

## Laboratorium

# Temat : Refaktoryzacja do wzorca *bridge*

## Historia zmian

[illegible]

# 1. Cel laboratorium

Głównym celem laboratoriów jest zapoznanie się ze złożonym strukturalnym wzorcem projektowym *Bridge* (pomost). Należy on do grupy wzorców strukturalnych. Zajęcia powinny pomóc studentom rozpoznawać omawiane wzorce w projektach informatycznych, samodzielnie implementować wzorce oraz dokonywać odpowiednich modyfikacji wzorca w zależności od potrzeb projektu. Istotnym elementem laboratoriów jest nauka wykorzystania zaawansowanego środowiska programistycznego IDE (na przykładzie Eclipse) do automatycznej generacji kodu oraz refaktoryzacji.

*Czas realizacji laboratoriów wynosi 2 godziny.*

*Insulation (pl. izolacja) - implementation details (type, data, or function) can be altered without forcing clients of the component to recompile - a physical property of design*

## 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:

<http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>

<http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm>

<http://www.vincehuston.org/dp/>

[http://en.wikipedia.org/wiki/Design\\_pattern\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science))

## 3. Laboratorium

1. Laboratorium jest kontynuacją poprzednich laboratoriów na temat utrzymywania wysokiej jakości kodu na przykładzie projektu biblioteki dostarczającej różne implementacje stosu.
2. Wadą obecnej implementacji jest zastosowanie statycznej tablicy w implementacji stosu. Z uwagi, iż w niektórych zastosowaniach rozmiaru stosu nie można z góry przewidzieć, należy umożliwić wykorzystanie wolniejszej, dynamicznej listy, której implementacja znajduje się w klasie *StackList*. Poprawienie tych wad jest celem zadań wymienionych w instrukcji.
3. **UWAGA:** pod koniec zajęć wyniki prac na laboratorium muszą być każdorazowo oznaczane w repozytorium jako osobny *release*. Braki w tym zakresie są równoważne z brakiem obecności na zajęciach.

---

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

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

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

#### 4. 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.

### 3.1. Refaktoryzacja do wzorca *bridge*

1. Skopiuj **implementację** stosu opartą na tablicy z klasy *Stack*, do nowej klasy *StackArray*.
2. Zaprojektuj na nowo klasę *Stack* w oparciu o lokalny atrybut typu *StackArray*:
  - wygeneruj konstruktor używając *Source* → *Generate Constructor using Fields*.
  - napisz konstruktor domyślny, żeby podklasy *Stack* mogły działać poprawnie.
  - używając opcji *Source* → *Generate Delegate Methods* wydeleguj realizację wszystkich operacji do obiektu *StackArray* (bez metod *toString*, *hashCode* i *equals*). Zweryfikuj działanie aplikacji demo *StacksDemo*,
  - w razie potrzeby zaktualizuj testy jednostkowe klasy *Stack*, tak aby domyślnie korzystały z **implementacji** *StackArray*.
3. Sprawdź działanie opcji *Navigate* → *Open Declaration (F3)* na wywołaniach metod w klasie *Stack*.
4. Według własnego pomysłu wprowadź modyfikacje, aby hierarchia klas *Stack* korzystała z implementacji stosu opartej na liście tj. *StackList*. **Nie zmieniaj interfejsu klasy *Stack***. Zweryfikuj działanie aplikacji demo *StacksDemo*. W komentarzu w klasie *StackList* wymień opcje Eclipse IDE użyteczne w wykonaniu zadania.
5. Zorganizuj klasy **implementujące** podstawowy mechanizm stosu *StackArray* oraz *StackList* we wspólną hierarchię. Do automatycznej generacji korzenia tej hierarchii użyj opcji *Refactor* → *Extract Interface*.
6. W zależności od sposobu postępowania, w klasie *Stack* mogło dojść do istotnych zmian: typ atrybutu mógł zostać zmieniony na interfejs (korzeń hierarchii **implementacji**). Jeżeli do tego nie doszło dokonaj zmiany (opcja *Refactor* → *Generalize Declared Type*).
7. W komentarzu w klasie *Stack* napisz jakie są konsekwencje zmiany omówionej w poprzednim punkcie.
8. Przenieś klasę *StackArray* do pakietu zawierającego klasę *StackList* (opcja *Refactor* → *Move* lub ***alt+shift+v***).
9. Zmień nazwę pakietu zawierającego powyższe klasy, tak aby bardziej odpowiadała zawartości (*Refactor* → *Rename* lub ***alt+shift+r***).
10. Czy masz stałe wspólne dla *StackArray* i *StackList*? Jeżeli nie przeanalizuj i popraw kod pod tym kątem. Następnie wspólne stałe przesuń do wspólnego interfejsu (*Refactor* → *Move* lub ***alt+shift+v***).
11. Zmień wartość zwracaną przez metody *peek* i *pop* w przypadku pustego stosu z -1 na 0. Czy pomogła Ci w tym realizacja zadania 10? Odpowiedź napisz w komentarzu we wspólnym interfejsie hierarchii implementacji.
12. W podklasach *Stack* wygeneruj odpowiednie konstruktory używając opcji *Source* → *Generate Constructors from Superclass*.
13. Ponownie sprawdź działanie opcji *Navigate* → *Open Declaration (F3)* na wywołaniach

metod w klasie `Stack`. Porównaj działanie z opcją `Navigate` → `Quick Type Hierarchy` (**ctrl+t**) oraz naciśniętego **ctrl** przy pracy kursora myszki. Wnioski napisz w komentarzu.

14. W celu optymalizacji w klasie `StackFIFO` zmień atrybut `temp` na stos z hierarchii implementacji (np. `StackArray`). Jaki wybór będzie najlepszy (napisz komentarz)?
15. Wzorując się na `DefaultStacksFactory` zaimplementuj dwie fabryki implementujące `StacksFactory` (opcja `New` → `Class` i `Add..` w celu wybrania interfejsu), które zwracają stosy oparte na implementacji wykorzystującej:
  - tablicę,
  - listę.UWAGA: metoda `getFalseStack` powinna zwrócić stos oparty na implementacji „przeciwnej” do domyślnej.
16. W aplikacji demo dodaj testy wykorzystujące fabryki zaimplementowane w poprzednim punkcie. Może się okazać konieczne skorzystanie z opcji `Refactor` → `Generalize Declared Type` lub `Refactor` → `Use Supertype Where Possible`.
17. Które z klas w hierarchii abstrakcji i w jaki sposób łamią zasadę izolacji? (tj. niezależność abstrakcji od implementacji). \*W jaki sposób należałoby to naprawić?
18. \*Naszkicuj diagram klas UML po refaktoryzacji do wzorca *bridge*.

### 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. \*Napisz testy jednostkowe dla pozostałych klas projektu.