helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Eclipse. Podręcznik programisty

Autor: Praca zbiorowa
Tłumaczenie: Adam Bochenek (wstęp, rozdz. 1 – 18),
Piotr Rajca (rozdz. 19 – 27), Jaromir Senczyk (rozdz. 28 – 34)
ISBN: 83-7361-691-8
Tytuł oryginału: Java Developers Guide to Eclipse
Format: B5, stron: 784

Poznaj możliwości środowiska Eclipse

- Korzystanie ze środowiska Eclipse
- Tworzenie programów w Javie w środowisku Eclipse
- Rozbudowa możliwości Eclipse za pomocą dodatkowych modułów

Eclipse to nie tylko doskonałe zintegrowane środowisko do tworzenia aplikacji w języku Java. To także platforma do tworzenia i integracji narzędzi programistycznych, umożliwiająca zbudowanie kompletnego warsztatu pracy zarówno dla pojedynczego programisty, jak i dla całego zespołu pracującego nad wspólnym projektem. Idea środowiska Eclipse została zainicjowana przez giganty rynku IT — firmy, które zdecydowały się na stworzenie uniwersalnego narzędzia, umożliwiającego dostosowanie sposobu pracy z nim do wymagań różnych typów projektów. Eclipse to narzędzie zbudowane w oparciu o potrzeby programistów.

Książka „Eclipse. Przewodnik programisty" opisuje nie tylko środowisko Eclipse i sposoby zastosowania go w codziennej pracy twórcy oprogramowania. Przedstawia również metody rozbudowania go o dodatkowe moduły zwiększające jego możliwości i przydatność do konkretnego zadania. Autorzy książki, wykorzystując swoje doświadczenie zarówno w stosowaniu, jak i w nauczaniu Eclipse, pokazują, jak można dostosowywać platformę w celu zwiększenia efektywności i wydajności pracy oraz jak unikać najczęściej pojawiających się błędów i problemów.

- Podstawowe wiadomości o Eclipse — interfejs, konfiguracja i połączenie z innymi aplikacjami
- Pakiet Java Development Tools
- Testowanie i usuwanie błędów aplikacji stworzonych za pomocą Eclipse
- Praca zespołowa — współpraca Eclipse z systemem CVS
- Zarządzanie konfiguracją Eclipse
- Architektura środowiska Eclipse
- Tworzenie modułów rozszerzających
- Okna dialogowe, kreatory i widoki
- Architektura projektów
- Tworzenie pakietów i systemów pomocy dla użytkowników
- Wykorzystanie technologii COM i OLE

Dodatkowe ćwiczenia przedstawiają sposoby wykonania najbardziej typowych zadań związanych z obsługą środowiska Eclipse.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 11.

Okna dialogowe i kreatory

W niniejszym rozdziale zajmiemy się wieloma typami okienek dialogowych i kreatorów dostępnych w Eclipse. Dowiemy się, jak z nich korzystać i jak je rozszerzać, by stały się częścią tworzonych przez nas narzędzi. Skoncentrujemy się na oknach dialogowych, za pomocą których da się rozbudować środowisko Eclipse. Będą wśród nich okienka służące do określania właściwości obiektów, strony definiujące parametry środowiska, a także kreatory wywoływane w celu stworzenia nowych zasobów lub obsługi procesu ich eksportu bądź importu. Każdy kreator jest bowiem rodzajem okna dialogowego.

Przy konstruowaniu nowych okienek dialogowych stosujemy istniejące szablony, co zwiększa naszą wydajność i jednocześnie podnosi stopień integracji wszystkich elementów środowiska. Użytkownik przyzwyczaja się do pewnego standardowego sposobu wykonywania typowych operacji.

Tak więc korzystanie z dostarczonych w ramach Eclipse klas przyspiesza proces tworzenia nowych narzędzi, jednak celem nadrzędnym jest zachowanie spójności i przestrzeganie zasad, które pozwalają na tak wysoki stopień współpracy różnych składników Eclipse.

Okna dialogowe — powszechny element aplikacji

Okna dialogowe to bardzo typowy składnik większości aplikacji. Ich definicja znajduje się w różnych miejscach kodu źródłowego Eclipse. Oczywiście, większość z nich pełni w środowisku dokładnie określoną funkcję i nie jest dostępna dla twórców narzędzi.

W lepszym zrozumieniu zasad posługiwania się oknami dialogowymi dostarczonymi wraz z Eclipse pomoże nam analiza roli, jaką pełnią one w ramach interfejsu użytkownika, i technik ich implementacji oraz dołączania do własnych modułów.

Istnieją dwa podstawowe sposoby dodania okienka dialogowego do Eclipse w ramach pluginu, który rozszerza możliwości środowiska:

- Definicja okna dialogowego dołączonego jako rozszerzenie i korzystającego z istniejącego, standardowego szablonu dostosowanego do naszych potrzeb. Dodajemy go do któregoś z dostępnych punktów rozszerzenia.
- Definicja własnego okienka pełniącego pewną specyficzną dla naszego narzędzia funkcję. Wywoływane jest ono z wnętrza modułu.

Okna dialogowe będące rozszerzeniem istniejącego schematu konstruuje się bardzo łatwo, nie wymagają one pisania dużej ilości kodu. Na przykład stworzenie nowego kreatora typu *New File* (czyli odpowiedzialnego za dodanie nowego zasobu), polega na wykonaniu pięciu prostych kroków, którymi są: dodanie rozszerzenia, wygenerowanie klasy, zdefiniowanie kilku pól, metody, a w niej sześciu wierszy kodu. Dodanie nowej strony parametrów środowiska za pomocą odpowiedniego kreatora wymaga jeszcze mniejszej liczby etapów, w których definiujemy jedną metodę i cztery wiersze kodu. W dalszej części rozdziału opiszemy dokładnie, jak to zrobić, zaprezentujemy także ilustrujące to przykłady.

Punkt wyjścia, czyli szablon

Okna dialogowe, a szczególnie zbudowane na ich podstawie kreatory realizujące pewne zadania stanowią jeden z najważniejszych składników budowanych przez nas nowych modułów.

Istnieją różne rodzaje okienek dialogowych: służą one do wyboru pewnej wartości, szukania i zastępowania fragmentu tekstu, wyświetlania komunikatów o błędach czy prezentacji właściwości. Wiele z nich wykorzystujemy bezpośrednio w naszym kodzie, inne budujemy za pomocą mechanizmu dziedziczenia. Dodatkowo trzy typy okien dialogowych mogą być do aplikacji dodane na zasadzie rozszerzenia. W ten sposób środowisko możemy rozbudować o nową stronę okienka parametrów, nowy kreator zasobów lub okno służące do importu lub eksportu. W tabeli 11.1 przedstawiamy pakiety, w których znajdziemy okna dialogowe i kreatory mogące stanowić podstawę naszych nowych rozwiązań.

Istnieje wiele zależności między klasami znajdującymi się w pakietach `org.eclipse.ui.*` oraz `org.eclipse.jface.*`-- klasy `org.eclipse.ui.*` dziedziczą po klasach zdefiniowanych w `org.eclipse.jface.*`. Biblioteka JFace ma charakter zbioru klas o charakterze bardziej uniwersalnym. Na ich podstawie implementowane są klasy typowe dla interfejsu użytkownika Eclipse, odpowiadające konkretnym funkcjom i zadaniom.

Tabela 11.1. *Pakiety wykorzystywane przy budowie własnych okien dialogowych i kreatorów*

Nazwa pakietu	Opis
`org.eclipse.jface.dialogs`	Pakiet zawiera definicję klasy `Dialog`, która jest podstawą wszystkich okienek dialogowych. Znajdziemy tu również obsługę okienek służących do wyświetlania komunikatów czy też informowania użytkownika o stopniu zaawansowania wykonywanych operacji.
`org.eclipse.jface.wizard`	Implementacja klas odpowiedzialnych za tworzenie kreatorów. Tutaj znajdziemy definicję samego kreatora, a także klasę reprezentującą jego stronę.
`org.eclipse.ui.dialogs`	Implementacja typowych dla Eclipse specjalizowanych okien dialogowych, takich jak strona parametrów środowiska, strona właściwości, okienko służące do wyboru i zapisywania plików, wyboru elementów z listy itp.
`org.eclipse.ui.wizards.datatransfer`	Klasy reprezentujące kreatory importu i eksportu.
`org.eclipse.wizards.newresource`	Kreator odpowiadający za tworzenie nowych zasobów.
`org.eclipse.swt.widgets`	Klasy SWT: `Dialog`, `FileDialog` oraz `DirectoryDialog`. Reprezentują one elementy zdefiniowane po stronie systemu operacyjnego; mają charakter elementów ogólnego przeznaczenia.

Punkty rozszerzenia dla okien dialogowych i kreatorów

Tabela 11.2 zawiera listę punktów rozszerzenia, które pozwalają na dołączenie do obszaru roboczego Eclipse własnych okien dialogowych i kreatorów.

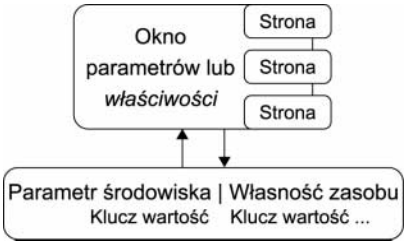
Tabela 11.2. *Punkty rozszerzenia obsługujące dodawanie okien dialogowych i kreatorów*

Typ dodawanego elementu	Punkt rozszerzenia
Strona parametrów środowiska	`org.eclipse.ui.preferencePages`
Strona właściwości obiektu lub zasobu	`org.eclipse.ui.propertyPages`
Kreator nowego zasobu	`org.eclipse.ui.newWizards`
Kreator importu	`org.eclipse.ui.importWizards`
Kreator eksportu	`org.eclipse.ui.exportWizards`

Użycie tych punktów rozszerzenia powoduje, że nasz dodatkowy element (okno dialogowe, kreator czy strona właściwości) zostaje zintegrowany z platformą Eclipse.

Rozszerzanie okienka parametrów środowiska i właściwości zasobu lub obiektu

W Eclipse występują dwa szczególne typy zintegrowanych okienek dialogowych. Jest to okno służące do definiowania parametrów środowiska (*Preferences*) oraz okienko dialogowe wyświetlające właściwości obiektów lub zasobów (*Properties*). Rysunek 11.1 przedstawia podstawową strukturę i możliwości tych elementów. Dołączenie do każdego z okienek tego typu nowej strony z właściwościami polega na dodaniu rozszerzenia w odpowiednim punkcie.



Rysunek 11.1. *Struktura okna dialogowego składającego się ze stron właściwości*

Struktura okien dialogowych w obu przypadkach jest taka sama, co widać na rysunku 11.1. Zauważ, że okno właściwości może odwoływać się do cech zasobu, choć nie jest to wcale obowiązkiem. Możemy w nim operować na dowolnym zestawie danych, które w jakikolwiek dotyczą zasobu lub innego wskazanego obiektu.

Wszystkie dołączone do platformy moduły mogą wstawiać własne strony do wspólnego okienka parametrów środowiska, które stanowi jeden z ważniejszych elementów obszaru roboczego. Jest to najlepszy przykład możliwości integracyjnych Eclipse, ponieważ strony definiowane przez różne narzędzia stają się częścią pojedynczego okienka. Okienko to posiada odpowiednie mechanizmy obsługujące nawigację wśród tych stron, pozwala także na ich ładowanie i aktywację dopiero w momencie, gdy są niezbędne. Zawartość poszczególnych stron różni się między sobą, inny jest też zestaw wartości, które są tam wyświetlane i następnie zapamiętywane. Jednak dzięki temu, że przestrzegamy pewnych ogólnych reguł, z punktu widzenia użytkownika mamy cały czas do czynienia z jednolitym środowiskiem. Nie ma też wątpliwości, w którym miejscu należy szukać ustawień dotyczących parametrów systemu. Oczywiście, istnieją również pewne parametry związane z działaniem poszczególnych widoków, takie jak bieżący filtr czy kolejność sortowania. Tych nie znajdziemy w okienku parametrów środowiska, ale wśród opcji poszczególnych widoków.

Na bardzo podobnej zasadzie funkcjonują okienka dialogowe służące do definiowania właściwości zasobu lub obiektu. Tak naprawdę istnieje jedno takie okno w środowisku Eclipse, choć występuje ono w wielu wariantach. Wywoływane jest w różnych miejscach, związane może być z dowolnym widokiem. Typowym rozwiązaniem jest jednak dodanie nowej strony do okna wywoływanego z widoku *Navigator* lub *Package Explorer*. W ramach każdej strony definiujemy nowe własności oraz sposób ich zapisywania za pomocą API obsługującego przestrzeń projektów (piszemy o tym w punkcie „Własności zasobów", w rozdziale 15.).

Obsługę okna dialogowego właściwości możemy też dołączyć do własnego widoku. Wolno nam też zdefiniować odpowiednie strony związane z danym rodzajem obiektu lub zasobu, które będą dostępne również dla twórców innych modułów.

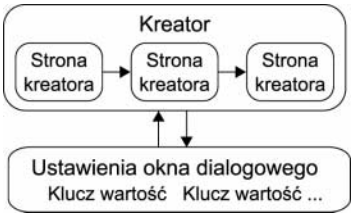
Strony okna dialogowego parametrów i właściwości dodawane są do środowiska Eclipse wyłącznie za pomocą punktów rozszerzenia. Stanowić one mogą wyłącznie fragment określonych z góry okienek dialogowych. Nie powinniśmy próbować wykorzystywać ich w innym celu.

Proponujemy rozważyć taki scenariusz, w którym cechy określające parametry środowiska lub własności obiektu posiadają pewną wartość domyślną, z dopuszczeniem jej modyfikacji.

Kreatory dołączane w punktach rozszerzenia

Kreator używany jest przez użytkownika do przeprowadzania pewnych nieco bardziej złożonych operacji. Zadania takie składają się najczęściej z kilku kroków, każdy z nich wykonywany jest w ramach kolejnej, osobnej strony. Rysunek 11.2 przedstawia podstawy konstrukcji kreatora.

Rysunek 11.2.
Struktura kreatora



Kreator może być traktowany przez nas jako kontroler, który zarządza składającymi się na niego stronami. One odpowiadają za kształt interfejsu użytkownika, sam kreator natomiast za nawigację i nadzór całego procesu. Dlatego każda ze stron jest odpowiednio identyfikowana. Oprócz tego, że definiujemy każdą stronę, wpływ mamy także na tytuł i inne elementy dekoracyjne kreatora.

Kreatory są narzędziami o wiele bardziej elastycznymi niż strony właściwości. Możemy dodawać je do środowiska za pomocą punktów rozszerzenia, ale też wywoływać samodzielnie z wnętrza własnych modułów. Jeśli korzystamy z pierwszego wariantu, mamy możliwość obsługi trzech typowych sytuacji i miejsc:

- Menu *File/New*, gdzie tworzymy nowy zasób.
- Menu *File/Import...*, za pomocą którego importujemy zasoby do środowiska.
- Menu *File/Export...*, skąd eksportujemy zasoby na zewnątrz Eclipse.

Kreator wywołany może też być z dowolnego miejsca naszego pluginu, jeśli tylko zaistnieje taka potrzeba. Nietrudno sobie wyobrazić sytuację, gdy edytor lub widok wymaga wprowadzenia pewnego zestawu danych, którego ręczna edycja jest zbyt skomplikowana. Wtedy z pomocą przychodzi nam odpowiednio skonstruowany kreator. Przykładem może być widok *Properties* wchodzący w skład perspektywy *PDE*. Możemy w nim ręcznie wpisać nazwę klasy rozszerzenia, ale istnieje także wariant, w którym z widoku wywołujemy odpowiedni kreator. Pomoże nam on w wykonaniu tej czynności, pozwalając skorzystać z klasy już istniejącej bądź ułatwiając wygenerowanie nowej.

W kreatorze, podobnie jak w każdym okienku dialogowym, możemy oczywiście korzystać z odpowiednich ustawień i zapisywać stan interakcji z użytkownikiem. Zapisywane mogą być wartości, które użytkownik wprowadził wcześniej, lub lista ostatnio wybranych elementów. Dzięki temu da się w pewnych sytuacjach przewidzieć czynności, które użytkownik będzie chciał wykonać. Przykładem jest tutaj kreator importu zapamiętujący miejsce, z którego ostatnio pobieraliśmy pliki, lub też kreator nowych zasobów, który podpowiada nam ostatnio wprowadzone w nim wartości.

Okna dialogowe ogólnego zastosowania

W bibliotekach dołączonych do Eclipse znajdziemy implementację gotowych okien dialogowych, które służą do wykonywania wielu często powtarzanych czynności i mogą zostać bez problemu użyte wewnątrz naszych modułów. Ich definicje zawierają się w kilku warstwach składających się na interfejs użytkownika Eclipse: SWT, JFace czy też w klasach definiujących obszar roboczy.

Pakiet `org.eclipse.swt.widgets` oferuje nam dwa typowe okna dialogowe:

- `DirectoryDialog`, służące do przeglądania i wyboru katalogu.
- `FileDialog`, za pomocą którego wskazujemy plik do otwarcia lub do zapisu danych.

Te dwa okna dialogowe są elementami sterującymi SWT. Oznacza to, że korzystają z odpowiadających im obiektów zdefiniowanych na poziomie konkretnego systemu operacyjnego i mogą się między sobą różnić. Dzięki temu jednak mają na każdej platformie, na której działa aplikacja, naturalny dla użytkownika wygląd. Natomiast interfejs programowy, z którego korzystamy (czyli API), jest identyczny.

W tabeli 11.3 znajdziemy listę okienek dialogowych, które zdefiniowane są w pakiecie `org.eclipse.jface.dialogs`. Możemy z nich korzystać w samodzielnie tworzonych narzędziach.

Tabela 11.3. *Okna dialogowe zdefiniowane w pakiecie org.eclipse.jface.dialogs*

Nazwa klasy	Opis
`MessageDialog`	Wyświetla typowe okienko komunikatu. Może zawierać ilustrujący kategorię komunikatu obrazek. Do dyspozycji mamy następujące możliwości: `none` (brak obrazka), `error` (błąd), `information` (informacja), `question` (pytanie) oraz `warning` (ostrzeżenie).
`InputDialog`	Okienko dialogowe pozwalające użytkownikowi na wprowadzenie pojedynczego łańcucha znaków.
`ProgressMonitorDialog`	Z tego okienka dialogowego korzystamy w sytuacji, gdy wykonujemy długotrwałą operację i chcemy o stanie jej zaawansowania informować użytkownika na bieżąco. Od nas zależy, czy zadanie to zdecydujemy się uruchomić w osobnym wątku. Jeśli tak, możemy obsłużyć funkcję przerwania tego zadania przez użytkownika. Wszystkie te opcje mogą być obsłużone przez opisywane okno.
`ErrorDialog`	Okno dialogowe służące do wyświetlania komunikatu o błędzie. Posiada część, w której mogą znaleźć się szczegółowe informacje o zaistniałej sytuacji.

Również pakiet `org.eclipse.ui.dialogs` oferuje nam zbiór okien dialogowych, które potrafią wesprzeć budowany przez nas moduł:

- `ListSelectionDialog` pozwala użytkownikowi wybrać elementy znajdujące się na liście.
- `ResourceSelectionDialog` służy do wyboru jednego elementu z listy zasobów.
- `ElementTreeSelectionDialog` pozwala wybierać elementy tworzące strukturę drzewa. Podczas selekcji możemy korzystać z mechanizmów filtrowania i kontroli poprawności.

- `SaveAsDialog` odpowiada typowemu oknu dialogowemu „Zapisz jako”. Wskazujemy plik (także taki, który jeszcze nie istnieje), do którego chcemy zapisać zawartość obiektu.

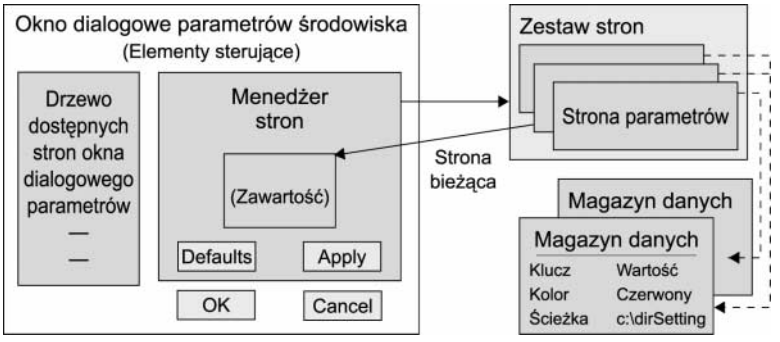
Przedstawiliśmy tylko wybrane okna dialogowe zdefiniowane w pakiecie. Naszym zdaniem są one najbardziej przydatne. Radzimy, by w sytuacji, gdy potrzebne jest nowe okienko dialogowe, sprawdzić, czy w wymienionych pakietach nie istnieje już klasa, która spełnia nasze wymagania. Jeśli jednak stwierdzimy, że konieczne jest zbudowanie własnego okienka, warto jego implementację oprzeć na jednej z wymienionych klas bazowych:

- `org.eclipse.jface.dialogs.Dialog`
- `org.eclipse.jface.dialogs.TitleAreaDialog`
- `org.eclipse.ui.dialogs.SelectionDialog`

Proponujemy również, by opierać się w miarę możliwości na komponentach dostarczanych w ramach SWT i nie tworzyć na przykład własnej implementacji okienka służącego do otwarcia pliku. Stracimy bowiem cechy charakterystyczne dla natywnych w danym systemie aplikacji.

Tworzenie własnej strony parametrów

Strona parametrów pozwala nam definiować opcje dotyczące któregoś z zainstalowanych modułów. Informacje te zapisywane są jako lokalne dane dotyczące wybranego pluginu. Rysunek 11.3 przedstawia strukturę okienka parametrów i zawarte w nim strony. Widać na nim także sposób, w jaki strona komunikuje ze swym magazynem danych (ang. *preference store*).



Rysunek 11.3. *Struktura okienka parametrów systemu i zawartej w nim strony*

Okno dialogowe parametrów środowiska wyświetla wszystkie dostępne strony; zarządza tym moduł zwany menedżerem stron. Każda z nich składa się z pojedynczego komponentu typu `Composite`, który staje się aktywnym obiektem po wskazaniu w drzewie pozycji odpowiadającej danej stronie. Przypominamy, że `Composite` to klasa zdefiniowana w bibliotece SWT (szczegóły znajdują się w rozdziale 10.). Stanowi ona kontener zawierający inne elementy sterujące, zdefiniowane przez nas podczas projektowania strony.

Aktywna strona korzysta z klasy pluginu w celu znalezienia miejsca, w którym zapamiętujemy wartości naszych właściwości (jest to tzw. magazyn danych).

Za komunikację z magazynem danych odpowiada klasa pluginu. Sama strona parametrów nie zarządza zapisywaniem wartości. To klasa pluginu ma dostęp do aktualnej przestrzeni projektów, w której znajduje się przypisany jej folder służący do przechowywania bieżących danych. Jest to zawsze folder lokalny, nie repozytorium. Domyślne wartości parametrów mogą być zdefiniowane wewnątrz pluginu. O wprowadzonych przez użytkownika zmianach informować mogą zdarzenia płynące od poszczególnych elementów interfejsu użytkownika. Wartości parametrów zapisywane są lokalnie, ale mogą być importowane i eksportowane, tak więc istnieje możliwość przenoszenia ich pomiędzy różnymi przestrzeniami projektów.

W magazynie danych związanym ze stroną parametrów możemy przechowywać wartości, które nigdy nie będą widoczne dla użytkownika. Możemy także zapisywać własności zdefiniowane w jednej części narzędzia, a odczytywać je w drugiej. Wszystkie one będą wzięte pod uwagę w czasie wykonywania operacji eksportu oraz importu.

Za większość czynności związanych z dodawaniem strony parametrów odpowiada szkielet, z którego zwykle korzystamy. Naszym podstawowym zadaniem jest definicja znajdujących się w niej pól. Oto lista kolejnych etapów tworzenia własnej strony parametrów:

1. Definicja rozszerzenia dotyczącego strony parametrów.
2. Implementacja strony parametrów.
3. Przygotowanie interfejsu użytkownika strony.
4. Określenie sposobu zapisu i odczytu własności.
5. Ustalenie domyślnych wartości parametrów.
6. Dodanie kodu zarządzającego wartościami parametrów.
7. Obsługa zdarzeń na stronie parametrów.

Oto szczegółowy opis przedstawionych kroków.

Definicja rozszerzenia dotyczącego strony parametrów

Nową stronę parametrów definiujemy, dodając do pliku manifestu punkt rozszerzenia o nazwie `org.eclipse.ui.preferencePages`. Określamy tam wartości trzech atrybutów: nazwę strony, jej identyfikator oraz nazwę klasy zawierającej implementację. Opcjonalnie możemy podać kategorię — na tej podstawie strony parametrów mogą tworzyć strukturę hierarchiczną widoczną po lewej stronie okna parametrów. Poniżej znajduje się przykład, w którym definiujemy dwie strony parametrów. Występuje między nimi zależność polegająca na tym, że druga z nich jest elementem węzła, który stanowi pierwsza.

```
<extension point="org.eclipse.ui.preferencePages">
  <page
     name="QRS Tool Primary Options"
```

```
      class="qrs.tool.preferences.PrimaryPreferencePage"
      id="qrs.tool.preferences.primary">
  </page>
</extension>

<extension point="org.eclipse.ui.preferencePages">
  <page
      name="QRS Tool Secondary Options"
      category="qrs.tool.preferences.primary"
      class="qrs.tool.preferences.SecondaryPreferencePage"
      id="qrs.tool.preferences.secondary">
  </page>
</extension>
```

Powyższe punkty rozszerzenia tworzą dwie strony parametrów. W drugim z nich występuje atrybut `category` — określa on stronę, która jest elementem w stosunku do danej nadrzędnym. Dlatego też druga ze stron znajduje się na drugim poziomie hierarchii.

Od momentu, kiedy punkty rozszerzenia zostaną zdefiniowane, w okienku dialogowym parametrów środowiska pojawią się odpowiadające im pozycje. Nie będzie miał znaczenia fakt, że klasy zawierające implementacje stron jeszcze nie istnieją. Na zawartość drzewa znajdującego się po lewej stronie okna parametrów wpływ mają atrybuty zdefiniowane w punktach rozszerzenia. Klasa z implementacją wymagana jest dopiero w chwili, gdy użytkownik dany element wskaże.

Implementacja strony parametrów

Klasa zawierająca kod obsługujący stronę parametrów musi implementować interfejs `IWorkbenchPreferencePage`. Najlepiej skorzystać z typu bazowego o nazwie `PreferencePage`. Początkowa postać naszej definicji wygenerowana przez narzędzia wchodzące w skład PDE wygląda następująco:

```
public class PrimaryPreferencePage extends PreferencePage
             implements IWorkbenchPreferencePage {
  /**
   * Konstruktor
   */
  public PrimaryPreferencePage() {
  }

  /**
   * @see PreferencePage#init
   */
  public void init(IWorkbench workbench) {
  }

  /**
   * @see PreferencePage#createContents
   */
  protected Control createContents(Composite parent) {
    return null;
  }
}
```

Naszym zadaniem jest wzięcie odpowiedzialności za wygląd strony oraz zapis wartości, pozostałe operacje związane ze współpracą z elementami obszaru roboczego bierze na siebie klasa, po której dziedziczymy.

Interfejs użytkownika definiowany jest za pomocą metody `createContents`. To tutaj określamy wygląd strony i wstawiamy do niej wymagane komponenty.

Wszystkie czynności inicjalizacyjne związane z tworzeniem strony umieszczamy w metodzie `init`. Zostanie ona wywołana tylko raz, gdy po raz pierwszy wejdziemy na daną stronę w oknie parametrów. Jeśli chcemy, aby metoda `init` została użyta ponownie, należy zamknąć okno parametrów i wywołać je po raz kolejny.

Następną czynnością jest implementacja funkcji przypisanych standardowych przyciskom, które pojawiają się na każdej stronie parametrów. Polega ona na pokryciu odpowiadających przyciskom metod. Listę przycisków i metod prezentujemy w tabeli 11.4.

Tabela 11.4. *Metody odpowiadające przyciskom widocznym w okienku parametrów*

Nazwa przycisku	Metoda
OK	`performOk()`
Apply	`performApply()`
Restore Defaults	`performDefaults()`
Cancel	`performCancel()`

Metodę `performOk` pokrywamy w celu zdefiniowania czynności obsługujących zapis aktualnych wartości parametrów. Jest ona wywoływana w odpowiedzi na kliknięcie przycisku *OK* oraz *Apply*. Przycisk *Restore Defaults* może być użyty w celu przywrócenia wartości domyślnych naszych parametrów. Zdefiniowane są one na poziomie pluginu i magazynu danych, który mu odpowiada (piszemy o tym w punkcie „Ustalenie domyślnych wartości parametrów"). W większości przypadków nie ma konieczności implementowania pozostałych metod.

Przyciski *Restore Defaults* oraz *Apply* pojawiają się zwykle na stronie parametrów. Możemy z nich jednak zrezygnować: wystarczy w tym celu pokryć metodę `noDefaultAndApplyButton` w ten sposób, by zwracała wartość `false`.

W momencie, gdy mamy już punkty rozszerzenia oraz projekt klasy implementującej stronę parametrów, przechodzimy do definiowania jej wyglądu oraz sposobu, w jaki zarządzać będziemy wprowadzonymi wartościami.

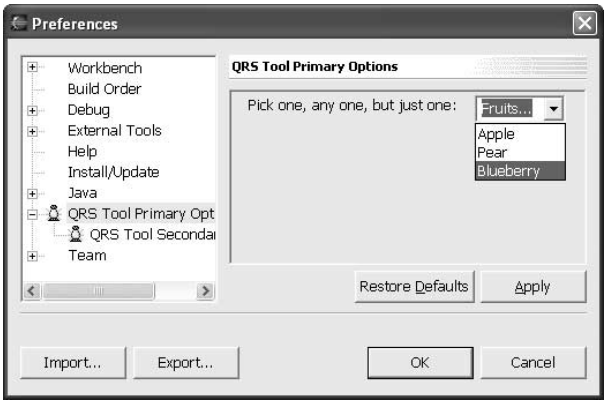
Przygotowanie interfejsu użytkownika strony

W celu zdefiniowania wyglądu strony pokrywamy metodę `createContents`, w której wstawiamy kolejne elementy sterujące tworzące zawartość okienka. Obiektem w stosunku do nich nadrzędnym jest przekazany do metody parametr `parent` typu `Composite`. Szczegóły dotyczące programowania z użyciem komponentów SWT opisaliśmy w rozdziale 10.

Metoda zaprezentowana poniżej dodaje do strony etykietę oraz listę rozwijaną.

```
protected Control createContents(Composite parent) {
  Composite composite = new Composite(parent, SWT.BORDER);
  RowLayout rowLayout = new RowLayout();
  rowLayout.justify = true;
  rowLayout.marginLeft = 5;
  rowLayout.marginRight = 5;
  rowLayout.spacing = 5;
  composite.setLayout(rowLayout);

  Label label = new Label(composite, SWT.NONE);
  label.setText("pick one, any one, but just one:");
  Combo combo = new Combo(composite, SWT.NONE);
  combo.setItems(new String[]
    {"Apple", "Pear", "Blueberry"});
  combo.setText("Fruits...");

  return composite;
}
```

Metoda `createContents` zwraca obiekt typu `Control`, na podstawie którego wyliczany jest rozmiar strony i jej układ w okienku parametrów. Rezultat naszej pracy zobaczymy po wskazaniu odpowiedniej pozycji w oknie dialogowym parametrów, tak jak na rysunku 11.4.



Rysunek 11.4. *Okno dialogowe parametrów środowiska zawierające nową stronę*

Określenie sposobu zapisu i odczytu własności

Zadaniem zdecydowanej większości stron parametrów jest wyświetlenie i możliwość edycji zbioru wartości. Aktualny ich stan zapisywany jest w magazynie danych, którego lokalizację określa związany ze stroną plugin i który znajduje się w odpowiednim folderze naszej przestrzeni projektów. Poszczególne czynności związane z odczytem i zapisem wartości nie są skomplikowane:

- Plugin udostępnia nam obiekt zawierający metody `get` i `set` obsługujące odczyt i zapis wartości (jest to zwykle obiekt klasy `PreferenceStore` lub `Preferences`).
- Ustalamy domyślne wartości parametrów dla danej strony.

- Używamy tego obiektu w implementacji naszej strony parametrów.
- Sprawdzamy, czy zawartość obiektu jest poprawnie zapisywana na dysku po zamknięciu Eclipse.

Eclipse oferuje dwa sposoby zapisywania parametrów. W obu przypadkach obiekt, z którego korzystamy, otrzymujemy za pomocą funkcji zawartych w klasie pluginu.

- Używamy klasy `AbstractUIPlugin` i obiektu `PreferenceStore`.
- Korzystamy z klasy `Plugin` i obiektu `Preferences`.

Obiekty typu `PreferenceStore` i `Preferences` obsługują następujące funkcje:

- Zapis i odczyt pary klucz-wartość.
- Ustawianie i pobieranie wartości domyślnych dla poszczególnych kluczy.
- Zapis wartości odpowiadających typom prostym (`boolean`, `double`, `float`, `int`, `String` i `long`).
- Obsługa obiektów słuchaczy, które odbierają zdarzenia związane ze zmianą wartości.

Dlaczego mamy aż dwa sposoby utrwalania wartości parametrów? Z przyczyn historycznych. API `AbstractUIPlugin` i implementacja `PreferenceStore` pochodzą z pierwszej wersji Eclipse. Z wariantu tego korzystać możemy tylko wtedy, gdy klasa pluginu dziedziczy po `AbstractUIPlugin`. To z kolei wymaga dołączenia pluginu `org.eclipse.ui` do sekcji `<requires>` pliku manifestu. W tym podejściu magazyn danych został tak zaprojektowany, by zapewnić możliwość zapisu ustawianych przez użytkownika parametrów widocznych w oknie *Preferences*.

API `Plugin` i implementacja obiektu `Preferences` pojawiły się w Eclipse 2.0. Nie korzystamy w tym wypadku z pluginu `org.eclipse.ui`, opieramy się na funkcjach zawartych w module `org.eclipse.core.runtime`. Dzięki nim możliwościami zapisu i odczytu parametrów dysponuje każda aplikacja Eclipse, która dziedziczy po klasie `Plugin` (zamiast `AbstractUIPlugin`), w szczególności nieposiadająca własnego interfejsu użytkownika (nie jest wymagany plugin `org.eclipse.ui`). Większość istniejących pluginów implementuje ogólny interfejs `AbstractUIPlugin`. Jeśli jednak chcielibyśmy zbudować lekki, pozbawiony części interfejsu moduł i korzystać w nim z zapisu i odczytu pewnych właściwości, warto postawić na implementację abstrakcyjnej klasy `Plugin`.

Ustalenie domyślnych wartości parametrów

Domyślne wartości parametrów definiowane są wewnątrz pluginu. Istnieje wiele wariantów metody `setDefault`, która służy do ich określania i zapewnia obsługę różnych typów. Jeśli dla danej właściwości zdefiniowano wartość domyślną, jest ona zwracana podczas wywołania metody typu `get` (chyba że wcześniej została zmodyfikowana przez użytkownika). Możemy także do wartości domyślnych odwoływać się bezpośrednio.

Jeśli nasza klasa pluginu dziedziczy po `AbstractUIPlugin`, w celu ustalenia wartości domyślnych należy pokryć jedną z zaprezentowanych poniżej metod:

- `initializeDefaultPreferences(IPreferenceStore store)`
- `initializeDefaultPluginPreferences()`

Metody te wywoływane są w momencie, gdy prosimy plugin o udostępnienie obiektu `PreferenceStore` lub `Preferences`.

Gdy korzystamy z obiektu `PreferenceStore`, pokrywamy metodę `initializeDefaultPreferences` klasy `AbstractUIPlugin`.

```
protected void
initializeDefaultPreferences(IPreferenceStore store) {
  // każdy wiersz określa jedną wartość domyślną
  store.setDefault("pref_key", "myDefaultValue");
}
```

Gdy natomiast zamierzamy użyć obiektu `Preferences`, pokrywamy metodę `initializeDefaultPluginPreferences` klasy `AbstractUIPlugin`.

```
protected void initializeDefaultPluginPreferences() {
  // każdy wiersz określa jedną wartość domyślną
  getPluginPreferences().setDefault(
    "pref_key",
    "myDefaultValue");
}
```

Dodawanie kodu zarządzającego wartościami parametrów

Określenie sposobu, w jaki zapisywane będą dane znajdujące się na stronie parametrów, wymaga kilku czynności. Konieczne jest uzyskanie dostępu do obiektu, który wykorzystamy do utrwalania właściwości. Opcjonalnie kojarzymy go ze stroną parametrów, pobieramy wartości w celu wyświetlenia, obsługujemy ewentualne żądanie przywrócenia wartości domyślnych, w końcu dołączamy funkcjonalność związaną z zapisem danych. Kolejne punkty zawierają szczegółowy opis wszystkich tych etapów.

Pobranie referencji do magazynu danych

Referencję do obiektu pełniącego funkcję magazynu danych uzyskamy, wywołując jedną z poniższych metod:

```
IPreferenceStore ps = DialogsPlugin.getDefault().getPreferenceStore();
Preferences prefs = DialogsPlugin.getDefault().getPluginPreferences();
```

Lista metod typu `set` i `get`, za pomocą których realizowany jest dostęp do poszczególnych parametrów, w obu przypadkach jest taka sama. Jako twórca modułu możesz zdecydować, czy dostęp do parametrów będzie możliwy za pośrednictwem jednej lub drugiej klasy. Wyjątkiem jest tu metoda definiująca wartości domyślne. Możemy zdefiniować nawet strony parametrów, które posługują się różnymi wariantami dostępu w obrębie jednego pluginu. Przy posługiwaniu się klasą `FieldEditorPreferencePage` (co zostanie opisane nieco dalej), możemy korzystać wyłącznie z obiektu typu `PreferenceStore`.

Teraz, gdy uzyskaliśmy dostęp do magazynu danych pluginu, definiujemy reguły dotyczące wartości.

Podłączenie magazynu danych do strony parametrów

Gdy uzyskamy dostęp do zdefiniowanego w ramach pluginu obiektu `PreferenceStore`, możemy bezpośrednio przypisać go do strony parametrów. Służy do tego metoda `setPreferenceStore`, a całą operację najlepiej przeprowadzić w ramach wspominanej już wcześniej metody `init` wywoływanej podczas pierwszego odwołania do strony.

```
ToolPlugin myPlugin = ToolPlugin.getDefault();
if (myPlugin instanceof AbstractUIPlugin) {
  AbstractUIPlugin uiPlugin = (AbstractUIPlugin) myPlugin;
  this.setPreferenceStore(uiPlugin.getPreferenceStore());
}
```

Dzięki takiemu przyporządkowaniu kod zawarty na stronie i odpowiadający za wyświetlanie oraz późniejsze zapisanie wartości może posługiwać się metodą `getPreferenceStore` i w ten sposób, z pominięciem obiektu pluginu, może odwoływać się bezpośrednio do magazynu danych. Pozwala to zdefiniować klasę strony parametrów stanowiącą wzorzec, w którym podłączenie do magazynu danych odbywać się będzie na poziomie konstruktora.

W hierarchii klas dotyczących strony parametrów nie znajdziemy natomiast funkcji, która obsługiwałyby w ten sam sposób obiekt typu `Preferences`. Jeśli chcielibyśmy skorzystać z podobnego mechanizmu, w klasie implementującej stronę parametrów należy samodzielnie zdefiniować odpowiednie pole i metody.

Pobieranie wartości parametrów

Zadaniem strony parametrów jest wyświetlenie bieżących ustawień aplikacji. Użytkownik dzięki temu może je przeglądać i modyfikować. Do pobierania wartości stosujemy metody z rodziny `get`; każda z nich odpowiada innemu typowi. Tak więc łańcuchy znakowe odczytujemy metodą `getString`, a wartości logiczne — `getBoolean`. Oto przykład pobrania własności typu `String`, którą następnie wstawiamy do elementu sterującego:

```
// Wariant z użyciem obiektu PreferenceStore
combo.setText(getPreferenceStore().getString("pref_key"));

// Wariant z użyciem obiektu Preferences
combo.setText(ToolPlugin.getDefault().getPluginPreferences().getString("pref_key"))
;
```

Wartości pobieramy zwykle raz, w ramach tworzenia interfejsu użytkownika strony. Może zdarzyć się jednak taka sytuacja, że tę samą właściwość umieścimy na więcej niż jednej stronie parametrów. W takim wypadku konieczna jest operacja odświeżania podczas zmiany stron. Nie polecamy takiego rozwiązania, jednak czasami jesteśmy do tego zmuszeni. W takim przypadku kod odpowiadający za ponowne pobranie i wyświetlenie wartości umieścić możemy w metodzie `setVisible`.

Przywracanie wartości domyślnych

Za pomocą znajdującego się na stronie parametrów przycisku *Restore Defaults* użytkownik może przywrócić wartości domyślne. Poniżej prezentujemy kod, który stanowi przykład implementacji takiej operacji. Zakładamy w nim, że nasz plugin zawiera już definicję wartości domyślnych:

```
// Wariant z użyciem obiektu PreferenceStore
combo.setText(getPreferenceStore().getDefaultString("combo_field_key"));

// Wariant z użyciem obiektu Preferences
combo.setText(
  ToolPlugin.getDefault().
             getPluginPreferences().
             getDefaultString("combo_field_key"));
```

Zapis wartości parametrów

Gdy użytkownik kliknie przycisk *Apply* lub *OK*, odczytujemy bieżące wartości widoczne na ekranie i na ich podstawie modyfikujemy właściwości znajdujące się w magazynie danych. Kod, który to realizuje, znajduje się w metodzie `performOk`. Jest ona wywoływana w odpowiedzi na kliknięcie zarówno przycisku *Apply*, jak też *OK*.

Istnieje tylko jedna metoda, `setValue`, służąca do ustawiania wartości parametru. Jednak zdefiniowano kilka jej wariantów, a to za sprawą konieczności obsługi wielu typów zapisywanych danych (`boolean`, `double`, `float`, `int`, `String` i `long`). Metodę zawsze wywołujemy z dwoma parametrami, z których pierwszy stanowi klucz, a drugi wartość. Oto przykład:

```
// Wariant z użyciem obiektu PreferenceStore
getPreferenceStore().setValue(
  "combo_field_key",
  combo.getText());

// Wariant z użyciem obiektu Preferences
ToolPlugin.getDefault().getPluginPreferences().setValue(
  "combo_field_key",
  combo.getText());
```

Metoda `setValue` nie powoduje jednak zapisu wartości na dysk. Magazyn danych zarządzany przez plugin jest automatycznie utrwalany podczas zamykania Eclipse. Gdy plugin dziedziczy po klasie `AbstractUIPlugin`, zajmuje się tym metoda `shutdown`. Istnieje jednak sposób na natychmiastowe zapisanie wartości. Należy w tym celu wywołać zdefiniowaną na poziomie pluginu metodę o nazwie `savePluginPreferences`. Tak postępujemy, gdy nasz moduł dziedziczy po klasie `Plugin`.

Możemy również osobno zaimplementować metodę `performApply`, która wywoływana będzie po kliknięciu przycisku *Apply*. Należy jednak pamiętać, że stanowi ona jedynie uzupełnienie, a nie alternatywę dla metody `performOk` (`performOk` będzie i tak wywołana po kliknięciu *Apply*).

Obsługa zdarzeń na stronie parametrów

Jeśli chcemy na bieżąco obserwować zmiany zachodzące na stronie parametrów i od razu na nie reagować, dodajemy swój obiekt będący słuchaczem zdarzeń. Operacja ta dotyczyć może zarówno wariantu korzystającego z `PreferenceStore`, jak też `Preferences`. Implementacja obiektu odpowiadającego na zdarzenia pochodzące ze strony parametrów zależy od sposobu magazynowania danych. Obiekt nasłuchujący zostanie poinformowany o zdarzeniu związanym ze zmianą właściwości.

Naszym zadaniem jest ustalenie na podstawie parametrów zdarzenia, czy stan danego parametru ma wpływ na inne wartości, i ewentualne podjęcie odpowiednich, zgodnych z logiką aplikacji czynności. Poniższe fragmenty kodu pokazują, jak za pomocą słuchacza zdefiniowanego w postaci klasy wewnętrznej wypisać identyfikator własności.

```
// Wariant z użyciem obiektu PreferenceStore
ToolPlugin
  .getDefault()
  .getPreferenceStore()
  .addPropertyChangeListener(new IPropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent event) {
    System.out.println("a preference setting changed");
    System.out.println(event.getProperty());
  };
});

// Wariant z użyciem obiektu Preferences
ToolPlugin
  .getDefault()
  .getPluginPreferences()
  .addPropertyChangeListener(
    new Preferences.IPropertyChangeListener() {
    public void propertyChange(
      Preferences.PropertyChangeEvent event) {
        System.out.println("a preference setting changed");
        System.out.println(event.getProperty());
    };
});
```

Edytor pól jako typ strony parametrów

Gdy zdecydujemy się na stworzenie strony parametrów, która dziedziczy po klasie `PreferencePage`, scenariusz dalszego postępowania jest następujący:

- Implementacja interfejsu użytkownika strony za pomocą komponentów biblioteki SWT i JFace.
- Pobranie poszczególnych parametrów stanowiących stan początkowy strony.
- Pokrycie metod obsługujących zapis własności oraz powrót do wartości domyślnych.

Istnieje jednak inne podejście. Możemy zbudować stronę opierającą się na wzorcu noszącym nazwę edytora pól. W takim przypadku proces definiowania strony przebiega inaczej:

1. Dodajemy rozszerzenie dotyczące strony.
2. Tworzymy klasę, która dziedziczy po `FieldEditorPreferencePage` i implementuje interfejs `IWorkbenchPreferencePage`. Zawiera ona w sobie obsługę edytora pól.
3. Definiujemy bezparametrowy konstruktor. W nim określamy układ edytora pól, wywołując konstruktor klasy bazowej i przekazując mu parametr `GRID` lub `FLAT`.
4. Dołączamy do strony parametrów obiekt będący magazynem danych (w tym przypadku musi to być obiekt typu `PreferenceStore`).
5. Implementujemy metodę `createFieldEditors`, która tworzy i dodaje do strony odpowiednie pola (obiekty dziedziczące po klasie `FieldEditor`).

Klasa `FieldEditorPreferencePage` sama zajmuje się takimi operacjami, jak obsługa, zapis i odczyt wartości. Dodawanie gotowych komponentów typu `FieldEditor` jest zadaniem łatwiejszym niż budowanie kompletnego interfejsu użytkownika od początku i zapewnienie poprawnej jego współpracy z obiektem pełniącym funkcję magazynu danych.

Poniżej przedstawiona została implementacja strony parametrów opartej na klasie `FieldEditorPreferencePage`, która wywołuje konstruktor klasy bazowej z parametrem `GRID` i podłącza obiekt magazynu danych:

```
public class MyFieldEditorPrefPage
  extends FieldEditorPreferencePage
  implements IWorkbenchPreferencePage {

  public MyFieldEditorPrefPage() {
    super(GRID);
    setDescription("My Field Editor Preference Page \n");
    IPreferenceStore store = ToolPlugin.getDefault()
      .getPreferenceStore();
    setPreferenceStore(store);
  }
  .
  .
  .
}
```

Obiekty odpowiadające kolejnym polom strony również zawierają w sobie podstawową funkcjonalność i współpracują z innymi składnikami tworzącymi szablon. Stanowią one określone komponenty interfejsu użytkownika, ładują wartości, obsługują ich zapis oraz przywracanie wartości domyślnych. Za szczegóły związane z logiką przetwarzania odpowiada już implementacja klasy reprezentującej naszą własną stronę, która dziedziczy po `FieldEditorPreferencePage`.

W tabeli 11.5 prezentujemy rodzaje elementów, które można wstawić do strony edytora pól.

Tabela 11.5. *Typy obiektów, które mogą być składnikami strony edytora pól*

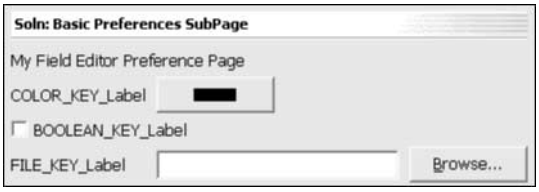
Rodzaj pola	Opis
`BooleanFieldEditor`	Pole obsługujące wartość typu `boolean`.
`RadioGroupFieldEditor`	Kilka wartości widocznych jako lista pól opcji.
`IntegerFieldEditor`	Pole służące do edycji wartości typu `integer`.
`ColorFieldEditor`	W ten sposób obsługujemy wybór koloru.
`StringFieldEditor`	Element, w którym wprowadzamy łańcuch tekstowy.
`StringButtonFieldEditor`	Abstrakcyjne pole służące do wprowadzania wartości będącej łańcuchem znaków, wyposażone w przycisk pozwalający dodać obsługę (np. okno dialogowe) związaną z edycją pola.
`ListEditor`	Abstrakcyjny element zarządzający wyborem wartości z listy. Wyświetla poszczególne wartości, przyciski służące do dodawania nowych i usuwania istniejących oraz przyciski *Up* i *Down* zmieniające ich kolejność.
`PathEditor`	Pole związane z wyborem ścieżki dostępu. Zbudowane w oparciu o klasę `ListEditor`.
`DirectoryFieldEditor`	Pole obsługujące parametry będące ścieżką katalogu. Po kliknięciu przycisku `Change` pojawia się standardowe okno dialogowe zawierające listę folderów.
`FileFieldEditor`	W tym polu zawarta jest obsługa wyboru pliku wraz z pełną ścieżką dostępu. Standardowe okno dialogowe wywołujemy przyciskiem `Change`.
`FontFieldEditor`	W ten sposób obsługujemy wybór czcionki.
`PropagatingFontFieldEditor`	Pole implementuje zależność między wartościami znajdującymi się w dwóch magazynach danych: źródłowym i docelowym. Każda zmiana, która zachodzi w pierwszym, pociąga za sobą modyfikację drugiego.
`WorkbenchChainedTextFont FieldEditor`	Pole podobne do poprzedniego. Źródłowym magazynem danych są tutaj ustawienia obszaru roboczego.

Poniżej znajduje się przykład implementacji metody `createFieldEditors`, w której do strony dodajemy trzy pola różnego typu:

```
protected void createFieldEditors() {
  // Pierwsza wartość typu String jest kluczem
  // służącym do identyfikacji właściwości,
  // drugi łańcuch to etykieta widoczna na stronie

  ColorFieldEditor colorField = new ColorFieldEditor(
    "COLOR_KEY", "COLOR_KEY_Label",
    getFieldEditorParent());

  BooleanFieldEditor choiceField = new BooleanFieldEditor(
    "BOOLEAN_KEY", "BOOLEAN_KEY_Label",
    org.eclipse.swt.SWT.NONE,
    getFieldEditorParent());

  FileFieldEditor fileField = new FileFieldEditor(
    "FILE_KEY", "FILE_KEY_Label",
    true, getFieldEditorParent());
    fileField.setFileExtensions(
      new String[] {"*.jar", "*.txt", "*.zip" });
```

```
        addField(colorField);
        addField(choiceField);
        addField(fileField);
    }
```

Zaprezentowany kod definiuje stronę, której kształt prezentujemy na rysunku 11.5 (oczywiście, strona jest w rzeczywistości elementem okna dialogowego parametrów systemu).

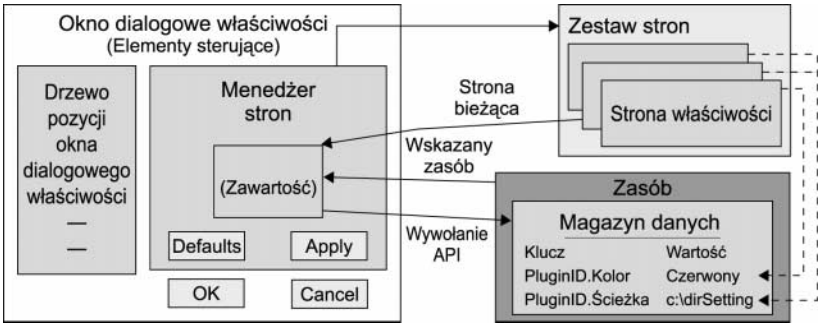


Rysunek 11.5. *Strona będąca edytorem pól*

Tworzenie własnej strony właściwości

Po zdobyciu odpowiednich doświadczeń w budowie stron parametrów (*Preferences*) przechodzimy do stron właściwości (*Properties*). Implementacja obu tych składników środowiska Eclipse ma wiele wspólnych elementów.

Rolą okienka dialogowego właściwości jest prezentacja i ewentualna modyfikacja cech obiektu, któremu przypisane jest omawiane okno. W wielu widokach, takich jak *Navigator*, *Package Explorer* czy *Error Log*, po wskazaniu danego obiektu i wybraniu z menu polecenia *Properties* wyświetli się okienko zawierające listę cech charakterystycznych dla tego właśnie elementu. Chociaż nową stronę właściwości możemy dodać do każdego okienka typu *Properties*, najczęściej wiążemy ją z określonym typem zasobu lub obiektem z danym zasobem związanym. Dlatego też typowym miejscem, do którego ją dołączamy, jest okienko *Properties* widoku nawigatora i przeglądarki pakietów. Możliwość powiązania określonego zbioru własności z danym zasobem jest dużym ułatwieniem dla użytkownika. Z jego punktu widzenia środowisko dostosowuje się na bieżąco do wykonywanych przez niego czynności. Rysunek 11.6 prezentuje strukturę okna dialogowego *Properties* i osadzonych w nim stron. Widzimy tam również sposób, w jaki strona właściwości współpracuje z zasobem wskazanym tuż przed wywołaniem okienka.



Rysunek 11.6. *Struktura okna dialogowego właściwości i znajdujących się tam stron*

Jak widać na rysunku, okno dialogowe *Properties* prezentuje użytkownikowi pewien zbiór stron właściwości zarządzanych przez obiekt będący ich menedżerem. Każda strona właściwości stanowi pojedynczy panel, który staje się aktywnym i widocznym elementem interfejsu po wskazaniu odpowiadającej mu pozycji w drzewie zawierającym listę dostępnych stron. Strona posiada dostęp do wskazanego zasobu bądź innego obiektu. Jeśli jest to zasób, za pomocą API związanego z przestrzenią projektu możemy odwoływać się do przypisanych mu parametrów. Ich zbiór jest przechowywany w magazynie danych, którym zarządza Eclipse. Magazyn danych stanowią wartości, które znajdują się w przestrzeni projektów, nie są natomiast wysyłane do repozytorium.

Pozostałe komponenty pokazane na rysunku 11.6 są częścią szablonu, na podstawie którego tworzymy własne rozszerzenie. Również w tym przypadku w wielu czynnościach mogą wyręczyć nas klasy będące szkieletem dla naszych własnych rozwiązań. Stworzenie typowej strony właściwości składa się z czterech etapów:

1. Definicja rozszerzenia dotyczącego strony właściwości.
2. Implementacja klasy strony.
3. Definicja wyglądu strony.
4. Napisanie kodu, który odpowiada za jej logikę.

Poniżej zamieszczamy dokładny opis tych punktów.

Definicja rozszerzenia dotyczącego strony właściwości

Pierwszym krokiem do dodania nowej strony właściwości jest zdefiniowanie w pliku manifestu rozszerzenia typu `org.eclipse.ui.propertyPages`. W ramach tej operacji należy:

- Nadać stronie nazwę oraz identyfikator.
- Określić rodzaj obiektu lub zasobu, z którym nową stronę chcemy połączyć.
- Wskazać klasę zawierającą implementację strony.
- Opcjonalnie dołączyć ikonkę widoczną na stronie.
- Zdefiniować (również opcjonalnie) filtr pozwalający precyzyjnie określić warunki, jakie muszą zostać spełnione, by strona została włączona do okna dialogowego *Properties*.

W ten sposób dołączamy stronę do okna dialogowego wywoływanego w celu edycji właściwości wskazanego typu zasobów.

Atrybut `objectClass` określa rodzaj składnika, z którym strona będzie związana. Mamy tu na myśli ogólny typ elementu, tzn. plik, folder lub projekt wskazany w widoku *Navigator*. Wymienionym w poprzednim zdaniu typom elementów odpowiadają kolejno interfejsy `IFile`, `IFolder` oraz `IProject`.

Zaprezentowana poniżej definicja rozszerzenia dotyczy strony właściwości, która pojawi się, gdy wskazanym elementem będzie plik o rozszerzeniu *.java*. Jej implementacja znajdzie się w klasie `JavaPropertyPage`. Zostanie ona dołączona do okienka dialogowego

Properties w chwili, gdy plik tego typu wskażemy w widoku *Navigator*. Nie pojawi się natomiast po jego wskazaniu w przeglądarce pakietów — widok ten traktuje zawarte w nim obiekty jako elementy języka Java a nie pliki.

```
<extension point="org.eclipse.ui.propertyPages">
  <page
     objectClass="org.eclipse.core.resource.IFile"
     name="Java File Properties"
     nameFilter="*.java"
     class="qrs.tool.properties.JavaPropertyPage"
     id="qrs.tool.properties.qrsfile">
  </page>
</extension>
```

Jednak zakładamy, że jeśli definiujemy stronę dla plików typu *.java*, chcielibyśmy móc ją zobaczyć w okienku właściwości, które otworzymy podczas pracy w przeglądarce pakietów. Tam jednak ten sam plik traktowany jest jako zasób innego rodzaju, którym jest tzw. jednostka kompilacji (ang. *compilation unit*). Rozwiązanie tego problemu jest możliwe i polega na tym, że korzystamy z atrybutu `adaptable`, wskazującego na to, że dany obiekt może zostać potraktowany jako określony zasób, tak by można było skorzystać z funkcji służących do zapisu i odczytu jego właściwości. Przypisanie `adaptable="true"` pozwoli nam zobaczyć stronę właściwości zarówno w widoku *Navigator*, jak też *Package Explorer*.

```
<extension point="org.eclipse.ui.propertyPages">
  <page
     objectClass="org.eclipse.core.resource.IFile"
     adaptable="true"
     name="Java Resource Properties"
     nameFilter="*.java"
     class="qrs.tool.properties.JavaPropertyPage"
     id="qrs.tool.properties.qrsfile">
  </page>
</extension>
```

Widok *Package Explorer* nie wyświetla plików, tylko obiekty (elementy typu *IJava-Elemet*) zdefiniowane wewnątrz modułu JDT. Tak więc to, co dla widoku *Navigator* jest plikiem typu *.java* (odpowiada składnikowi typu `IFile`), dla przeglądarki pakietów (która wchodzi w skład JDT) jest jednostką kompilacji (interfejs `ICompilationUnit`). Jednak wskazany element może zostać odpowiednio przystosowany i traktowany jako typ `IResource`, udostępniając tym samym metody `get` i `set`, za pomocą których uzyskujemy dostęp do jego parametrów.

Gdybyśmy chcieli, by nasza strona została dołączona wyłącznie do okienka *Properties* widoku *Package Explorer*, moglibyśmy atrybutowi `objectClass` przypisać wartość `org.eclipse.jdt.core.ICompilationUnit`.

Możemy także wyposażyć nasze rozszerzenie w mechanizm filtrowania elementów, z którymi chcemy związać nowo dodaną stronę. W tym celu należy do węzła `<page>` wstawić element `<filter name="" value="">`, definiując w ten sposób dodatkowe warunki. Listę opcji filtrowania prezentuje tabela 11.6.

Tabela 11.6. *Opcje filtrowania strony właściwości*

Typ obiektu	Nazwa	Wartości
Zasoby	`extension`	Rozszerzenie, np. *.txt* lub *.java*.
	`name`	Nazwa zasobu.
	`path`	Ścieżka dostępu (dozwolony jest znak „*”).
	`projectNature`	Identyfikator natury projektu.
	`readOnly`	Czy zasób jest tylko do odczytu (`true` lub `false`).
Projekty	`nature`	Identyfikator natury projektu.
	`open`	Czy projekt jest otwarty (`true` lub `false`).

Implementacja klasy strony

Klasa, która odpowiada za obsługę strony właściwości, implementować powinna interfejs `IWorkbenchPropertyPage`. Możemy oczywiście definiować ją od podstaw samodzielnie, znacznie wygodniej jednak będzie skorzystać z klasy bazowej `PropertyPage`.

```
public class JavaPropertyPage extends PropertyPage
             implements IWorkbenchPropertyPage {
  /**
   * @see PropertyPage#createContents
   */
  protected Control createContents(Composite parent) {
    return null;
  }
}
```

Jeśli używamy klasy, która stanowi szablon strony, możemy pominąć kodowanie standardowych operacji i skoncentrować się wyłącznie na jej cechach indywidualnych, takich jak wygląd czy odpowiednie zarządzanie wartościami.

Jeśli strona ma charakter czysto informacyjny, wymagane jest pokrycie wyłącznie metody `createContents`. Dodajemy w niej elementy sterujące; komponentem w stosunku do nich nadrzędnym jest przekazany metodzie parametr typu `Composite`. Gdy jedynie wyświetlamy własności obiektu warto pokryć metodę `noDefaultAndApplyButton` i zwrócić w niej wartość `false` — w ten sposób ukryjemy zbędne przyciski *Restore Defaults* oraz *Apply*. Pozostaną tylko przyciski *OK* i *Cancel*, które służą do zamknięcia okna dialogowego *Properties*.

Gdy strona właściwości dopuszcza, oprócz wyświetlania, również modyfikację cech danego zasobu, czeka nas jeszcze nieco pracy. Operacje te w dużym stopniu związane będą z obsługą przycisków, które wraz z odpowiadającymi im metodami przedstawiamy w tabeli 11.7.

Mamy w tym momencie definicję punktu rozszerzenia oraz szkielet klasy obsługującej stronę właściwości. Kolejne punkty poświęcimy analizie kodu źródłowego, który odpowiada za jej wygląd i logikę.

Tabela 11.7. *Metody pokrywane w czasie implementacji strony*

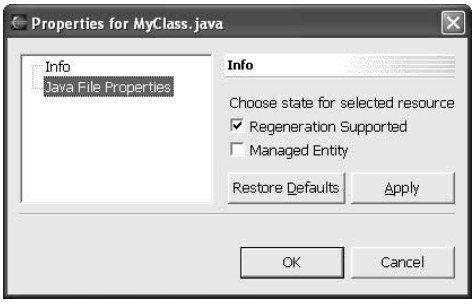
Nazwa przycisku	Metoda
OK	`performOk()`
Apply	`performApply()`
Restore Defaults	`performDefaults()`
Cancel	`performCancel()`

Definicja wyglądu strony

W celu dodania do strony elementów sterujących pokrywamy metodę `createContents`. W przykładzie zaprezentowanym poniżej komponentami tymi są etykieta oraz dwa pola wyboru:

```
protected Control createContents(Composite parent) {
  Composite composite = new Composite(parent, SWT.NONE);
  GridLayout gridLayout = new GridLayout();
  composite.setLayout(gridLayout);

  Label label = new Label(composite, SWT.NONE);
  label.setText("Choose state for selected resource.");
  Button regenB = new Button(composite, SWT.CHECK);
  regenB.setText("Regeneration Supported");
  regenB.setSelection(getRegenPropertyState());
  Button managedB = new Button(composite, SWT.CHECK);
  managedB.setText("Managed Entity");
  managedB.setSelection(getManagedPropertyState());

  return composite;
}
```

Wartość pól wyboru odpowiada parametrom zasobu. Wynik działania metody `createContents` widzimy na rysunku 11.7.



Rysunek 11.7. *Interfejs użytkownika strony właściwości*

Kod odpowiadający za logikę strony

API służące do obsługi przestrzeni projektów (szczegóły w rozdziale 15. — „Obsługa zasobów przestrzeni projektów”) zawiera funkcje pozwalające odczytywać i zapisywać parametry danego zasobu. Interfejs `IResource` definiuje zbiór metod z wyraźnie zazna-

czonym podziałem na te, które ustalają własności zasobu na stałe, i na te, które robią to tylko na czas trwania bieżącej sesji.

```
setSessionProperty(QualifiedName key, Object value);
getSessionProperty(QualifiedName key);
setPersistentProperty(QualifiedName key, String value);
getPersistentProperty(QualifiedName key);
```

Wartość parametru `QualifiedName`, który określa nazwę własności, składa się w tym przypadku z dwóch części. Pierwsza z nich zwyczajowo określa nazwę pluginu, druga to nazwa własności. Te z nich, które obowiązują tylko na czas trwania sesji, przechowywane są wyłącznie w pamięci. Własności mające charakter trwały zapisywane są na dysku, w aktualnej przestrzeni projektów. Te pierwsze mogą być reprezentowane przez obiekt dowolnego typu, właściwości utrwalane na dysku to wyłącznie łańcuchy znakowe (typ `String`).

Poniżej widzimy fragment kodu pobierający wartości, które określać będą początkowy stan strony właściwości:

```
private static QualifiedName REGEN_PROPERTY_KEY =
  new QualifiedName("qrs.toolPlugin", "regen");

...

private boolean getRegenPropertyState() {
  IResource resource = (IResource) getElement();
  try {
    String propValue =
      resource.getPersistentProperty(REGEN_PROPERTY_KEY);
    if ((propValue != null) && propValue.equals("True"))
      return true;
    else
      return false;
  } catch(CoreException e) {
    // obsługa wyjątku
  }
  return false;
}
```

Teraz natomiast przedstawiamy przykład implementacji metody `performOk`, w której zapisywana wartość konwertowana jest do typu `String`:

```
public boolean performOk() {
  setRegenPropertyState(regenB.getSelection());
  return super.performOk();
}

private void setRegenPropertyState(boolean value) {
  IResource resource = (IResource) getElement();
  try {
    if (value) {
      resource.setPersistentProperty(
        REGEN_PROPERTY_KEY, "True");
    } else {
```

```
            resource.setPersistentProperty(
              REGEN_PROPERTY_KEY, "False");
          }
        } catch (CoreException e) {}
      }
```

Metoda `performOk` jest wywoływana w odpowiedzi na kliknięcie przycisku *OK*. Jest ona również domyślną częścią `performApply`.

Zasady przyporządkowywania właściwości

Podstawowym zadaniem strony właściwości i prezentowanych na niej wartości jest możliwość przypisania określonej cechy do konkretnego, pojedynczego zasobu. Korzystamy przy tym z API, które zawiera funkcje dotyczące przestrzeni projektów. Czasami jednak definiujemy własność dotyczącą nie pojedynczego pliku, ale wszystkich zasobów danego typu. Możemy wtedy zastosować rozwiązanie polegające na tym, że wartość widoczna na stronie właściwości zapisywana będzie w magazynie parametrów całego systemu przy pomocy jednej pary klucz-wartość. Tak jak dzieje się w przypadku wartości widocznych w okienku parametrów systemu. Implementacja takiego wariantu nie jest skomplikowana, klasa strony właściwości dziedziczy po klasie strony parametrów. Możemy więc z jej poziomu odwoływać się do wspólnego dla całej przestrzeni projektów magazynu danych.

Jednak w takim momencie należy zweryfikować to podejście i upewnić się, czy jest to słuszna decyzja projektowa. Możemy zadać sobie przecież pytanie, czy w sytuacji, gdy mamy do czynienia z jedną, globalną wartością, jej miejscem nie powinno być okienko parametrów środowiska i strona związana z którymś z narzędzi?

Musimy być przekonani co do przesłanek takiego rozwiązania. A użytkownik intuicyjnie powinien kojarzyć daną opcję z okienkiem właściwości zasobu. Jeśli tak nie jest, edycję wartości należy przenieść do dialogu *Preferences*.

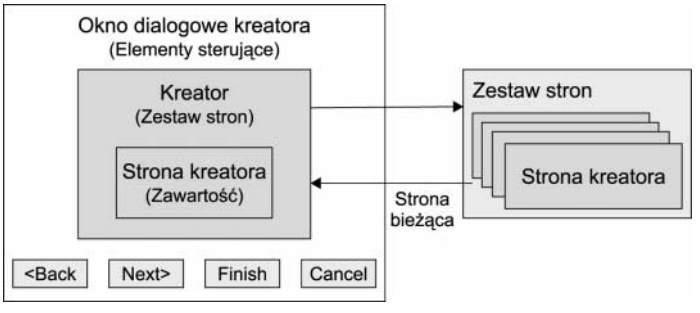
Implementacja własnego okna dialogowego Properties

Dotychczas zakładaliśmy, że rozszerzać będziemy okno dialogowe *Properties* związane z widokiem *Navigator*, wspominaliśmy też o wyświetlaniu właściwości elementów zawartych w widoku *Package Explorer*, gdzie dokonaliśmy pewnego rodzaju rzutowania zasobu typu `IFile` na obiekt widziany jako jednostka kompilacji.

Jednak swoją stronę właściwości dołączyć możemy do dowolnego okienka dialogowego *Properties*, jeśli tylko znamy typ obiektów wyświetlanych w widoku, z którym jest ono związane. W takich przypadkach często dokonujemy operacji **adaptacji** zasobów do typu obiektów znajdujących się w widoku. W tym przypadku konieczne jest, aby obiekty prezentowane w widoku implementowały interfejs `IAdaptable`.

Kreatory tworzenia nowych zasobów, importu oraz eksportu

Definiując odpowiednie rozszerzenie i posługując się dostępnym szablonem, w prosty sposób jesteśmy w stanie skonstruować kreator odpowiadający za tworzenie nowych zasobów, a także za ich import i eksport. Rysunek 11.8 przedstawia strukturę kreatora oraz zawartych w nim stron i ich relacje z klasą `WizardDialog`, która stanowi ramy całego rozwiązania.



Rysunek 11.8. *Struktura kreatora i zawartych w nim stron*

Kreator reprezentuje pewną, złożoną zwykle z paru kroków, operację. Składa się on z kilku stron, każda z nich odpowiada jednemu etapowi zadania. Bardzo często przejście do kolejnej strony zależy od tego, czy poprawnie wypełniono stronę poprzednią. Gdy kompletna jest ostatnia z nich, kończy się działanie kreatora i wywoływana jest metoda `performFinish`.

Nowy kreator dodajemy do środowiska za pomocą punktu rozszerzenia. Zostaje on automatycznie wstawiony w ramy standardowego okna dialogowego. Zarządza ono zestawem stron oraz zawiera elementy pozwalające na nawigację.

Kreator występuje przede wszystkim w roli kontrolera. To on decyduje, która strona jest w danym momencie wyświetlana, czy można przejść do następnej itp. Użytkownik zaś posługuje się przyciskami *Next* i *Back* — ich zadaniem jest wykonanie operacji przejścia do etapu następnego lub poprzedniego. Nie zawsze przyciski te są aktywne, zależy to od liczny stron, aktualnego stanu kreatora i kompletności wprowadzonych danych. Własne kreatory budujemy w oparciu o istniejący schemat, dodając rozszerzenie w pliku manifestu, projektując kolejne strony oraz pisząc kod odpowiadający za sterowanie i obsługę zadania. Oto lista czynności, które składają się na proces tworzenia własnego kreatora:

1. Definicja rozszerzenia.
2. Implementacja klasy kreatora.
3. Implementacja strony (stron) kreatora.
4. Dopasowanie wyglądu strony (stron) kreatora.
5. Napisanie kodu, który odpowiada za sposób działania.

Kroki te, a także dodatkowe zagadnienia związane z dostosowaniem kreatora do naszych potrzeb, są tematem kolejnych punktów.

Definicja rozszerzenia

Istnieją trzy rodzaje rozszerzeń, które możemy wykorzystać w celu dodania do środowiska nowego kreatora. Odpowiadają one operacji stworzenia nowego zasobu, procesowi importu oraz eksportu.

- `org.eclipse.ui.newWizards`
- `org.eclipse.ui.importWizards`
- `org.eclipse.ui.exportWizards`

Oto przykład punktu rozszerzenia, w którym dołączamy do środowiska kreator importu:

```
<extension
  id="impWiz"
  point="org.eclipse.ui.importWizards">
  <wizard
    name="QRS File Import"
    class="qrs.tool.wizards.ImportWizard"
    id="qrs.tool.wizards.import">
  </wizard>
</extension>
```

Za pomocą przedstawionych punktów rozszerzenia wstawiamy nowe pozycje w trzech dobrze znanych użytkownikowi Eclipse miejscach:

- `org.eclipse.ui.newWizards` dodaje nową opcję umieszczaną wśród tych, które są widoczne po wybraniu polecenia *File/New/Projects...* i *File/New/Other...*.
- `org.eclipse.ui.importWizards` to nowy kreator dostępny po wskazaniu opcji *File/Import...*.
- `org.eclipse.ui.exportWizards to nowy sposób eksportu danych -- menu` *File/Export...*.

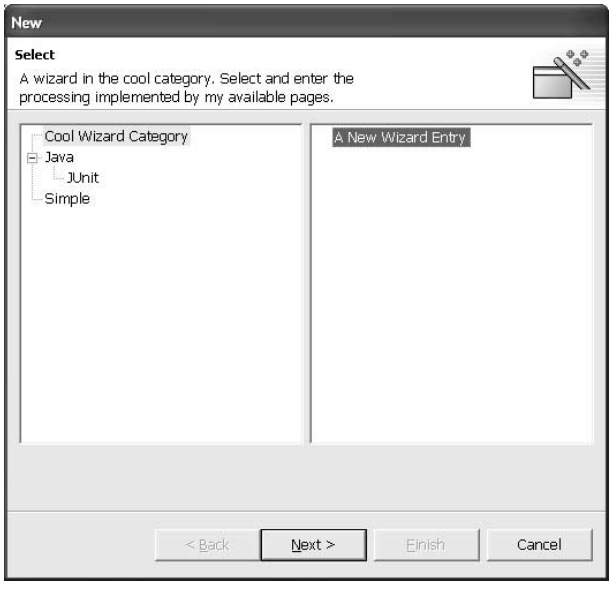
Każde z wymienionych poleceń menu wyświetla okienko dialogowe, w którym możemy wybrać jedną pozycję z dostępnej listy kreatorów. Opcja *Projects...* uwzględnia tylko te punkty rozszerzeń, w definicji których atrybut `project` ma wartość `true`. Również przy okazji definiowania rozszerzenia możemy podać informacje na temat kategorii i w ten sposób odpowiednio zorganizować dostępne opcje.

```
<extension
   id="newWiz"
   name="qrs new wizards"
   point="org.eclipse.ui.newWizards">
  <category
    name="Cool Wizard Category"
    id="qrs.tool.wizards.category">
  </category>
  <wizard
    name="A New Wizard Entry"
    category="qrs.tool.wizards.category"
    class="qrs.tool.wizards.QRSNewWizard"
    id="qrs.tool.wizards.new">
    <description>
```

```
      A wizard in the cool category. Select and enter the
      processing implemented by my available pages.
      </description>
    </wizard>
  </extension>
```

Na rysunku 11.9 prezentujemy okno dialogowe, w którym widać zdefiniowaną nową kategorię oraz przyporządkowany jej kreator (wraz z krótkim opisem). Definicja tego rozszerzenia znajduje się powyżej.

Rysunek 11.9. *Nowy kreator zasobów*



Okno dialogowe z rysunku 11.9, służące do wyboru kreatora, samo też jest obiektem tego typu i posiada przyciski pozwalające na nawigację. Po dokonaniu określonego wyboru i akceptacji wywoływany jest z niego kolejny, zdefiniowany oddzielnie kreator zajmujący się już bezpośrednio tworzeniem nowego projektu bądź zasobu.

Implementacja klasy kreatora

Podstawą konstrukcji kreatora jest klasa dziedzicząca po klasie `Wizard` i implementująca odpowiedni interfejs (zależnie od typu kreatora, którego dotyczy). To ona pełni funkcję kontrolera. Lista interfejsów, które odpowiadają typom punktów rozszerzenia, znajduje się w tabeli 11.8.

Tabela 11.8. *Punkty rozszerzenia służące do wstawienia kreatora*

Punkt rozszerzenia	Interfejs
`org.eclipse.ui.newWizards`	`INewWizard`
`org.eclipse.ui.importWizards`	`IImportWizard`
`org.eclipse.ui.exportWizards`	`IExportWizard`

Narzędzia wchodzące w skład PDE generują dla każdego typu rozszerzenia inny szkielet klasy. Przykład zamieszczony poniżej dotyczy kreatora, który odpowiada za stworzenie nowego zasobu. Klasa kontrolera nosi nazwę `QRSNewWizard`, implementuje interfejs `INewWizard`:

```
public class QRSNewWizard extends Wizard
  implements INewWizard {

  /**
   * Konstruktor
   */
  public QRSNewWizard() {
  }

  /**
   * @see Wizard#performFinish
   */
  public boolean performFinish() {
    return false;
  }

  /**
   * @ see Wizard#init
   */
  public void init(
    IWorkbench workbench,
    IStructuredSelection selection) {
  }
}
```

Naszym zadaniem jest zdefiniowanie i dodanie klas reprezentujących strony kreatora, napisanie kodu, który odpowie za logikę modułu, oraz implementacja metody `performFinish`, która zwraca wartość `true` na znak, że proces zakończył się sukcesem.

Implementacja strony (stron) kreatora

Implementacja strony kreatora nie jest czynnością skomplikowaną i polega na stworzeniu klasy dziedziczącej po `WizardPage`, w której pokrywamy metodę `createControl` odpowiadającą za kształt strony.

```
public void createControl(Composite parent) {
  Composite composite = new Composite(parent, SWT.NONE);
  RowLayout rowLayout = new RowLayout();
  rowLayout.justify = true;
  rowLayout.marginLeft = 5;
  rowLayout.marginRight = 5;
  rowLayout.spacing = 5;
  composite.setLayout(rowLayout);

  Label label = new Label(composite, SWT.NONE);
  label.setText("Select a processing option:");
  Combo combo = new Combo(composite, SWT.NONE);
  combo.setItems(
    new String[] {"Automatic", "Manual", "Hybrid" });
  setControl(composite);
}
```

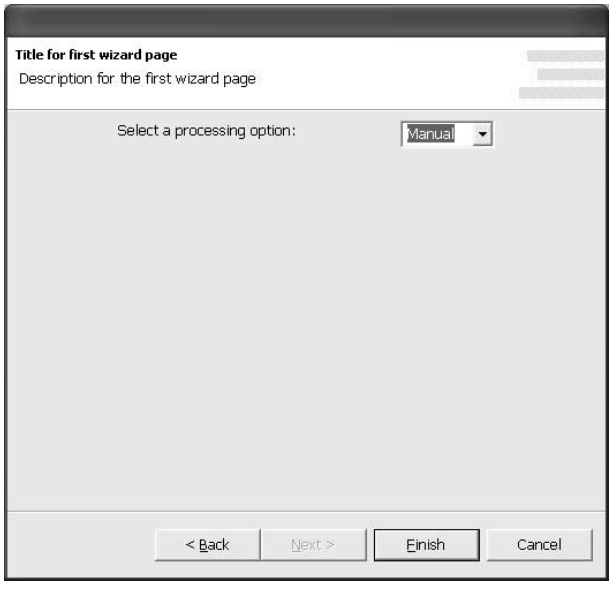
W powyższym kodzie metoda `setControl` została użyta w celu wskazania obiektu będącego kontenerem zawierającym elementy sterujące strony. Jest to podejście nieco inne od tego, które poznaliśmy podczas prezentacji analogicznej metody `createContents` służącej do definiowania zawartości strony parametrów.

Kolejne strony wstawiamy do kreatora, pokrywając jego metodę `addPages`. W następnym przykładzie do kreatora `QRSNewWizard` dodajemy stronę, której odpowiada klasa `SimpleWizardPage`.

```
public void addPages() {
  SimpleWizardPage testPage =
    new SimpleWizardPage("simplePage");
  testPage.setTitle("Title for the first wizard page");
  testPage.setDescription(
    "Description for the first wizard page");
  addPage(testPage);
}
```

Do naszego kreatora możemy wstawiać także strony zdefiniowane wewnątrz Eclipse (piszemy o tym w punkcie „Predefiniowane strony kreatora").

Po zdefiniowaniu klasy będącej kontrolerem kreatora, klasy strony kreatora i połączeniu ich w jedną całość możemy już wypróbować, jak działa nasza konstrukcja. W tym celu w oknie dialogowym służącym do tworzenia nowych zasobów wskazujemy dodaną przed chwilą pozycję i rozpoczynamy działanie kreatora (rysunek 11.10).



Rysunek 11.10. *Podstawowa implementacja kreatora i jego strony*

Na rysunku widać tytuł i opis pierwszej jego strony. Jeśli zdecydujemy, że strona w czasie działania może sygnalizować błędy (np. niepoprawne wartości), komunikat taki pojawi się w miejscu, w którym znajduje się opis. W obu przypadkach korzystamy bowiem z tego samego elementu sterującego.

Dopasowanie wyglądu strony (stron) kreatora

Kolejnym krokiem, który należy wykonać, jest dopasowanie kreatora do zadania, za które odpowiada.

Polega to na określaniu szczegółów dotyczących jego wyglądu. Chodzi tu zarówno o wstawienie tytułu, ikonki czy paska postępu, jak też o obsługę wartości z nim związanych (nie zawsze jest to konieczne).

W przypadku interfejsu użytkownika najlepszym miejscem na wykonanie wspomnianych czynności inicjalizacyjnych jest metoda `init`. Wywołamy w niej metodę `setWindowTitle`, która umieści określony tekst w linii tytułowej okienka. Funkcja `setDefaultPageImageDescriptor` wstawi do kreatora ikonkę widoczną na każdej stronie (za wyjątkiem sytuacji, gdy strona sama odpowiada za swą ikonkę i korzysta w tym celu z metody `setImageDescriptor`).

Przykład implementacji metody `init` prezentujemy poniżej:

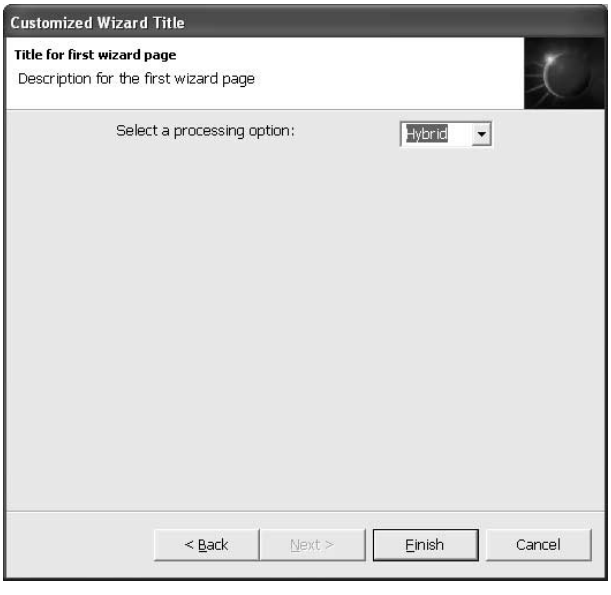
```
public void init(
  IWorkbench workbench, IStructuredSelection selection) {

  setWindowTitle("Customized Wizard Title");
  setDefaultPageImageDescriptor(
    getImageDescriptor("eclipse.jpg"));
  setNeedsProgressMonitor(true);
}
```

A kolejny fragment kodu, czyli metoda `getImageDescriptor`, pokazuje nam, jak pobrać ikonkę z folderu przypisanego pluginowi:

```
private ImageDescriptor getImageDescriptor(
  String relativePath) {

  String iconPath = "icons/";
  try {
    ToolPlugin plugin = ToolPlugin.getDefault();
    URL installURL = plugin.getDescriptor().getInstallURL();
    URL url = new URL(installURL, iconPath + relativePath);
    return ImageDescriptor.createFromURL(url);
  } catch (MalformedURLException e) {
    // To nie powinno się zdarzyć
    return null;
  }
}
```

Kilka powyższych operacji zmienia wygląd naszego kreatora — widzimy to na rysunku 11.11.

Jeszcze raz przypomnimy, że metoda `setImageDescriptor` zdefiniowana w klasie reprezentującej stronę pozwala nam użyć tam ikonki innej niż zdefiniowanej globalnie dla całego kreatora.



Rysunek 11.11. *Dostosowana strona kreatora*

Nawigacja wśród stron kreatora

Kiedy kreator składa się z więcej niż jednej strony, odziedziczony z szablonu kod powoduje, że przycisk *Next* staje się aktywny. Reguły związane ze sprawdzeniem, czy można przejść do kolejnej strony, również są składnikiem klasy bazowej. Za porządek określający stronę poprzednią i następna odpowiada kolejność ich dodawania w czasie definiowania kreatora. Na logikę odpowiadającą za zachowanie kreatora mamy jednak wpływ; wystarczy w tym celu skorzystać z metody `isPageComplete`, która informuje o statusie strony. Dopóki nie uznamy w niej, że strona jest kompletna, nie przejdziemy do następnej. Korzystamy z tej możliwości w przypadkach, gdy poprawnie wprowadzone, pełne dane są warunkiem powodzenia całej operacji.

Oto scenariusz, który odpowiada najbardziej typowej sytuacji, gdy wewnątrz metody `addPages` dodajemy kilka stron:

- Do kolejnych stron przechodzimy, posługując się przyciskiem *Next*.
- Przycisk *Next* jest aktywny, gdy bieżąca strona nie jest ostatnią i jednocześnie ma status strony kompletnej.
- Przycisk *Finish* aktywny staje się dopiero w momencie, gdy wszystkie strony uważane są za kompletne.

Krok w przód

Jeśli istnieje pole, które musi zostać obowiązkowo wypełnione, lub jeśli na stronie znajdują się elementy, wśród których jeden musi zostać wybrany, strona nie powinna być uznana za kompletną przed wprowadzeniem przez użytkownika odpowiednich wartości. Dopiero wtedy metoda `isPageComplete` może zwrócić wartość `true`. Powyższa reguła dotyczy także sprawdzania poprawności danych. Powrót do stron, które zostały już wy-

świetlone, powinien mieć miejsce jedynie wtedy, gdy użytkownik chce zmienić którąś w wprowadzonych wartości. Niedopuszczalna jest natomiast sytuacja, gdy na kolejnych stronach dowiadujemy się, że dane wprowadzone wcześniej są niepoprawne bądź niekompletne. Zasady zdefiniowane w kodzie obsługi kreatora powinny wykluczać taką możliwość. Zawartość poszczególnych stron musi odpowiadać logice procesu, który obsługujemy.

Zatwierdzanie operacji

Metody `isPageComplete` zdefiniowane dla każdej z dodanych do kreatora stron określają, czy wszystkie z nich należy odwiedzić przed zakończeniem całego procesu. Domyślnie metoda `isPageComplete` zwraca wartość `true`, w takim przypadku użytkownik może praktycznie w dowolnym momencie skorzystać z przycisku *Finish* i zakończyć pracę kreatora. Przycisk ten związany jest bezpośrednio z metodą `canFinish`, a ta domyślnie zwraca wartość `true`, gdy wszystkie strony są kompletne.

Gdy przycisk *Finish* jest aktywny, wciskając go, powodujemy wywołanie metody `performFinish`. Jej zadaniem jest wykonanie wszystkich operacji związanych z kończeniem działania kreatora i zwrócenie wartości `true`, gdy proces się powiódł — wtedy dopiero kreator zostanie zamknięty. Wartość `false` powoduje, że jest on dalej aktywny.

Kreator można tak skonfigurować, by wyświetlał i obsługiwał pasek postępu. Może mieć to znaczenie w przypadku wykonywania operacji długotrwałych.

Wybór następnej strony

Gdy kolejność wyświetlanych stron jest wynikiem bieżącej wartości wprowadzonych danych lub decyzji użytkownika, kod odpowiedzialny za nawigację znaleźć się może w metodzie `getNextPage` kontrolera lub poszczególnych stron.

Predefiniowane strony kreatora

Istnieje pewien zbiór gotowych stron związanych z typowymi operacjami, takimi jak tworzenie projektu, pliku czy folderu. Mogą być one częścią projektowanego przez nas kreatora, co więcej, wolno nam je dostosować do własnych potrzeb.

Gdy spojrzymy na klasy będące rozszerzeniem typu `WizardPage`, będziemy w stanie dosyć szybko zorientować się, jakimi możliwościami dysponuje w tej dziedzinie Eclipse i z jakich elementów możemy skorzystać we własnych aplikacjach. Oto kilka klas, które mogą się nam przydać:

- `WizardExternalProjectImportPage`
- `WizardNewFileCreationPage`
- `WizardNewFolderMainPage`
- `WizardNewProjectCreationPage`
- `WizardNewProjectReferencePage`

Oprócz tego możemy rozważyć wykorzystanie kolejnych klas, które zawiadują procesem importu i eksportu:

- `WizardExportPage` (nie zalecana, lepiej używać `WizardExportResourcePage`)
- `WizardExportResourcePage`
- `WizardImportPage` (nie zalecana, lepiej używać `WizardResourceImportPage`)
- `WizardResourceImportPage`
- `WizardSelectionPage`

Gdy zamierzamy rozbudowywać zestaw narzędzi służących do tworzenia aplikacji w języku Java, do dyspozycji mamy strony związane z dodawaniem podstawowych elementów każdego programu pisanego w tym języku:

- `JavaCapabilityConfigurationPage`
- `NewPackageWizardPage`
- `NewClassWizardPage`
- `NewInterfaceWizardPage`

Zaleca się korzystanie z klasy `JavaCapabilityConfigurationPage` zamiast `NewJavaProjectWizardPage`. Jest to opisane w dokumentacji.

Dwie poniższe klasy stanowić mogą natomiast typ bazowy dla tworzonych przez nas stron:

- `NewContainerWizardPage`
- `NewTypeWizardPage`

Wśród wymienionych stron znajdziemy takie, które mogą być bezpośrednio użyte w ramach nowego kreatora oraz takie, które mają charakter klas bazowych i wymagają szczegółowej implementacji (niektóre z nich mogą wystąpić jednocześnie w obu rolach).

Na przykład, jeśli chcemy w ramach nowego kreatora obsłużyć operację dodania nowego pliku, możemy skorzystać z gotowej strony o nazwie `WizardNewFileCreationPage`. Wymagać to będzie zdefiniowania w klasie będącej kontrolerem kreatora pola, które zapamięta wartość parametru `selection` przekazanego metodzie `init` (będzie on później potrzebny przy tworzeniu wspomnianej strony). Oto ilustrujący takie działanie fragment kodu:

```
...
private IStructuredSelection selection;
private WizardNewFileCreationPage myFilePage;
...
public void init(
  IWorkbench workbench, IStructuredSelection selection) {

  this.selection = selection;
}
...
public void addPages() {
```

```
    myFilePage =
      new WizardNewFileCreationPage("filePage", selection);
    addPage(myFilePage);
  }
```

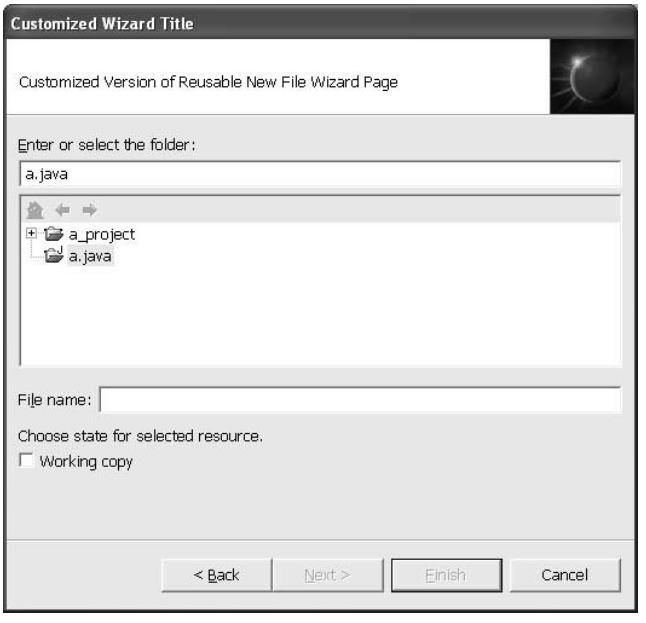
W ten sposób zbudujemy kreator, w którym znajdzie się strona obsługująca operację dodania pliku. Wystarczy w tym momencie zdefiniować odpowiednią metodę `performFinish`, która zostanie wywołana po kliknięciu przycisku *Finish*:

```
public boolean performFinish() {
  myFilePage.createNewFile();
  return true;
}
```

Ale możemy zdecydować się na inne rozwiązanie, w którym na bazie istniejącej klasy stworzymy swój typ, a w nim własny wygląd strony i kod odpowiedzialny za jej działanie. W takim przypadku definiujemy podklasę klasy `WizardNewFileCreationPage` i pokrywamy wybrane metody:

```
public class MyNewFileWizardPage
  extends WizardNewFileCreationPage {
...
  public void createControl(Composite parent) {
    super.createControl(parent);
    Composite composite = (Composite)getControl();
    Label label = new Label(composite, SWT.NONE);
    label.setText("Choose state for selected resource.");
    Button button2 = new Button(composite, SWT.CHECK);
    button2.setText("Working copy");
  }
```

Na rysunku 11.12 widzimy rezultat naszej pracy.



Rysunek 11.12. *Własny interfejs użytkownika strony kreatora*

Wywoływanie kreatora z poziomu kodu źródłowego

Kreatory mogą być wywoływane z dowolnego miejsca tworzonego przez nas modułu. Proces ich definiowania pozostaje ten sam, zmienia się natomiast sposób użycia. Nie korzystamy już z punktów rozszerzenia i pliku manifestu, zamiast tego stworzone przez nas obiekty wstawiamy do okna dialogowego `WizardDialog` i otwieramy je samodzielnie.

Poniżej znajduje się przykład, w którym kreator jest tworzony i wywoływany w kodzie aplikacji, a jego parametrem jest wskazany wcześniej zasób.

```
ToolWizard wizard = new ToolWizard();
wizard.init(getWorkbench(), mySelection);

WizardDialog dialog =
  new WizardDialog(
    getWorkbench().getActiveWorkbenchWindow().getShell(), wizard);
dialog.open();
```

Zapis cech okna dialogowego

Istnieje mechanizm pozwalający na zapis ustawień dotyczących każdego okienka dialogowego. Podczas tworzenia własnych narzędzi często z niego korzystamy. Obsługa cech okna dialogowego przypomina nieco magazyn danych związany ze stroną parametrów. Także tutaj posługujemy się obiektem, do którego dostęp uzyskujemy za pośrednictwem pluginu. Oferuje nam on zbiór funkcji typu `get` i `put`, za pomocą których odczytujemy i zapisujemy właściwości. Za ich utrwalenie odpowiada natomiast sam plugin.

W tabeli 11.9 znajdziemy przykład kodu korzystającego z opisywanego rozwiązania wraz z odpowiadającą mu reprezentacją danych w pliku *dialog_setting.xml*, który jest zapisywany w katalogu odpowiadającym pluginowi.

Tabela 11.9. *Zapis cech okna dialogowego*

Kod źródłowy	Zapisane dane

`IDialogSettings dset =`
`  DialogPlugin.getDefault()`
`  .getDialogSettings();`
 	
`String[] sList = {"a", "b", "c"};`
 	
`IDialogSettings dSection =`
`  dset.addNewSection(`
`  "MyValuesSection");`
 	
`dSection.put("int", 1);`
`dSection.put("double", 2000000000);`
`dSection.put("long", 300000000);`
`dSection.put("float", 400000000);`
`dSection.put("trueFalse", false);`
`dSection.put("string", "myValue";`
`dSection.put("stringArray", sList);`
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<section name="Workbench">`
	`  <section name="MyValuesSection">`
	`  <item key="trueFalse" value="false"/>`
	`  <item key="int" value="1"/>`
	`  <item key="string" value="myValue"/>`
	`  <item key="double" value="2.0E9"/>`
	`  <item key="long" value="300000000"/>`
	`  <item key="float" value="4.0E8"/>`
	`  <list key="stringArray">`
	`  <item value="a"/>`
	`  <item value="b"/>`
	`  <item value="c"/>`
	`  </list>`
	`  </section>`
	`</section>`

W praktyce z mechanizmu tego korzystamy w przypadku zapisu wprowadzonych ostatnio w oknie dialogowym danych, budowania listy wybranych dotychczas pozycji itp. Wybór należy do nas, a zakres możliwości jest szeroki — tym bardziej, że zapisywane dane możemy organizować w sekcje. Każda z nich może np. odpowiadać innemu oknu dialogowemu modułu.

Ćwiczenia

Niniejszy rozdział poświęciliśmy dodawaniu do własnych modułów okienek dialogowych. Omówiliśmy różne warianty takiej operacji. Teraz radzimy przyjrzeć się przykładom znajdującym się na płytce CD, tworzącym projekt *com.ibm.lab.soln.dialogs*. Opisany tam plugin definiuje kilka rozszerzeń, za pomocą których dołączamy stronę właściwości, dwie strony parametrów środowiska oraz dwa kreatory. Pojawia się też kilka opcji menu, które odpowiadają nowym akcjom.

Strona właściwości pozwala nam przypisać do każdego zasobu wskazanego w widoku *Navigator* i *Package Explorer* dwie cechy. Są to wartości typu `String` i `Boolean`.

W oknie dialogowym parametrów systemu pojawiają się dwie strony. Układ pierwszej z nich zaprojektowany jest w całości przez nas, druga korzysta z edytora pól. Strona prosta zawiera dwie opcje. Edytor pól demonstruje użycie kilku wariantów wstawianych do niego obiektów. Dzięki nim w łatwy sposób edytować można wartości różnego typu.

W przykładzie znajdziemy również dwa kreatory. Pierwszy używa gotowego elementu obsługującego tworzenie plików oraz dwóch stron, które pokazują, w jaki sposób realizować sprawdzanie poprawności wprowadzonych danych i wpływać tym samym na proces nawigacji. Drugi kreator zawiera początkowo tylko jedną stronę, pozostałe są dodawane dynamicznie, ich układ zależy od użytkownika. W obu przypadkach kreator dodaje folder oraz plik.

Do środowiska dodajemy również trzy akcje znajdujące się w menu. Za ich pomocą wywołujemy kreator bezpośrednio z wnętrza naszego pluginu. Określamy także, czy dodać do strony właściwości słuchacza zdarzeń, który zmiany wprowadzane w okienku będzie wyświetlał również w konsoli.

Podsumowanie

Po lekturze niniejszego rozdziału umiesz już dołączyć do swego modułu okno dialogowe, zarówno korzystające z dostępnych punktów rozszerzenia, jak też wywoływane z wnętrza pluginu. Przedstawiliśmy proces dodawania nowych stron w oknach dialogowych właściwości oraz parametrów. Opisaliśmy szczegóły definiowania własnych kreatorów, służących do tworzenia nowych zasobów oraz do wykonywania operacji importu i eksportu. Ale nie ograniczyliśmy się wyłącznie do rozwiązań już przewidzianych — pokazaliśmy, jak stworzyć od podstaw własne okienko dialogowe wywoływane w dowolnym fragmencie narzędzia.

Ta wiedza jest niezbędna w sytuacji, gdy zamierzamy rozszerzać środowisko Eclipse. I to zarówno korzystając z przewidzianych w ramach platformy scenariuszy, jak również w sposób bardziej indywidualny.

Bibliografia

Cooper Ryan, „Simplifying Preference Pages with Field Editors”, *http://www.eclipse.org*, 21 sierpnia 2002.