

## Laboratorium

# Temat : Podstawowe techniki utrzymania wysokiej jakości kodu

## Historia zmian

<i>Data</i>	<i>Wersja</i>	<i>Autor</i>	<i>Opis zmian</i>
2013.03.08	1.0	Tomasz Kowalski	Utworzenie dokumentu i opracowanie zadań
2013.04.22	1.1	Tomasz Kowalski	Podział na dwa laboratoria i ogólna aktualizacja treści
2013.10.14	1.2	Tomasz Kowalski	Aktualizacja treści – testy jednostkowe
2016.10.02	1.3	Tomasz Kowalski	Zmiana repozytorium z svn-a na github.
2017.03.01	1.4	Tomasz Kowalski	Instrukcja i kody na organizacji na github + aktualizacja
2018.05.07	2.0	Tomasz Kowalski	Przeróbka projektu

# 1. Cel laboratorium

Głównym celem laboratoriów jest przegląd podstawowych technik mających na celu poprawienie i utrzymanie wysokiej jakości kodu w języku Java. Po pierwsze studenci zapoznają się z powszechnie stosowanymi konwencjami dotyczącymi organizacji struktury projektów Java oraz kodu języka Java (m. in. formatowanie i nazewnictwo). Kolejnym istotnym elementem laboratoriów jest nauka wykorzystania zaawansowanego środowiska programistycznego IDE (na przykładzie Eclipse) do automatycznej generacji kodu oraz refaktoryzacji.

*Czas realizacji laboratoriów wynosi 3 godziny.*

## 2. Zasoby

### 2.1. Wymagane oprogramowanie

Polecenia laboratorium będą dotyczyły programowania wzorców w języku Java. Potrzebne będzie środowisko dla programistów (JDK – Java Development Kit<sup>1</sup>) oraz zintegrowana platforma programistyczna (np. Eclipse<sup>2</sup>) z zainstalowaną wtyczką do obsługi narzędzia Maven (np. m2eclipse<sup>3</sup>).

### 2.1. Materiały pomocnicze

Materiały dostępne w Internecie:

[Code Conventions for the Java TM Programming Language](#)

[Eclipse help - refactoring](#)

## 3. Laboratorium

1. Na platformie github zrób **fork** projektu biblioteki *powp\_rhymers\_bridge* dostarczającej egzotyczne warianty stosu.
2. Fork projektu należy pobrać lokalnie (np. *git clone*) i zaimportować do Eclipse IDE wybierając *File* → *Import...* → *Maven* → *Existing Maven Projects*. Następnie należy wybrać katalog zawierający plik *pom.xml* jako *Root Directory* i kliknąć *Finish*.
3. Zawarte w projekcie warianty *wyliczanek* służą przeprowadzaniu rozrywek skomplikowanych gier całkowito liczbowych. O ile działanie bazowej klasy *wyliczankowej DefaultCountingOutRyhmer* w swoim działaniu przypomina stos:
  - *FIFORyhmer* – dostarczający pod klasycznym interfejsem *wyliczankę* opartą na podejściu kolejki First In First Out.
  - *HanoiRyhmer* – na którym nie jest możliwe *zgłoszenie* (*countIn*) liczby większej niż bieżąca (*peekaboo*).

Zapoznaj się z projektem, jego strukturą, klasami. Uruchom aplikację demo *RhymersDemo*. Uruchom również testy jednostkowe: menu kontekstowe projektu *Run as ...* → *Maven test*. Zapoznaj się z testami jednostkowymi klasy *DefaultCountingOutRyhmer* (znajdują się w *src/test/java*).

4. \*Pracując nad projektem dbaj o poprawność testów jednostkowych.

---

1 <http://java.sun.com/javase/downloads/index.jsp>

2 <http://www.eclipse.org/>

3 <http://www.sonatype.org/m2eclipse>

## 5. UWAGI:

- Przeczytaj każdy podpunkt instrukcji do końca przed rozpoczęciem jego realizacji.
- Wszędzie gdzie jest to napisane wykorzystuj narzędzia środowiska IDE.
- Wykonanie każdego podpunktu nie powinno wprowadzać nowych błędów.
- W razie kłopotów korzystaj z pomocy prowadzącego.
- Alternatywne pomysły na rozwiązanie zadań zgłoś prowadzącemu.

## 6. Praca z systemem kontroli wersji:

- Wykonuj *commit* do każdego wykonanego podpunktu laboratorium. W sytuacjach wyjątkowych (np. bardzo małe zmiany) *commit* może obejmować większą część instrukcji. Komentarz przy *commicie* powinien ułatwić identyfikację podpunktów.
- Pod koniec zajęć wyniki prac na laboratorium muszą być każdorazowo oznaczane w repozytorium jako osobny *release/tag*. Braki w tym zakresie są równoważne z brakiem obecności na zajęciach.
- Przechowywanie folderów generowanych automatycznie przez narzędzia (np. *bin*, *target*) w repozytorium utrudnia prace i jest niewskazane. Dla wygody możesz skorzystać z pliku *.gitignore*.

### 3.1. Pół-automatyczna poprawa jakości kodu.

Domyślnie (tj. o ile nie napisano inaczej) poprawki stosuj w klasach w *src/main/java*.

1. Popraw błędy związane z formatowaniem kodu źródłowego Java we wszystkich klasach. Na zaznaczonym fragmencie używaj klawisza *tab* lub kombinacji *shift+tab* do regulacji wcięć. W komentarzu do *commita* napisz, które wiersze w klasie *HanoiRyhmer* były źle sformatowane (automatyczna generacja komentarza na zaznaczeniu *ctrl+)*.
2. Sprawdź czy wszystkie błędy formatowania zostały usunięte uruchamiając automatyczne formatowanie (*Source* → *Format* lub *ctrl+shift+f*)
3. Zweryfikuj działanie kombinacji klawiszy *alt* + ← oraz *alt* + →. Komentarz na ten temat zamieść w ostatnio edytowanym pliku.
4. Popraw błędy konwencji nazewnictwa klas, metod, atrybutów, itp. kodu źródłowego Java we wszystkich klasach. Użyj do tego opcji *Refactor* → *Rename* (skrót klawiszowy *alt+shift+r*).
5. Przejrzyj kod w poszukiwaniu literałów (napisy, liczby), które można by zastąpić deklaracjami stałych (np. w klasie *DefaultCountingOutRyhmer* liczby -1 i 12). Wygeneruj odpowiednie stałe używając opcji *Refactor* → *Extract Constant*.
6. Przeanalizuj metody i atrybuty pod kątem widoczności (modyfikatory *public*, *private*, *etc.*). Wszędzie zastosuj możliwe najmniejszą widoczność.
7. Wygeneruj getter dla pola *total* w klasie *DefaultCountingOutRyhmer* (opcja *Source* → *Generate Getters and Setters*).
8. Dokonaj hermetyzacji nieprywatnych atrybutów (polecenie *Refactor* → *Encapsulate Field*, użyte z opcją *keep field reference*). W komentarzu przy tych atrybutach opisz jakie nastąpiły automatyczne zmiany w pozostałych klasach w związku z hermetyzacją.
9. Usuń nieużywane settery. Do analizy wywołań użyj opcji *Navigate* → *Open Call Hierarchy* (*ctrl+alt+h*).

10. Ustaw modyfikator *final* przy niemutowalnych (nie zmieniających wartości) atrybutach klas (\*opcjonalnie przy lokalnych zmiennych i parametrach metod).
11. Użyj anotacji *@Override* przy metodach tam gdzie jest to możliwe.
12. Przejrzyj projekt pod kątem klas, które nie muszą być publiczne. Przeorganizuj projekt tak, aby usunąć pliki związane z takimi klasami. Uzasadnij w komentarzu do commita dlaczego wybrane klasy zostały publiczne.
13. Uporządkuj aplikację demo *RyhmersDemo* (znajdującą się w *src/test/java*) rozbijając metodę *main*. Utwórz metodę statyczną *testRyhmers* przyjmującą jako argument fabrykę stosów. W tym celu użyj opcji *Refactor* → *Extract Method* na zawartości funkcji *main* z pominięciem deklaracji lokalnej zmiennej *factory*.

### 3.2. Testy jednostkowe

1. Dokonaj walidacji projektu testami jednostkowymi. W razie potrzeby popraw testy i projekt.  
\*Jeżeli występują błędy określ gdzie i przy realizacji, których punktów powstały.
2. \*Popraw jakość kodu testów jednostkowych i napisz testy dla pozostałych klas projektu.

### 3.3. Komentarze i diagramy

1. Naszkicuj diagram klas UML występujących w projekcie.<sup>4</sup>
2. Wygeneruj automatycznie szkielet dokumentacji do wybranej klasy i jej metod (np. opcja *Source* → *Generate Element Comment*) oraz ją uzupełnij.
3. \*Przy okazji przejrzyj implementację pod kątem jakości. Jeżeli znajdziesz jakieś potencjalne miejsca, które można naprawić, dodaj notkę „TODO:” w komentarzach, np.:  
`// TODO: needs refactoring to the bridge pattern :)`

---

<sup>4</sup> Można użyć np. narzędzia [yuml.me](http://yuml.me) lub aplikacji Visual Paradigm.