Laboratorium

Temat : Refaktoryzacja do wzorca ***bridge***

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	Ogólna przebudowa treści i aktualizacja celu
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.
2018.04.28	2.0	Tomasz Kowalski	Przeróbka projektu

1. Cel laboratorium

Głównym celem laboratoriów jest zapoznanie się ze złożonym strukturalnym wzorcem projektowym *Bridge* (pomost). Należy on do grupy wzorców strukturalnych. Zajęcia powinny pomoc 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[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:
Code Conventions for the Java TM Programming Language
Eclipse help - refactoring
http://www.vincehuston.org/dp/
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 tzw. *wyliczanek*.
2. Wadą obecnej implementacji jest zastosowanie statycznej tablicy w implementacji wewnętrznej struktury danych **stosu** wykorzystywanej przez *wyliczanki*. 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 *IntLinkedList*. 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 *tag*. 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 opartego na tablicy z klasy *DefaultCountingOutRyhmer*, do nowej klasy *IntArrayStack.*
2. Zaprojektuj na nowo klasę *DefaultCountingOutRyhmer* w oparciu o lokalny atrybut typu *IntArray*:
    - wygeneruj konstruktor używając *Source → Generate Constructor using Fields.*
    - napisz konstruktor domyślny, żeby podklasy *wyliczanek* mogły działać poprawnie.
    - używając opcji *Source → Generate Delegate Methods* wydeleguj realizację wszystkich operacji do obiektu *IntArray* (bez metod *toString*, *hashCode* i *equals*). Zweryfikuj działanie aplikacji demo *RyhmersDemo*,
    - w razie potrzeby zaktualizuj testy jednostkowe, tak aby domyślnie korzystały z **implementacji** *IntArrayStack*.

3. Sprawdź działanie opcji *Navigate → Open Declaration* (***F3***) na wywołaniach metod w klasie *DefaultCountingOutRyhmer.*

4. Według własnego pomysłu wprowadź odpowiednie modyfikacje, aby hierarchia klas *wyliczanek* korzystała z implementacji stosu opartej na liście tj. *IntLinkedList*. **Nie zmieniaj interfejsu klasy *DefaultCountingOutRyhmer.*** Zweryfikuj działanie aplikacji demo *RyhmersDemo*. W komentarzu do commitu wymień opcje Eclipse IDE użyteczne w wykonaniu zadania.

5. Zorganizuj klasy **implementujące** podstawowy mechanizm stosu *IntArrayStack* oraz *IntLinkedList* we wspólną hierarchię. Do automatycznej generacji korzenia tej hierarchii użyj opcji *Refactor → Extract Interface. (Można rozważyć skorzystanie z adaptera.)*
6. W zależności od sposobu postępowania, w klasie *DefaultCountingOutRyhmer* 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 *DefaultCountingOutRyhmer* napisz jakie są konsekwencje zmiany omówionej w poprzednim punkcie.
8. Przenieś klasę *IntArrayStack* do pakietu zawierającego klasę *IntLinkedList* (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 *IntArrayStack* i *IntLinkedList*? Jeżeli nie przeanalizuj i w razie potrzeby popraw kod pod tym kątem. Wspólne stałe przesuń do wspólnego interfejsu (*Refactor → Move* lub ***alt*+*shift*+*v***).

11. Zmień wartość zwracaną przez metody *peekaboo* i *countOut* 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 hierarchii *wyliczanek* (t.j. **abstrakcji**) 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 *FIFORyhmer* zmień atrybut *temp* na stos z **hierarchii implementacji**. Jaki wybór będzie najlepszy (napisz komentarz)?

15. Wzorując się na *DefaultRyhmersFactory* zaimplementuj dwie fabryki implementujące *RyhmersFactory* (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. Z jakim wzorcem projektowym w hierarchii klas *RyhmersFactory* mamy do czynienia?

17. 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*.

18. Które z klas w hierarchii abstrakcji i w jaki sposób łamią zasadę izolacji? (tj. niezależność abstrakcji od implementacji). *W jaki sposób należałoby to naprawić?

19. *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.