothEnabled(true)
    .setGraphicsCardEnabled(true)
    .build();
```

Builder – kiedy stosujemy

- konstruktor obiektu zawiera wiele argumentów (dobra praktyka > 2)
- konstruktor zawiera argument tego samego typu `public Point(int x, int y)`
- cześć argumentów konstruktora jest wymagana a część jest opcjonalna

Builder – kiedy stosujemy?

```
public class Person {
  private final String lastName;
  private final String firstName;
  private final String middleName;
  private final String salutation;
  private final String suffix;
  private final String streetAddress;
  private final String city;
  private final String state;
  private final boolean isFemale;
  private final boolean isEmployed;
  private final boolean isHomewOwner;
  public Person(String lastName, String firstName, String middleName,
          String salutation, String suffix, String streetAddress,
          String city, String state, boolean isFemale, boolean isEmployed, boolean
isHomewOwner) {}
}
```

Builder - zalety

- implementacja tworzenia obiektu w jednym miejscu
- dodanie nowej nowych pól do klasy wymaga zmiany kodu w tylko jednym miejscu
- łatwiejsze tworzenie niemutowanych struktur danych
- generowany przez IDE

Builder klasyczny

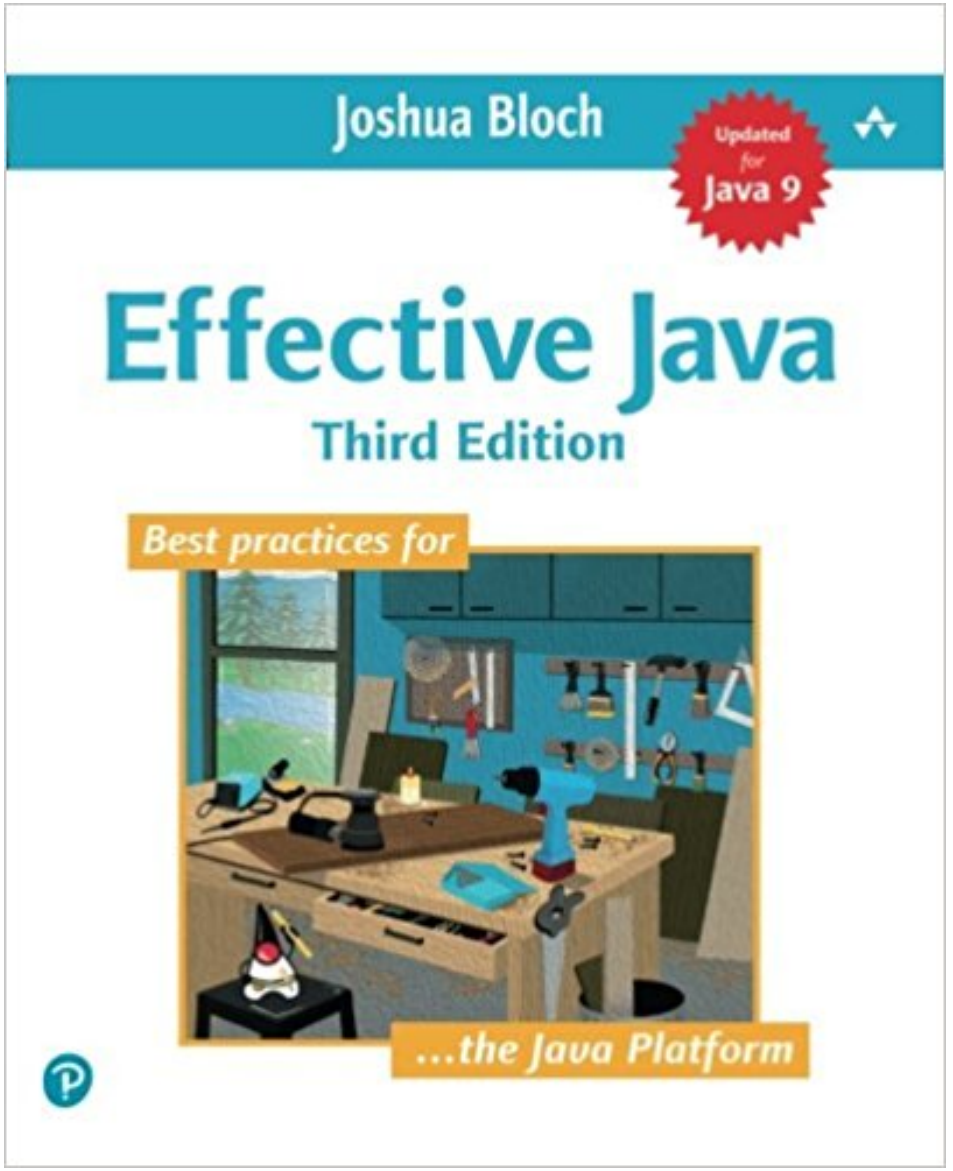
```
public class Task {
  private final String description;
  private final String name;

  Task(String description, String name) {
    this.description = description;
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public String getDescription() {
    return description;
  }
}
```

```
public class TaskBuilder {
  private String description;
  private String name;
  public TaskBuilder setDescription(String
description) {
    this.description = description;
    return this;
  }
  public TaskBuilder setName(String name) {
    this.name = name;
    return this;
  }
  public Task createTask() {
    return new Task(description, name);
  }
}
```

Builder by Joshua Bloch

- do konstruktora klasy właściwej przekazywana jest statyczna klasa buildera
- przykład poniżej
  - *pl.anicos.patterns.exercises.builder.joshuabloch .ComputerBuilderTest*

Biblia programisty Java!!

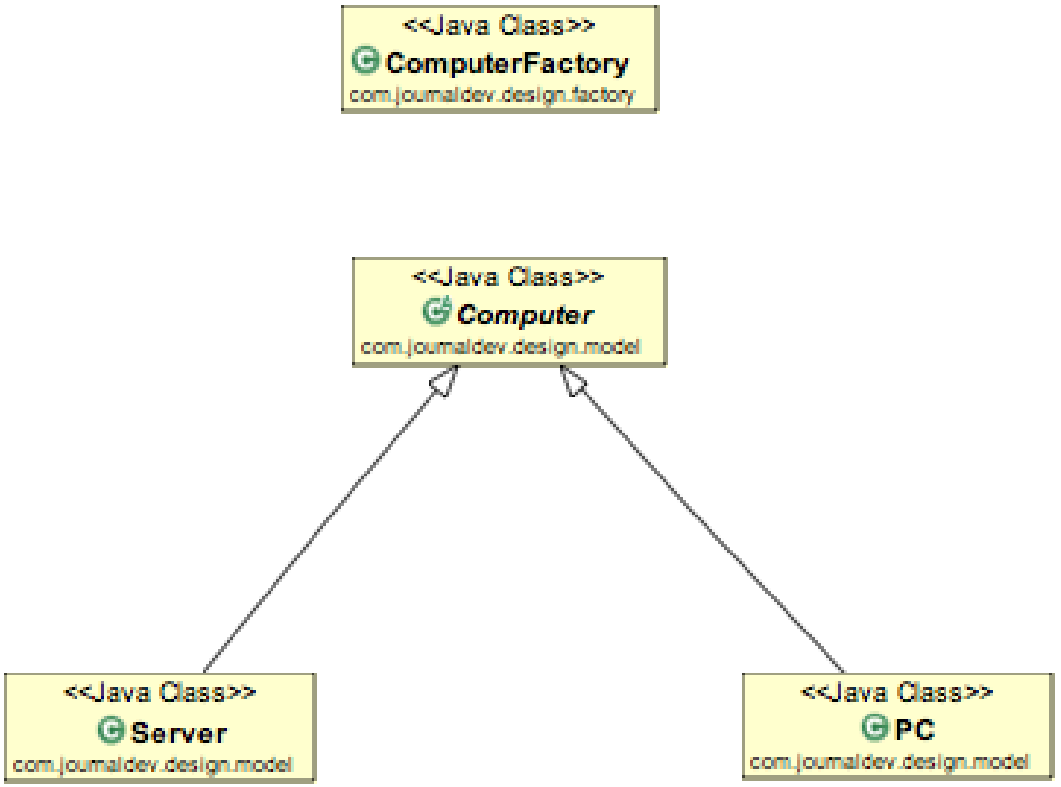


Przykłady builder-a w JDK

- **java.lang.StringBuilder**#append() (unsynchronized)
- **java.lang.StringBuffer**#append() (synchronized)
- **java.nio.ByteBuffer**#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
- **javax.swing.GroupLayout.Group**#addComponent()
- All implementations of **java.lang.Appendable**

Factory method

- służy do tworzenia nowych obiektów, nieokreślonych, lecz związanych z jednym, wspólnym interfejsem



factory method - przykład

```
public class ComputerFactoryTest {
  @Test
  public void shouldCreatePC() {
    Computer testObj = ComputerFactory.getComputer("PC", "2 GB", "500 GB", "2.4
GHz");
    assertTrue(testObj instanceof PC);
  }
  @Test
  public void shouldCreateServer() {
    Computer testObj = ComputerFactory.getComputer("Server", "16 GB", "1 TB",
"2.9 GHz");
    assertTrue(testObj instanceof Server);
  }
  @Test(expected = RuntimeException.class)
  public void shouldNotCreateMac() {
    ComputerFactory.getComputer("Mac", "16 GB", "1 TB", "2.9 GHz");
  }
}
```

factory method - zalety i wady

- Zalety:
  - Niezależność od konkretnych implementacji zasobów oraz procesu ich tworzenia.
  - Wzorzec hermetyzuje proces tworzenia obiektów, zamykając go za ściśle zdefiniowanym interfejsem.
  - Spójność produktów – w sytuacji, gdy pożądane jest, aby klasy produkty były z określonej rodziny.
- Wady:
  - Nie znam…

Przykłady factory method w JDK

- java.util.Calendar#getInstance()
- java.util.ResourceBundle#getBundle()
- java.text.NumberFormat#getInstance()
- java.nio.charset.Charset#forName()
- java.net.URLStreamHandlerFactory#createURLS treamHandler(String) (Returns singleton object per protocol)
- java.util.EnumSet#of()

Abstract Factory

- Zadaniem fabryki abstrakcyjnej jest określenie interfejsu do tworzenia różnych obiektów należących do tego samego typu (rodziny).
- Interfejs ten definiuje grupę metod, za pomocą których tworzone są obiekty.

Abstract Factory

```
public class AbstractFactoryApplicationTest {
  @Test
  public void shouldPaintMacOsUi() {
    //given
    AbstractFactoryApplication testObj = new AbstractFactoryApplication(new MacOSFactory());
    //when
    String[] paint = testObj.paint();
    //then
    Assert.assertEquals(MacOSButton.YOU_HAVE_CREATED_MAC_OSBUTTON, paint[0]);
    Assert.assertEquals(MacOSCheckbox.YOU_HAVE_CREATED_MAC_OSCHECKBOX, paint[1]);
  }
  @Test
  public void shouldPaintWindowsUi() {
    //given
    AbstractFactoryApplication testObj = new AbstractFactoryApplication(new WindowsFactory());
    //when
    String[] paint = testObj.paint();
    //then
    Assert.assertEquals(WindowsButton.YOU_HAVE_CREATED_WINDOWS_BUTTON, paint[0]);
    Assert.assertEquals(WindowsButton.YOU_HAVE_CREATED_WINDOWS_BUTTON, paint[1]);
  }
}
```

Abstract Factory - zalety

- odseparowanie klas konkretnych – klienci nie wiedzą jakich typów konkretnych używają, posługują się interfejsami abstrakcyjnymi.
- łatwa wymiana rodziny produktów.
- spójność produktów – w sytuacji, gdy pożądane jest, aby klasy produkty były z określonej rodziny, fabryka bardzo dobrze to zapewnia.

Abstract Factory - zastosowanie

- można wykorzystać między innymi do stworzenia uniwersalnego sterownika do obsługi różnych typów baz danych (MySQL, Oracle SQL itp.).

Przykłady *Abstract Factory w JDK*

- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

Prototype

- Opisuje mechanizm tworzenia nowych obiektów poprzez klonowanie jednego obiektu macierzystego.
- mechanizm klonowania wykorzystywany jest wówczas, gdy należy wykreować dużą liczbę obiektów tego samego typu lub istnieje potrzeba tworzenia zbioru obiektów o bardzo podobnych właściwościach

Prototype

```
public class Employees implements Cloneable {
  private List<String> empList = new ArrayList<String>();
  public Employees(List<String> list) {
    this.empList = list;
  }
  public void addNew(String newEmp) {
    empList.add(newEmp);
  }
  public List<String> getEmpList() {
    return empList;
  }
  @Override
  public Object clone() throws CloneNotSupportedException {
    List<String> temp = new ArrayList<String>();
    for (String s : this.getEmpList()) {
      temp.add(s);
    }
    return new Employees(temp);
  }
}
```

Prototype - zalety

- Użycie metody *clone* z JDK jest szybsze niż tworzenie obiektów przez konstruktor

Prototype w JDK

- java.lang.Object#clone()

Prototype – uwaga!!!

- bardzo często klonowanie obiektów jest skutkiem złego zaprojektowania kodu. **Zastanówmy się dwa razy nim skopiujemy jakiś obiekt**