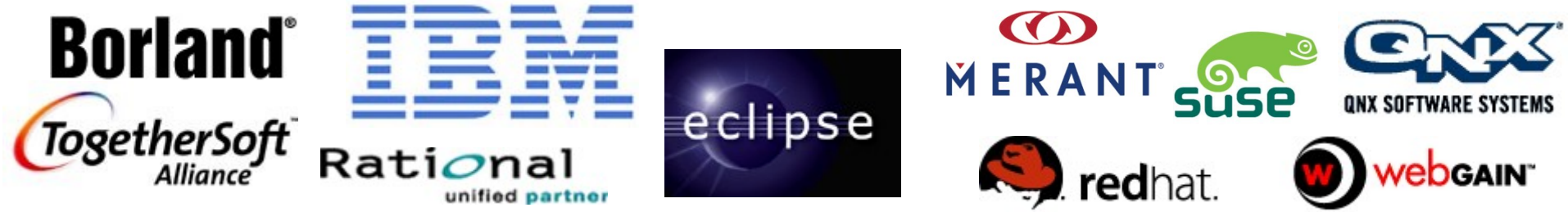


Eclipse jako uniwersalne środowisko programistyczne

Uniwersalność – dzięki zastosowaniu technologii JAVA, Eclipse jest środowiskiem multiplatformowym (dotychczas współpracuje z systemami:Linux, HP-UX, AIX, Solaris, QNX, OSx oraz Windows)

Dostępność – sposób dystrybucji Eclipse sprawia, że każdy w prosty sposób i nie ponosząc dodatkowych kosztów może zdobyć oprogramowanie korzystać z narzędzi udostępnianych w ramach środowiska, a także innych z nim związanych

Otwartość – sposób licencjonowania (specjalnie zaprojektowana licencja - EPL, zgodna ze wskazaniami OSI i zaakceptowana przez tę organizację) sprawia, że chętni mogą brać czynny udział w rozwoju „ekosystemu" Eclipse

Elastyczność – wbrew pozorom, Eclipse nie jest nierozłącznie związany z programowaniem w języku JAVA, istnieje możliwość zastosowania tego środowiska także do innych języków (przy wykorzystaniu odpowiednich wtyczek, np. CDT)

Wygoda – Eclipse jest stworzony z myślą o tym, by ułatwiać programistom pracę. Jego popularność i ciągły rozwój dowodzi, że ten cel został osiągnięty

Trochę historii

Visual Age for Java – VA4J, narzędzie do programowania w *Javie*, stworzone przez OTI (*Object Technologies Consortium* – firma zależna od IBM, wchłonięcie nastąpiło w 1996), napisane w *Smalltalku*.

Listopad 2001 – pierwsza wersja Eclipse.

Dziś – **Eclipse** występuje w wersji 3.0, w pracy nad środowiskiem współpracuje 45 przedsiębiorstw, w tym tacy giganci jak Oracle, IBM, HP czy Intel.

Sam IBM podobno zainwestował w prace nad Eclipse $40 000 000.

O licencji

Na potrzeby Eclipse stworzono specjalny rodzaj licencji, **EPL**. Charakteryzuje się ona tym, że zezwala na bardzo swobodne zarządzanie kodem Eclipse, tak uzyskanym jak i stworzonym, w szczególności zezwala na komercyjne używanie Eclipse i jego elementów, a konkretnie – oprogramowanie korzystające ze środowiska bądź je rozszerzające moze podlegać licencjonowaniu bardziej restrykcyjnemu.

Jeszcze w wersji 2.1.1 licencją, którą był objęty Eclipse była **CPL**, nieco bardziej ogólna od EPL.

Zawieramy znajomość

Instalacja – ten proces jest wyjątkowo prosty. Dysponując systemem z zainstalowanym i właściwie skonfigurowanym środowiskiem JAVA (np. JRE autorstwa SUN), wystarczy pobrany z Internetu plik archiwalny zawierający Eclipse rozpakować do katalogu docelowego. Znajdzie się tam gotowy do uruchomienia **plik wykonywalny**.

Miła niespodzianka dla użytkowników Windows: Eclipse nie korzysta z rejestru systemowego, zatem znikają problemy występujące przy reinstalacji systemu, etc.

Architektura.

Istotne elementy:

przestrzeń projektów (ang. *workspace*),

moduł kontroli wersji (ang. *team component*)

moduł pomocy (ang. *help component*)

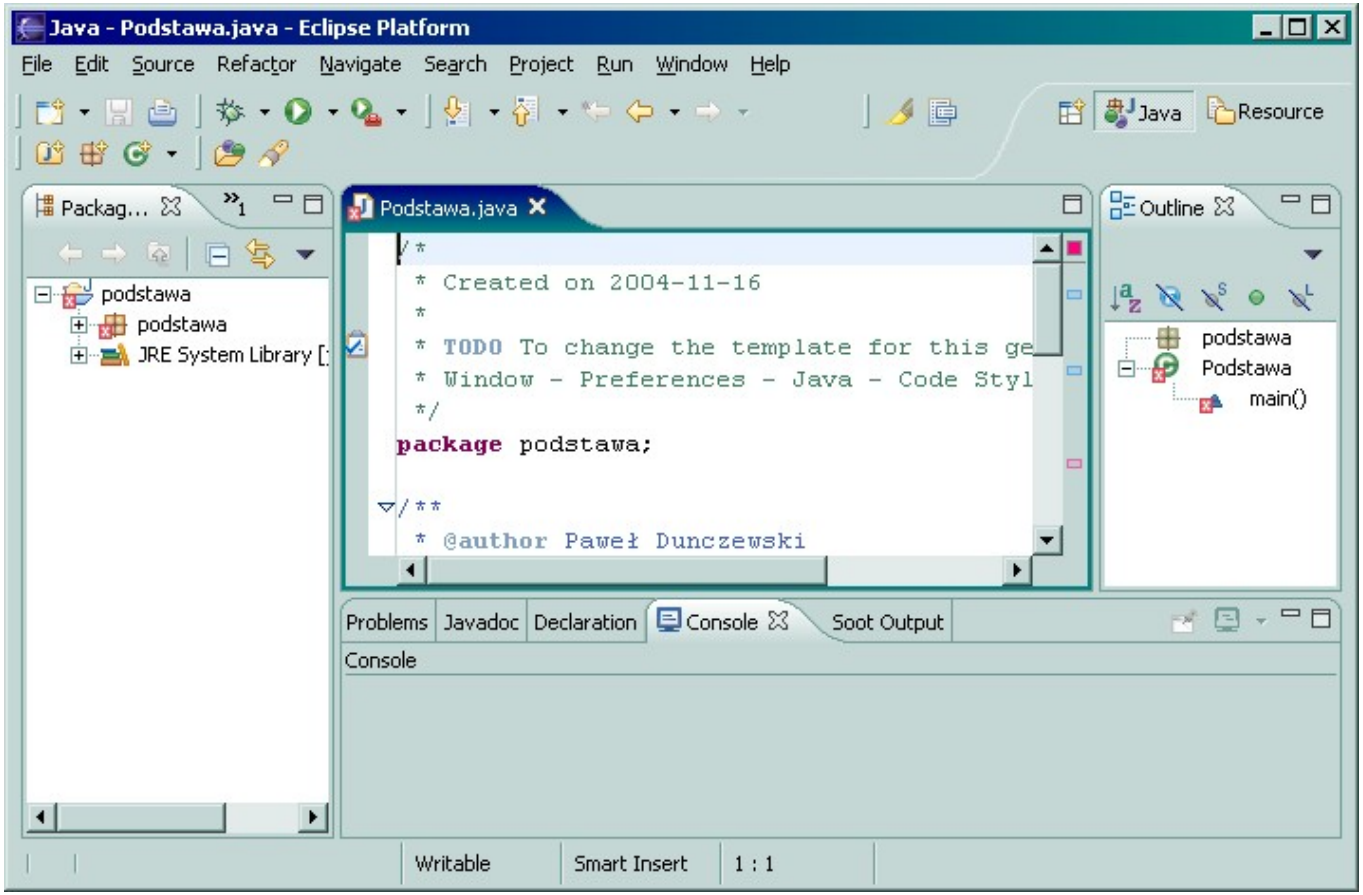
Widoki i perspektywy (na wydajną współpracę)

Widok (ang. *view*) jest podstawową jednostką organizacyjną interfejsu użytkownika. Każde pojedyncze „okienko” wewnątrz Eclipse jest właśnie widokiem.

Perspektywa (ang. *perspective*) to grupa skonfigurowanych uprzednio widoków. Do różnych zadań (np tworzenie kodu, testowanie, wyszukiwanie błędów) wygodnie jest używać różnych perspektyw.

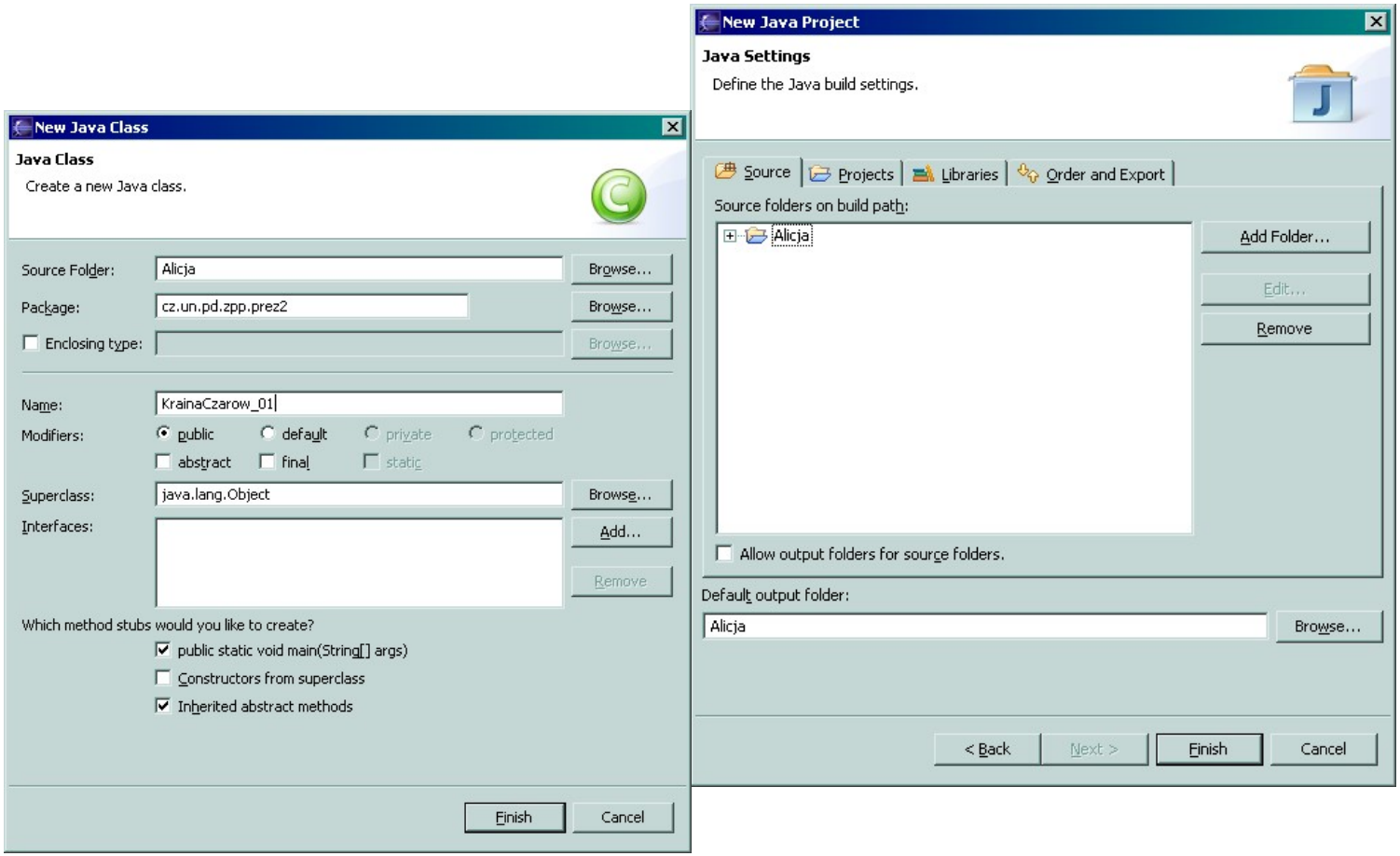
Zmiany aktualnej perspektywy dokonuje się przez menu *Window/Open Perspective*, gdzie wyberamy perspektywę z wyświetlonej listy bądź z okienka *“Select Perspective”* otwieranego opcją *Other*.

Niektóre czzęsto używane perspektywy to: *Java*, *Java Browsing*, *Debug*, *Plugin Development*, itd.



Nowy projekt, nowa klasa

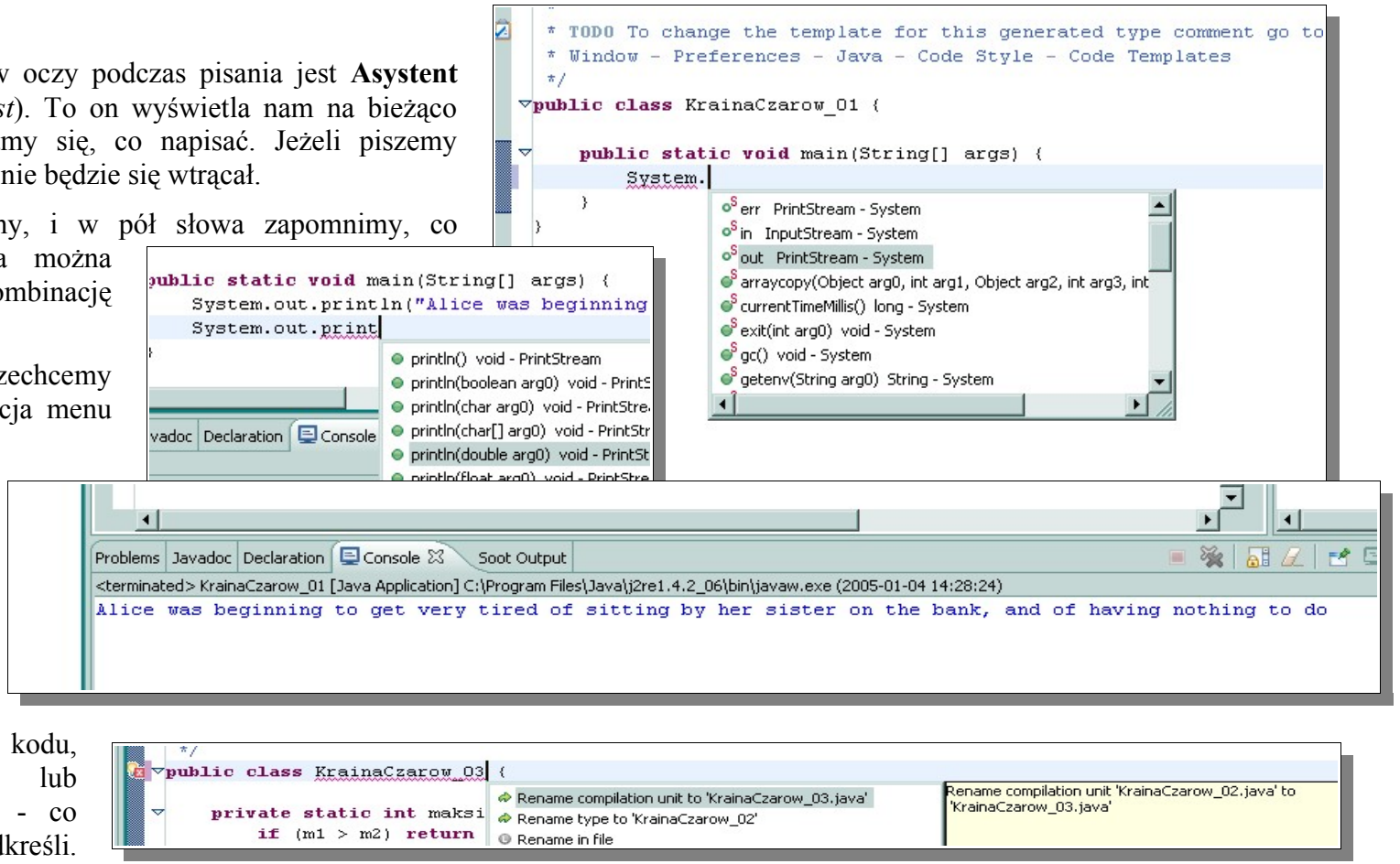
Poza intuicyjnym sposobem na utworzenie nowego projektu (czyli skorzystaniem z menu) możemy skorzystać także ze skrótu klawiaturowego *Alt+Shift+N*. W ten sposób uruchomimy wróżkę (ang. *wizard* ;-) która poprowadzi nas prosto do celu, zadając pytania o typ, nazwę i dodatkowe ustawienia naszego projektu. Na początek wartości domyślne powinny być zadowalające.



Kiedy mamy gotowy szkielet projektu, pora utworzyć klasę – w końcu zamierzamy coś zaprogramować. Widzimy dodatkowe opcje, z których możemy skorzystać, np generowanie szkieletu podstawowych metod.

Warto zwrócić uwagę na stosowany sposób nazewnictwa pakietów gwarantujący unikalność ich nazw (nie jest to cecha szczególna Eclipse, lecz *Javy* jako takiej).

Kraina czarów



Pierwszą rzeczą rzucającą się w oczy podczas pisania jest **Asystent wprowadzania** (ang. *code assist*). To on wyświetla nam na bieżąco podpowiedzi, kiedy zastanawiamy się, co napisać. Jeżeli piszemy szybko i zdecydowanie, asystent nie będzie się wtrącał.

Jeżeli natomiast się rozpędzimy, i w pół słowa zapomnimy, co mieliśmy na myśli, asystenta można zawsze przywołać naciskając kombinację *Ctrl+Spacja*.

Napisany kod pewnie zechcemy uruchomić, służy do tego pozycja menu *Run/Run as/Java Application*.

Eclipse wymusi zapisanie plików na dysku, a następnie skompiluje kod. Jeśli nie było błędów, program ruszy. Wyniki prostego programu ujrzymy w widoku konsoli.

Cały czas jest sprawdzana poprawność wprowadzanego kodu, występujące pomyłki lub niedokładności są wyróżniane - co Eclipse uzna za błąd, to podkreśli. Podpowiedzi można uzyskać – po naciśnięciu *Ctrl+1*[1]. W powyższym przykładzie – nazwy klasy i pliku, w którym jest zapisana są nieidentyczne, tak nie może być..

1 Znak „*1*”, czyli jedynka, a nie małe L. Klikając w wybraną podpowiedź sprawimy, że Eclipse sam ją zrealizuje, wyręczając w tym nas... trochę to straszne, swoją drogą.

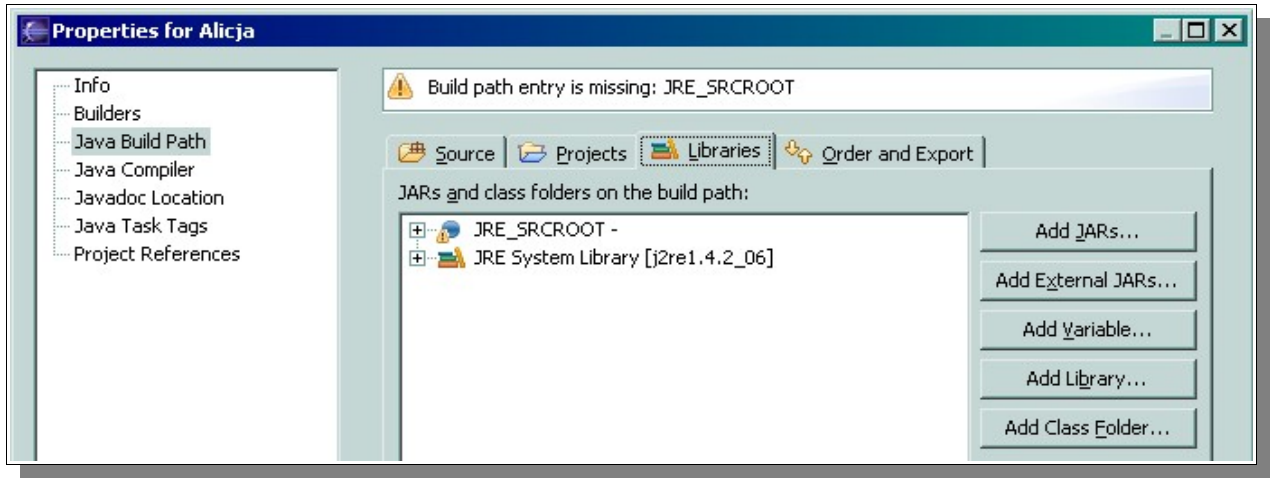
Konfiguracja uruchomieniowa

Jeżeli nasz program potrafi przyjmować parametry z linii poleceń, to możemy je zadać także uruchamiając go w środowisku Eclipse. Lista wprowadzanych parametrów (Arguments) jest dostępna w menu głównym, *Run/Run...* :

W tym samym miejscu, w innych zakładkach możemy sprecyzować np. której z dostępnych w naszym systemie wersji JRE chcemy używać do uruchomienia programu, ustawić szczegóły ładowania bibliotek lub zmienne środowiskowe, itp.

Dołączamy bibliotekę

Często tworząc projekt chcemy użyć zasobów znajdujących się w zewnętrznych bibliotekach. Jak sprawić, by zostały one odnalezione przez środowisko uruchomieniowe? Aby wskazać odpowiednie pliki, należy w widoku *Package Explorer* wskazać nasz projekt i z menu kontekstowego wybrać *Properties*, w wyświetlonym zaś drzewku wskazać pozycję *Java Build Path*. Zakładka *Libraries* jest tym, czego szukaliśmy.



W pozostałych zakładkach możemy wskazać (co bywa ważne!) dokładną kolejność czytania bibliotek, a także zależności między projektami, jakie chcemy ustawić.

Przeglądarka hierarchiczna

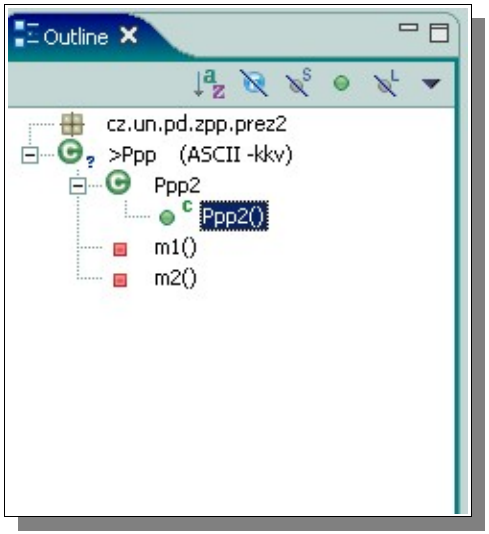
Umiemy już stworzyć nową klasę, spójrzmy teraz na to, co możemy z taką klasą zrobić.

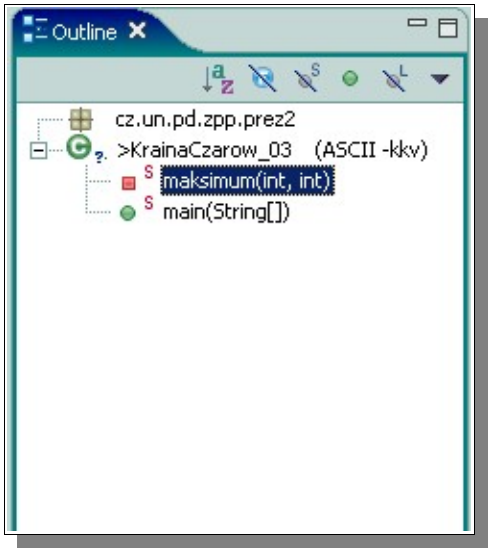
Możemy ją np. obejrzeć – w przeglądarce hierarchii :-)

Tworząc nową klasę przy pomocy kreatora, możemy wskazać nadklasę, po której nowoutworzona ma dziedziczyć. Eclipse potrafi np automagicznie wstawić do takiej nowej klasy konstruktor, który wstawi np. kreator klasy, a w nim odwołanie do kreatora nadklasy: Kod takiego kreatora (automatycznie wygenerowany) wygląda następująco:

```
public class Ppp2 {

    /**
     *
     */
    public Ppp2() {
        super();
        // TODO Auto-generated constructor stub
    }
```



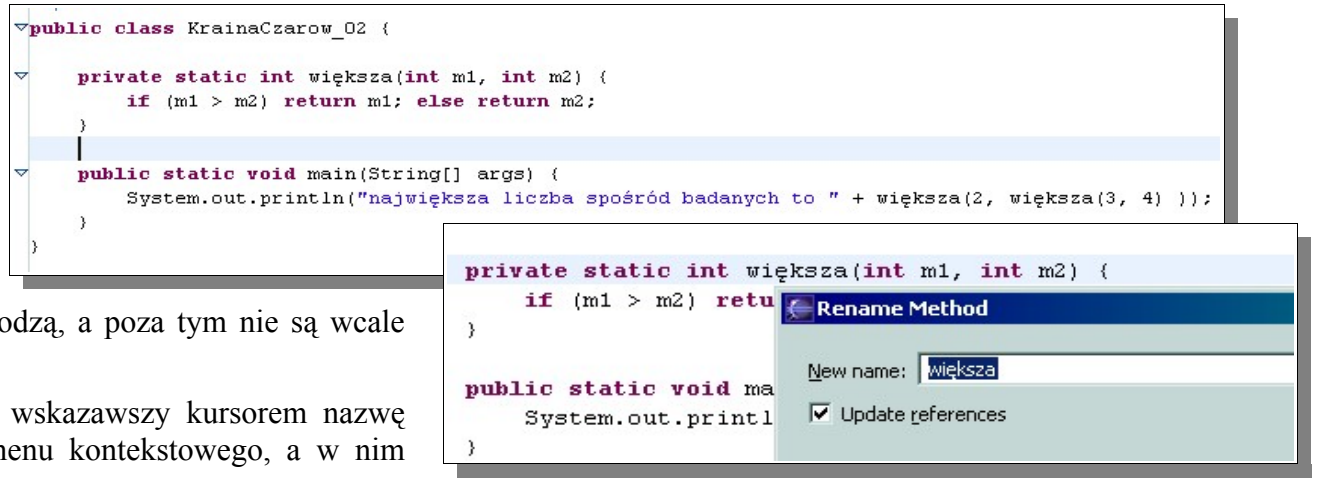


Przeglądarka może (podobnie, jak w większości IDE) bardzo ułatwić dostęp do poszczególnych elementów hierarchii obiektowej, dla każdego z nich oferuje obfite menu kontekstowe, dzięki któremu można bardzo szybko i swobodnie poruszać się po strukturze naszego kodu.

Refaktoring i inne sztuczki

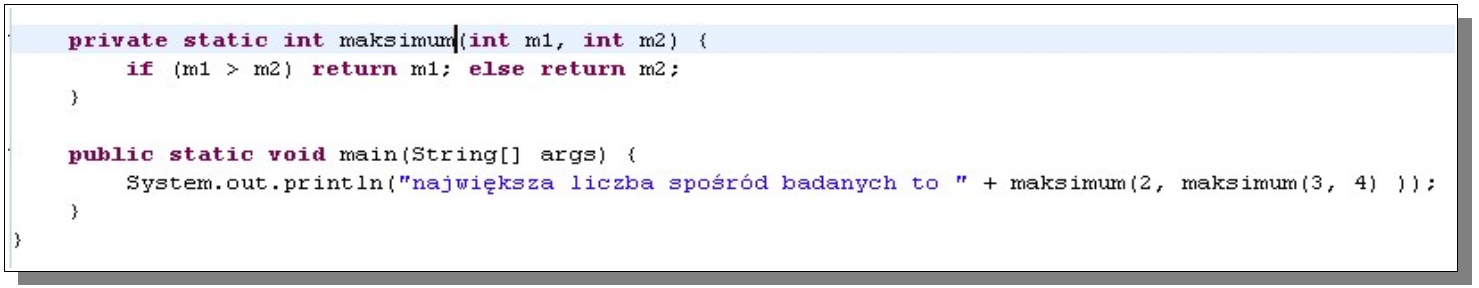
Jedną z wartościowszych cech środowiska typu Eclipse jest możliwość zautomatyzowanej zmiany i przenoszenia niektórych, napisanych już fragmentów programu. Takie działania nazywamy *refaktoringiem*.

Zmiana nazwy – bywa, że z jakichś względów powstaje potrzeba zmiany nazwy metody. Pół biedy, póki metoda jest „świeżo dodana" do kodu, jeżeli jednak jest już „dobrze zadomowiona" i wielokrotnie się do niej odwołujemy... lepiej nie mówić.



Chałupnicze sposoby w stylu masowej wymiany wszystkich odpowiednio brzmiących wyrazów w kodzie czasami zawodzą, a poza tym nie są wcale wygodne.

Na szczęście używamy Eclipse, i możemy, wskazawszy kursorem nazwę metody, nacisnąć *Alt+Shift+R* (albo użyć menu kontekstowego, a w nim *Refactor/Rename*) i po chwili mieć zmienioną nie tylko nazwę metody, ale i wszystkie do tej nazwy odwołania.



W przykładowym kodzie zmieniona została nazwa „*większa*" na „*maksimum*".

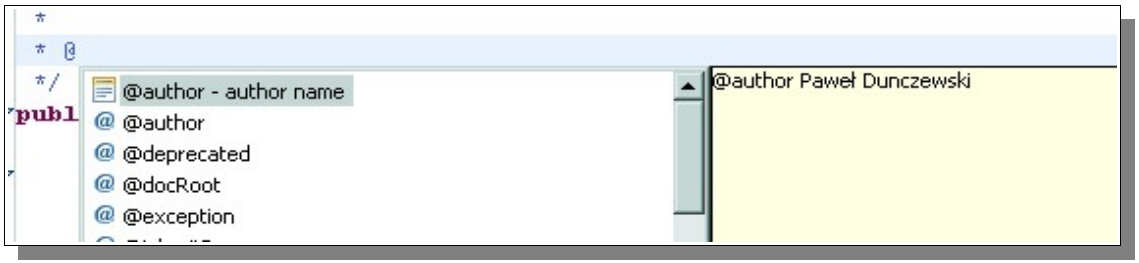
To tylko jeden przykład na przydatność refaktoringu, są liczne inne – np **przenoszenie elementów programu** (np całych metod z jednej klasy do innej, pociągające za sobą automagiczną zmianę także w miejscu odwołań do tej metody – o ile ma to sens) i **ekstrakcja interfejsów**.

Sztuczki

Zasada nr 1: **menu kontekstowe**. Tam znajdziesz to, czego szukasz.

Tworzenie dokumentacji – jak to w *Javie*, przez odpowiednie komentarze. Tradycyjnie, możemy liczyć na *asystenta wprowadzania kodu*.

Jak zawsze, podpowiedzi uzyskamy naciskając *Ctrl+Spacja*. Kiedy np napiszemy w komentarzu znak **@** i poprosimy o podpowiedź, uzyskamy pełną listę możliwych znaczników *Javadoc*.



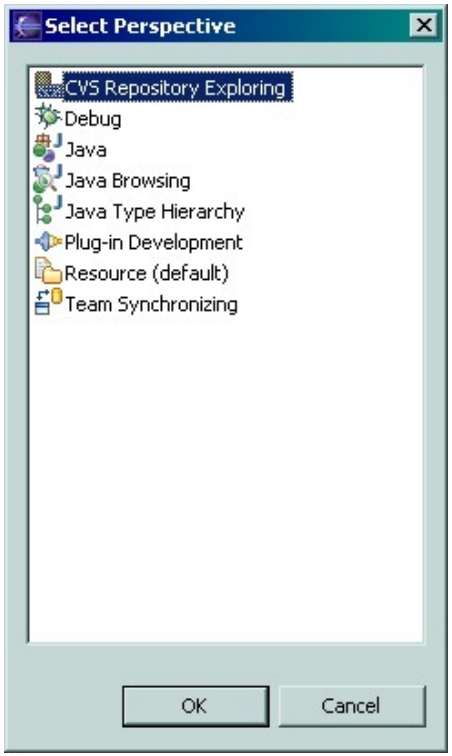
Pisząc **@exception**, dostaniemy zestaw wszystkich wyjątków, które mogą być zgłoszone przez metodę. Podobnie dla **@param**, i innych znaczników wspieranych przez Javadoc, których szkielet może być stworzony na podstawie kodu *Javy*.

Warto przyswoić sobie liczne udostępniane przez Eclipse skróty klawiaturowe (wymienione są zwykle przy konkretnych opcjach menu kontekstowego) – przyspieszy to pracę, bo nie będzie potrzeby przerywać pisania i sięgać po myszkę :-)

Przykładem niech będzie zakomentowanie kawałka kodu: zaznaczamy niechciane (bądź, chciane, bo tak samo jak się komentuje, można także odkomentować) fragmenty kodu – używając kursorów i przycisku *Shift*, bądź myszki, a następnie naciskamy *Ctrl+/* - w podobny sposób możemy nałożyć lub zdjąć całe bloki komentarzy, skrótami – odpowiednio – *Ctrl+Shift+/* oraz *Ctrl+Shift+*

To samo można uzyskać z menu kontekstowego, ale o ile wygodniej jest korzystać ze skrótów.

Wspomaganie pracy zespołowej, - CVS



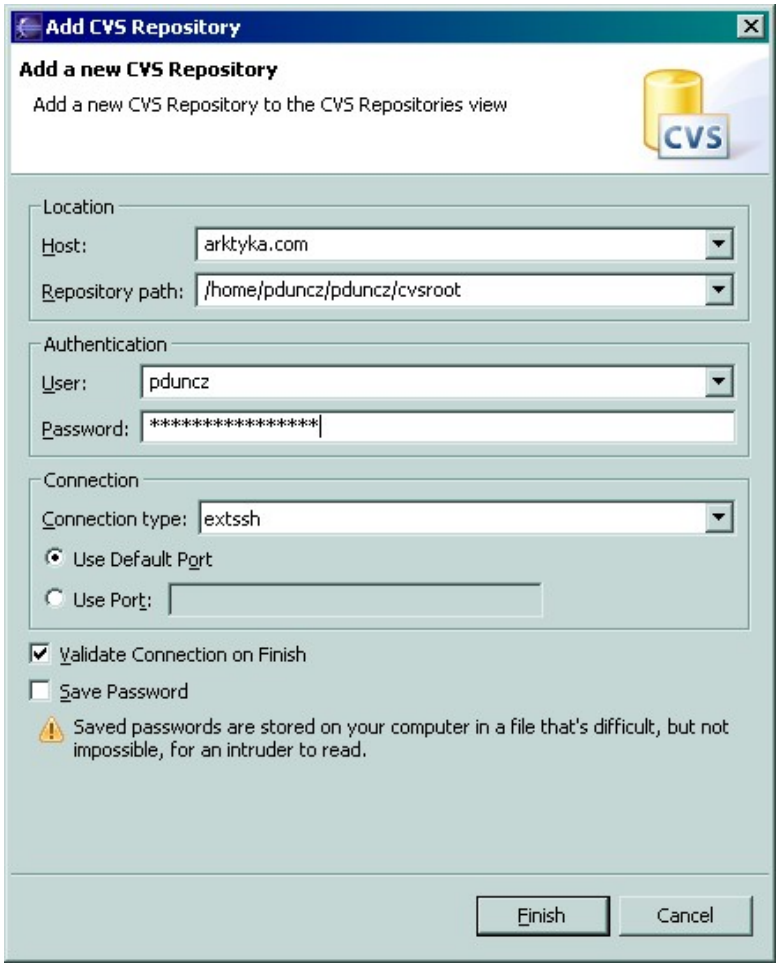
Eclipse potrafi współpracować z systemem CVS w celu umożliwienia współdzielenia kodu pomiędzy wielu programistów jednocześnie, śledzenia wprowadzanych modyfikacji, synchronizacji zmian i zarządzania wersjami projektu.

Serwer jest zjawiskiem niezależnym od Eclipse i nie będziemy się nim tu zajmować. Zakładamy, że dysponujemy już uruchomiony serwerem.

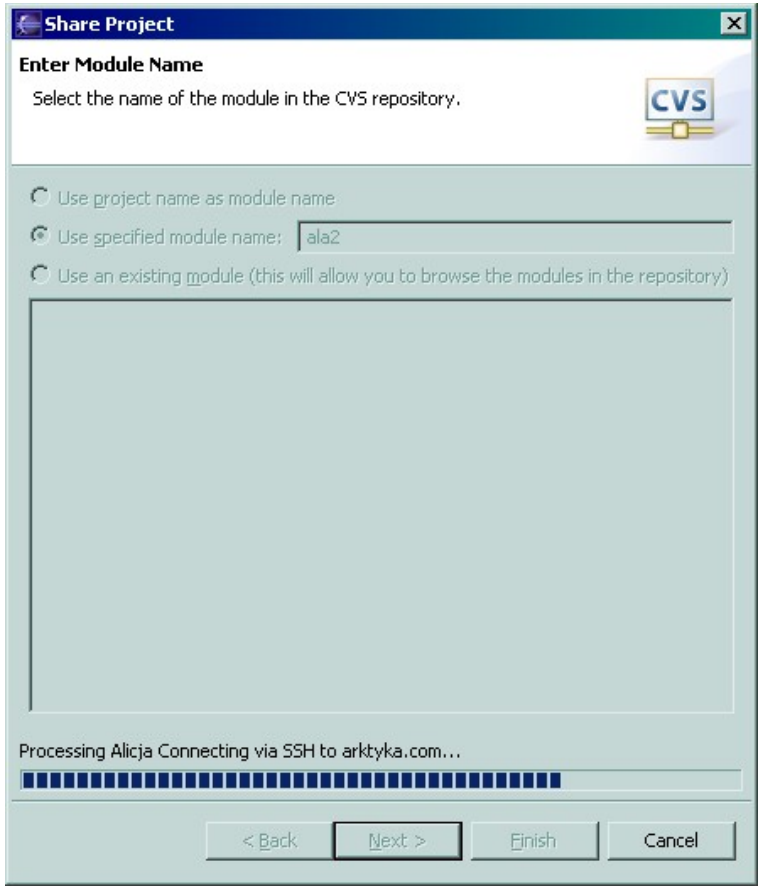
Aby korzystać z dobrodziejstw CVS, do perspektywy w której pracujemy musimy dodać widok o nazwie *CVS Repositories*. Aby to zrobić, używamy menu: *Window/Open Perspective/Other*, następnie w dialogu *Select Perspective* wskazujemy opcję *CVS Repository Exploring*.

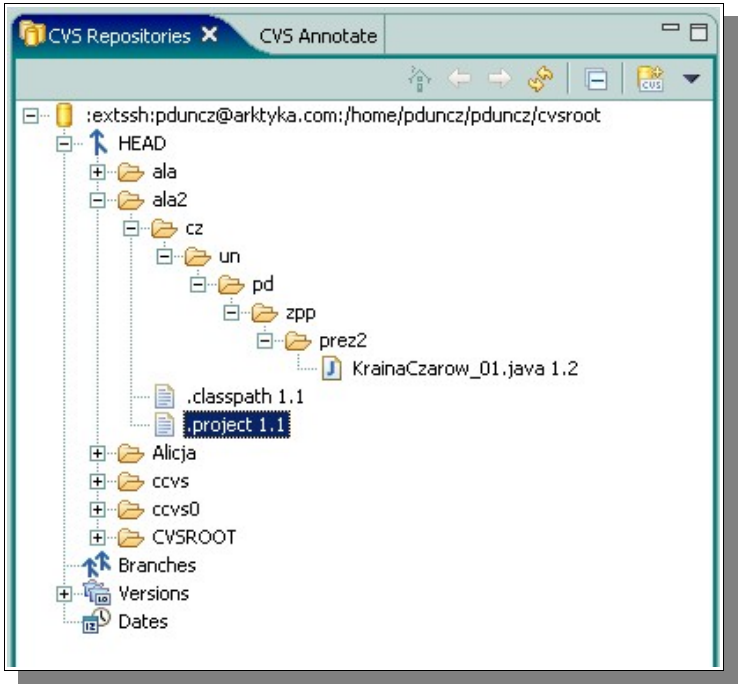
W nowopowstałym widoku *Repositories* otwieramy menu kontekstowe i wybieramy *New/Repository Location*. W oknie wpisujemy dane serwera i repozytorim, oraz wybieramy metodę łączenia się.

Przy pierwszym połączeniu przez *extssh* zostaniemy zapytani o zgodność odcisku palca klucza serwera oraz o utworzenie lokalnych katalogów dla ssh.



Chcemy wyeksportować jeden z naszych projektów do repozytorium CVS. Najwygodniej bedzie nam to zrobić wracając do perspektywy *Java*, by następnie w przeglądarce pakietów otworzyć menu kontekstowe dla pożądanego projektu i wybrać opcję *Team/Share Project*. W serii okien dialogowych wybieramy nasz serwer CVS i nazwę dla modułu CVS. Eclipse (w wersji 3., poprzednie wersje mogą zachowywać się w tym miejscu nieco odmiennie) zaproponuje nam od razu dodanie plików (oraz katalogów, pochodzących np. z hierarchicznej struktury nazewnictwa projektów w Javie) projektu do repozytorium. Zostaniemy poproszeni o wskazanie komentarza, a następnie nasz projekt zostanie wyeksportowany.





Gdy przejdziemy do perspektywy *CVS Repository Exploring* i odświeżymy gałęzie naszego repozytorium, nowowprowadzony projekt już tam będzie, jak widać na rysunku obok.

Aby korzystać z istniejących już na serwerze projektów, połączenie z serwerem musimy mieć skonfigurowane, jak wcześniej. Z drzewa CVS wskazujemy interesujący nas moduł i z jego menu kontekstowego wybieramy *Check Out As*, i przechodzimy przez serię pytań. Po chwili projekt zostaje zaimportowany.

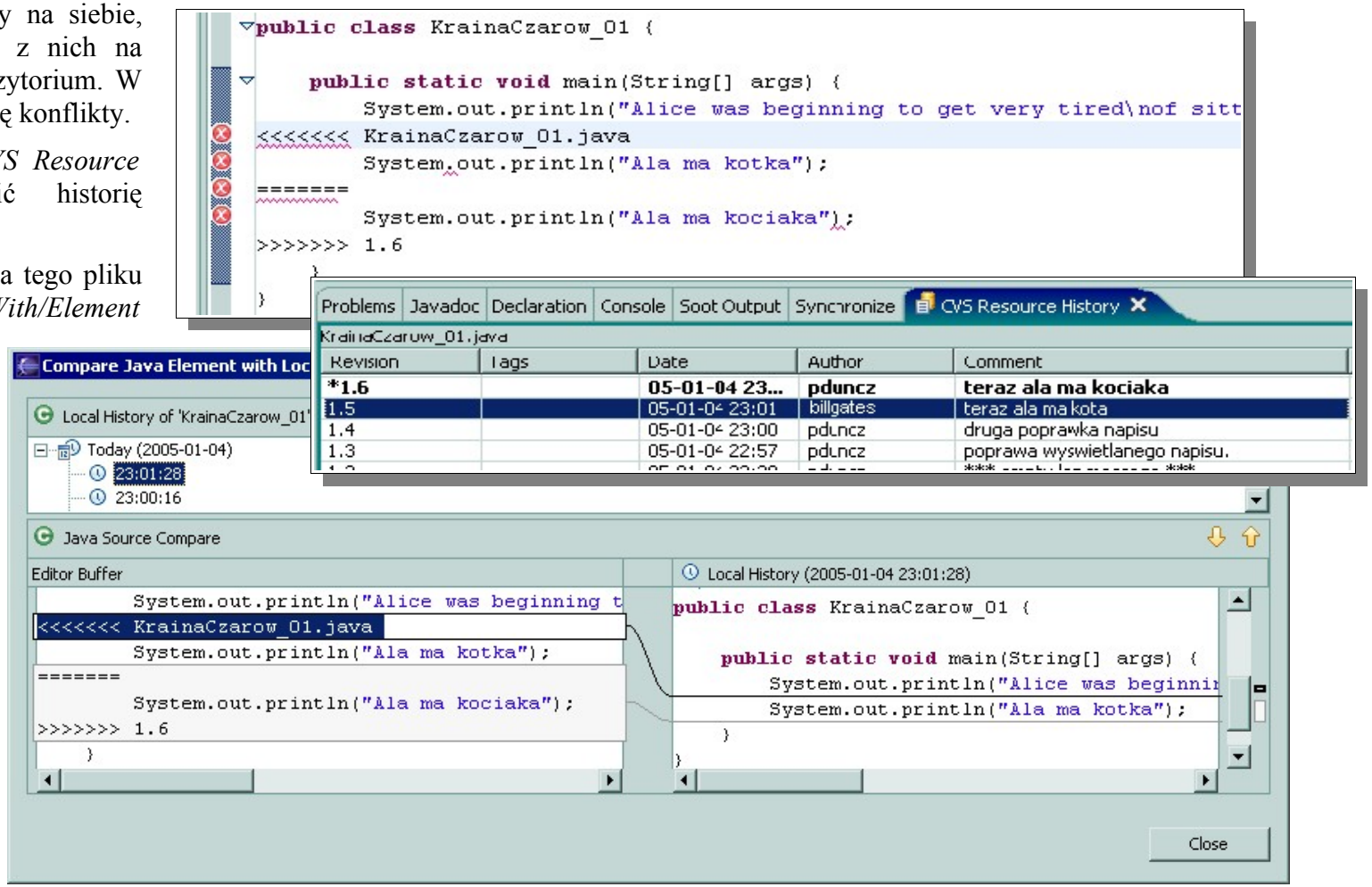
Najważniejsza idea CVS to synchronizacja modyfikacji plików. Załóżmy, że wprowadziliśmy modyfikacje w tym samym miejscu kodu, w którym ktoś z naszych współpracowników w tym samym czasie wprowadził własne poprawki.

Gdyby poprawki nie nachodziły na siebie, CVS po prostu złożyłby obie z nich na najnowszą wersję pliku w repozytorium. W patologicznej sytuacji zdarzają się konflikty.

U dołu ekranu w zakładce *CVS Resource History* możemy prześledzić historię wprowadzanych zmian.

Jeżeli z menu kontekstowego dla tego pliku wybierzemy opcję *Compare With/Element from Local History*, w którym zmiana po zmianie będzie można prześledzić historię modyfikacji interesującego nas rejonu.

Musimy zdecydować się na jedną z wersji lub wybrać jakiś kompromis, i wykonać *Commit*. Ta wersja zostanie umieszczona w repozytorium jako najaktualniejsza (pod warunkiem, że w międzyczasie ktoś nie dokonał jeszcze innych zmian w tych samych miejscach ;-)

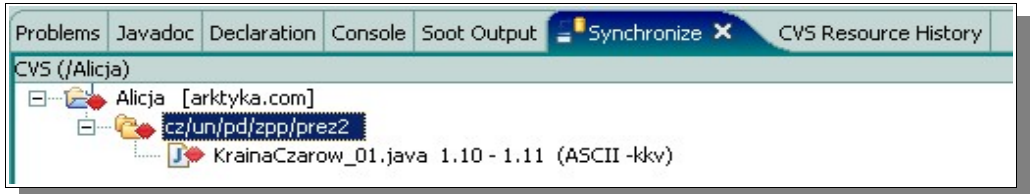


Problem nie jest skomplikowany, jeśli różnice są drobne – możemy wprowadzić poprawkę ręcznie. Jeżeli jednak jest ich sporo, lepiej wykorzystać metodę **synchronizacji**.

W zasymulowanej sytuacji lokalnie zmieniamy zwierzę na *tygrysa*, zaś inny użytkownik w tym samym czasie zmieni je na *psa*. Ponadto, dołoży nowy wiersz do naszego wspólnego kodu.

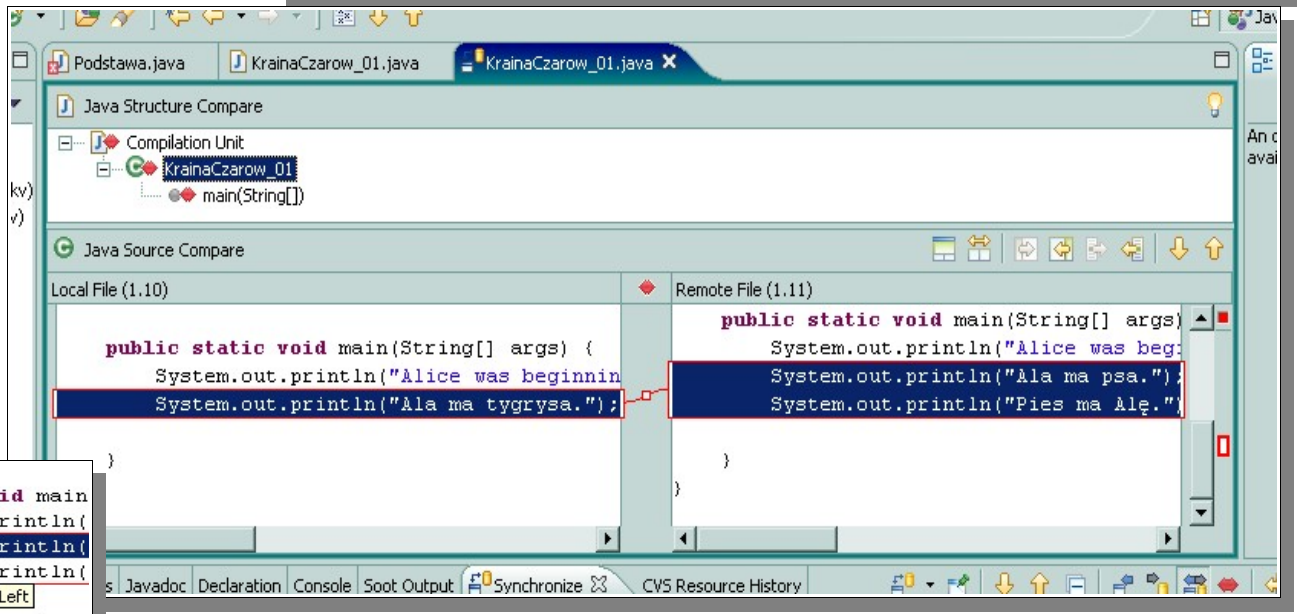
Wykonujemy z menu kontekstowego projektu *Team/Synchronize with Repository*. Na ekranie pojawia się nowy widok – *Synchronize*.

Wybieramy plik, który trzeba zsynchronizować. Pojawia się widok *Java Structure Compare*, w którym wybieramy część do synchronizacji (po kolei, jeśli jest ich więcej).

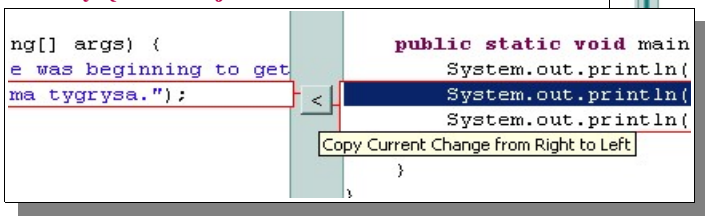


Widzimy obie wersje: lokalną i zdalną.

Kiedy najedzie się myszą na kwadrat łączący zmiany, zmienia się on w przycisk kopiujący kod z jednej strony na drugą. Poprawkę można wtedy doszlifować ręcznie, a następnie zapisać do pliku (*Ctrl*+S lub menu kontekstowe na okienku "*Local File*" i wtedy *Save*.



Napotkałem pewne problemy przy współpracy CVS w wersji 1.11.2 z Eclipse 3.0.1 (przy synchronizacji); sądzę, że istnieje prawdopodobieństwo, że to wcale nie jest wyłącznie moja wina...



Co powinno „obić się o uszy" każdemu użytkownikowi Eclipse

JUnit – środowisko pomocne przy debugowaniu kodu, wspomaga tworzenie specjalnych klas testujących, dzięki czemu można przeprowadzać testy poprawności działania naszej aplikacji.

Ant – wspierany przez Eclipse (pod postacią specjalnego pluginu) pakiet służący do zarządzania tym, co chcemy stworzyć – np do składania archiwów JAR zawierających owoce naszej pracy. Z grubsza można powiedzieć, że jest to odpowiednik nieśmiertelnego pakietu *make*, który jest znany chyba wszystkim

Tomcat – serwer internetowy do uruchamiania aplikacji JSP. Wtyczka do Eclipse umożliwiająca współpracę z Tomcat nazywa się *Sysdeo Tomcat*.

SWT – *Standard Widget Toolkit*, przenośna i uniwersalna biblioteka do programowania graficznego interfejsu użytkownika w Javie, łącząca zalety poprzednich rodzajów takich bibliotek (AWT i Swing). Eclipse jest przykładem użycia tejże biblioteki.

Prezentacja przygotowana na podstawie książki aut. **Steve Holzner** pt. „*Eclipse*", wyd. O'Reily/Helion 2004, ISBN 83-7361-648-9

Do pisania i składu użyto pakietu OpenOffice w wersji 1.9.65

autor: *Paweł Dunczewski,* http://pd.un.cz/