Laboratorium

Temat : Podstawowe techniki utrzymania wysokiej jakości kodu

Historia zmian

Data	*Wersja*	*Autor*	*Opis zmian*
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

1. Cel laboratorium

Głównym celem laboratoriów jest przegląd podstawowych technik mających na celu poprawienie i utrzymanie wysokiej jakość 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[1]) oraz zintegrowana platforma programistyczna (np. Eclipse[2]) z zainstalowaną wtyczką do obsługi narzędzia Maven (np. m2eclipse[3]).

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

3. Laboratorium

1. Na platformie github zrób **fork** projektu biblioteki *powp_stacks_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 struktury danych stos służą do przechowywania liczb całkowitych. Wśród niestandardowych implementacji stosu jest:
   - *StackFIFO* – dostarczający pod klasycznym interfejsem stosu funkcjonalność kolejki First In First Out.
   - *StackHanoi* – na którym nie jest możliwe położenie liczby większej niż obecna na wierzchu stosu.

   Zapoznaj się z projektem, jego strukturą, klasami. Uruchom aplikację demo *StacksDemo*. Uruchom również testy jednostkowe: menu kontekstowe projektu *Run as ... → Maven test*. Zapoznaj się z testami jednostkowymi klasy *stack* (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 identyfikacje podpunktów.**
    - **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.**
    - **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.*

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 klasie StacksDemo napisz w komentarzu, które wiersze 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 nazewniczej 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 *Stack* 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 *Stack* (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.
11. Użyj annotacji @Override przy metodach tam gdzie jest to możliwe.
12. Przejrzyj projekt pod kątem klas (oprócz klas *Stack**), które nie muszą być publiczne. Przeorganizuj projekt tak, aby usunąć pliki związane z takimi klasami.

13. Uporządkuj aplikację demo *StacksDemo* (znajdującą się w *src/test/java*) rozbijając metodę *main.* Utwórz metodę statyczną *testStacks* 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. *Napisz testy jednostkowe dla pozostałych klas projektu.

3.3. *Komentarze i diagramy*

1. Naszkicuj diagram klas UML występujących w projekcie.[4]

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 :)
```

4 Można użyć np. narzędzia yuml.me lub aplikacji Visual Paradigm.