JAVA exPress

Numer 1 (1) Sierpień 2008

Eclipse RCP
oraz
NetBeans Platform

A także:

Alternatywny kurs Javy

Odpluskwianie w Eclipse

Porządki w kodzie, czyli nie tylko o refaktoringu

Hello World!

Oddaję w Wasze ręce pierwszy numer JAVA exPress. Zupełnie za darmo, bez żadnych ukrytych opłat możecie przeczytać ciekawe artykuły o Javie, a także o technologiach i narzędziach z nią związanych.

Gazeta będzie miała charakter cykliczny. Przewiduje publikację kolejnych numerów co 3 miesiące. Jeśli chcesz dostawać gazetę mailem, zapisz się do grupy http://groups.google.com/group/javaexpress lub na Polish Java Podcast (http://grzegorzduda.blogspot.com), gdzie także będą pojawiały się informacje o kolejnych wydaniach JAVA exPress.

Chciałbym, aby ta gazeta jak najlepiej Wam służyła, dlatego proszę o komentarze i wskazówki co należy poprawić. Przesyłajcie je na adres JavaExpressTeam@gmail.com

Każdy numer będzie podzielony na działy: Maszynista - czyli kilka słów ode mnie, Megafon - krótkie informacje i ogłoszenia, Poczekalnia - kącik dla początkujących, Bocznica - temat poboczny numeru, Konduktor - jakośc, wzorce i metodyki, Dworzec Główny - temat główny numeru, Maszynownia - narzędzia i biblioteki, Więcej węgla - przyszłość gazety i inne inicjatywy.

Jeśli chciałbyś zostać autorem lub pomóc w redagowaniu gazety, także zapraszam do kontaktu na powyższy adres. Wśród autorów tego i kolejnego numeru rozlosuję jedną darmową wejściówkę na tegoroczną konferencję Java Develpers Day oraz 50% zniżkę na JAOO.

Chciałem także podziękować wszystkim, dzięki którym powstał ten numer. A więc autorom artykułów: Bartkowi Kuczyńskiemu, Jakubowi Jurkiewiczowi, Mariuszowi Sieraczkiewiczowi, Jackiwi Pospychale i Markowi Klisiowi. Chciałem także podziękować Billowi Merlavage, który za darmo udostępnił zdjęcie na okładkę (więcej zdjęć na http://www.merlavageimages.com/).

Pozdrawiam,
Grzegorz Duda

Plan podróży

Zniżka na Konferencję JAOO

Na przełomie września i października (28 wrzesień - 3 październik) odbędzie się w Danii konferencja JAOO (http://jaoo.dk/conference/). Jest to jedna z najlepszych konferencji informatycznych na świecie. Można na niej posłuchać wykładów takich sław jak: Martin Fowler, Bill Venners, James Ward, Jeff Sutherland, Kirk Pepperdine, Linda Rising, Neal Ford, Ola Bini i wielu innych.

Dla naszych czytelników przygotowaliśmy specjalną **15% zniżkę**. Podczas rejestracji na konferencję należy użyć kodu promocyjnego "**dudareaders2008**".
Wśród autorów artykułów dla JAVA exPress zostanie rozlosowana 50% zniżka na JAOO. Więcej na ten temat w dziale "Więcej węgla".

Zniżka na IntelliJ IDEA

Drugą zniżkę jaką możemy Wam zaoferować, to **10% zniżki** na zakup licencji IntelliJ IDEA (http://www.jetbrains.com/?java_GRZEGORZDUDA). Jako kod promocyjny należy użyć **37986**.
Jest to najlepsze IDE do Javy, a raczej powinienem napisać IDE dla programisty Java ponieważ oprócz Javy znajdziecie w nim wsparcie dla wielu innych technologii. Także dla ostatnio modnych Groovy, Scala, Flex czy Seam. I w przeciwieństwie do innych modnych IDE, wszystko dostajecie w pakiecie, bez konieczności szukania odpowiedniego pluginu.

Java Rebel dla brązowego pasa

JavaRebel to plugin do JVM, który umożliwia wprowadzanie zmian w klasach Javy podczas działania aplikacji, bez konieczności jej restartu.
Wszyscy posiadacze brązowych pasów na JavaBlackBelt są nagradzani darmową licencją, a więc kolejny powód żeby zdać kilka egzaminów na JBB (http://www.javablackbelt.com/).
Więcej informacji o JavaRebel znajdziecie na stronie http://www.zeroturnaround.com/javarebel/.

Własny FindBugs

Znacie produkt FindBugs? Brakuje Wam w nim pewnych rzeczy? A więc warto przyjrzeć się SemmleCode (http://semmle.com/), który udostępnia język zapytań podobny do SQL, dzięki któremu możecie napisać swoje własne inspekcje.

Konferencja JDD w Krakowie

Już 16 października Kraków stanie się stolicą Javową Polski. A to za sprawą już trzeciej edycji konferencji Java Developers' Day (http://08.jdd.org.pl/).
W tym roku po raz pierwszy będą prowadzone dwa wykłady jednocześnie. A więc będzie można wybrać, co nas bardziej interesuje.
Agenda jeszcze nie jest znana, ale już wiadomo, że do Krakowa przyjedzie Ted Neward i Neal Ford. Są to światowej klasy prelegenci, a więc warto przyjechać do Krakowa w październiku.
I tutaj także niespodzianka. Wśród autorów artykułów dla JAVA exPress zostanie rozlosowana 1 bezpłatna wejściówka na JDD. Więcej informacji w dziale "Więcej węgla".

Kubek Kawy - czyli alternatywny kurs Javy

Bartek Kuczyński

Java - rodzaj kawy produkowany na wyspie Jawa. Słowo to jest też slangowym okresleniem kawy w Stanach Zjednoczonych. W Indonezji zwrot Kopi Java nie oznacza kawy jako takiej, a rodzaj mocnej, czarnej i bardzo słodkiej kawy podawanej z mielonymi ziarnami.

Kawa została sprowadzona na Jawę przez Duńczyków w XVIIw. Pierwotnie była to kawa gatunku Arabica, ale po pladze w 1880 roku została zastąpiona najpierw odmianą Liberica, a potem Robusta. Obecnie określenia Java używa się tylko w stosunku do niektórych odmian produkowanych na wyspie.

Wstęp

Cześć właśnie dorwałeś się do alternatywnego kursu języka Java. Ja nazywam się... nie inaczej. W internecie spotkasz mnie pod xwką Koziołek. Osobiście możesz mnie spotkać na imprezach oraganizowanych przez warszawski JUG

Poniższy kurs jest wynikiem długiej praktyki z językiem Java i różnymi materiałami edukacyjnymi dotyczącymi tegoż języka. Przez ten czas zebrało mi się wiele pomysłów oraz myślałem czego brakuje mi w tradycyjnych kursach. Na ich podstawie powstał ten oto kurs. Żeby ułatwić nam pracę na początek omówię jak organizujemy pracę.

Wychodzę z założenia, że nigdy nie programowałeś lub bardzo mało programowałeś. Jeżeli jesteś bardziej doświadczonym programistą lub znasz już jakiś język programowania w stopniu eksperckim to z góry przepraszam za uproszczenia. W pierwszej części kursu przybliżę Ci czym jest Java, jakie są podstawowe koncepcje i pojęcia z nią związane. Opowiem o maszynie wirtualnej i przybliżę pojęcia takie jak klasa, obiekt, interfejs. Zaprezentuję też narzędzia pozwalające na łatwia i przyjemną pracę i naukę. W drugiej części poznamy podstawowe elementy składni języka. Będzie to najnudniejsza i najbardziej najbardziej sztampowa część kursu. Niestety nie da się tego pominąć. W kolejnych częściach zajmiemy się już poważnymi sprawami. Poznasz zasady przekazywania i wysyłania danych w programach, sposób zarządzania interfejsem użytkownika czy też zasady zdroworozsądkowego programowania.

Od początku będę też odsyłał Cię do innych źródeł. Ma to na celu wyrobienie w Tobie umiejętności poszukiwania wiedzy. Jeżeli chcesz złaczyć swoją przyszłość z programowaniem to bardzo ważne jest abyś potrafił szybko i sprawnie wyszukiwać rozwiązania problemów. Nie oszukujmy się większość z nich już dawno rozwiązano, a czas poświęcony na znalezienie tego rozwiązania jest zazwyczaj krótszy niż wymyślanie koła od nowa. Tu muszę podkreślić, że samo wymyślanie koła od nowa nie zawsze jest złe, w końcu nasze koło może być bardziej okrągłe.

Czego nie znajdziesz w tym kursie. Nie będę odowływał się do innych języków porgramowania. Nie będę męczył cię przydługimi listingami kodu. Nie będę używał techniczno komputerowego bełokotu w ilościach większych niż jest to potrzebne.

Skoro już ustaliliśmy co i jak czas wziąć się do pracy.

Część 1: Kubek Javy

Otwieramy puszkę z kawą.

Tak jak kawa (zazwyczaj w wersji żużel – kawa pół na pół z wodą) jest paliwem programistów tak język java stał się jednym z najważnieszych składników współczesnego świata komputerów. Czym jest java? Jest to zorientowany obiektowo, kompilowany i uruchamiany na maszynie wirtualnej javy (JVM) język programowania. Trochę skomplikowane? Rozpracujmy tą definicję krok po kroku.

Java jest zorientowana obiektowo. Oznacza to, że w języku tym posługujemy się pojęciami takimi jak obiekt, klasa, interfejs. Java nie jest językiem wpełni obiektowym ponieważ istnieje w niej grupa typów prostych (pierwonych), które nie są obiektami. Zaraz opowiem co to są obiekty.

Java jest kompilowana to znaczy, że zanim uruchomimy nasz program jego kod jest poddawany specjalnemu procesowi, który tłumaczy nasze wypociny na coś zrozumiałego dla kom-

putera. Szczegółow co to jest kompilator poszukaj sobie w Googlu.

Java uruchamiana jest na Maszynie Virtualnnej Javy (JVM od ang. Java Virtual Machine). Jest to najważniejsza i najbardziej rozpoznawalna cecha javy. Maszyna wirtualna to specajlu program, który znajduje się pomiędzy naszym kodem, a systemem operacyjnym. Jej zadaniem jest ujednolicenie interfejsu tak aby nasz kod mógł zostać napisany raz i uruchamiany na każdym komuperze niezależnie od jego architektury. Jest to bardzo skomplikowane pojęcie, ale spróbuję je uprościć w następujący sposób.

Zapewne spotkaleś się z uniwersalnymi czytnikami kart pamięci na USB. Te małe urządzenia pozwalają na przeglądanie zawartości karty nie zależnie jakiego jest typu. Złaczka karty jest „magicznie" zamieniana na zwykłe USB. Jeżeli teraz nasz kod jest tym portem USB, a systemy operacyjne są reprezentowane przez różne typy kart to JVM jest naszym czytnikiem, który pozwala nam na stworzenie kodu raz i nie martwienie się jakiego typu jest karta, którą chcemy przeczytać. Mam nadzieję, że łapiesz o co chodzi.

Obiecałem napisać coś o obiektach no to do dzieła.

Esencja aromatu

No właśnie czas przyswoić sobie trzy podstawowe pojęcia związane z językiem java. Są to Klasa, Obiekt i Interfejs. Nie będę bawił się w rozwlekłe opisywanie co i jak. Czas skorzystać ze źródeł. W swoim czasie popełniłem „klasyczny" kurs javy na 4programmers.net (http://4programmers.net/Java/Podstawy_javy)

Klasa

Abstrakcyjny byt określający zbiór Obiektów o takich samych właściwościach.

Klasa definiuje zestaw Metod i Pól dla swoich Obiektów. Przykładem klasy może być samochód. Może on podejmować różne działania np. jechać i ma pewne właściwości np. kolor. Jest jednak abstrakcyjny, mówiąc inaczej "nienamacalny". Jeżeli mówimy o klasie Samochód to oznacza, że mówimy o jakimś samochodzie.

Ogólny wzór definicji klasy w Javie wygląda w następujący sposób:

```
[modyfikator dostępu] [abstract] class NazwaKlasy{
  // definicja pól i metod
}
```

Nasz wzorcowy samochód:

```
public class Samochod{
  public Color kolor = new Color(255,255,255);
  // pole kolor określa kolor samochodu

  public void jedzie(){
    //kod odpowiedzialny za jazdę samochodu
  }
}
```

Klasa może implementować Interfejs. Oznacza to, że klasa ma wszystkie metody, zachowania, definiowane przez interfejs. Klasa musi implementować wszystkie metody interfejsu. Tu jednak rodzi się pytanie co zrobić jeżeli nie chcemy implementować wszystkich metod? Można oczywiście zaimplementować je w taki sposób by zwracały wartości null. Jednak nie jest to dobre rozwiązanie. Można powiedzieć nawet więcej, jest to najgorsze z możliwych rozwiązań, ponieważ ukrywa fakt braku implementacji metod. Znacznie lepszym rozwiązaniem jest uczynienie danej klasy abstrakcyjną.

Klasa abstrakcyjna zawiera w sobie zwykłe metody, czyli takie które mają jawną implementację, oraz metody abstrakcyjne. Metoda abstrakcyjna nie ma implementacji. Poniżej przykład klasy abstrakcyjnej:

```
public abstract class KlasaAbstrakcyjna {
  public int pole;
  public abstract void metodaAbstrakcyjna();
  public void metodaZwykła(){
    //Kod metody
  }
}
```

Po co to wszystko? Dochodzimy do bardzo ważnego elementu programowania obiektowego jakim jest dziedziczenie. Załóżmy, że chcemy stworzyć klika klas odpowiadających różnym markom i typom samochodów. Interesuje nas to że samochody jeżdżą i mogą mieć doczepioną naczepę. Najbardziej prawidłowe podejście powinno wyglądać w następujący sposób:

- Definiujemy interfejs Samochód. Posiada on metody które odpowiadają działaniom wszystkich samochodów np.
- jedź
- załóż naczepę
- Tworzymy klasę abstrakcyjną, która implementuje interfejs Samochód w taki sposób, że metoda jedź jest wspólna dla wszystkich samochodów,

a metoda załóż naczepę abstrakcyjna.

- Tworzymy poszczególne klasy samochodów. Dziedziczą, rozszerzają, one klasę AbstrakcyjnySamochód i implementują zakładanie naczepy w zależności od potrzeb.

Nasze samochody to:

- Ferrari
- Star

Kod takiego programu powinien wyglądać mniej więcej w taki sposób:

```
// plik Samochód
public interface Samochód {
  public void jedź();
  public void załóżNaczepę();
}

//plik AbstrakcyjnySamochód
public abstract class AbstrakcyjnySamochód imple-
ments Samochód {
  public void jedź() {
    //kod odpowiedzialny za poruszanie się samochodu
  }
  public abstract void załóżNaczepę();
}

//plik Ferrari
public class Ferrari extends AbstrakcyjnySamochód {
  public void załóżNaczepę() {
    //nic nie robim. Ferrari nie może mieć naczepy
  }
}

//plik Star
public class Star extends AbstrakcyjnySamochód {
  public void załóżNaczepę() {
    //Czynności związane z zakładaniem naczepy
  }
}
```

Kilka uwag o dziedziczeniu i interfejsach w javie:

- Można dziedziczyć tylko po jednej klasie
- Klasa może implementować kilka interfejsów, ale trzeba uważać na konflikty nazw
- Interfejs może dziedziczyć po innym interfejsie
- Wszystkie obiekty dziedziczą po klasie Object

Obiekt

Byt fizyczny stanowiący instancję klasy

Obiekt jest fizyczną "manifestacją" Klasy. Oznacza to że ma własne miejsce w pamięci komputera i możemy nim manipulować. Jeżeli mówimy o obiekcie samochód będącym instancją Klasy Samochód to oznacza, że mówimy o konkretnym samochodzie.

Obiekt definiujemy jako Zmienną.

Zmienna

Byt fizyczny o którym mówimy że ma wartość

Przez pojęcie zmiennej w Javie rozumiemy referencję, wskazanie, do określonego miejsca w pamięci komputera. Zmienne reprezentują dane i pozwalają za swoim pośrednictwem nimi manipulować.

Stała

Rodzaj zmiennej, której wartość nie podlega modyfikacji

Stała to zmienna, która po nadaniu jej wartości nie może być już zmieniona. Zazwyczaj tej konstrukcji używa się do definiowania np. ustawień. Chcąc uzyskać taki efekt należy użyć słowa kluczowego final

Metoda

Działanie, które mogą podjąć Obiekty danej Klasy

Metoda określa Działanie. Dlatego najczęściej stosuje się do jej nazwania czasowniki. samochód.jedzie() - jedzie() to metoda, działanie. Metody wywołujemy za pomocą znaku . po nazwie zmiennej.

Pole

Właściwość Obiektu danej Klasy

Samochód może mieć kolor. Pole Klasy rozumiemy właściwość lub relację MA. Pole jest zmienną która należy do klasy.

Interfejs

Dobrze zdefiniowany zestaw czynności - metod, które może wykonać obiekt danej klasy

Interfejs jest jednym z najtrudniejszych do zrozumienia pojęć OOP. Jednocześnie jest stosunkowo prostym pojęciem jeżeli posłużymy się przykładem. Załóżmy że Samochód Jest Interfejsem. Oznacza to że możemy korzystać z jego Metod i Stałych. Jednocześnie możemy zdefiniować kilka różnych klas implementujących

interfejs Samochód. Wszystkie one będą miały dobrze zdefiniowane metody interfejsu. Oznacza to że jeżeli wiemy że dany obiekt implementuje interfejs samochód to możemy "w ciemno" używać metod tego interfejsu. Nie interesuje nas jak one działają, a jedynie ich parametry wywołania i to co otrzymamy w wyniku ich działania.

W życiu codziennym spotykamy się z interfejsami które zapewniają funkcjonalności, a jednocześnie nie musimy znać zasad ich działania. Przykład to pilot od telewizora. Jest interfejsem pozwalającym na włączania, zmianę kanałów i wyłączanie urządzenia.

W języku Java definicja jest następująca:

```
[modyfikator dostępu] interface NazwaInterfejsu{
  // definicja stałych i metod
}
```

W naszym przykładzie z samochodem:

```
public interface Samochod{
  public final Color kolor = new Color(0,0,0);
  // pole będące stałą. Produkujemy Fordy T ;)

  public void jedzie();

}
```

Modyfikatory dostępu

Od początku zarówno w definicji klasy jak i interfejsu, metody i pola znajduje się tajemniczy zapis [modyfikator dostępu].

Modyfikator dostępu określa w jaki sposób inne obiekty mogą otrzymać dostęp do danego pola, metody, definicji klasy czy interfejsu. W języku Java wyróżniamy cztery modyfikatory dostępu: publiczny, prywatny, chroniony i domyślny.

Modyfikator publiczny

Określany jest słowem kluczowym public. Oznacza iż do danej metody, pola, definicji klasy i interfejsu ma dostęp każdy obiekt w uniwersum javy (JU). Dostęp ten nie jest uzależniony o tego w jakim pakiecie znajduje się udostępniana własność i czy to jest ten sam pakiet co pakiet obiektu z którego pochodzi żądanie.

Modyfikator prywatny

Jest przeciwieństwem modyfikatora publicznego. Pola, metody, klasy i interfejsy oznaczone słowem private są całkowicie niewidoczne poza definicją swojego właściciela. Przykładowa klasa zawierająca zarówno pola, metody, klasy i interfejsy prywatne:

```
class PrywatnaKlasa {
  private int polePrywatne;

  private void metodaPrywatna(){}
  private interface prywatnyInterfejs{}
  private class prywatnaKlasa implements     pry-
watnyInterfejs{}
}
```

Żaden obiekt w JU nie będzie wstanie odwołać się do zawartości PrywatnaKlasa. Bardzo istotnym faktem jest to, że klasa znajdująca się na najwyższym poziomie, czyli PrywatnaKlasa nie może być prywatna.

Modyfikator chroniony

Modyfikator chroniony oznaczamy słowem kluczowym protected. Elementy oznaczone w ten sposób są widoczne tylko dla innych obiektów znajdujących się poniżej w tej samej hierarchii klas. Innymi słowy dziedziczące po klasie zawierającej elementy chronione. Oczywiście są też widoczne dla innych elemntów znajdujacych się w tym samym pakiecie i pod pakietach.

Modyfikator domyślny

Nie jest oznaczony żadnym słowem kluczowym. Nie zrozumienie jak działa modyfikator domyślny jest źródłem popularnych błędów związanych z próbą wykorzystania oznaczonych w ten sposób pól i metod. Modyfikator domyślny można zdefiniować w następujący sposób, jeżeli dostęp do pola, metody, klasy lub intrerfejsu jest oznaczony jako domyślny to mogą go otrzymać tylko te obiekty, które znajduja się w tym samym pakiecie. Oznacza to, że klasy dziedziczące po klasie zawierającej elementy z dostępem domyślnym nie będą mogły użyć tego elementu chyba, że znajdują się w tym samym pakiecie. Ten sam pakiet oznacza literalnie ten sam katalog w strukturze pakietów. Nie może to być np. podkatalog o innej strukturze pakietów nie wspominając.

Inne modyfikatory

Poza modyfikatorami dostępu funkcje i pola obiektów mogą być opisane za pomocą innych

modyfikatorów.

Modyfikator final

Pole obiektu oznaczone jako final po inicjacji nie może być modyfikowane. Oznacza to, że inicjacji pola można dokonać tylko na dwa sposoby. Poprzez jawne inicjowanie w definicji klasy:

```
class Klasa{
  final Object pole = new Object();
}
```

lub w konstruktorze:

```
class Klasa{
  final Object pole;

  public Klasa(){
    pole = new Object();
  }

  public Klasa(Object obj){
    pole = obj;
  }
}
```

Wartość pola po zainicjowaniu nie może być już zmieniana. Jeżeli pole wskazuje na jakiś obiekt to dalej można używać metod ustawiających tego obiektu i bezpośrednich odwołań do pól do ustawiania wartości pól tego obiektu.

W przypadku metody oznaczenie jej jako final powoduje, że klasy dziedziczące po klasie nie mogą przesłonić metody:

```
public class KlasaA {
  final void metoda(){}
}

class KlasaB extends KlasaA{
  // Niedozwolone! Nie można przesłonić
  // metody final
  // void metoda(){}
}
```

Jezeli klasa jest oznaczona jako final to nie można rozszerzyć tej klasy:

```
public final class KlasaA {
}
// nie można rozszerzyć klasy oznaczonej jako final
// class KlasaB extends KlasaA{}
```

Modyfikator static

Modyfikator static oznacza iż pole obiektu ma taką samą wartość dla wszystkich obiektów danej klasy. Formalnie oznacza to iż wszystkie obiekty danej klasy odwołują się do tego samego miejsca w pamięci.

Jeżeli metoda jest oznaczona jako statyczna to może być wywołana bez potrzeby tworzenia obiektu klasy definiującej tą metodę. W tym miejscu należy wspomnieć iż do zarówno metod, jak i pól oznaczonych jako static dobieramy się w inny sposób iż do normalnych metod i pól. Chcąc odwołać się do statycznego elementu należy użyć wzorca:

```
<<nazwa_klasy>>.<<nazwa_metody/nazwa_pola>>
```

Jeżeli chcemy użyć tradycyjnego odwołania:

- obiekt.metoda();
- obiekt.pole;

to kompilator zwróci ostrzeżenie The static field/method obiekt.pole/obiekt.metoda() should be accessed in a static way.

Program powinien kompilować się bez błędów i ostrzeżeń.

Modyfikator strictfp

Metoda lub klasa oznaczona w ten sposób będzie wykonywana tak by wszystkie obliczenia były zgodne ze standardem IEEE-754.

Modyfikator native

Zaawansowany modyfikator, którym oznaczane są metody wykonywane przez JNI (Java Native Interface). W praktyce oznacza to, że metody te są implementowane w języku innym niż Java, a JVM wywołując je odwołuje się do mechanizmów pozwalających na komunikację z np C++.

Modyfikator transient

Zaawansowany modyfikator związany z JPA (Java Persistence Api). Pola oznaczone w ten sposób nie są utrwalane.

Modyfikator volatile

Obok modyfikatora transient, jeden z rzadziej wykorzystywanych modyfikatorów. Oznacza, że pole oznaczone w ten sposób może być modyfikowane przez zewnętrzny proces lub inny wątek.

Na razie wystarczy tej wiedzy. W kolenym odcinku omówię składnię i polecenia języka. Kofeina w żyłach się już wypaliła więc miłej nocy.

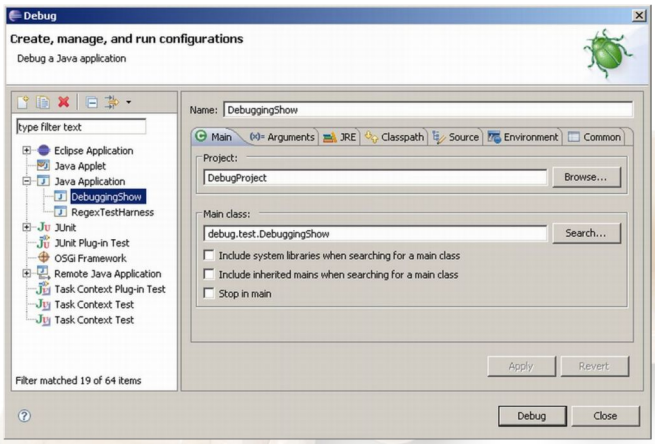
W walce o lepszą jakość - odpluskwianie

Jakub Jurkiewicz

W pracy każdego programisty przychodzi taki moment, gdy okazuje się, że nie wszystko działa tak jak przewidział – testy nie przechodzą, produkt nie chce się zbudować, a menedżer projektu pisze maile, że będzie trzeba zostać po godzinach. Na szczęście programista nie pozostaje w takiej sytuacji bezbronny, może wykorzystać sprawdzony od lat mechanizm odpluskiwania (ang. debugging) kodu w celu znalezienia błędów. Eclipse IDE oferuje, niezastąpiony w pracy developera, zestaw narzędzi do analizy kodu podczas jego wykonywania. W artykule tym zostaną przedstawione zarówno podstawowe, jak i bardziej zaawansowane rozwiązania dla odpluskwiania kodu napisanego w języku Java oferowane przez Eclipse IDE w wersji 3.3.

Jak zacząć?

Uruchomienie programu w trybie umożliwiającym odpluskwiania różni się niewiele od uruchomienia aplikacji w trybie standardowym. Z menu głównego wybieramy `Run->Open Debug Dialog`. Otworzy się nowe okno (Rysunek 1.)



Rysunek 1. Okno z nową konfiguracją uruchomienia dla odpluskwiania

Następnie z drzewa po lewej stronie wybieramy istniejącą już konfigurację uruchomienia (ang. launch configuration) dla aplikacji Java (ang. Java Application) lub też tworzymy nową (przycisk New launch configuration). Ustawiamy wszystkie interesujące nas opcje i klikamy przycisk Debug znajdujący się w prawym dolnym rogu okna.

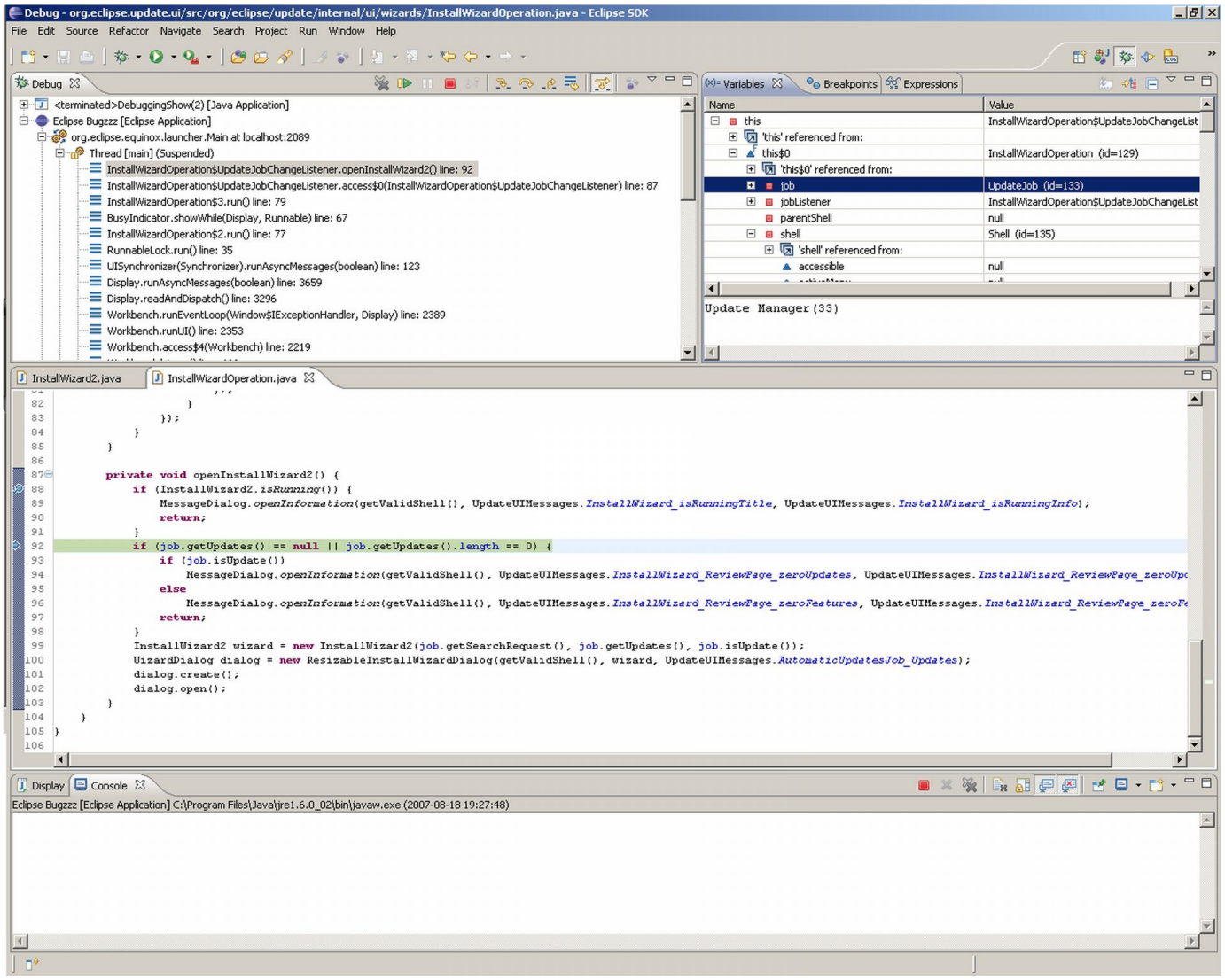
Co istotne Eclipse IDE potrafi uruchomić w tym trybie kod, który zawiera błędy kompilacji (Eclipse IDE do kompilacji wykorzystuje własny kompilatora). Jeśli ścieżka wykonywania się kodu osiągnie punkt, w którym istnieje błąd to wykonywanie się zatrzyma i Eclipse IDE pozwoli poprawić błąd, aby móc kontynuować wykonywanie kodu.

Będąc w odpowiedniej perspektywie

W celu rozpoczęcia procesu debugowania potrzebne są jeszcze tzw. punkty wstrzymania (ang. breakpoints) ustawione (przez podwójny klik na bocznym pasku przy linijce, na której taki punkt ma zostać stworzony lub przez ustawienie kursora na tejże linijce i wciśnięcie skrótu `[Ctrl+Shift+B]`) na ścieżce wykonywania się programu. Gdy uruchomimy aplikację w trybie odpluskwiania i natrafimy na któryś z punktów wstrzymania to Eclipse IDE zapyta się nas czy chcemy się przełączyć do perspektywy `Debug`. Odpowiadając pozytywnie powinniśmy zobaczyć zestaw widoków przeznaczonych do wydajnego odpluskwiania kodu. Rysunek 2. przedstawia przykładowy wygląd perspektywy Debug.

Najważniejsze widoki to:

- `Debug` - pozwala zobaczyć jaki jest stos wywołania metod w momencie zatrzymania wykonywania kodu. Możemy również cofnąć się do którejś z wcześniej wywołanych metod, aby zobaczyć jakie wartości miały zmienne w momencie wywołania kolejnej metody. Możemy również przejść do kolejnej linijki kodu (przycisk Step Over), wejść do kolejnej wywoływanej metody (przycisk Step into), wyjść z aktualne wykonywanej metody (przycisk Step return), kontynuować normalne wykonywania (przycisk Resume), lub też przerwać wykonywanie (przycisk Terminate).
- `Variables` – pozwala sprawdzać oraz edytować wartości (opcja Change Value... z menu kontektowego) pól klas oraz zmiennych dostępnych w danym bloku kodu. Po kliknięciu na danej zmiennej (lub polu) wyświetli się jej wartość (dla



Rysunek 2. Zrzut ekranu przedstawiający erspektywę Debug

typu prostego) lub wartość metody toString (dla obiektu).

- `Breakpoints` – pozwala zarządzać wszystkimi zdefiniowanymi punktami wstrzymania oraz pozwala dodawać nowe. Więcej informacji o punktach wstrzymań w dalszej części artykułu.

- Expressions – umożliwia obserwowanie wartości przez siebie zdefiniowanych wyrażeń opartych na polach klas lub zmiennych. Aby dodać nowe wyrażenie wystarczy zaznaczyć je w kodzie (np. nazwę zmiennej) kliknąć prawym przyciskiem myszy na edytorze i z menu kontekstowego wybrać opcję Watch. Można to zrobić również klikając prawym przyciskiem w polu widoku i wybierając opcję Add Watch Expression, a następnie wpisując interesujące nas wyrażenie.

Punkty wstrzymań

Jak już zostało wspomniane w celu odpluskwiania kodu potrzebne są odpowiednio ustawione punkty wstrzymań. Każdy taki punkt ma swoje właściwości, do których można się dostać klikając prawym przyciskiem na danym punkcie (na bocznym pasku przy kodzie lub w widoku `Breakpoints`) i wybierając opcję `Breakpoint properties...`. Dla każdgo punktu wstrzymań można ustawić wartość `Hit Count`, która mówi ile razy dany punkt musi zostać osiągnięty, aby wykonywanie kodu zostało wstrzymane. Pozostałe właściwości punktów wstrzymań różnią się w zależności od ich rodzaju. Eclipse IDE oferuje pięć różnych rodzajów punktów wstrzymań:

- dla linii – standardowe, dotyczące jednej konkretnej linii kodu

- dla metody – aktywne przy wejściu/wyjściu do/z metody (do ustawienia we właściwościach – odpowiednio: `Method Entry` i `Method Exit`). Możliwe jest również ustawienie punktów wstrzymań na metodach, do których nie mamy źródeł – taki punkt można stworzyć w widoku Outline (opcja `Taggle Method Breakpoint` w menu kontekstowym po kliknięciu prawym przyciskiem myszy na wybranej metodzie).

- dla pola (tzw. watchpoint) - aktywne przy dostępie lub modyfikacji wartości pola klasy (można to ustawić we właściwościach – odpowiednio: `Field access` i `Field Modyfication`).

- dla klasy – aktywne przy pierwszym ładowaniu danej klasy. Podobnie jak w przypadku metod można ustawić punkt wstrzymań na klasie, do której nie mamy źródeł – w tym celu również wykorzystujemy widok `Outline` (opcja `Taggle Class Load Breakpoint` w menu kontekstowym po kliknięciu prawym przyciskiem myszy na nazwie klasy).

- dla wyjątku – aktywne w momencie, gdy wybrany wyjątek jest rzucony. Aby stworzyć takiego breakpointa należy w widoku Brakpoints kliknąć przycisk `Add Java Exception Breakpoint` a następnie w nowym oknie określić jaka klas wyjątku nas interesuje.

Dodatkowo dla punktów wstrzymań metod i linii można definiować warunki przy których dany punkt będzie aktywny (np. gdy chcemy się zatrzymać, gdy określona zmienna przyjmie interesującą nas wartość).

Zastępowanie kodu na gorąco

Wyobraźmy sobie sytuacje, w której debugujemy kod już drugą godzinę i w końcu dochodzimy do miejsca, w którym wydaje nam się, że znaleźliśmy błąd. Dla pewności chcielibyśmy zmienić kawałek kodu i zobaczyć, czy wszystko zadziała tak jak byśmy tego chcieli. Co musimy zrobić? Zatrzymać proces odpluskwiania, zmienić kod i rozpocząć cały proces od nowa? Z Eclipse IDE nie jest to konieczne! Pozwala on na zmianę kodu w czasie jego wykonywania. Mechanizm ten, zwany zastępowaniem kodu na gorąco (ang. hot code replace), jest bardzo przydatny aby sprawdzić szybkie rozwiązania bez potrzeby restartu i reprodukcji stanu w którym akurat jesteśmy podczas odpluskwiania. Aby skorzystać z tego rozwiązania muszą być spełnione następujące warunki:

- Wirtualna Maszyna Javy (ang. Java Virtual Machine), z której korzystamy, wspiera zastępowanie kodu na gorąco.

- W Eclipse IDE włączony jest mechanizm automatycznej komplilacji (opcja w menu głównym- `Project->Build Automatically`).

- Zmiana dokonana w kodzie nie może zmieniać „kształtu” klasy, czyli żadna metoda ani pole nie mogą zostać usunięte ani dodane.

Jeśli podczas zmiany kodu wystąpi błąd kompilacji to może się okazać, że jedna z ramek na stosie wywołania metod jest niepoprawna i dalsze przetwarzanie może okazać się niemożliwe.

Odpluskwianie zdalne

Eclipse IDE jest zatem potężnym narzędziem pozwalającym na analizę kodu podczas jego wykonywania. Warto się zastanawiać, czy możliwe jest przeanalizowanie aplikacji uruchomioną lokalnie, lecz poza samym Eclipse IDE lub też zupełnie zdalnie – na innym komputerze? Oczywiście Eclipse IDE pozwala również i na to! Wygląda to podobnie to odpluskiwania aplikacji lokalnych, jedynie odpluskwiana aplikacja musi być uruchomiona z odpowiednimi parametrami (parametry te czytelnik znajdzie w ramce).

Parametry uruchamiania aplikacji do odpluskwiania zdalnego.

```
-Xdebug -Xnoagent -Xrunjdwp:transport=
dt_socket,server=y,suspend=y, address=8000
-Djava.complier=NONE
```

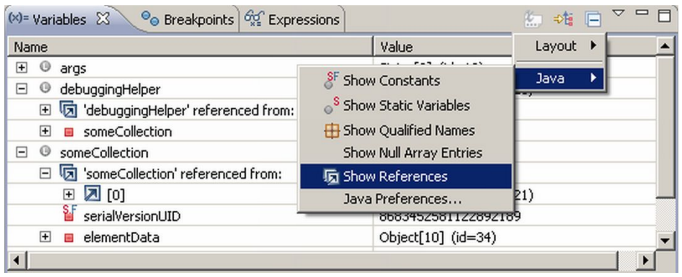
Gdy program jest uruchomiony z tymi parametrami to startuje i czeka na połączenie z narzędziem do odpluskiwania (ang. debuger). W Eclipsie IDE wybieramy z menu głównego `Debug->Open Debug Dialog` następnie tworzymy nową instancję Remote Java Aplication specyfikujemy adres pod jakim dostępna jest zdalna aplikacja oraz odpowiedni port (wartość parametru address z jakim została aplikacja uruchomiona, np. 8000). Po kliknięciu przycisku Debug Eclipse IDE łączy się ze zdalnym programem i rozpoczyna sesję odpluskwiania. Kod aplikacji jest brany z aktualnej przestrzeni pracy (ang. workspace).

Co nowego w Eclipse IDE 3.3?

W wersji 3.3 Eclipse IDE pojawiło się wiele nowości. Z punktu widzenia odpluskwiania najciekawsze są trzy nowinki (wszystkie wymagają Javy SE 6). Po pierwsze Eclipse IDE umożliwia nam zobaczenie wszystkich referencji danego obiektu. Są swa sposoby dostępu do tej informacji:

- W menu widoku Variables wybieramy `Java->Show References` (tak jak jest to zaprezentowane na Rysunku 3.). Referencje zostaną wyświetlone jako część widoku Variables.

- W widoku `Variables` klikamy prawym przyciskiem na wybranej zmiennej i z menu kontekstowego wybieramy All References lub wciskamy skrót [`Ctrl+Shift+R`]. Pojawi się wówczas dodatkowe okienko z informacją o referencjach.



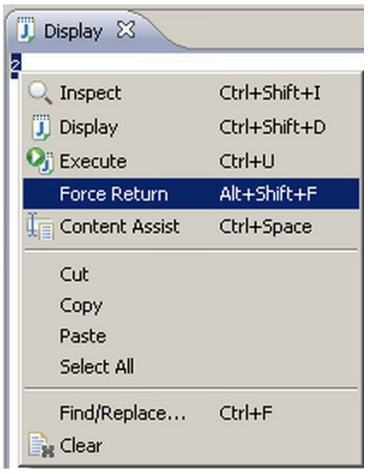
Rysunek 3. Dodawanie widoczności referencji wybranej zmiennej

Po drugie możliwe jest wyświetlenie wszystkich instancji danego typu – wystarczy tylko w edytorze kliknąć prawym przyciskiem myszy na wybranym typie lub na zmiennej w widoku Variables i z menu kontekstowego wybrać opcję All Instances, można również skorzystać ze skrótu [Ctrl + Shift + N].

```
35  public static int returnOne() {
36      return 1;
37  }
```

Rysunek 4. Edytor z metodą zwracającą nieodpowiadającą nam wartość

Po trzecie można wymusić wcześniejsze wyjście z metody, w której akurat zatrzymane jest przetwarzanie. Aby skorzystać z tej opcji należy zaznaczyć w edytorze wyrażenie, które ma zostać zwrócone i z menu kontekstowego wybrać pozycję `Force Return`. Jeśli chcemy zwrócić określoną przez nas samych wartość należy w widoku Display wpisać wartość, która ma zostać zwrócona, zaznaczyć ją, kliknąć prawym przyciskiem myszy na zaznaczeniu i z menu kontekstowego wybrać opcję `Force return` lub użyć skrót [`Alt+Shift+F`]. Przykładowo jeśli jesteśmy akurat w metodzie zwracającej wartość int (Rysunek 4.) i zgodnie z przetwarzaniem metoda zwróciłaby wartość 1, a my chcemy tę wartość zmienić na 2 to w widoku `Display` wpisujemy wartość 2, zaznaczamy ją i z menu kontekstowego wybieramy odpowiednią opcję (Rysunek 5.).



Rysunek 5. Wymuszenie zwrócenia wybranej wartości

Podsumowanie

W artykule zostały przedstawione możliwości Eclipse IDE w zakresie analizy wykonywanego programu. Używając Eclipse IDE programista nigdy nie zostaje sam z błędem, który wkradł się do programu. Eclipse IDE stara się ułatwić żmudne szukanie błędów przy pomocy wielu użytecznych widoków, różnorakich rodzajów punktów wstrzymań, mechanizmu zastępowania kodu na gorąco, a także odpluskwiania aplikacji zdalnych.

Nie tylko refaktoring

Mariusz Sieraczkiewicz

Początkowo moim zamysłem było stworzenie artykułu o refaktoringu. Jednak im bardziej zastanawiałem się nad tematem, tym bardziej utwierdzałem się w przekonaniu, iż nie będę pisał tylko i wyłącznie o refaktoringu. Chodzi o coś znacznie istotniejszego, o przelanie bardzo rozległej wiedzy, a w zasadzie doświadczenia związanego z tworzeniem kodu. Kodu, który nie tylko działa, nie tylko jest dobrze zaprojektowany, ale przede wszystkim doskonale się czyta. Kiedy osiągamy tę umiejętność, stajemy u progu profesjonalizmu. Programistycznego profesjonalizmu.

Zatem będzie to artykuł między innymi o refaktoringu, ale wzbogacony o zbiór przemyśleń, sugestii, czasami również wątpliwości, którą mają pobudzić Cię, Czytelniku, do refleksji, zweryfikowania swoich programistycznych poczynań. Wierzę, że spowodują cały proces zmian – wprowadzenia nowych, dobrych nawyków.

Przede wszystkim czytelność

Programowanie bardzo szybko ewoluuje. Pamiętam jeszcze dość dobrze czasy, kiedy rozpocząłem swoją przygodę z kodowaniem jakieś dziesięć lat temu. Programy pisało się wtedy całkiem inaczej. Ceniono pomysłowość, zwięzłość i enigmatyczność. Im kod był bardziej niezrozumiały, tym programista był lepszy.

Jednak z czasem systemy informatyczne stawały się coraz bardziej skomplikowane, wymagały coraz większej wiedzy i co najważniejsze, stały się produktem pracy zespołowej. Obecnie pojedynczy programista nie jest w stanie zdziałać zbyt wiele. Być może stworzy rozbudowany program desktopowy, natomiast nie będzie w stanie w wystarczająco skończonym czasie stworzyć rozproszonego systemu, opartego o architekturę trójwarstwową, zapewniającego odpowiedni poziom bezpieczeństwa, umożliwiającego zarządzanie prawami dostępu do wybranych części aplikacji, realizującym wielojęzyczność itp. itd. Takie systemy tworzy obecnie kilkunastu lub kilkudziesięciu programistów, w zależności od wielkości projektu, przez kilka lub kilkanaście miesięcy. Programista przestał być nierozumianym przez nikogo indywidualistą, a stał się graczem zespołowym, nastawionym na współpracę.

Co za tym idzie, sposób kodowania też musiał się zmienić. Wyłonił się podstawowy postulat dotyczący kodowania:

Przede wszystkim czytelność

Istnieją przynajmniej trzy podstawowe powody, które potwierdzają ważność tego stwierdzenia:

- wymagania się zmieniają,
- programowanie to umiejętność zespołowa,
- projekty są zbyt duże, aby pojedyncza osoba była w stanie ogarnąć całość.

Z tych właśnie powodu w ciągu ostatnich kilku lat bardzo mocno rozwijają się takie techniki jak refaktoring, pisanie testów oraz zwraca się ogromną uwagę na standard kodowania.

To właśnie Czytelność będzie głównym bohaterem tego artykułu. Będzie on zawierać sugestie i przemyślenia, które ułatwią realizację powyższego postulatu. Niektóre wskazówki będą stanowić moją subiektywną opinię, inne będą wyrażać mądrość doświadczeń społeczności programistycznej. Oczywiście należy pamiętać o pewnej zasadzie: “Jedyną niezmienną zasadą jest to, że nie ma niezmiennych zasad”. Uogólniając, należy stwierdzić, iż przedstawiane wnioski sprawdziły się w wielu sytuacjach, co nie znaczy, że są zasadne w 100% przypadkach. Dlatego należy uważnie się przyglądać pojawiającym się na co dzień problemom i odważnie stosować przytoczone wskazówki. Warto krytycznie spojrzeć na swoje nawyki lub ich brak i rozpocząć zmiany. Zatem do dzieła!

Poprzez przykład do celu

Analiza kodu mniej doświadczonych programistów, często doprowadzała mnie do zaskakujących spostrzeżeń, umożliwiających znalezienie źródła problemów młodych (ale również i tych doświadczonych) adeptów sztuki programowania.

Dlatego artykuł ten oparty będzie o przykład nie najlepiej napisanej klasy, która będzie analizowana i stopniowo udoskonalana.

Celem, postawionym przed autorami poniższego kodu, było zaimplementowanie klasy pochodnej klasy java.util.BitSet (wektora bitowego) wzbogaconej o:

- możliwość konkatenacji,
- właściwość narzuconej długości wektora (pole length),
- specyficznego mnożenia dwóch wektorów bitowych polegającego na zwróceniu wartości 0, jeśli jedynki w obu wektorach bitowych powtarzają się na parzystej ilości miejsc, oraz wartości 1, jeśli jedynki pokrywają się na nieparzystej ilości miejsc,
- operację zamiany wektora na ciąg znakowy (w określonym z góry formacie),
- operację zamiany wektora w ciąg bajtów.

Pragnę zaznaczyć, iż treść przykładu nie ma tu większego znaczenia. Przytoczony kod służy tylko jako ilustracja często występujących niedoskonałości programistycznych. Ponadto, ponieważ nieodłączną częścią refaktoringu są testy, sprawdzające testowany kod, jako dodatek do artykułu została zamieszczona klasa testowa do analizowanej klasy.

Oto zaproponowana implementacja nowej wersji wektora bitowego:

```
import java.util.* ;

public class ExtendedBitSet extends BitSet {
  int length ;

  public ExtendedBitSet(int size, String str) {
    super(size) ;
    length = size ;
    int strLength = str.length();
    for(int i = 0; i < strLength; ++i) {
      if(str.charAt( strLength - 1 - i) == '1') {
        set(i) ;
      }
    }
  }

  public ExtendedBitSet(String str) {
    this( str.length(), str );
    int strLength = str.length();
    for(int i = 0; i < strLength; ++i) {
      if(str.charAt( strLength - 1 - i) == '1') {
        set(i) ;
      }
    }
  }

  public static ExtendedBitSet merge(
    ExtendedBitSet a, ExtendedBitSet b) {
    StringBuffer str = new
      StringBuffer(a.convertToBitString() +
      b.convertToBitString()) ;
    return new ExtendedBitSet(a.length + b.length,
      str.toString()) ;
  }

  public static int boolMultiply(
    ExtendedBitSet a, ExtendedBitSet b) {
    int sum = 0 ;
    int len ;
    if(a.length < b.length) {
      len = a.length ;
    } else {
      len = b.length ;
    }
    for(int i = 0; i < len; i++) {
      if (a.get(i) && b.get(i)) {
        sum++ ;
      }
    }
    return sum % 2 ;
  }

  public byte[] toByteArray() {
    int bytesNumber ;
    if(length % 8 == 0) {
      bytesNumber = length / 8 ;
    } else {
      bytesNumber = length / 8 + 1 ;
    }
    byte[] arr = new byte[bytesNumber] ;
    for(int j = bytesNumber - 1, k = 0; j >= 0 ;
      j--, k++) {
      for(int i = j * 8 ; i < (j + 1) * 8; i++) {
        if(i == length) {
          break ;
        }

        if(get(i)) {
          arr[k] += (byte)Math.pow(2, i % 8) ;
        }
      }
    }
    return arr ;
  }

  public String convertToBitString( int size ) {
    char [] resultArray = new char[ size ];
    for ( int i = 0; i < size; ++i ) {
      resultArray[ i ] = '0';
    }
    for ( int i = this.nextSetBit(0); i >= 0;
      i = this.nextSetBit(i + 1) ) {
      resultArray[ size - 1 - i ] = '1';
    }
    return new String( resultArray );
  }

  public String convertToBitString() {
    return convertToBitString( this.length );
  }
}
```

W pierwszej kolejności spójrzmy na klasę całościowo. Jedna z pierwszych rzeczy, która rzuca się w oczy to fakt, że metody konkatenacji i mnożenia wektorów są statyczne. Jest to sprzeczne z bardzo ważną zasadą:

Twórz spójne interfejsy i klasy

Jeśli przyjrzymy się klasie bazowej `BitSet`, łatwo zauważymy, iż żadna publiczna metoda nie jest statyczna. Dostępne są m. in. niestatyczne metody or (`Bitset`), xor (`Bitset`), których

celem jest modyfikacja obiektu na rzecz którego są one wywoływane (operacja na this), a nie udostępnienie metody zewnętrznej (statycznej), która tworzy nowy obiekt, będący efektem implementowanej operacji. Zatem obydwie metody (merge i boolMultiply) swoją postacią wprowadzają rozdźwięk w strukturze nowej klasy, prowadząc do niespójnego interfejsu klasy `ExtendedBitSet`. W tym przypadku utrzymanie spójności poprzez zamianę metod statycznych na metody niestatyczne, uprości używanie klasy ExtendedBitSet, gdyż będzie się z niej korzystać tak samo jak z klasy `BitSet`.

Istnieje jeszcze jedna zasada, którą warto przytoczyć analizując metody `merge` i `boolMultiply`:

Unikaj statycznych elementów w programowaniu

Elementy statyczne to pozostałość po programowaniu proceduralnym, gdyż statyczność oznacza globalność. A przecież jedną z konsekwencji programowania obiektowego jest zamykanie implementowanych funkcjonalności w autonomicznych i możliwie jak najbardziej niezależnych obiektach. Dlatego elementów statyczne używaj tylko wtedy, kiedy nie ma innego wyjścia lub kiedy informacja lub operacja ma rzeczywiście charakter globalny. Zatem używaj pól statycznych jako stałych, szczególnie stałych globalnych, zaś metod statycznych używaj dla operacji globalnych. Przykładem użycia metod i pól statycznych jest wzorzec Singletonu, jednak „wzorcowość" tego wzorca bywa kwestionowana (więcej przeczytasz na stronie http://c2.com/cgi/wiki?SingletonsAreEvil). Ponadto należy pamiętać, że metody statyczne nie są polimorficzne, co oznacza, że nie możemy dostarczyć ich alternatywnych implementacji oraz że nie możemy ich zastępować za pomocą mocków. Zatem ich użycie powoduje usztywnienie kodu oraz utrudnia testowanie.

Zmieńmy zatem nieco przytoczony kod, zgodnie z pierwszymi dwoma regułami:

```
  public void merge(ExtendedBitSet extendedBitSet)
{
    for ( int i = extendedBitSet.nextSetBit(0);
      i >= 0;  i = extendedBitSet.nextSetBit(i+1))
{
      this.set( this.length + i );
    }
    this.length = this.length +
      extendedBitSet.length;
  }

  public int boolMultiply(
    ExtendedBitSet extendedBitSet ) {
    int sum = 0 ;
    int len ;
    if(this.length < extendedBitSet.length) {
      len = this.length ;
    } else {
      len = extendedBitSet.length ;
    }
    for(int i = 0; i < len; i++) {
      if (this.get(i) && extendedBitSet.get(i)) {
        sum++ ;
      }
    }
    return sum % 2 ;
  }
```

Eclipse RCP

Jacek Pospychała

RCP, czyli Rich Client Platform, doskonale wpisuje się w trend w jakim zmierzają wszyscy twórcy aplikacji użytkowych. Bogactwo środowisk, czy „frameworków", upraszczających budowę interfejsów użytkownika, oraz konstrukcję aplikacji (także internetowych) świadczy o zapotrzebowaniu na narzędzia, dzięki którym twórcy oprogramowania będą mogli więcej czasu poświęcać logice swoich aplikacji, a mniej szczegółom technicznym, unikać pisania tzw. stałych części aplikacji i czynić je łatwiejszymi do zarządzania, utrzymania i dalszego rozwoju.

Niniejszy artykuł prezentuje platformę wspomagającą tworzenie aplikacji biurkowych, opartych o te same elementy co popularne środowisko programistyczne Eclipse. W dalszej części artykułu zakładam że czytelnik miał kontakt ze środowiskiem Eclipse oraz zna podstawy programowania w Javie.

Zanim zaczniemy

By zacząć pracę z RCP należy się upewnić że Eclipse, którym dysponujemy zawiera odpowiednie wtyczki. Na potrzeby tego artykułu korzystano z instalacji Eclipse for RCP/Plug-in Developers w wersji 3.3, dostępnej na www.eclipse.org. Jeśli już posiadamy Eclipse 3.3, będzie się on nadawał pod warunkiem że ma perspektywę Plug-in Development. Jeżeli jej nie ma, wystarczy doinstalować opcję Eclipse Plug-in Development Environment (PDE) korzystając z menu `Help -> Software Updates -> Find and Install...` i wybierając następnie Search for new features to install i Europa Discovery Site.

W dalszej części stworzymy prostą aplikację RCP, korzystając z szablonów dostępnych w PDE. W raz z rozwojem naszej aplikacji można ją wzbogacać o elementy standardowo dostępne w Eclipse, a więc np. własne widoki (charakterystyczne „małe okienka", które znamy z codziennej pracy z Eclipse), preferencje użytkownika, system pomocy, a nawet cały mechanizm automatycznej aktualizacji. Swoją aplikację można rozbudowywać niemal jak z klocków, dodając wtyczki dostarczane przez innych, np. do raportowania, czy do graficznej edycji danych. Do tego tematu wrócimy w dalszej części artykułu.

Podstawy architektury

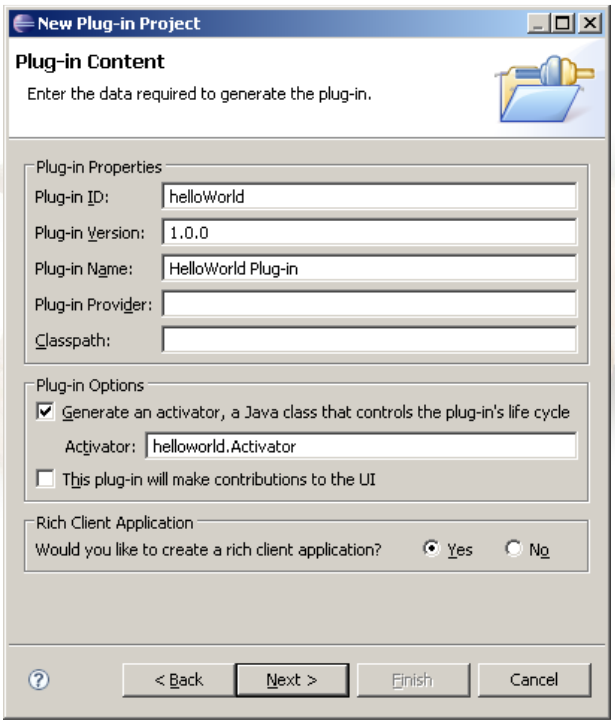
Zanim jednak stworzymy pierwszą aplikację RCP, by lepiej zrozumieć miejsce i rolę jaką będzie odgrywał nasz przyszły kod, pokrótce przyjrzyjmy się budowie samego Eclipse. Eclipse słynie z wtyczek (ang. plug-in), w rzeczywistości całość kodu w Javie jest podzielona na nie i są one domyślnie przechowywane w katalogu plugins w głównym katalogu Eclipse. Wtyczki które realizują wspólne funkcje, np. możliwości edycji Javy, czy mechanizm pomocy są pogrupowane w składniki, czy bardziej znane z angielskiego interfejsu Eclipse features. Składniki są przechowywane w katalogu features. Składnik w najprostszej postaci ma tylko plik xml definiujący listę wtyczek, jakie się na niego składają. Składniki znacznie upraszczają zarządzanie wtyczkami, których mogą być setki. Z poziomu managera aktualizacji dostępnego w `Help -> Software Updates` widzimy tylko składniki, możemy je dodawać, włączać i wyłączać, a także diagnozować problemy. Składniki mają także dodatkowe opisy i ikony ułatwiające użytkownikowi ich identyfikację. Tworząc własne wtyczki z myślą o wykorzystaniu istniejącego mechanizmu aktualizacji, warto więc przygotować także własny składnik. Poza wtyczkami i składnikami w katalogu Eclipse, znajduje się także folder configuration zawierający informacje pozwalające wystartować całe środowisko, oraz wszelkiego rodzaju informacje gromadzone w trakcie działania, np. cache, preferencje, pamięć o ostatnich operacjach użytkownika itp. Ostatnim niezbędnym elementem jest binarny plik wykonywalny, np. w Windows jest to plik eclipse.exe. Ten bardzo prosty program służy do uruchomienia Eclipse w nowej maszynie wirtualnej Javy z parametrami przekazanymi z linii poleceń oraz z pliku konfiguracyjnego (eclipse.ini).

Wracając do wtyczek, najczęściej w ramach jednej funkcjonalności, np. środowiska do Javy, istnieje podział na osobne wtyczki zawierające wła-

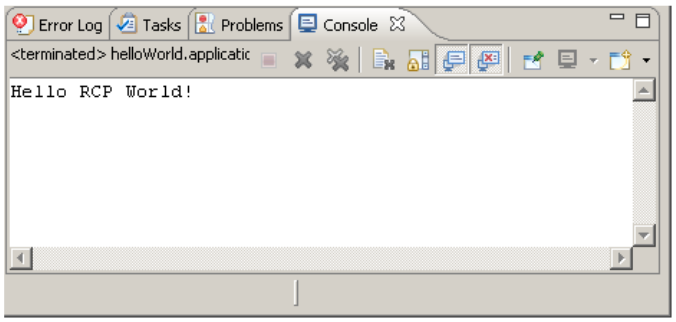
ściwą logikę (np. kompilator – org.eclipse.jdt.core), elementy interfejsu graficznego (org.eclipse.jdt.ui), dokumentację (org.eclipse.jd.doc), itp. Jest to podział czysto umowny, bo całość na ogół równie dobrze mogłaby być reprezentowana w postaci jednej wtyczki, jednak ułatwia zarządzanie wtyczkami i np. podmianę tylko wybranych w zależności od potrzeb. Również użytkownicy wykorzystujący istniejące wtyczki na tym korzystają, bo w swoich aplikacjach RCP mogą załączać np. same narzędzia kompilacji, bez związanych z nimi elementów interfejsu, czy dokumentacji.

HelloWorld!

W naszym przykładzie dla wygody ograniczymy się do jednej wtyczki. Po jej stworzeniu przejdziemy do uruchamiania i testowania a następnie wyeksportujemy jako produkt gotowy dla docelowego użytkownika. Nową wtyczkę tworzymy wybierając `New -> Project...` a następnie Plug-in Project. W kreatorze podajemy nazwę helloWorld, naciskamy Next, na kolejnej stronie upewniamy się że opcja Would you like to create rich client application jest wybrana i jednocześnie wyłączamy opcję This plug-in will make contributions to UI, potwierdzamy Next, wybieramy Headless Hello RCP i Finish. Akceptujemy propozycję środowiska, aby przejść do perspektywy Plug-in development - jest ona prawie taka sama, jak perspektywa Javy. Zanim przyjrzymy się bliżej wygenerowanemu projektowi, najlepiej od razu zaryzykować i uruchomić go klikając prawym przyciskiem na projekcie i wybierając z menu kontekstowego `Run as -> Eclipse Application`. Uruchomienie zajmie tylko chwilę i spowoduje wypisanie na konsoli Hello RCP World!. To nasze pierwsze RCP!



Rysunek 1. Kreator podczas tworzenia projektu helloWorld



Rysunek 2. Wyniki działania programu helloWorld

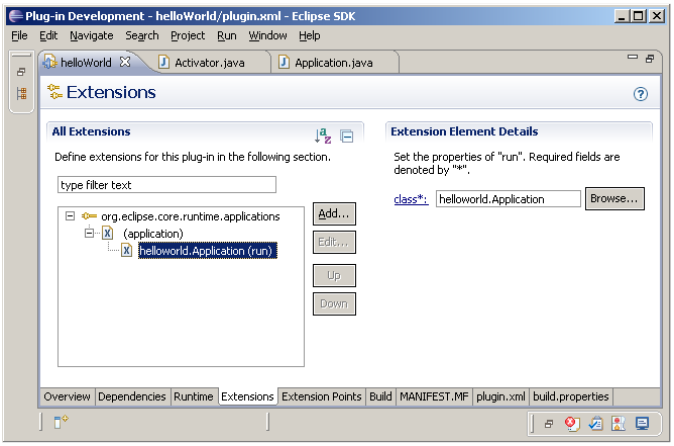
Pora zajrzeć do projektu helloWorld. Składa się na niego kod źródłowy, w katalogu src. Jest to pakiet helloworld z dwiema klasami: `Activator` oraz `Application`. Kolejny element, to zależności niezbędne do działania projektu – biblioteka JRE – wymagana do kompilacji wszelkiego kodu w javie, oraz Plug-in Dependencies – lista innych wtyczek wymaganych przez tą stworzoną przez nas. Niemal każda wtyczka ma jakieś wymagania co do swojego środowiska pracy. Pozostałe elementy, tj. META-INF/MANIFEST.MF, plugin.xml oraz build.properties to pliki konfiguracyjne. Zarówno MANIFEST.MF jak i plugin.xml są edytowane korzystając z jednego, wspólnego edytora - Plug-in Manifest Editor i definiują własności całej wtyczki. Zmieniając wartości w tym edytorze tak naprawdę modyfikujemy MANIFEST.MF i plugin.xml. MANIFEST.MF jest zgodny z typowym manifestem plików JAR. Jest on jednak rozszerzony o informacje specyficzne dla standardu OSGI, na którym opiera się Eclipse. Niesie m.in. takie informacje jak nazwa i wersja wtyczki, wymagane inne wtyczki, wymagana wersja Javy, oraz wiele innych, dodatkowych pozwalających poprawnie zainicjalizować wtyczkę. Z kolei plugin.xml wiąże się z punktami rozszerzeń.

Koncepcja punktów rozszerzeń pozwala na zmniejszenie zależności między klasami z różnych wtyczek, oraz umożliwienie rozszerzania dowolnych koncepcji aplikacji przez innych dostawców. Brzmi to trochę abstrakcyjnie, ale spoglądając na pierwszy lepszy element interfejsu użytkownika, np. widoki, zauważymy, że nie są one statyczne. Wręcz przeciwnie, zmieniają się zależnie od sytuacji, perspektywy i zbioru zainstalowanych wty-

czek. Dzieje się tak dlatego, że jedna z głównych wtyczek interfejsu graficznego Eclipse (w tym wypadku org.eclipse.ui) ma zdefiniowany punkt rozszerzeń (org.eclipse.ui.views – nazewnictwo punktów rozszerzeń ma podobną konwencję jak nazwy klas, jednak to tylko identyfikator, nie klasa). Do tak zdefiniowanego punktu rozszerzeń każdy może się podpiąć i dostarczyć własną implementację (w tym przypadku własnym widok) z dowolnej wtyczki. Mechanizm rozszerzeń pozwala zredukować zależności między klasami Javy, oraz uniezależnić się od szczegółów implementacji.

Rozszerzenie jest dodawane poprzez wpis w pliku plugin.xml. W zależności od typu rozszerzenia czasem konieczne jest zaimplementowanie odpowiedniej klasy, jednak nie jest to regułą. W samym tylko interfejsie użytkownika jest kilkadziesiąt punktów rozszerzeń, pozwalających dodawać m.in. edytory, perspektywy, menu, kreatory, akcje, skróty klawiszowe – by wymienić tylko kilka. Poza GUI wiele wtyczek udostępnia bardzo wyspecjalizowane punkty rozszerzeń, np. system pomocy pozwala dodawać kolejne rozdziały pomocy, słowa kluczowe, czy też silniki wyszukiwania (ang. search engines).

Nasz pierwszy projekt nie korzysta z GUI, jednak korzysta z jednego punktu rozszerzeń, o nazwie org.eclipse.core.runtime.applications, by zdefiniować aplikację która ma być uruchamiana (tą klasę, która wypisuje Hello RCP World!). Poprzez dwuklik na pliku plugin.xml otwieramy edytor manifestu wtyczki, na karcie Extensions, gdzie widzimy rozszerzenie org.eclipse.core.runtime.applications a po rozwinięciu drzewka, element helloworld.Application, oraz po prawej stronie nazwę klasy helloworld.Application. Na ogół klasa wskazywana w punkcie rozszerzeń musi implementować odpowiedni interfejs, co pozwoli środowisku we właściwy sposób skorzystać z klasy. W tym przypadku, gdy otworzymy klasę helloworld.Application, zauważymy że faktycznie implementuje ona interfejs IApplication i posiada dwie metody start i stop. Klasa ta jest bardzo prosta bo nasza aplikacja ma ubogi – tekstowy interfejs. W przyszłości zobaczmy analogiczną klasę, która inicjalizuje środowisko graficzne. Jeszcze chwilę uwagi należy poświęcić drugiej klasie – helloworld.Activator. Na pierwszy rzut oka jest bardzo podobna do Application – również ma metody start i stop. Jednak o ile klasa Application reprezentuje całą aplikację – na ogół jest tylko jedna dla całego zbioru wtyczek w naszym RCP, to klasy Activator są specyficzne dla poszczególnych wtyczek i każda osobno może mieć tam kod niezbędny do wykonania w momencie inicjalizacji, np. wykonanie połączenia z bazą danych, lub załadowanie zasobów.



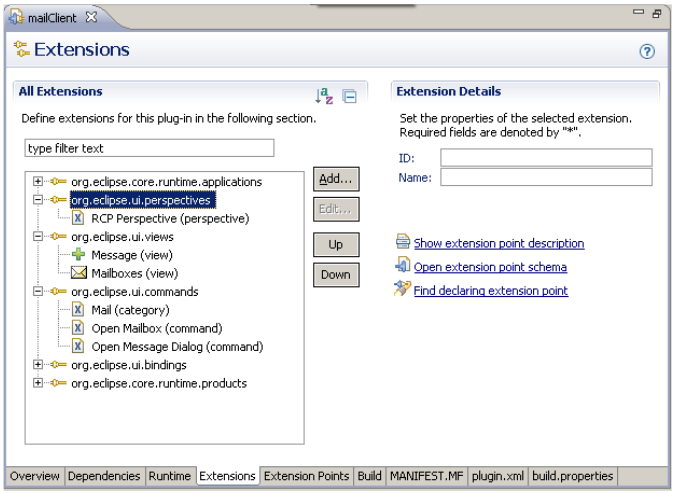
Rysunek 3. Punkty rozszerzeń w projekcie helloWorld

Drugi przykład

Teraz, gdy znamy już strukturę wtyczek, wróćmy do kreatora w menu `New -> Project...-> Plug-in Project`. Stworzymy ciekawszą aplikację – z interfejsem użytkownika. Będzie to klient poczty. Tym razem w kreatorze, na ekranie Plug-in Content zaznaczamy This plug-in will make contributions to UI i oczywiście Would you like to create a rich client application?. Po naciśnięciu Next pojawia się inny zestaw przykładów. Wybieramy ostatni - RCP Mail Template. Wygenerowany projekt jest tym razem znacznie bardziej rozbudowany od poprzedniego. Co prawda nie jest to w pełni funkcjnalna aplikacja pocztowa, a jedynie interfejs użytkownika, jednak nie powinno to przeszkadzać w poznawaniu zagadnień RCP.

Warto zwrócić uwagę plik plugin.xml. Po otwarciu go, na karcie Extensions zauważymy dużo więcej wykorzystanych punków rozszerzeń, m.in. definicję własnej perspektywy (org.eclipse.ui.perspectives), czy własnego widoku (org.eclipse.ui.views). Perspektywa definiuje w jaki sposób mają być ułożone główne elementy w oknie aplikacji, czyli widoki. W tym przypadku okno składa się z widoku folderów (klasa NavigationView) – poczty przychodzącej, wychodzącej, itp. W głównej części jest natomiast miejsce na widoki wiadomości (klasa View). Ponadto jest zdefiniowany szereg komend – abstrakcyjnych czynności, jak np. otwarcie wiadomości (punkt rozszerzeń org.eclipse.ui.commands), oraz definicje skrótów klawiszowych (punkt rozszerzeń org.eclip-

se.ui.bindings). Jeśli wspomnimy poprzedni przykład, była tam klasa Application, która jest także i w tym przykładzie. Jednak tym razem jest bardziej rozbudowana i uruchamia interfejs graficzny naszego programu. By uruchomić klienta pocztowego ponownie należy kliknąć prawym klawiszem myszy na projekcie i z menu wybrać `Run as -> Eclipse Application`.



Rysunek 4. Punkty rozszerzeń w projekcie mailClient

Poniżej omówione zostaną poszczególne klasy klienta pocztowego.

- `Activator` – Niejako punkt zaczepienia wtyczki w środowisku Eclipse. Jej położenie wskazuje nieobowiązkowe pole Bundle-Activator w pliku MANIFEST.MF. W klasie tej można zdefiniować akcje wykonywane podczas inicjalizacji, lub przed usunięciem wtyczki z środowiska. Warto przy tym pamiętać, że wtyczki są w tym względzie bardzo elastyczne, a usunięcie wybranej z nich powoduje zniknięcia wszystkich jej funkcji i nie wymaga restartu środowiska. Ponadto ta klasa zawiera id wtyczki (PLUGIN_ID), ustawienia preferencji (`getPreferenceStore()`), czy pozwala odwoływać się do jej plików.

- Application – Ta klasa implementuje interfejs IApplication. Typowo w całej aplikacji RCP jest tylko jedna klasa IApplication, chociaż nie jest to z góry ograniczone. Klasa tego typu pozwala zaimplementować dodatkowe akcje wykonywane podczas uruchamiania lub zamykania aplikacji.

- `ApplicationActionBarAdvisor` – Konfiguracja pasków i menu aplikacji, w szczególności górnego paska opcji (`MenuBar`), czesto znajdującego się pod nim paska z ikonami (`CoolBar`), paska statusu (`StatusLine`). W przykładowej aplikacji, klasa ta definiuje kilka standardowych akcji korzystając z fabryki `ActionFactory`:

```
exitAction = ActionFactory.QUIT.create(window);
register(exitAction);
aboutAction = ActionFactory.ABOUT.create(window);
register(aboutAction);
```

Tutaj rejestrowane są także własne akcje klienta poczty – akcja do otwierania widoków (`OpenViewAction`) i do wyświetlania komunikatów (`MessagePopupAction`).

Ponadto, w metodzie fillMenuBar(IMenuManager) definiowana jest struktura menu. W menu można umieszczać bezpośrednio utworzone wcześniej akcje, lub tylko zaznaczać miejsca dodania nowych akcji (ang. Group Markers). Dzięki temu programiści innych wtyczek z łatwością będą mogli dodać własne akcje do menu.

- `ApplicationWorkbenchAdvisor` – Konfiguracja wyglądu aplikacji. W przykładowej implementacji, w tym miejscu zdefiniowana jest tylko domyślna perspektywa, wg. której program ma skonfigurować okno przy uruchomieniu aplikacji. Ta klasa dostarcza także szczegółowej konfiguracji głównego okna (w metodzie `createWorkbenchAdvisor(IWorkbenchWindowConfigurer)`)

- `ApplicationWorkbenchWindowAdvisor` – Konfiguracja głównego okna (zwracana przez `ApplicationWorkbenchAdvisor`). Zawiera szczegóły techniczne wyglądu okna, jak np. jego rozmiar, położenie, elementy które mają być widoczne (pasek stanu, menu). Rozszerzając implementacje metod nadrzędnej klasy, (`WorkbenchWindowAdvisor`) można zdefiniować dodatkowe akcje, które miałyby być wykonane w różnych etapach cyklu życia okna, jak np. przed jego utworzeniem, po jego otwarciu, itp.

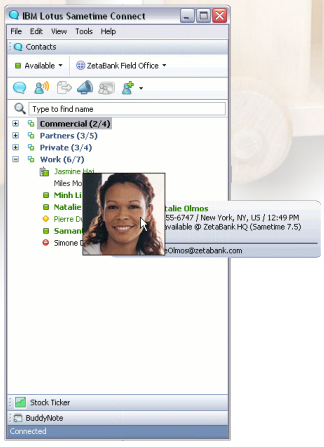
- `ICommandIds` – Definicje nazw akcji klienta pocztowego. Klasa pomocnicza pozwala uniknąć każdorazowego powtarzania nazw bezpośrednio w kodzie. Nazwy te odnoszą się do akcji zdefiniowanych w pliku plugin.xml.

- `MessagePopupAction`, `OpenViewAction` – Przykładowe akcje. Akcje w Eclipse reprezentują elementarne czynności, jakie może wykonać użytkownik. W przykładowym kliencie nie jest ich zbyt wiele, jednak niezależnie od liczby, wszystkie akcje cechują pewne wspólne elementy. Każda akcja ma swój unikalny identyfikator, nazwę czytelną dla użytkownika, ikonę oraz oczywiście samą implementację czynności

(w metodzie `run()`). Listę wszystkich dostępnych w danej chwili akcji można przejrzeć otwierając interfejs `IAction` (skrót klawiszowy `Ctrl+Shift+T` i wpisując `IAction`), a następnie przeglądając hierarchię klas implementujących ten interfejs (po zaznaczeniu w edytorze interfejsu fragmentu tekstu `IAction`, naciskamy `Ctrl+T`). Na liście znajdziemy więc poza definicjami własnych klas, także całą listę standardowych akcji domyślnie zaimplementowanych w środowisku.

- `NavigationView, View` – implementacje przykładowych widoków. Podobnie jak akcje, również i widoki mają swoje unikalne identyfikatory, ponadto aby widok był dostępny i można go było wyświetlić w oknie aplikacji, musi być zarejestrowany w punkcie rozszerzeń org.eclipse.ui.views. Widoki muszą implementować interfejs `IViewPart`, jednak najczęściej rozszerzają klasę `ViewPart`, zawierającą implementacje wybranych podstawowych metod. Najczęściej w widoku będziemy samodzielnie implementować metodę `createPartControl(Composite)` gdyż to ona jest wywoływana przy pierwszym otwarciu widoku i to ona definiuje sam jego wygląd. Obiekt `Composite` dostarczany w parametrze metody to miejsce gdzie można wstawiać wszelkie przyciski, listy, panele, itp. zarówno korzystając z API biblioteki SWT jak i JFace.

- `Perspective` – konfiguracja domyślnej perspektywy aplikacji, tj. domyślnego ułożenia wszystkich widoków w oknie. Widoki można rozmieszczać wokół obszaru edytora, który ze względu na pierwotne zastosowanie Eclipse jako IDE, był centralnym punktem aplikacji. Rozmieszczając widoki podaje się dodatkowo jaką część okna mają zajmować. Widoki można także grupować przy wykorzystaniu tzw. folderów. Przykładowe rozmieszczenie widoków w perspektywie przedstawia zrzut ekranu komunikatora Sametime bazującego na Eclipse (Rysunek 5) – domyślne okno składa się z trzech widoków (Contacts, Stock Ticker, BuddyNote), z których pierwszy jest rozwinięty, a dwa kolejne zminimalizowane.



Rysunek 5. Aplikacje RCP nie zawsze przypominają wyglądem Eclipse.

Narodziny produktu

Pierwszy element aplikacji, którego nie widzieliśmy do tej pory to okno powitalne z logo. Często jest to pierwszy krok, od którego zaczyna się „upiększanie" produktu – ku uciesze oka jego przyszłych użytkowników. Produkt musi mieć identyfikowalną z nim ikonę, musi być łatwo spawdzić kto jest jego producentem oraz którą mamy wersję. Po utworzeniu takiej definicji produktu, zawierającej wszystkie tego typu informacje Eclipse w prosty sposób wyeksportować program, gotowy do instalacji na komputerach użytkowników.

Aby stworzyć definicję produktu, należy z menu `File` wybrać `New -> Other` i odszukać `Product Configuration`. Jako lokalizację podać projekt mailClient, nazwę pliku np. product.product i pozostawić zaznaczoną opcję Use an existing product.... Okazuje się bowiem, że w punkcie rozszerzeń org.eclipse.core.runtime.products w plugin.xml projektu był już zdefiniowany produkt, jednak edytowanie go w ten sposób nie jest zbyt wygodne.

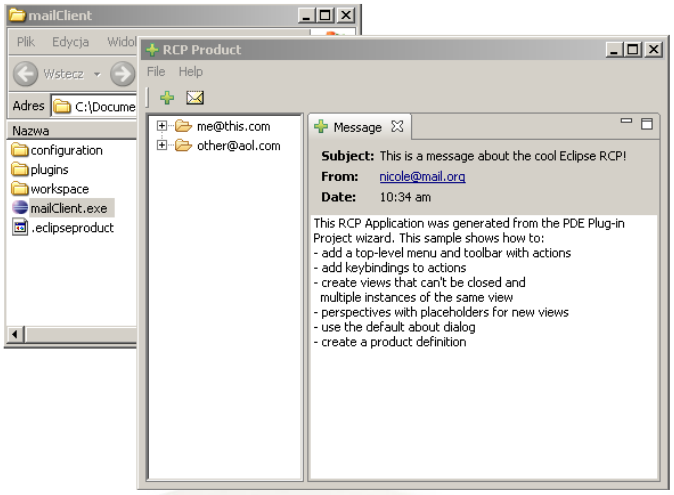
Po utworzeniu konfiguracji produktu naszym oczom ukazuje się edytor z następującymi kartami: Overview, Configuration, Launching, Splash oraz Branding. Po kolei, na karcie Overview, można zdefiniować nazwę naszego programu, uruchamiać go (ramka Testing), oraz eksportować jako zewnętrzną aplikację – do tematu eksportu jeszcze wrócimy.

Na karcie Configuration jest lista wtyczek, które będzie zawierał produkt. Jest to niezwykle ważna lista, ponieważ jeśli produkt nie będzie zawierał wtyczek które są wymagane, może się nie uruchomić. Jednocześnie im więcej wtyczek dodamy na liście, tym produkt będzie większy i cięższy do wysyłania.

Na karcie Launcher zauważymy że każda ramka zawiera takie same opcje dla czterech rodzajów systemów operacyjnych – linux, macos, solaris, win32. Może już zapomnieliśmy, ale aplikacje na bazie Eclipse są multiplatformowe, więc można je uruchamiać na różnych systemach operacyjnych. Jednak dla każdego systemu jest wymagana osobna wersja instalacyjna i będzie utworzony specjalny plik binarny, jak np. dla Windows znany nam eclipse.exe. Na tej karcie jest także miejsce by podać własną nazwę pliku wykonywalnego (pole Launcher Name), np. mailClient. Tutaj można także podać parametry maszyny wirtualnej programu, jak np. potrzebny rozmiar pamięci. Jeśli mamy

obawy, czy przyszli użytkownicy będą mieli zainstalowaną Javę, możemy od razu załączać ją do programu. Pozostałe dwie karty, tj. Splash i Branding pozwalają zdefiniować ekran powitalny oraz dodatkowe informacje, jakie mają być wyświetlane w okienku About aplikacji.

Po ustawieniu wszystkich opcji, zawsze warto. a jeśli dodajemy lub usuwamy wtyczki na karcie Configuration nawet trzeba, wrócić na kartę Overview i przetestować, jak nasza aplikacja działa. W ramce Testing trzeba kliknąć Synchronize i Launch Eclipse application. Jeśli wszystko jest tak, jak oczekiwaliśmy, przechodzimy do eksportu aplikacji, czyli do ramki Exporting i klikamy link Eclipse Product export wizard. Podajemy tylko katalog docelowy (lub archiwum) i klikamy Finish. We wskazanym katalogu zostanie wygenerowana cała aplikacja, o strukturze niemal identycznej jak struktura Eclipse – opisywana na początku, czyli z plikiem wykonywalnym, tym razem mailClient.exe, katalogiem plugins, oraz configuration.



Rysunek 6. Końcowy efekt, wyeksportowany program mailClient

Jak widać, przy pisaniu aplikacji RCP, wyłania się powoli schemat: na ogół dodanie nowej funkcji zaczynamy od dodania właściwego punktu rozszerzeń, oraz klas implementujących stosowne interfejsy. W podobny sposób własną aplikację można rozszerzyć o pomoc, podpowiedzi, własne kreatory, skróty klawiszowe, funkcje wyszukiwania, zarządzanie zasobami użytkownika (Resources).

Co dalej...

Oczywiście, im większe będziemy mieli wymagania wobec przyszłej aplikacji, tym lepiej trzeba poznać API platformy Eclipse. Warto więc także wypróbować kilka innych przykładowych aplikacji, jakie są dostępne w kreatorze wtyczek. Dalsze kroki najlepiej natomiast skierować na strony takie jak np. http://www.eclipse.org/evangelism, gdzie znajdziemy kolejne przykłady a także dokumentację RCP. Poza standardowym środowiskiem, to co naprawdę stanowi o sile RCP to bogactwo innych projektów gotowych do integracji z naszymi wtyczkami. Są to np. środowisko do tworzenia narzędzi graficznych – GMF, czy silne wsparcie dla MDA (Model Driven Architecture), w postaci EMF – omówione w niniejszym numerze, BIRT (Business Inteligence and Reporting Tools) dostarczający funkcjonalność raportów biznesowych, czy projekt ECF (Eclipse Communication Framework) - moduł komunikacji korzystający z popularnych otwartych standardów jak np. XMPP, znany z Jabbera, lub Google Talk, czy IRC.

Podsumowanie

W artykule przyjżeliśmy się budowie aplikacji Eclipse RCP, tworząc dwie proste aplikacje oraz budując program gotowy do dostarczenia klientowi. Na pierwszy rzut oka RCP nie różni się zbyt wiele od tworzenia zwykłych aplikacji w Javie. Pozwala uniknąć programowania wielu stałych elementów aplikacji a skupić się tylko na interfejsie użytkownika głównych elementów, oraz na logice i kluczowych modułach systemu. Jednocześnie znacząco ułatwia personalizację aplikacji, dzięki dużej elastyczności w definiowaniu wyglądu i zachowania aplikacji.

Linki do innych materiałów o Eclipse Rich Client Platform:

http://wiki.eclipse.org/index.php/Rich_Client_Platform
http://www.eclipsercp.org/
http://wiki.eclipse.org/RCP_Custom_Look_and_Feel
http://wiki.eclipse.org/RCP_FAQ
http://www.eclipsezone.com/eps/10minute-rcp/
http://wiki.eclipse.org/index.php/The_Official_Eclipse_FAQs
http://www.eclipsecon.org/2007/index.php?page=sub/&area=rich-client
http://www.eclipse.org/evangelism

NetBeans Platform – czyli jak budować szybko i skutecznie standardowe aplikacje biurkowe

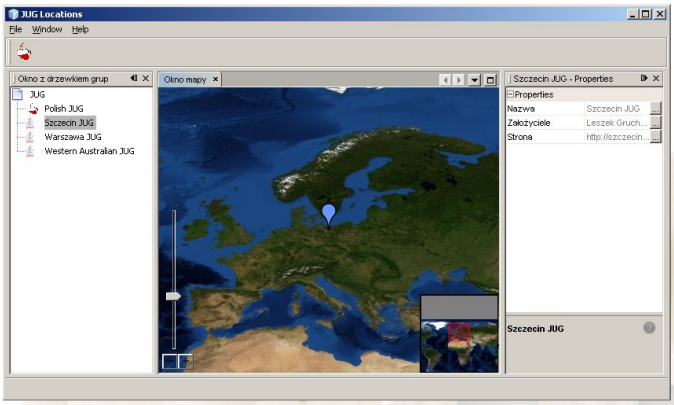
Marek Kliś

Wśród wielu programistów panuje przekonanie, że język Java nie nadaje się do tworzenia aplikacji biurkowych. Argumentują to słabą wydajnością aplikacji napisanych w Javie bądź wyglądem odbiegającym od wyglądu aplikacji uruchamianych w danym systemie.

Ten artykuł ma na celu próbę pokazania, że w Javie można budować z dobrym efektem standardowe aplikacje biurkowe. Przykładem niech będzie choćby środowisko programistyczne NetBeans IDE, które w całości jest napisane właśnie w Javie.

Zastanówmy się więc nad stworzeniem przykładowej aplikacji biurkowej. Chcielibyśmy, aby miała ona w miarę nowoczesny i przyjazny wygląd, możliwość prostej aktualizacji itp. itd. Jednocześnie, jak większość programistów, chcielibyśmy zrobić to jak najmniejszym nakładem pracy. I tutaj z pomocą przychodzi nam firma Sun Microsystem i jej produkt NetBeans Platform.

Za przykład niech posłuży nam aplikacja, która wykorzystuje komponent JXMapKit do pokazania położenia geograficznego kilku wybranych grup JUG (Java User Group).



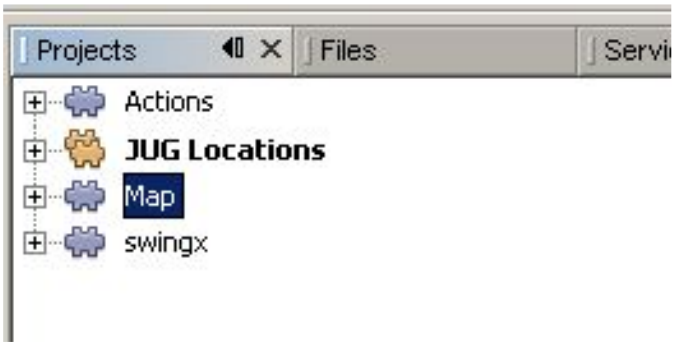
Rysunek 1: Aplikacja JUG Locations

Co to właściwie jest platforma NetBeans?

Najkrócej można zdefiniować, że jest to szkielet aplikacyjny oparty na bibliotece Swing. Dzięki temu możemy wykorzystywać wszelkie dostępne w sieci komponenty Swingowe w naszej aplikacji.

Aplikacja zbudowana na platformie NetBeans jest podzielona na moduły (plugin). Każdy z modułów posiada swoją nazwę, wersję oraz listę zależności od innych modułów. W każdej chwili aplikacja może być rozszerzona poprzez dodanie do niej nowego modułu.

Nasza przykładowa aplikacja składa się z trzech modułów: Map, Actions oraz swingx (Rysunek 2).



Rysunek 2: Moduły składowe aplikacji

DataObject, Node, ExplorerManager

Obiekty danych (DataObjects), węzły (Nodes) oraz ExplorerManager to chyba jedne z najbardziej pomocnych mechanizmów platformy. Obiekty danych to własne obiekty encyjne, specyficzne dla naszej aplikacji. Węzeł jest warstwą prezentacji danych (sam w sobie nie jest obiektem zawierającym dane; przykrywa DataObject). Platforma NetBeans oferuje nam komponenty, takie jak tabele, listy czy drzewka, potrzebne do prezentacji struktur reprezentowanych przez węzły. Do najpopularniejszych należą BeanTreeView, ListView czy ContextTreeView. ExplorerManager zarządza zaznaczaniem i nawigacją po tych komponentach.

Dla węzłów są już gotowe metody obsługujące operacje „przeciągnij i puść”, kopiuj, wklej, menu kontekstowe.

Obiektem danych w naszej aplikacji jest klasa `JUGDataObject`:

```
package org.myorg.node;

import java.awt.Image;
import org.jdesktop.swingx.mapviewer.GeoPosition;

public class JUGDataObject {
  private GeoPosition position;
  private String leaders;
```

```
  private String page;
  private String name;
  private Image image;

  public JUGDataObject(String name,
      GeoPosition position,String leaders,
      String page, Image image){
    this.name = name;
    this.position = position;
    this.leaders = leaders;
    this.page = page;
    this.image = image;
  }

  public String getName() {
    return name;
  }

  public String getLeaders() {
    return leaders;
  }

  public String getPage() {
    return page;
  }

  public GeoPosition getPosition() {
    return position;
  }

  public Image getImage() {
    return image;
  }
}
```

Węzłem jest klasa `JUGNode` będąca rozszerzeniem klasy abstrakcyjnej `org.openide.nodes.AbstractNode`.

```
package org.myorg.node;

import org.openide.nodes.AbstractNode;
import org.openide.nodes.Children;
import org.openide.nodes.PropertySupport;
import org.openide.nodes.Sheet;
import org.openide.util.Exceptions;
import org.openide.util.lookup.Lookups;

public class JUGNode extends AbstractNode {

  public JUGNode() {
    super(Children.create(new JUGChildFactory(),
      true));
    setDisplayName("JUG");
  }

  public JUGNode(JUGDataObject dataObject) {
    super(Children.LEAF, Lookups.singleton(
      dataObject));
    setDisplayName(dataObject.getName());
  }
  @Override
  public Image getIcon(int arg0) {
    JUGDataObject jug = getLookup().lookup(
      JUGDataObject.class);
    Image image = Utilities.loadImage(
      "org/myorg/node/jlogo.png");
    if(jug != null){
      image = jug.getImage();
    }
    return image;
  }

  @Override
  protected Sheet createSheet() {
    Sheet sheet = Sheet.createDefault();
    Sheet.Set set = Sheet.createPropertiesSet();
    JUGDataObject jug = getLookup().lookup(
      JUGDataObject.class);
    if (jug != null) {
      try {
        Property name =
          new PropertySupport.Reflection(jug,
            String.class, "getName", null);
        Property leaders =
          new PropertySupport.Reflection(jug,
            String.class, "getLeaders", null);
        Property site =
          new PropertySupport.Reflection(jug,
            String.class, "getPage", null);
        name.setName("Nazwa");
        leaders.setName("Założyciele");
        site.setName("Strona");
        set.put(name);
        set.put(leaders);
        set.put(site);
      } catch (NoSuchMethodException ex) {
        Exceptions.printStackTrace(ex);
      }
      sheet.put(set);
    }
    return sheet;
  }
}
```

W domyślnym konstruktorze węzła `JUGNode` wywołujemy konstruktor klasy nadpisywanej, gdzie jako argument podajemy, jakie dzieci powinien posiadać tworzony przez nas węzeł (`super(Children.create(new JUGChildFactory(), true))`). Potomkowie są tworzeni przez obiekt klasy `JUGChildFactory` przedstawiony poniżej. Drugi argument (w tym przypadku wartość `true`) informuje czy tworzenie potomków ma się odbywać w osobnym wątku. Jest to przydatne, jeśli budowanie drzewa zajmuje trochę czasu – jeśli np. informacje na temat obiektów są pobierane z bazy danych lub z jakiegoś serwisu internetowego. W takim przypadku klasa `Children` zadba za nas o to, aby na czas budowania drzewa wyświetlić informację np. „Please wait...”.

```
package org.myorg.node;

import java.util.List;
import org.jdesktop.swingx.mapviewer.GeoPosition;
import org.openide.nodes.ChildFactory;
import org.openide.nodes.Node;

public class JUGChildFactory extends
  ChildFactory<JUGDataObject>{

  @Override
  protected boolean createKeys(
      List<JUGDataObject> list) {
    list.add(new JUGDataObject(
      "Polish JUG",
      new GeoPosition(50.061822,19.937181),
      "G. Duda, R. Holewa and A. Nowak",
      "http://java.pl/",
      Utilities.loadImage(
        "org/myorg/node/pjug.gif")));
    list.add(new JUGDataObject(
      "Szczecin JUG",
```

```
      new GeoPosition(53.416667,14.583333),
      "Leszek Gruchala",
      "http://szczecin.jug.pl/",
      Utilities.loadImage(
        "org/myorg/node/jlogo.png")));
    list.add(new JUGDataObject(
      "Warszawa JUG",
      new GeoPosition(52.211780,20.982340),
      "Jacek Laskowski",
      "http://www.warszawa.jug.pl",
      Utilities.loadImage(
        "org/myorg/node/jlogo.png")));
    list.add(new JUGDataObject(
      "Western Australian JUG",
      new GeoPosition(-31.95227,115.85007),
      "Michael Mok",
      "https://wajug.dev.java.net",
      Utilities.loadImage(
        "org/myorg/node/jlogo.png")));
    return true;
  }

  @Override
  protected Node createNodeForKey(
      JUGDataObject dataObject) {
    return new JUGNode(dataObject);
  }
}
```

Obiekt klasy `JUGDataObject` przechowuje informacje na temat konkretnej grupy, czyli nazwę grupy, jej założycieli, adres strony, logo w postaci ikonki oraz położenie geograficzne.

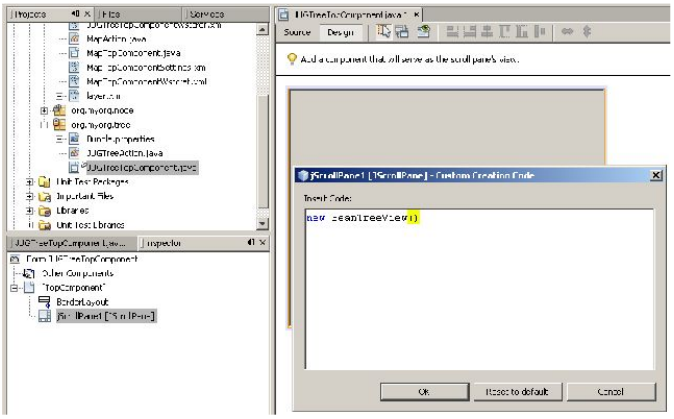
Klasa `JUGChildFactory` nadpisuje dwie metody z `ChildFactory`: `createKeys` i `createNodeForKey`. Pierwsza z nich odpowiada za utworzenie kluczy (DataObjects) a druga, która jest wywoływana osobno dla każdego wcześniej utworzonego klucza, tworzy obiekt węzła reprezentujące obiekty danych. W klasie tej, zamiast tworzyć w kodzie obiekty dla poszczególnych grup, możemy napisać metodę, która będzie odczytywać informację o grupach javowych ze strony https://sv-web-jug.dev.java.net/kml/jug-leaders.kml.

System okien

Platforma NetBeans daje nam do dyspozycji specjalny komponent okna zwany TopComponent. W skrócie można przyjąć, że jest to odpowiednik JPanel. Możemy na nim rozkładać komponenty, ustawiać layout w taki sam sposób jak to czynimy w Swingu właśnie z JPanel. Każdy TopComponent w aplikacji może być dokowany lub oddokowywany, można zmieniać jego rozmiar lub położenie. Położenie jest inaczej zwane jako tryb (mode) i dla przykładu standardowe okno explorera po lewej stronie to tryb Explorer, tryb Properties to pionowe okienko po prawej stronie, a tryb Editor to okienko umiejscowione centralnie.

W naszej przykładowej aplikacji mamy stworzone dwa okna: okno z drzewkiem grup oraz okno mapy. Okno Properties jest standardowo dostępne w platformie. Do jego obsługi wystarczy nadpisać metodę `createSheet()` klasy `AbstractNode` (patrz klasa `JUGNode`).

Na rysunku (Rysunek 3) pokazano w jaki sposób można utworzyć okno z drzewkiem wykorzystując TopComponent oraz BeanTreeView (BeanTreeView dziedziczy po klasie JScrollPane). Korzystając z kreatora dla utworzenia nowego komponentu typu „Window Component" wystarczy tylko, że dodamy w konstruktorze nowo utworzonej klasy informację, jak ma być budowane drzewko (podajemy obiekt, który ma pełnić rolę korzenia drzewa) oraz dodajemy implementację interfejsu ExplorerManager.Provider (wprowadza on tylko jedną metodę getExplorerManager()).



Rysunek 3: Przykład tworzenia okna TopComponent z drzewkiem

```
package org.myorg.tree;
...
final class JUGTreeTopComponent
  extends TopComponent
  implements ExplorerManager.Provider {
...
  private ExplorerManager manager =
    new ExplorerManager();

  private JUGTreeTopComponent() {
    initComponents();
    setName(NbBundle.getMessage(
      JUGTreeTopComponent.class,
      "CTL_JUGTreeTopComponent"));
    setToolTipText(NbBundle.getMessage(
      JUGTreeTopComponent.class,
      "HINT_JUGTreeTopComponent"));
    manager.setRootContext(new JUGNode());
    associateLookup(ExplorerUtils.createLookup(
      manager, getActionMap()));
  }
  ...

  public ExplorerManager getExplorerManager() {
    return manager;
  }
}
```

W powyższym kodzie wytłuszczono te linijki

kodu, które trzeba dodać po zakończeniu kreatora. Nieistotne fragmenty kodu zastąpiono wielokropkiem(...). Widzimy tutaj przykład użycia ExplorerManager'a, o którym była już wcześniej mowa (dla przypomnienia - zarządza zaznaczaniem). Zaznaczony węzeł (a właściwie związany z nim obiekt danych) jest rejestrowany w globalnym rejestrze – `Lookup`. Okno mapy implementuje interfejs LookupListener, który nasłuchuje zmian generowanych między innymi przez ExplorerManager w Lookup.

```
package org.myorg.map;
...
import org.jdesktop.swingx.JXMapKit;

final class MapTopComponent extends TopComponent
  implements LookupListener {

  private JXMapKit mapKit;

  private MapTopComponent() {
    initComponents();
    setName(NbBundle.getMessage(
      MapTopComponent.class,
      "CTL_MapTopComponent"));
    setToolTipText(NbBundle.getMessage(
      MapTopComponent.class,
      "HINT_MapTopComponent"));
    mapKit = new JXMapKit();
    add(mapKit, BorderLayout.CENTER);
  }

  @Override
  public void componentOpened() {
    Lookup.Template tmp = new Lookup.Template(
      JUGDataObject.class);
    result = Utilities.actionsGlobalContext().
      lookup(tmp);
    result.addLookupListener(this);
  }

  @Override
  public void componentClosed() {
    result.removeLookupListener(this);
    result = null;
  }
...
  private Lookup.Result result = null;
  public void resultChanged(
      LookupEvent lookupEvent) {
    Lookup.Result r =
      (Lookup.Result) lookupEvent.getSource();
    Collection c = r.allInstances();
    if (!c.isEmpty()) {
      JUGDataObject jug =
        (JUGDataObject) c.iterator().next();
      mapKit.setAddressLocation(jug.getPosition());
    }
  }
}
```

`MapTopComponent` nasłuchuje wszelkich zmian obiektu typu `JUGDataObject` w globalnym rejestrze Lookup i jeśli nastąpiła zmiana, odczytuje dane na temat nowego obiektu i wyświetla jego dane geograficzne (`mapKit.setAddressLocation(jug.getPosition())`) w kontrolce mapy.

Akcje i menu

Menu i paski narzędzi to główne udogodnienia dla użytkownika aplikacji. Menu jest zorganizowane hierarchicznie i podzielone według intuicyjnych kryteriów (np. Edycja, Plik, Narzędzia, Widok,...). Każdy moduł określa gdzie w menu mają się pojawić akcje przez ten moduł wprowadzane.

Zarządzanie stanem akcji, które zależą od stanu obiektów wprowadzonych przez inne moduły, byłoby niezwykle trudne z wykorzystaniem typowych akcji dostarczanych przez Swinga. Dlatego platforma NetBeans daje nam do dyspozycji własne, bogatsze klasy akcji. Przykładem niech będzie klasa `JUGDescAction` . Jest ona rozszerzeniem klasy `CookieAction`. W skrócie można powiedzieć, że jest to akcja, której stan zależy od aktualnie zaznaczonego węzła (lub obiektu danych DataObject). W naszym przykładzie akcja `JUGDescAction` jest dostępna, jeśli jest zaznaczony węzeł dla `JUGDataObject`.

```
package org.myorg.actions;
public final class JUGDescAction extends CookieAc-
tion {

  protected void performAction(
      Node[] activatedNodes) {
    JUGDataObject dataObject =
      activatedNodes[0].getLookup().lookup(
        JUGDataObject.class);
    if(dataObject != null) {
      String text = "<html><center>" +
        dataObject.getLeaders() + "<br>" +
        dataObject.getPage() + "</center></html>";
      JLabel label = new JLabel(text);
      DialogDescriptor dd =
        new DialogDescriptor(label,
          dataObject.getName());
      DialogDisplayer.getDefault().notify(dd);
    }
  }
...
  protected Class[] cookieClasses() {
    return new Class[]{JUGDataObject.class};
  }

  @Override
  protected String iconResource() {
    return "org/myorg/actions/pjug.gif";
  }
...
}
```

Po implementacji nowej klasy akcji należy ją zarejestrować w pliku layer.xml, który jest głównym plikiem konfiguracyjnym aplikacji zbudowanej na platformie (tzw System File System). W większości przypadków nie trzeba robić tego ręcznie, kreator zrobi to za nas.

Platforma NetBeans jest produktem sprawdzo-

nym przez tysiące programistów wykorzystujących na co dzień w swojej pracy NetBeans IDE. Dzięki dużemu wsparciu ze strony Sun Microsystem, platforma i produkty na niej oparte są stale rozwijane. Nie do przecenienia jest także wsparcie ze strony firm trzecich udostępniających najróżniejsze moduły. Wykorzystując w swojej pracy platformę możemy zaoszczędzić dużo cennego czasu i zamiast zajmować się interfejsem użytkownika możemy skupić się nad jeszcze lepszą implementacją logiki aplikacji. Dodatkowym argumentem jest to, że są dostępne kody źródłowe NetBeans IDE, gdzie zawsze można podglądnąć w jaki sposób programiści NetBeans IDE wykorzystują mechanizmy platformy.

Przydatne linki:

http://platform.netbeans.org/
http://planetnetbeans.org/

Oferta dla sponsorów

Pierwsze wydanie gazety JAVA exPress powstało dzięki bezinteresowanej pomocy autorów artykułów, którzy poświęcili swój czas, aby pomóc zrealizować pomysł stworzenia pierwszej gazety o Javie. Gazety, która będzie dostępna za darmo, dla każdego, kto chciałby poszerzyć swoja wiedzę o języku Java.
Jeśli więc chciałbyś, aby gazeta nadal się ukazywała, a artykuły były coraz lepszej jakości możesz wspomóc naszą inicjatywę.
Niezależnie czy jesteś osobą prywatną, czy przedstawicielem firmy, skontaktuj się z nami pod adresem JavaExpressTeam@gmail.com.
Pomóż rozwijać zdolności swoich obecnych i przyszłych pracowników.

Google Developer Day

24 października (piątek) w Pradze odbędzie się konferencja Google Developer Day (http://code.google.com/events/developerday/).
Jeśli jesteś zainteresowany zbiorowym wyjazdem na konferencję, zgłoś się pod adresem JavaExpressTeam@gmail.com.
Koszt przejazdu z Krakowa wynosiłby ok. 100 zł. Możliwy powrót w piątek w nocy, lub pozostanie na weekend w Pradze i powrót w niedzielę wieczorem. Koszt noclegu nie jest wliczony w cenę. Przy dużym zainteresowaniu możliwa będzie zbiorowa rezerwacja miejsc noclegowych.
W przypadku małego zainteresowania, wyjzad zostanie odwołany.

Zostań autorem

Gazeta JAVA exPress będzie ukazywała się co 3 miesiące. Jest to pierwsze wydanie gazety, dlatego zapraszamy wszystkich do współpracy.
Pierwsze wydanie powstało dzięki bezinteresowanej pomocy autorów artykułów. Oddając w moje ręcę swoje artykuły nie byli świadomi, że mogą cokolwiek za to otrzymać. Tym bardziej wielkie brawa za ich czas i zaangażowanie.
Poszukujemy osoby, które chciałyby zostać autorami artykułów do gazety lub pomóc w jej rozwoju (np. poprzez przygotowanie graficzne i skład).
Jeśli chcesz zostać autorem lub w inny sposób pomóc gazecie, wyślij swoja propozycję na adres JavaExpressTeam@gmail.com.
Wśród osób zaangażowanych w tworzenie pierwszych dwóch numerów gazety (poprzez artykuły lub pomoc przy przygotowanu gazety) zostaną rozlosowane następujące bonusy:
- 50% zniżki na konferencję JAOO (http://jaoo.dk)
- 1 darmowa wejściówka na konferencję JDD (http://08.jdd.org.pl)