Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jacek Pasternak
Nr albumu: 189430

Wizualny edytor skryptów programu Ant

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2005

Oświadczenie kierującego pracą

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data | Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data | Podpis autora (autorów) pracy

Streszczenie

W pracy opisuję wykonany przeze mnie edytor skryptów programu Ant. Stanowi on rozszerzenie środowiska Eclipse i umożliwia edycję skryptów Anta w sposób graficzny. Został wykonany z wykorzystaniem zestawu bibliotek do tworzenia edytorów graficznych – GEF. W pracy omawiam zarówno Anta jak i GEF-a. Część pracy stanowi też opis metodologii pisania programów funkcyjnych w języku Java, którą zastosowałem do implementacji edytora.

Słowa kluczowe

edytor graficzny, Ant, GEF, Draw2D, MVC, Eclipse, Java, wtyczka, programowanie funkcyjne, modelmanipulator, filtrowanie modeli, modele leniwe

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.1 Programming Techniques
D.1.1 Applicative (Functional) Programming
D.2 Software Engineering
D.2.3 Coding Tools and Techniques – Program editors

Spis treści

Wprowadzenie 5

1. Ant – narzędzie do budowania projektów 7
- 1.1. Wprowadzenie 7
- 1.2. Praca z Antem 7
- 1.3. Budowa skryptu Anta 8
- 1.4. Wsparcie dla Anta na różnych platformach 11
  - 1.4.1. Antidote 11
  - 1.4.2. JBuilder 12
  - 1.4.3. Eclipse 12
  - 1.4.4. Grand: Graphical Representation of Ant Dependencies 12
  - 1.4.5. Vizant 12
  - 1.4.6. Visual Ant Editor 13
- 1.5. Podsumowanie 13

2. GEF – biblioteka do pisania edytorów graficznych 15
- 2.1. Edytory w środowisku Eclipse 15
- 2.2. Wprowadzenie do GEF-a 17
- 2.3. MVC 17
  - 2.3.1. Model 18
  - 2.3.2. Widok 18
  - 2.3.3. Kontroler 19
- 2.4. Proces edycji 20
  - 2.4.1. Narzędzia 20
  - 2.4.2. Żądania 21
  - 2.4.3. Komendy 21
  - 2.4.4. Polityki edycyjne 21
  - 2.4.5. Połączenia 22
- 2.5. Podsumowanie 23

3. Programowanie funkcyjne w języku Java 25
- 3.1. Motywacja 25
- 3.2. Obiekty niezmienialne 26
  - 3.2.1. Obiekty proste 27
  - 3.2.2. Niezmienialne kolekcje 35
  - 3.2.3. Obiekty funkcyjne 36
- 3.3. Podsumowanie 39

4. WESA – Wizualny Edytor Skryptów programu Ant 41
4.1. Główne cechy narzędzia 41
4.2. Architektura 43
4.2.1. Interfejs użytkownika 44
4.2.2. Warstwa pośrednicząca 44
4.2.3. Część funkcyjna 46
4.3. Modele 46
4.4. Budowa prostego skryptu przy pomocy edytora WESA 48

5. Zakończenie 55

A. Zawartość płyty CD 57

Bibliografia 59

Wprowadzenie

Narzędzia do wspomagania budowania projektów informatycznych są dziś powszechnym standardem. Dawno minęły już czasy, gdy aby z kodu źródłowego otrzymać gotowy program, należało uruchomić po kolei i z właściwymi parametrami serię programów i czekać cierpliwie na zbudowanie od początku całego projektu. W dzisiejszych czasach narzędzia takie jak *make* [Make00], czy *Ant* [ANTMan05] automatyzują i optymalizują cały proces budowania czy też instalowania aplikacji, a w dodatku są darmowe. Narzędzia te nie są jednak idealne. Do używania ich wymagana jest znajomość specyficznego formatu plików opisujących budowany projekt, co może stanowić trudność szczególnie dla początkujących programistów, którzy chcą po prostu, aby program się skompilował i działał, bez zbędnego wdawania się w szczegóły składniowe narzędzia do wspomagania budowania.

W niniejszej pracy prezentuję edytor, które napisałem w celu ułatwienia tworzenia skryptów dla programu Ant. Jego głównym wyróżnikiem jest graficzny sposób edycji. Edytor stanowi rozszerzenie środowiska Eclipse. Dzięki graficznej edycji tworzenie skryptów nie wymaga takiej wiedzy dotyczącej ich budowy, jak przy edycji tekstowej. Tworzenie skryptów jest przez to dużo łatwiejsze, szczególnie dla początkujących użytkowników.

W rozdziale 1 przedstawię narzędzie Ant, jego główne cechy i sposób używania. Dokonam również krótkiego przeglądu innych narzędzi i edytorów, które w różnym stopniu ułatwiają pracę z Antem.

W rozdziale 2 zaprezentuję GEF – zestaw bibliotek do tworzenia edytorów graficznych w środowisku Eclipse, który znacząco ułatwia to zadanie. Omówię ogólnie sposób działania edytorów w środowisku Eclipse, a następnie przedstawię metodologię pisania edytorów z wykorzystaniem bibliotek GEF.

W rozdziale 3 omówię metodologię pisania programów funkcyjnych w Javie, którą zastosowałem w części projektu niezależnej od GEF, czyli w logice edytora. Szczególną uwagę poświęcę obiektom trwałym i realizacji trwałości w nietrywialnych przypadkach, jakim są kolekcje.

W rozdziale 4 przedstawię edytor, który powstał w ramach tej pracy, jego architekturę i możliwości. Omówię problemy, jakie napotkałem w czasie jego realizacji oraz sposoby ich rozwiązania. Przedstawię też technikę, która pozwala łączyć kod wykonywany w sposób funkcyjny z kodem, który operuje na obiektach stanowych.

Rozdział 5 stanowi podsumowanie pracy. Opiszę w nim m.in. potencjalne kierunki rozwoju edytora.

Rozdział 1

Ant – narzędzie do budowania projektów

1.1. Wprowadzenie

Jak piszą autorzy, Ant (oryginalnie skrót od *Another Neat Tool* [Neill03]) to oparte na języku Java narzędzie do budowania projektów pozbawione ułomności narzędzia *make* [ANTMan05]. Program powstał w roku 2000 i do dnia dzisiejszego pozostaje narzędziem darmowym. Główną motywacją do stworzenia tego narzędzia były ograniczenia, jakie niosły ze sobą istniejące narzędzia do budowania projektów takie, jak *make*, *gnumake*, *nmake* czy *jam*. Wszystkie one bazują na interpretatorze poleceń: ewaluują zbiór zależności, a następnie wykonują polecenia. Oznacza to, że można w łatwy sposób rozszerzyć te narzędzia używając bądź pisząc programy pod dany system operacyjny. Ale to jest ich słabość. Są związane z konkretnym systemem operacyjnym. Ant, będąc narzędziem w 100% napisanym w języku Java, jest wolny od tych ograniczeń. Rozszerzanie Anta sprowadza się do napisania klas w języku Java. Oczywiście Ant umożliwia wykonywanie poleceń zależnych od platformy, ale nie jest to często stosowana praktyka. Innym problemem jest własny format plików, jaki stosują wymienione narzędzia. Niektóre z nich wymagają np. wpisania znaku [tab] w odpowiednim miejscu. Pominięcie go powoduję błędy podczas budowania, które trudno jest znaleźć. Pliki z opisem sposobu budowania programu Ant (zwane dalej *skryptami Anta*) używają powszechnie znanej składni XML. Wykonanie skryptu Anta polega na sekwencyjnym wykonywaniu zadań (ang. *task*) zawartych w celach (ang. *target*) zgodnie z drzewem zależności pomiędzy celami.

1.2. Praca z Antem

W najprostszym przypadku użycie Anta składa się z następujących kroków:

- napisanie w edytorze tekstowym skryptu, który opisuje cały proces budowania, a w ramach tego:
  - zdefiniowanie zmiennych, które będą używane w dalszej części skryptu,
  - zdefiniowanie celów i zadań w ramach tych celów,
  - zdefiniowanie zależności pomiędzy celami,
  - zdefiniowanie celu domyślnego,

- w katalogu ze skryptem wykonanie polecenia `ant`, które powoduje wykonanie domyślnego celu i wszystkich celów od niego zależnych, o ile jest taka konieczność (np. zmieniły się pliki źródłowe i konieczna jest ponowna kompilacja).

1.3. Budowa skryptu Anta

Żeby wyjaśnić budowę skryptu Anta posłużę się przykładem. Poniższy skrypt prezentuje główne elementy składowe skryptów dla tego narzędzia.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 1 -->
<project name="MiniUSOSWeb" basedir="." default="war"> <!-- 2 -->

    <!-- 3 -->
    <property environment="myenv"/>
    <property name="base.dir" value="WebRoot/WEB-INF"/>
    <property name="classes.dir" value="${base.dir}/classes"/>
    <property name="src.dir" value="${base.dir}/src"/>
    <property name="lib.dir" value="${base.dir}/lib"/>
    <property name="jonas.root" value="${myenv.JONAS_ROOT}"/>
    <property name="jonaslib.dir" value="${jonas.root}/lib"/>
    <property name="beans.dir" value="../MiniUSOSEJB/build/classes"/>
    <property name="build.dir" value="build"/>

    <!-- Project settings -->
    <property name="project.distname" value="MiniUSOSWeb"/>

    <!-- classpath for Struts 1.1 -->
    <path id="compile.classpath">
        <pathelement path="lib/struts.jar"/>
        <pathelement path="${beans.dir}"/>
        <fileset dir="${lib.dir}">
            <include name="*.jar"/>
        </fileset>
        <fileset dir="${jonaslib.dir}">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${classes.dir}"/>
    </path>

    <!-- Check timestamp on files -->
    <!-- 4 -->
    <target name="prepare">
        <!-- 5 -->
        <tstamp/>
        <mkdir dir="${classes.dir}"/>
        <mkdir dir="${build.dir}"/>
    </target>
```

```
    <!-- Remove classes directory for clean build -->
    <!-- 6 -->
    <target name="clean" description="Prepare for clean build">
        <delete dir="${classes.dir}"/>
        <delete dir="${build.dir}"/>
    </target>

    <!-- Copy any resource or configuration files -->
    <target name="resources">
        <copy todir="${classes.dir}" includeEmptyDirs="no">
            <!-- 7 -->
            <fileset dir="${src.dir}">
                <patternset>
                    <include name="**/*.conf"/>
                    <include name="**/*.properties"/>
                    <include name="**/*.xml"/>
                </patternset>
            </fileset>
        </copy>
    </target>

    <!-- Normal build of application -->
    <!-- 8 -->
    <target name="compile" depends="prepare,resources">
        <javac srcdir="${src.dir}" destdir="${classes.dir}">
            <classpath refid="compile.classpath"/>
        </javac>
    </target>

    <!-- Build entire project -->
    <!-- 9 -->
    <target name="build" depends="prepare,compile"/>
    <!-- Create binary distribution -->
    <target name="war" depends="build">
        <war basedir="${base.dir}/.."
         warfile="${build.dir}/${project.distname}.war"
         webxml="${base.dir}/web.xml">
            <exclude name="WEB-INF/${build.dir}/**"/>
            <exclude name="WEB-INF/src/**"/>
            <exclude name="WEB-INF/web.xml"/>
        </war>
    </target>
</project>
```

Zwróćmy uwagę na najciekawsze jego elementy, które stanowią przegląd możliwości Anta.

1. `<?xml version="1.0"?>` ponieważ skrypty Anta są dokumentami XML-owymi, więc zgodnie ze specyfikacją na początku skryptu powinna się znajdować preambuła XML. Jak pokazuje praktyka nie jest ona jednak konieczna do prawidłowego działania skryptu, podanie jej jednak należy do dobrej praktyki programistycznej.

2. `<project ...>` głównym elementem skryptu Anta jest element `project`. Zawiera on 3 atrybuty:

   - `name` – nazwa projektu;
   - `basedir` – katalog, względem którego będą później definiowane inne ścieżki. Domyślnie jest to katalog, w którym znajduje się skrypt;
   - `default` – nazwa domyślnego celu. Atrybut wymagany.

3. W każdym większym skrypcie występują definicje pewnych zmiennych, które są używane w dalszej części skryptu. Definicje te mogą mieć różną postać.

   - `<property name="base.dir" value="WebRoot/WEB-INF" />` – przykład najprostszej postaci. Zawiera nazwę zmiennej i jej wartość (w tym przypadku względną ścieżkę, ale może to być dowolny napis[1]).
   - `<property name="classes.dir" value="${base.dir}/classes" />` – przykład odwołania się w definicji do innych wartości. Odwołanie do wartości zmiennej odbywa się poprzez wpisanie jej nazwy pomiędzy znaki `${` a `}` i może wystąpić w wartości dowolnego atrybutu. Wartości zmiennych występujących jako dzieci elementu `project` są wyliczane przed wykonaniem jakichkolwiek zadań. Możliwe jest warunkowe ustawianie wartości zmiennych za pomocą zadania `condition`.
   - `<property environment="myenv" />` – przykład definicji prefiksu używanego do odwołań do zmiennych środowiskowych. Po takiej definicji chcąc użyć wartości takiej zmiennej wystarczy poprzedzić ją zdefiniowanym prefiksem. Wygląda to następująco: `<property name="jonas.root" value="${myenv.JONAS_ROOT}" />`

4. Zasadniczym elementem skryptu Anta są definicje celów. W najprostszym przypadku definicja celu wygląda następująco: `<target name="prepare">` Oprócz obowiązkowego atrybutu name zawierającego nazwę celu mogą występować następujące atrybuty:

   - `depends` – oddzielona przecinkami lista celów, od których ten cel zależy. Cele te muszą zostać wykonane zanim zostanie wykonany bieżący cel;
   - `if` – zawiera nazwę atrybutu, który musi być ustawiony, aby cel został wykonany;
   - `unless` – zawiera nazwę atrybutu, który nie może być ustawiony, aby cel został wykonany;
   - `description` – krótki opis celu.

5. Wewnątrz celu zdefiniowana jest seria zadań, które są wykonywane sekwencyjnie. Mogą to być bardzo proste zadania jak `<tstamp />` powodujące ustawienie zmiennych `DSTAMP`, `TSTAMP` i `TODAY` na bieżącą datę, którą można wykorzystać np. w nazwach

[1] Napis musi być zgodny z wymaganiami XML-a co do wartości atrybutów. Nie może np. zawierać znaku `<`.

plików. Zadanie można parametryzować za pomocą atrybutów i elementów zagnieżdżonych. Przykładowo `<mkdir dir="${classes.dir}" />` powoduje stworzenie katalogu o nazwie zdefiniowanej za pomocą zmiennej `classes.dir` (w tym przypadku rozwija się do `WebRoot/WEB-INF/classes`).

6. Cele niekoniecznie muszą być ze sobą powiązane. O ile cel `prepare` jest potrzebny do wykonania innych celów, to cel `clean` nie jest wykorzystywany przez żaden inny cel i aby go wykonać, trzeba podać go jawnie w wierszu poleceń: `ant clean`.

7. Niektóre zadania związane z operacjami na plikach pozwalają zdefiniować zbiory plików, których dotyczą za pomocą zagnieżdżonych elementów fileset i dirset. Wewnątrz takich elementów mogą dodatkowo znajdować się podelementy, które rozszerzają, bądź zawężają bazowy zbiór. W opisie zbiorów oprócz zwykłych znaków można używać znaków specjalnych takich jak:

   - `?` – zastępuje dokładnie 1 znak,
   - `*` – zastępuje dowolną liczbę znaków,
   - `**` – zastępuje dowolną liczbę znaków wraz z podkatalogami.

   Dopasowanie jest robione na poziomie 1 katalogu. Aby uniezależnić się od lokalnej struktury katalogów należy użyć `**`. Przykładowe zbiory:

   - `a/b/c/**` – wszystkie pliki w podkatalogu `a/b/c` (np. `a/b/c/a.txt`, `a/b/c/r/u.txt`),
   - `**/abc/**/*.txt` – wszystkie pliki z rozszerzeniem `txt`, które w swej ścieżce do korzenia mają `abc`.

8. Niektóre cele mogą zależeć od więcej niż 1 innego celu

9. Inne cele mogą w ogóle nie mieć zadań do wykonania w swoim wnętrzu i odgrywać rolę elementu porządkującego wykonanie skryptu.

1.4. Wsparcie dla Anta na różnych platformach

Ponieważ Ant stał się popularnym narzędziem, więc nie dziwi fakt, że jest wspierany przez wiele dedykowanych narzędzi, a także większych środowisk zintegrowanych. W większości przypadków wsparcie polega na kolorowaniu składni skryptów, czasami także podpowiadaniu poszczególnych elementów, a także wykonywaniu skryptów i podglądzie ich wyników. Istnieją też narzędzia do wizualizacji struktury skryptów, a także takie, które z założenia mają służyć do wizualnego tworzenia skryptów.

1.4.1. Antidote

Antidote to narzędzie, które powstaje wraz z Antem i jest rozwijane od roku 2000 jako jego podprojekt [Antidote05]. Początkowo jego założeniem była łatwość integracji z istniejącymi środowiskami IDE (ang. *Integrated Development Environment*). Ponieważ prace nad narzędziem postępowały wolno, do środowiska IDE zostało niezależnie dodane wsparcie dla Anta i projekt Antidote stracił na znaczeniu. Obecnie jest rozwijany jako niezależna aplikacja. Wciąż jest w fazie projektowej i nie jest powszechnie dostępny w wersji gotowej do uruchomienia.

1.4.2. JBuilder

JBuilder [JBuilder05] to rozbudowane środowisko IDE do tworzenia aplikacji w języku Java. Wersja Professional i Enterprise zawiera wbudowane wsparcie dla Anta. Obejmuje ono podświetlanie składni, podgląd struktury skryptu bez uwzględnienia specyfiki skryptów (brak wyróżnienia dla celów i celu domyślnego) oraz wykonywanie skryptów. Jest to narzędzie komercyjne. Darmowa wersja Personal nie zawiera wsparcia dla Anta, ale istnieje darmowe narzędzie AntRunner [AntRunner04], które dodaje obsługę Anta do JBuildera.

1.4.3. Eclipse

Eclipse to środowisko IDE ogólnego przeznaczenia [Eclipse05]. Nie jest związane z jakimś szczególnym językiem programowania, oferuje jednak bardzo rozbudowane wsparcie dla języka Java. Można je rozszerzać poprzez mechanizm wtyczek (ang. *plug-in*). Oferuje duże wsparcie do pracy ze skryptami Anta. Obejmuje ono:

- kolorowanie składni,
- podpowiadanie nazw atrybutów,
- definiowanie zmiennych,
- uruchamianie wybranych celów,
- podgląd struktury skryptu z uwzględnieniem specyfiki skryptu,
- własny moduł wykonywania skryptów w postaci wtyczki (istnieje jednak możliwość skorzystania z zewnętrznego Anta).

Mimo, że rozbudowane, wsparcie dla Anta w Eclipse ma pewne luki. Otóż brakuje podglądu zależności pomiędzy celami. Nie ma też możliwości rozszerzenia obsługiwanej składni o dodatkowe elementy zdefiniowane przez użytkownika.

1.4.4. Grand: Graphical Representation of Ant Dependencies

Grand [Grand05] to narzędzie służące do wizualizacji zależności skryptów Anta. Może być uruchamiane jako osobna aplikacja, bądź też zadanie Anta, które tworzy plik o rozszerzeniu `dot` wymagający dalszej obróbki za pomocą narzędzia GraphViz [GraphViz05]. Umożliwia znajdowanie zależności zarówno statycznych (wynikających z atrybutu `depends` celu), jak i dynamicznych, wynikających z niektórych zadań takich jak `ant` czy `antcall` i przedstawianie ich za pomocą grafu zależności. Korzysta przy tym z API (ang. *Application Program Interface*) Anta.

1.4.5. Vizant

Vizant [Vizant02] to kolejne narzędzie do wizualizacji zależności w skryptach. Działa jako zadanie Anta. Na wejściu dostaje jako parametr nazwę pliku ze skryptem, który na zwizualizować, a na wyjściu generuje plik `dot`. Aby otrzymać graf zależności trzeba użyć programu GraphViz. Wizualizowane są tylko statyczne zależności.

1.4.6. Visual Ant Editor

Narzędzie, które miało umożliwić edycję plików Anta w sposób graficzny [VAE05]. Wciąż pozostaje we wczesnej fazie rozwojowej. Działa jako wtyczka do środowiska Eclipse. Wersja, która jest dostępna nie umożliwia graficznej edycji skryptów ani nawet graficznego podglądu. Klasa za to odpowiedzialna jest oznaczona jako „TODO Implement the GraphicalView class". Edytor jak na razie nie spełnia swojego zadania.

1.5. Podsumowanie

Ant jest powszechnie stosowanym narzędziem. Oferuje wiele przydatnych funkcji, takich jak XML-owy format skryptów, definiowanie własnych zadań, bogata biblioteka zdefiniowanych zadań, czy też łatwość definiowania własnych. W Internecie można znaleźć wiele materiałów i kursów dotyczących Anta [ANTMan05, Nazar05, Mills02]. Wsparcie dla niego jest oferowane przez wiele narzędzi, zarówno dedykowanych, jak i środowisk IDE. Istnieją programy do wizualizacji skryptów, brak jest jednak działających narzędzi, które umożliwiłyby edytowanie skryptów w sposób wizualny. Obecny model pracy polega na pisaniu skryptu przy pomocy narzędzia z podświetlaniem składni (bardzo dobrze w tej roli spisuje się Eclipse) i sporadycznym kontrolowaniu powstałych zależności za pomocą narzędzia do wizualizacji.

W dalszej części pracy opisze narzędzie WESA (Wizualny Edytor Skryptów programu Ant), którego zadaniem jest wypełnienie luki w zbiorze narzędzi ułatwiających pracę z Antem. Jak sama nazwa wskazuje, umożliwia ono edycję w sposób wizualny. Zostało napisane z użyciem zestawu bibliotek do tworzenia edytorów graficznych GEF i przeznaczone do pracy w środowisku Eclipse.

W następnym rozdziale zaprezentuję ten ciekawy zestaw bibliotek, dzięki któremu możliwe jest łatwe i szybkie stworzenie dowolnego edytora graficznego.

Rozdział 2

GEF – biblioteka do pisania edytorów graficznych

2.1. Edytory w środowisku Eclipse

Eclipse jest platformą ogólnego przeznaczenia, która może być rozszerzana za pomocą mechanizmu wtyczek [Eclipse05]. Wtyczki mogą pełnić najrozmaitsze funkcje. Najczęściej spotykane to wtyczki będące różnego rodzaju edytorami. Proces tworzenia wtyczki polega na przygotowaniu pliku deskryptora opisującego sposób łączenia wtyczki ze środowiskiem oraz klas implementujących odpowiednie interfejsy [PluginGuide05]. Wtyczki komunikują się ze środowiskiem za pomocą tzw. punktów rozszerzeń (ang. *extension point*). Dla wtyczki realizującej funkcje edytora najważniejszym punktem rozszerzeń, które musi zaimplementować, jest org.eclipse.ui.editors. Informuje ono środowisko m.in. o tym, jakie rozszerzenia plików mają być przypisane do danego edytora.

Plik opisu dla edytora WESA wygląda następująco:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension
         id="Wesa Editor"
         name="Wesa Editor"
         point="org.eclipse.ui.editors">
      <editor
            filenames="build.xml, buildfile.xml"
            icon="icons/ant_buildfile.gif"
            class="wesa.WesaMainEditor"
            default="false"
            name="Wesa Editor"
            contributorClass="wesa.WesaActions"
            id="wesa"/>
   </extension>
</plugin>
```

Każda implementacja punktu rozszerzeń ma swoją unikatową nazwę (w tym przypadku „Wesa Editor”). Z tego opisu możemy się dowiedzieć, że wtyczka implementuje tylko jeden punkt rozszerzeń, a mianowicie `org.eclipse.ui.editors`, edytor obsługuje pliki o nazwach `build.xml` i `buildfile.xml`, nie jest edytorem domyślnym, będzie widziany jako „Wesa Editor”, klasa edytora to `wesa.WesaMainEditor`, a klasa, która przekierowuje akcje ze środowiska takie jak undo, czy delete, to `wesa.WesaActions`. Oczywiście klasy te nie mogą być całkowicie dowolne. Muszą implementować interfejsy wymagane przez dany punkt rozszerzeń. W tym przypadku jest to `org.eclipse.ui.IEditorPart` i `org.eclipse.ui.IEditorActionBarContributor`. Ponieważ pisanie edytorów jest zjawiskiem powszechnym, została standardowo dostarczona abstrakcyjna klasa, implementująca interfejs `org.eclipse.ui.IEditorPart`, a mianowicie `org.eclipse.ui.part.EditorPart`. W celu napisania działającego edytora wystarczy zaimplementować wszystkie abstrakcyjne metody, a w szczególności istotna jest `createPartControl(Composite parent)`. Dzięki niej można stworzyć edytor o dowolnym wyglądzie i zachowaniu. Z edytorami są również związane różnego rodzaju widoki pokazujące użytkownikowi różne aspekty edytowanego dokumentu, bądź też pozwalające te aspekty zmieniać. Dodatkowe widoki można definiować, podobnie jak edytory, za pomocą mechanizmu wtyczek.

Tworzenie od początku całego wyglądu edytora jest żmudnym zadaniem mimo, iż biblioteka SWT (ang. *Standard Widget Toolkit* – biblioteka kontrolek używana w Eclipse) zawiera spory zestaw gotowych kontrolek. Nie wystarcza on jednak do bezpośredniego użycia w edytorze. Dlatego wraz z Eclipse dostarczona jest abstrakcyjna klasa `org.eclipse.ui.texteditor.AbstarctTextEditor`, będąca podstawową implementacją edytora tekstowego – najpopularniejszej odmiany edytora. Oferuje ona wiele gotowych do użycia mechanizmów, takich jak przewijanie tekstu, zaznaczanie, znaczniki, menu kontekstowe i wiele innych funkcjonalności przydatnych w edytorze tekstowym. Zadaniem programisty jest po prostu przedefiniowanie metod, których działanie uzna za nieadekwatne do jego potrzeb. Niestety nie ma standardowych klas do tworzenia edytorów graficznych. Jest to zrozumiałe, gdyż każdy edytor graficzny ma swoją specyfikę i w przeciwieństwie do tekstowego trudno jest powiedzieć jak konkretnie wygląda edytor graficzny.

Edytory graficzne wprawdzie trudno ogólnie opisać w kategoriach wyglądu, jednak można wyróżnić wymagania funkcjonalne, które mają spełniać. Należą do nich:

- interakcja z użytkownikiem odbywa się głównie za pomocą myszki bądź innego urządzenia wskazującego (w przypadku edytorów tekstowych za pomocą klawiatury),
- użytkownik tworzy obiekty, definiuje relacje pomiędzy nimi, zmienia ich położenie bądź też usuwa, używając do tego celu głównie mechanizmu „przeciągnij i upuść” (ang. *drag and drop*),
- użytkownik definiuje różne dodatkowe własności obiektów,
- do interakcji z obiektami użytkownik wykorzystuje narzędzia dostępne z palety, dzięki czemu rzadko zachodzi konieczność dodatkowego używania klawiatury.

Wymagania te dostrzegli twórcy GEF-a (ang. *Graphical Editing Framework*) [GEFRed04]. Jak sama nazwa wskazuje jest to zestaw bibliotek specjalnie stworzony do budowy edytorów graficznych w środowisku Eclipse, w przeciwieństwie do używanego powszechnie w środowisku Eclipse SWT, czy JFace, będących bibliotekami komponentów ogólnego przeznaczenia. W dalszej części rozdziału przedstawię filozofię, jaka stoi za tym zestawem bibliotek oraz metodologię tworzenia graficznych edytorów z ich użyciem.

2.2. Wprowadzenie do GEF-a

Wszystkie edytory mają pewne wspólne elementy. Są to:

- model danych, które podlegają edycji,
- widok złożony z obiektów, które definiują pewien graficzny układ na ekranie,
- możliwość modyfikacji przez użytkownika widoku za pomocą urządzeń takich jak mysz czy klawiatura,
- połączenie pomiędzy widokiem a modelem określające co się dzieje, gdy zmianie ulegnie model bądź też widok.

Edytor graficzny umożliwia modyfikowanie modelu w sposób graficzny. Ogólnie w edytorze połączenie pomiędzy modelem a widokiem może być dowolne. W szczególności jednemu stanowi modelu może odpowiadać wiele stanów widoku (gdy np. użytkownik zmienia położenie jakiś obiektów) i jednemu widokowi może odpowiadać wiele stanów modelu (gdy pewne aspekty modelu nie są wizualizowane).

O ile fakt, że wiele stanów modelu odpowiada jednemu widokowi nie stanowi większego problemu, gdyż mając model możemy jednoznacznie określić jaki jest jego widok, to wiele widoków skojarzonych z jednym stanem modelu powoduje niejednoznaczność. Wynika to z tego, że w takim przypadku część modelu tak na prawdę przechowywana jest w postaci niejawnej w widoku. Rozwiązaniem problemu jest trzymanie w modelu zarówno danych, które bezpośrednio podlegają edycji zwanych *modelem biznesowym*, jak i modelu danych, które wpływają na wizualne przedstawianie modelu biznesowego (np. pozycja obiektu na ekranie, którą użytkownik może definiować) zwanych *modelem widoku*. Otrzymujemy wtedy klarowny model interakcji z użytkownikiem, uproszczenie architektury aplikacji. Wystarczy bowiem interpretować wszystkie akcje użytkownika w interfejsie graficznym jako zmiany modelu (zawierającego w sobie model biznesowy i widok), natomiast zmiany modelu powodują jednoznacznie określone zmiany w widoku. Aby to osiągnąć należy zaimplementować:

- mechanizm, który automatycznie zbuduje widok i go wyświetli dla zadanego modelu,
- mechanizm, który odświeża widok, gdy zmianie uległ model,
- mechanizm, który przechwytuje akcje użytkownika w interfejsie graficznym i zamienia je na zmiany stanu modelu.

GEF ułatwia wykonanie każdego z tych kroków. Przyjrzyjmy się teraz jak to dokładnie jest realizowane.

2.3. MVC

U podstaw GEF-a leży dobrze znana architektura MVC (ang. *Model-View-Controler*) [Wiki04, Deacon00]. Stanowi ona wzorzec projektowy złożony z mniejszych wzorców, głównie: *Composite*, *Observer*, *Startegy*, a także *Decorator*, *Adapter* czy *Factory Method* [GHJV98, Metsker02]. Zakłada ona, że wszelkie zmiany w widoku i modelu są dokonywane za pośrednictwem kontrolera, który stanowi element pośredniczący. Dalszy opis dotyczy implementacji MVC w GEF-ie.

2.3.1. Model

GEF nie nakłada żadnych ograniczeń na model. Modelem może więc być dowolny obiekt Javy. Niemniej dobry model powinien spełniać pewne założenia, które pozwolą na łatwe korzystanie z niego:

- Musi zawierać wszelkie informacje, jakie użytkownik może zmieniać za pomocą edytora i które potem są przechowywane w postaci trwałej. Powinny się tam znaleźć również informacje na temat graficznych aspektów modelu biznesowego, które nie będą później utrwalane, ale które użytkownik może modyfikować (czyli model widoku).
- Nie może nic wiedzieć na temat widoku i innych części edytora. Stanowi jedynie pojemnik na dane.
- Musi posiadać jakiś mechanizm powiadamiania o swoich zmianach.

2.3.2. Widok

Widok to zbiór obiektów graficznych, które składają się na interfejs użytkownika. Powinien spełniać następujące wymagania:

- nie może zawierać istotnych danych, które nie są przechowywane w modelu,
- nie może nic wiedzieć na temat modelu i nie może uczestniczyć w żadnej części logiki edytora. W szczególności nie może zawierać żadnych referencji do innych części.

Mimo że GEF może jako widok bezpośrednio wykorzystywać SWT, to specjalnie dla niego został stworzony system graficznych komponentów Draw2D. W GEF zaimplementowano też wsparcie dla widoku w postaci drzewa opartego na komponencie Tree z SWT. Ponieważ w dalszych rozważaniach będę odwoływał się do Draw2D, przedstawię poniżej krótki opis tego zestawu komponentów.

Draw2D

Draw2D to system komponentów o filozofii działania zbliżonej do Swinga standardowo dostarczonego z Javą. W odróżnieniu jednak od Swinga, który sam jest zbudowany w architekturze MVC, Draw2D jest czysto graficzny. Za elementami graficznymi nie stoi żaden model, którym można by manipulować (jak to jest np. w przypadku JTable ze Swinga). Służy jedynie do wyświetlania elementów graficznych na ekranie, a nie do przechowywania czy manipulowania danymi.

W Draw2D obraz jest tworzony za pomocą tzw. figur, które tworzą drzewo. Każda figura ma jednego ojca i zero lub więcej dzieci. Istnieje tylko jedna figura, która nie ma ojca i jest to figura korzenia, która wchodzi w skład systemu komponentów. Figura może być dowolnym układem punktów na ekranie. Wystarczy jedynie, że klasa ją realizująca implementuje interfejs `org.eclipse.draw2d.IFigure`. Z Draw2D jest dostarczonych wiele standardowych figur, gotowych do bezpośredniego użycia.

Każda z figur potrafi się oczywiście narysować. Każda figura najpierw rysuje siebie, potem nakazuje wyrysować się swoim dzieciom, a na końcu rysuje swoją ramkę. Tak więc aby wyrysować całe drzewo figur wystarczy nakazać wyrysować się figurze w korzeniu. Każda figura ma określony prostokątny obszar, w którym musi się wyrysować, zwany dalej *prostokątem ograniczającym*, poza którym zawartość figury jest obcinana. Obszar ten może ulegać

zmianom. Dzięki temu można zapewnić, że dzieci danej figury nie zostaną narysowane poza jej obszarem. Przesunięcie prostokąta ograniczającego ojca automatycznie przesuwa dzieci.

Z każdą figurą można związać menedżera wyglądu (ang. *layout manager*). Jest to obiekt, którego zadaniem jest zarządzanie wzajemnym położeniem i rozmiarami dzieci danej figury. Informuje również figurę-ojca o preferowanym rozmiarze danej figury.

W Draw2D zaimplementowano algorytm testu trafień (ang. *hit-test*), którego zadaniem jest znalezienie najbliżej obserwatora figury, która znajduje się w danym punkcie (im niżej w hierarchii tym bliżej). Odbywa się to z uwzględnieniem obszaru figury i metody `containsPoint()` interfejsu Ifigure.

Z figurami można wiązać obiekty nasłuchujące pewnych zdarzeń. Są one kierowane do najbliższej figury, która potrafi je rozpoznać (tzn. został w niej zarejestrowany obiekt, który obsługuje dane zdarzenie). Mechanizm ten nie powinien być jednak nadużywany do definiowania jakiejś logiki. Może być wykorzystany np. do podświetlenia figury, gdy kursor znajduje się w jej obrębie (czyli akcja mająca związek tylko i wyłącznie z wyświetlaniem).

Draw2D oferuje wsparcie dla definiowania połączeń pomiędzy figurami. Nie jest to zadanie tak oczywiste, jakby się wydawało, ponieważ połączenia mogą występować pomiędzy różnymi poziomami hierarchii. Niemniej definiowanie połączeń należy do częstych zadań stawianych przed edytorami graficznymi, więc wsparcie dla nich po stronie systemu komponentów wydaje się nieodzowne.

Draw2D wspiera warstwy. Warstwa to figura, która jest przezroczysta dla testu trafień. Warstwy są umieszczane jedna na drugiej w porządku stosowym z użyciem specjalnego menedżera wyglądu. Istnieje specjalna warstwa służąca do przechowywania połączeń.

Z każdą figurą można związać kursor i tekst podpowiedzi, który się ukazuje, gdy kursor na dłużej zatrzyma się nad jej obszarem.

2.3.3. Kontroler

W GEF funkcję kontrolera pełnią implementacje `interfejsu org.eclipse.gef.EditPart`, zwane dalej elementami edycyjnymi. Ten interfejs nie jest jednak przeznaczony do bezpośredniego implementowania. Zamiast tego należy rozszerzać za pomocą mechanizmu dziedziczenia klasę `org.eclipse.gef.editparts.AbstractEditPart` i jest klasy pochodne. Z elementem edycyjnym związana jest klasa implementująca interfejs `org.eclipse.gef.EditPartViewer`, której zadaniem jest zarządzanie widokiem i cyklem życia elementu edycyjnego[1]. Przechowuje ona m.in. mapę z obiektów modelu na elementy edycyjne oraz mapę z figur na elementy edycyjne (dla `GraphicalViewer`). Jest też odpowiedzialna za dokonywanie testu trafień, dzięki któremu można znaleźć element edycyjny spełniający pewne narzucone założenia.

Kontroler jest odpowiedzialny za łączenie widoku z modelem. Przechowuje odniesienia zarówno do modelu, jak i widoku. Rejestruje się w modelu jako obserwator (wykorzystany wzorzec *Observer*), dzięki czemu wie kiedy występują jego zmiany. Potrafi też stosownie do nich zmienić widok. Uczestniczy również w procesie edycji i zmiany modelu.

Każdy element edycyjny może posiadać dowolną liczbę dzieci. Każde dziecko to pewna część modelu. Z elementem edycyjnym związana jest mapa odwzorowująca dzieci, wynikające z modelu, na elementy edycyjne, które potrafią ja obsługiwać. Przy odświeżaniu dzieci danego elementu edycyjnego wywoływana jest metoda `createChild` dla każdego dziecka umieszczonego w mapie. Domyślnie element edycyjny jest pobierany z fabryki przypisanej do klasy zarządzającej widokiem. Figury, które są związane z elementami edycyjnymi dzieci, są dodawane jako dzieci do figury przekazywanej przez metodę `getContentPane` danego elementu

[1] Dla systemu figur Draw2D, klasą zarządzającą widokiem jest `org.eclipse.gef.ui.parts.GraphicalViewerImpl`, która jest wspólna dla wszystkich elementów edycyjnych.

edycyjnego. Domyślnie jest to ta sama figura, która stanowi widok tego elementu edycyjnego, lecz często spotykaną praktyką jest przedefiniowanie jej tak, aby dzieci były dodawane do pewnego jej fragmentu.

Elementy edycyjne, podobnie jak figury, tworzą drzewo. Na jeden element edycyjny może przypadać więcej niż jeden element drzewa figur (i najczęściej tak właśnie jest). GEF nie zna szczegółów dotyczących figur. Rozumie tylko elementy edycyjne związane z tymi figurami. Jeśli więc element modelu ma być edytowalny, to musi być związany z elementem edycyjnym. Im mniej rodzajów elementów edycyjnych istnieje, tym stają się one bardziej skomplikowane, a wraz z nimi ich figury.

2.4. Proces edycji

Centralną rolę w procesie edycji odgrywa tzw. obszar edycji (ang. *edit domain*). Łączy on wszystkie części GEF-a w jedna spójną całość uzupełniając ją o niezbędne do edycji elementy, takie jak stos komend i paleta z narzędziami. Zadaniem programisty jest zaimplementowanie komend, polityk edycyjnych oraz stworzenie opisu palety. Cały proces edycji odbywa się zgodnie ze schematem:

- Użytkownik przy użyciu urządzeń takich jak klawiatura czy mysz korzysta z graficznego interfejsu użytkownika – GUI (ang. *Graphical User Interface*).
- Akcje użytkownika skutkują zdarzeniami, które są przekazywane przez instancję interfejsu `EditPartViewer` do aktywnego narzędzia.
- Aktywne narzędzie interpretuje serię zdarzeń i generuje żądania (podklasy klasy `org.eclipse.gef.Request`). Żądania to obiekty używane przez GEF do opisu operacji, jakie należy wykonać na modelu bez mówienia jak mają być zrealizowane.
- Aktywne narzędzie wysyła żądania do elementów edycyjnych, aby te stworzyły komendy realizujące żądanie oraz prosi elementy edycyjne o pokazanie reprezentacji graficznej odbywającego się procesu (np. szarego prostokątu podczas przeciągania elementów).
- Po otrzymaniu komendy aktywne narzędzie może ją wykonać za pośrednictwem stosu komend (klasa `org.eclipse.gef.commands.CommandStack`). Stanowi on implementację stosu undo – redo.
- Stos komend wykonuje operację, która powoduje zmiany w modelu. Zmiany te są przechwytywane przez elementy edycyjne i odwzorowywane w odpowiednie zmiany widoku.

2.4.1. Narzędzia

Narzędzia to obiekty zdolne do odbierania zdarzeń generowanych przez użytkownika. Stanowią automaty skończone, które śledzą sekwencje zdarzeń, jakie otrzymują i w zależności od stanu, w jakim się znajdują wysyłają żądania do właściwych elementów edycyjnych. GEF dostarcza standardowe implementacje wielu powszechnie używanych narzędzi, które służą do wskazywania, zaznaczania, tworzenia obiektów, czy też połączeń pomiędzy nimi. Wraz z narzędziami dostępne są też standardowe implementacje wielu żądań. Przez większość czasu aktywne narzędzie utrzymuje żądanie, które zostałoby wysłane, gdyby użytkownik zakończył właśnie interakcję. Kiedy żądanie jest aktualizowane, bądź też element edycyjny (bądź kilka elementów) się zmienił, aktywne narzędzie wysyła do elementów edycyjnych komunikaty, aby

te pokazały reprezentację graficzną odbywającego się procesu edycji, bądź też ją ukryły[2]. Umożliwia to elementom edycyjnym pokazanie użytkownikowi podglądu tego co by się stało, gdyby zakończył interakcję właśnie w tym momencie.

2.4.2. Żądania

Żądania zawierają informacje o modyfikacjach, jakie użytkownik chce przeprowadzić na modelu. Informacje te są opisane w terminologii elementów edycyjnych, ponieważ narzędzia nie mają informacji na temat modelu. Każde żądanie ma związany z nim atrybut określający jego znaczenie, tzn. co powinno być zrobione (jakiego rodzaju komenda powinna zostać stworzona), gdy element edycyjny je otrzyma. Sama informacja o klasie jaka jest użyta do reprezentowania danego żądania nie jest wystarczająca, ponieważ obiekty żądania tej samej klasy (przykładowo `org.eclipse.gef.requests.ChangeBoundsRequest`) mogą nieść informacje o różnym znaczeniu (w przypadku `ChangeBoundsRequest` informacje o obiektach, które należy skopiować w inne miejsce bądź też obiektach, których rozmiar należy zmienić). Typ żądania jest też używany do określenia polityk edycyjnych, które potrafią dane żądanie obsługiwać.

2.4.3. Komendy

Komendy to obiekty będące podklasami klasy `org.eclipse.gef.commands.Command`. Powstają z żądań i są generowane przez elementy edycyjne. Aktywne narzędzie wywołuje na elemencie edycyjnym metodę `getCommand`, która dla danego żądania przekazuje komendę, która je realizuje, bądź null, jeśli element edycyjny nie rozumie danego żądania. Komendy oprócz metody `execute`, realizującej główne zadanie, zawierają też metody `undo` i `redo` realizujące operacje związane z wycofaniem i ponawianiem akcji użytkownika. Stanowią one zastosowanie wzorca projektowego *Command*. Pojedyncze komendy mogą być łączone w większe bloki i wykonywane w całości.

2.4.4. Polityki edycyjne

Żądania tego samego rodzaju często wymagają podjęcia podobnych działań, bądź pokazania użytkownikowi odbywającego się procesu edycji w ten sam sposób. Dlatego zalecanym w GEF sposobem obsługi żądań jest delegowanie ich do polityk edycyjnych i taka jest domyślna implementacja metody `getCommand`. Polityki edycyjne to nic innego jak realizacja wzorca projektowego *Chain of Responsibility*. Tworzą je obiekty implementujące interfejs `org.eclipse.gef.EditPolicy`. Element edycyjny zamiast samemu generować komendy przegląda polityki edycyjne i każdej z nich zleca wygenerowanie komendy dla danego żądania, tworząc w ten sposób komendę złożoną. Jeśli dana polityka edycyjna nie rozumie żądania, to przekazuje null i taka komenda nie jest dodawana do powstającego łańcucha komend. Polityki edycyjne są przechowywane w elementach edycyjnych w postaci mapy, której klucze nazywane są rolami. Rolą może być dowolny obiekt, jednak w interfejsie `EditPolicy` jest zdefiniowany obszerny zestaw ról i należy z niego korzystać. Dzięki takiemu podejściu możliwe jest łatwe podmienianie polityk edycyjnych w trakcie działania aplikacji.

Z GEF dostarczony jest spory zestaw gotowych lub prawie gotowych (wymagających zaimplementowania jednej lub dwóch metod) polityk edycyjnych. Ich działanie polega na rozdzieleniu obsługi żądania na podstawie typu żądania (nie klasy żądania, ale atrybutu określającego jego typ) na kilka metod, często abstrakcyjnych, których implementacja jest już łatwiejsza. Często wcześniej dokonuje się wstępnego prze-

[2]Figury, które tworzą reprezentację graficzną trwającego procesu edycji istnieją w oddzielnej warstwie.

tworzenia żądania i docelowa metoda tworząca komendę otrzymuje już przetworzone argumenty (np. w metodzie `createAddCommand(EditPart child, EditPart after)` klasy `org.eclipse.gef.editpolicies.FlowLayoutEditPolicy`).

2.4.5. Połączenia

Połączenia to elementy modelu, które reprezentują pewną relację pomiędzy dwoma innymi wizualnymi reprezentacjami fragmentów modelu, zwanych źródłem i celem. Obiekty, które mogą być źródłem albo celem, są zwane węzłami. Połączenia stanowią szczególny rodzaj obiektów z kilku względów:

- Przede wszystkim mogą łączyć obiekty, które wizualnie są na różnych poziomach. Traktowanie ich jak inne obiekty spowodowałoby ich obcięcie do granic wyznaczonych przez rodzica.
- Zawsze muszą występować ponad innymi elementami diagramu.
- Zawsze łączą reprezentacje źródła oraz celu i nie mogą istnieć bez nich.

Z punktu widzenie Draw2D połączenia to po prostu figury jak każde inne. Zwykle tworzą ją obiekty klasy `org.eclipse.draw2d.Polyline`, składające się z zestawu punktów i linii pomiędzy nimi. Mają one jednak kilka interesujących cech:

1. Nie można dla nich ustawić prostokąta ograniczającego. Wynika on jednoznacznie z punktów tworzących dany wielokąt i jest najmniejszym prostokątem zawierającym wszystkie te punkty. Nie powinny więc być dodawane jako dzieci figur, których menedżerowie wyglądu ustawiają figurom potomnym prostokąt ograniczający.
2. Wielokąty są podklasami klasy `org.eclipse.draw2d.Shape`, więc można dla nich ustawiać takie atrybuty jak szerokość, czy styl linii.
3. Przedefiniowana jest w nich metoda containsPoint tak, aby dawała wartość *prawda*, gdy punkt znajduje się na wielokącie, bądź figurze będącym jego dzieckiem (np. etykietce).
4. Przesunięcie ojca wielokąta nie skutkuje przesunięciem wielokąta (z uwagi na pierwszą cechę).

Wielokąty nadają się do rysowania połączeń pomiędzy figurami, jednak nie spełniają oczekiwań stawianych przed prawdziwymi połączeniami. Połączenia powinny być w stanie automatycznie się rysować dla danego źródła i celu oraz zmieniać się adekwatnie do ich zmian. Wymagania te zostały zdefiniowane w interfejsie `org.eclipse.draw2d.Connection`.

Z każdym połączeniem związane są dwa końce: źródłowy i docelowy, będące implementacjami interfejsu `org.eclipse.draw2d.ConnectionAnchor`. Umożliwia on danemu połączeniu śledzenie jego zmian i w wypadku nastąpienia takowych odświeżenia całego połączenia. Końce połączeń służą przede wszystkim do wyznaczenia punktu zaczepienia bazując na właściwościach figury, która jest ich właścicielem (głównie na prostokącie ograniczającym) i zadanym punkcie odniesienia. GEF dostarcza kilku predefiniowanych końców połączeń. Z połączeniem związany jest również obiekt trasujący (implementuje interfejs `org.eclipse.draw2d.ConnectionRouter`), który jest odpowiedzialny za umieszczenie punktów połączenia na ekranie. Jego rola jest podobna do menedżerów wyglądu z tym, że wszelkie metody dotyczą połączeń. Obiekty trasujące są dzielone pomiędzy wieloma połączeniami. Przy wyznaczaniu punktów połączenia mogą brać pod uwagę ich wzajemne położenie bądź też właściwości ustawiane dla pewnych połączeń.

Wszystkie połączenia są standardowo przechowywane w specjalnie przeznaczonej dla nich warstwie.

Z GEF dostarczona jest klasa `org.eclipse.draw2d.PolylineConnection`, która jest wielokątem uzupełnionym o cechy połączenia. Posiada ona kilka cech, których nie posiada zwykły wielokąt, jak np. możliwość zdefiniowanie sposobu rysowania końców.

Dotychczasowy opis dotyczył samego sposobu wyświetlania połączeń. Oprócz wyświetlania połączenia muszą być podatne na edycję. Należy więc określić dla nich kontroler. W GEF funkcje kontrolera pełnią podklasy klasy `org.eclipse.gef.editparts.AbstractConnectionEditPart` (będąca implementacją interfejsu `EditPart`). Sposób pisania tych klas jest taki sam jak innych elementów edycyjnych z tym, że dodatkowo są dostępne funkcje do pobierania elementów edycyjnego będących źródłem lub celem danego połączenia.

Aby połączenia dodać do diagramu, należy przedefiniować w elementach edycyjnych, które mają mieć możliwość uczestniczenia w połączeniach, metody `getModelSourceConnections` i `getModelTargetConnections`, odpowiednio dla połączeń wychodzących i przychodzących. Istotne przy tym jest to, aby dane połączenie występowało dokładnie raz w charakterze połączenia wychodzącego i przychodzącego. Nie może zajść sytuacja, gdy pewien element edycyjny przekaże połączenie jako wychodzące, a żaden inny nie przekaże go jako przychodzącego.

Elementy edycyjne, które są źródłem bądź celem połączenia muszą obserwować zmiany modelu i jeśli zajdą takie, które wpływają na połączenia, odpowiednio odświeżać listy połączeń wychodzących i przychodzących.

Dla elementów edycyjnych będących końcami połączeń możliwe jest ustalenie wyglądu i sposobu zachowania się zakończeń. Wystarczy, że element edycyjny będący końcem będzie implementował interfejs `org.eclipse.gef.NodeEditPart`.

2.5. Podsumowanie

GEF jest bogatym zestawem bibliotek przeznaczonych do tworzenia edytorów graficznych dla platformy Eclipse. Zawiera wszystko co jest niezbędne programiście do napisania w pełni funkcjonalnego edytora. Pozwala skupić się bardziej na logice edytora, gdyż rozwiązuje wiele powszechnie spotykanych problemów związanych z edytorami graficznymi (jak np. test trafień, zarządzanie warstwami, paleta z narzędziami czy też interpretowanie niskopoziomowych zdarzeń przychodzących z klawiatury czy myszy). Programista ma dużą swobodę w używaniu bibliotek i może z powodzeniem wymieniać, bądź dostosowywać pewne jego części bez wpływu na pozostałe. Mimo że GEF jest zbudowany w sposób dosyć skomplikowany, to nauczanie się posługiwania nim w rozsądnym zakresie nie zajmuje dużo czasu. Choć wiele umożliwia, to nie zwalnia programisty z obowiązku myślenia, gdyż nawet najlepszy zestaw bibliotek nie pomoże, gdy jest używany w sposób niezgodny z przeznaczeniem i bezmyślny.

Ponieważ GEF jest nowym zestawem bibliotek, nie powstało jeszcze zbyt wiele materiałów pomocnych w jego nauce (w szczególności nie powstała jeszcze żadna książka opisująca dokładnie GEF). W Internecie można jednak znaleźć nieliczne artykuły omawiające pewne aspekty tej technologii. Dobrym punktem wyjścia jest artykuł napisany przez jednego z autorów GEF-a [Hudson03] oraz inny napisany przez osobę nie związaną bezpośrednio z GEF-em [rlemaigr04]. Na oficjalnych stronach projektu Eclipse dostępnych jest kilka artykułów poruszających wybrane aspekty GEF-a [Lee03, Zoio04]. W książce [GEFRed04] można znaleźć opis wielu aspektów GEF-a, nie jest to jednak pełny jego opis, a jedynie wprowadzenie z kilkoma ciekawostkami. O GEF-ie można również przeczytać w dodatkach do niektórych książek traktujących o systemach komponentów [SHNM05].

GEF mimo, że bardzo użyteczny, nie rozwiązuje jednak wszystkich problemów. W szczególności pozostawia programiście całkowitą swobodę, jeśli chodzi o wybór modelu, sposobu jego zmieniania i powiadamiania o tych zmianach. Jak powszechnie wiadomo, im więcej swobody, tym większa szansa na popełnienie błędu. Dlatego należy przyjąć pewien zestaw reguł, a później się go trzymać. Oczywiście te reguły powinny w jak największym stopniu umożliwiać pisanie kodu, który jest odporny na błędy programisty, a jeśli już takowe powstaną, pozwalać na szybkie ich znajdowanie. W ogólności jest to zadanie niewykonalne (inaczej można by było stworzyć system reguł, który potrafiłby orzekać o poprawności programów, co stoi w sprzeczności z twierdzeniem Gödla), istnieją jednak systemy pomagające w pisaniu kodu o wyższej jakości, a więc większej odporności na błędy.

Dobrym sposobem na podniesienie jakości kodu pisanego w Javie jest stosowanie się do reguł panujących w językach funkcyjnych, a w szczególności zapewnienie niezmienialności obiektów. W następnym rozdziale opisze jak można to osiągnąć.

Rozdział 3

Programowanie funkcyjne w języku Java

3.1. Motywacja

Na pierwszy rzut oka pomysł programowania funkcyjnego w Javie może się wydawać śmieszny. Po co programować w Javie funkcyjnie, skoro to język stworzony głównie z myślą o programowaniu obiektowym? Rzeczywiście Java została stworzona z myślą o programowaniu obiektowym i jest klasyfikowana jako język imperatywny [Belap04]. Imperatywny styl programowania charakteryzuje się tym, że wykonanie programu jest opisywane w kategoriach zmian stanu programu przez jego instrukcje. Programiści piszący w języku Java nie są jednak skazani na ten styl programowania. Stosując się do określonych reguł można w Javie pisać w sposób funkcyjny. Funkcyjny styl programowania opisuje wykonanie programu w kategoriach wyliczania wyrażeń. Poniżej przedstawiam krótką charakterystykę funkcyjnego stylu programowania:

- Wsparcie dla obiektów funkcyjnych[1] (ang. *closures*) i funkcji wyższego rzędu[2].
- Rekursja jako mechanizm kontroli przepływu sterowania w programie.
- Wymuszenie przezroczystości referencyjnej. Funkcja zawsze przekazuje tę samą wartość dla danego zestawu argumentów. Wartość wyrażeń nie zależy od globalnego stanu. Dzięki temu można określać własności funkcji polegając tylko i wyłącznie na własnościach podwyrażeń, a nie na kolejności wyliczenia, czy też efektach ubocznych wyrażeń.
- Brak efektów ubocznych. Efekt uboczny powstaje, gdy pewna konstrukcja języka modyfikuje stan systemu. W językach funkcyjnych wartości są przypisywane do zmiennych na stałe, więc nie ma możliwości ich późniejszej zmiany. Wywołanie funkcji skutkuje tylko wynikiem obliczeń, dzięki czemu nie ma możliwości powstania efektów ubocznych.

Dzięki takim właściwościom programy pisane w stylu funkcyjnym są mniej podatne na błędy i łatwiej te błędy znaleźć. Do wnioskowania o poprawności funkcji wystarczy znać właściwości podwyrażeń, które się na nią składają i dziedziny parametrów. Nie trzeba się martwić, że wynik obliczeń zależy od globalnego stanu, który może się w sposób nieprzewidziany zmienić, bądź też obliczenia mogą na niego wpłynąć.

[1] Są to obiekty, które zachowują się jak funkcje i mogą być przekazywane jak obiekty.
[2] Są to funkcje, które mogą jako argumenty przyjmować inne funkcje i przekazywać je jako wynik.

Doskonałym przykładem tego jak stan programu może prowadzić do błędów są powszechnie używane w Javie iteratory. Poniższy fragment programu ilustruje problem.

```
private static void writeAll(Iterator it) {
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }

    public static void main(String[] args) {
        ArrayList lista = new ArrayList();
        lista.add("1");
        lista.add("2");
        lista.add("3");
        Iterator it = lista.iterator();
        writeAll(it);
        lista.add("4");
    }
```

Zadaniem funkcji `writeAll` jest oczywiście wypisanie wszystkich elementów, przez które przechodzi iterator. Jak się można spodziewać na ekranie zostaną wypisane liczby 1, 2 i 3. Jednak patrząc tylko na kod funkcji writeAll nie można powiedzieć, że zawsze wypisze ona wszystkie elementy kolekcji, dla której została ona stworzona. Zmodyfikujmy trochę funkcję `main` pozostawiając funkcję `writeAll`. Zmodyfikowany kod funkcji `main` wygląda następująco.

```
public static void main(String[] args) {
        ArrayList lista = new ArrayList();
        lista.add("1");
        lista.add("2");
        lista.add("3");
        Iterator it = lista.iterator();
        lista.add("4");
        writeAll(it);
    }
```

Okazuje się teraz, że program kończy się błędem nic nie wypisując. Wynika to z tego, że stan obiektu przechowywanego na zmiennej `lista` jest związany z obiektem na zmiennej `it`. Funkcja `writeAll` przestała działać poprawnie mimo, że otrzymała poprawne argumenty. Programując w stylu funkcyjnym nie można doprowadzić do takiej sytuacji.

Aby móc programować w stylu funkcyjnym potrzebujemy dwóch rzeczy: obiektów niezmienialnych (albo inaczej obiektów klas niezmienialnych, czy też klas niezmienialnych) oraz możliwości przekazywania funkcji jako argumentów, z czego najistotniejsza wydaje się kwestia niezmienialnych obiektów, dlatego poświęcę jej najwięcej miejsca.

3.2. Obiekty niezmienialne

Istnienie obiektów niezmienialnych jest kluczowe, aby w ogóle móc myśleć o programowaniu w stylu funkcyjnym. Nawet jeśli nie chcemy w pełni wykorzystywać funkcyjnego stylu programowania, który mimo wielu zalet ma również wady, to używanie obiektów, których stanu

nie można zmieniać przynosi wiele korzyści. Jak można przeczytać w książce napisanej przez Joshua Bloch'a [Bloch01], będącej zestawem reguł dobrego programowania w języku Java, „klasy powinny być niezmienialne, chyba że jest naprawdę dobry powód, by były zmienialne". Jest wiele korzyści płynących z tego, że obiekty są niezmienialne. Oto najważniejsze z nich:

- Są proste. Obiekt niezmienialny pozostaje w stanie, w jakim został stworzony. Nie ma żadnych metod, które ten stan modyfikują. Jeśli pola obiektu mają spełniać jakieś zależności, to wystarczy je sprawdzać tylko w momencie tworzenia obiektu.
- Są z założenia bezpieczne w środowisku wielowątkowym. Nie wymagają żadnej synchronizacji. Nie mogą zostać „zepsute" przez wiele wątków jednocześnie się do nich odwołujących. Dlatego mogą być bez problemów współdzielone.
- Można bez obaw współdzielić części składowe obiektów niezmienialnych.
- Są dobrymi częściami składowymi bardziej skomplikowanych obiektów czy to zmienialnych, czy niezmienialnych. Łatwiej budować obiekty, gdy się wie, które ich części nie mogą się zmieniać w sposób nieprzewidywalny. Stanowią dzięki temu idealne klucze w słownikach.
- Nie muszą implementować metody `clone` ani dostarczać konstruktora kopiującego.

Jedyną tak na prawdę istotną wadą obiektów niezmienialnych jest to, że trzeba tworzyć nowy obiekt dla każdej nowej wartości.

Obiekty niezmienialne można podzielić na dwie kategorie wymagające oddzielnego potraktowania: obiekty proste i kolekcje. W dalszej części rozdziału przedstawię jeden ze sposobów zaimplementowania obu typów obiektów niezmienialnych.

3.2.1. Obiekty proste

Obiekty proste to takie, które w swoim wnętrzu zawierają z góry określoną liczbę elementów. Aby obiekty proste były niezmienialne należy stosować następujące reguły:

- Obiekt tworzony jest w całości. Nie ma sytuacji, gdy wykorzystywany jest konstruktor bezargumentowy, a następnie wykonywana jest seria metod `setXXX`.
- Nie ma żadnych metod, które zmieniają jego stan.
- Nie można przedefiniować żadnych jego metod. Klasa obiektu jest deklarowana z modyfikatorem `final`, który oznacza, że żadna inna klasa nie może z niej dziedziczyć.
- Wszystkie jej pola mają modyfikator `final` i są polami prywatnymi. W połączeniu z pierwszą regułą gwarantuje to, że tylko w konstruktorze będzie możliwe przypisanie wartości na pola.
- Jeśli jakieś pole jest obiektem klasy zmienialnej, to należy zapewnić, że klasa jest właścicielem tego obiektu i żadna inne metoda nie udostępnia go na zewnątrz w całości, bądź części. Aby to zapewnić, konieczne jest zrobienie kopii tego obiektu za pomocą konstruktora kopiującego, bądź metody `clone`. Mimo, że takie postępowanie pozwala na zagwarantowanie niezmienialności obiektu, to wprowadza niepotrzebną komplikację. W większości przypadków konieczność użycia obiektów klasy zmienialnej w obiekcie

klasy niezmienialnej wynika z potrzeby przechowywania kolekcji elementów i braku wsparcia dla niezmienialnych kolekcji ze strony języka Java[3]. Jeśli uda się zaimplementować niezmienialne kolekcje, to można przyjąć jako wymaganie, że wszystkie pola klasy niezmienialnej muszą być obiektami niezmienialnymi.

Poza wymaganymi cechami, dobrze jest, gdy klasa niezmienialna spełnia kilka innych reguł, które ułatwiają korzystanie z niej:

- Konstruktor jest prywatny. Do tworzenia obiektu wykorzystywana jest statyczna metoda `create`.
- Dla każdego pola klasy istnieje metoda `set`, której zadaniem jest przekazanie nowego obiektu z nową wartością danego pola.
- Dla pól, które mogą przyjmować wartość `null`, metody, które przekazują ich wartość mają prefiks „try", a metody, które ustawiając ich wartość (przekazując oczywiście nowy obiekt) mają w nazwie argumentu przyrostek „OrNull"

Kod przykładowej klasy niezmienialnej, zgodnej ze wszystkimi przytoczonymi wcześniej regułami, wygląda następująco:

```
public final class Klasa_niezmienialna {

    private final int pole1;
    private final String pole2;

    private Klasa_niezmienialna(int pole1, String pole2OrNull) {
        this.pole1 = pole1;
        this.pole2 = pole2;
    }

    public static Klasa_niezmienialna create(int pole1, String pole2OrNull) {
        return new Klasa_niezmienialna(pole1, pole2);
    }

    public int getPole1() {
        return pole1;
    }

    public Klasa_niezmienialna setPole1(int pole1) {
        return new Klasa_niezmienialna(pole1, pole2);
    }

    public String tryGetPole2() {
        return pole2;
    }
```

[3]W języku Java wszystkie tablice o niezerowej długości są obiektami zmienialnymi nawet, jeśli pole, na które są przypisane, zostało oznaczone jako `final`.

```
    public Klasa_niezmienialna setPole2(String pole2OrNull) {
        return new Klasa_niezmienialna(pole1, pole2);
    }
}
```

Klasę można użyć w następujący sposób:

```
public static void main(String[] args) {
        Klasa_niezmienialna obj = Klasa_niezmienialna.create(1,"a");
        System.out.println(obj.getPole2() + obj.getPole1());
        obj.setPole1(10);
        System.out.println(obj.getPole2() + obj.getPole1());
        obj = obj.setPole2("x");
        System.out.println(obj.getPole2() + obj.getPole1());
    }
```

Wynikiem wywołania powyższego kodu jest wypisanie na ekranie:

```
a1
a1
x1
```

Łatwo zauważyć, że ustawienie dla pola `pole1` wartości 10 pozostało bez efektu, gdyż wynik nie został nigdzie przypisany. Ustawienie zaś wartości „x" dla pola `pole2` skutkowało pozorną zmianą stanu obiektu — tak naprawdę powstał nowy obiekt z nową wartością i został przypisany na zmienną, na którą był przypisany poprzedni obiekt. Klasa więc rzeczywiście nie pozwala na żadną zmianę swojego stanu.

Klasy niezmienialne mają jednak wady, które wynikają z podstawowych wymagań. Jednym z nich jest niemożliwość dziedziczenia[4].

Często zachodzi jednak konieczność dziedziczenia spowodowana tym, że pewne byty modelowane przez klasy mogą być różnych rodzajów, a chcemy np. przechowywać je w jednym miejscu. Standardową drogą postępowania w takich sytuacjach jest stworzenie klasy abstrakcyjnej, w której będą zdefiniowane lub tylko zadeklarowane pewne ogólne metody wspólne dla wszystkich obiektów danego rodzaju, a następnie za pomocą dziedziczenia zdefiniowanie kilku klas odpowiadających interesującym nas typom obiektów. Przykładowo chcemy zamodelować drzewo binarne, w którym przechowywane są liczby całkowite wraz z operacją policzenia jego rozmiaru. Klasa realizująca te wymagania, napisana w często spotykany w Javie sposób, wygląda następująco:

```
public abstract class Wezel {

     public abstract int rozmiar();

}
```

[4] Zezwolenie na dziedziczenie klas niezmienialnych może spowodować, że niektóre metody zostaną nadpisane w niewłaściwy sposób i klasa straci własność niezmienialności.

```java
public class WezelWewnetrzny extends Wezel {

    private Wezel lewy;
    private Wezel prawy;
    private int zawartosc;

    public WezelWewnetrzny(Wezel lewy, Wezel prawy, int zawartosc) {
        this.lewy = lewy;
        this.prawy = prawy;
        this.zawartosc = zawartosc;
    }

    public int rozmiar() {
        return lewy.rozmiar() + 1 + prawy.rozmiar();
    }

    public int getZawartosc(){
        return zawartosc;
    }

    public void setZawartosc(int zawartosc){
        this.zawartosc = zawartosc;
    }

    public Wezel getLewy() {
        return lewy;
    }

    public void setLewy(Wezel lewy) {
        this.lewy = lewy;
    }

    public Wezel getPrawy() {
        return prawy;
    }

    public void setPrawy(Wezel prawy) {
        this.prawy = prawy;
    }
}

public class Lisc extends Wezel {

    public int rozmiar() {
        return 0;
    }

}
```

Ten sposób postępowania jest nazywany *dekompozycją typologiczną* (ang. *type decomposition*). Stosując ją zapewniamy, że każdy typ jest samopisującą się całością, ale definicje funkcji są porozrzucane po wszystkich typach [Roy03]. Technika ta jest popularna w językach umożliwiających pisanie w sposób obiektowy, do których oczywiście zalicza się Javę.

Przeciwieństwem tego podejścia jest to stosowane w językach funkcyjnych, a mianowicie *dekompozycja funkcjonalna* (ang. *functional decomposition*). Przy tym rodzaju dekompozycji każda funkcja jest jedną całością, a typy są porozrzucane po definicjach funkcji. Które z tych podejść jest lepsze?

- W dekompozycji funkcjonalnej można zmienić albo dodać nową funkcję bez zmieniania pozostałych. Jednakże dodanie lub zmienienie jakiegoś typu może nieść ze sobą konieczność zmieniania funkcji.
- W dekompozycji typologicznej można modyfikować typ, dodawać nowy (włączając w to dziedziczenie) bez zmieniania definicji innych definicji typów. Jednak zmiana bądź dodanie nowej funkcji może wymagać zmiany definicji wszystkich klas.

Aby możliwa była dekompozycja funkcjonalna potrzebne jest pojęcie typów wariantowych, powszechnie używanych w językach funkcyjnych. Cechą charakterystyczną typu wariantowego jest to, że jedyną informacją jaką bezpośrednio niesie jest informacja o tym, jakiego jest rodzaju. Dopiero po „otworzeniu” typu można wykorzystać jego zawartość. Dlatego z typami wariantowymi ściśle związana jest pojęcie dopasowania (ang. *matching*). Za pomocą dopasowania odbywa się „otwarcie” typu i dalsza jego obróbka. Następujący kod w języku SML ilustruje to zagadnienie:

```
datatype  int bin_tree = Leaf | Branch of  bin_tree * int * bin_tree;

val t_int = Branch (Branch (Leaf, 1, Leaf), 2, Branch (Leaf, 4, Leaf));

fun size Leaf = zero
     | size (Branch (t1, a, t2)) = (size t1) + 1 + (size t2);

fun zero () = 0;
```

Zdefiniowano tu typ dla drzewa binarnego oraz funkcję, która liczy jego rozmiar. Jest to klasyczny przykład dekompozycji funkcjonalnej. W jaki sposób wykorzystać tę technikę w języku Java i użyć jej nie łamiąc przy tym zasad tworzenia obiektów niezmienialnych? Zadanie to nie jest tak skomplikowane, jakby się mogło wydawać. Z pomocą przychodzi mechanizm rzutowania typów oraz konstrukcja *match*. W celu zdefiniowania typu wariantowego należy:

- Umieścić w klasie pole typu `Object` z modyfikatorem `final`.
- Zdefiniować tyle stałych całkowitoliczbowych, ile wariantów ma mieć dany typ wariantowy.
- Umieścić w klasie pole typu `int`, w którym przechowywana będzie informacja o tym, jaki konkretnie wariant zawiera ten typ wariantowy (przestrzeń wartości określają zdefiniowane wcześniej stałe).
- Dla każdego wariantu zdefiniować klasę niezmienialną, która niesie ze sobą informację o zawartości wariantu (ten krok można pominąć, jeśli jedyną informacją jaką niesie wariant, jest jego typ).

- Dla każdego wariantu zdefiniować metody `isXXX` (sprawdzenie, czy obiekt jest danym wariantem), `asXXX` (rozpakowanie wariantu) oraz statyczną `createXXX` (tworzy określony wariant).

Typy wariantowe można wykorzystywać na dwa sposoby:

1. przekazując je dalej,
2. rozpakowując je przy pomocy konstrukcji *match*.

O ile pierwszy sposób użycia typu wariantowego jest mało ciekawy, o tyle drugi wymaga wyjaśnienia. Konstrukcja *match* nie jest jakąś tajną konstrukcją języka Java. To seria konstrukcji `if else` stosowana w konkretnym celu – obsługi typu wariantowego. W ogólności wygląda następująco:

```
if(x.isA()){
// zrob cos z x.asA()
} else if (x.isB) {
// zrob cos z x.asB()
} ...
 else
throw new RuntimeException()
```

Najistotniejsze w konstrukcji match jest to, aby każdy `asXXX` był poprzedzony odpowiednim `isXXX`, żeby zostały w jednym miejscu wykorzystane wszystkie metody is danego typu wariantowego oraz aby w przypadku niedopasowania był zgłaszany wyjątek.

Znając powyższe zasady możemy już zdefiniować drzewo i operacje na nim w stylu funkcyjnym. Kod realizujący to zadanie wygląda następująco:

```
/**
 * FunWezel = variant {
 *      lisc
 *    | wezel of FunWezelWewnetrzny
 * }
 *
 */
public final class FunWezel {

    private final static int LISC = 0;
    private final static int WEZEL_WEWNETRZNY = 1;

    private final Object val;
    private final int type;

    private FunWezel(Object val, int type) {
        this.val = val;
        this.type = type;
    }
```

```
    public static FunWezel createLisc() {
        return new FunWezel(null, LISC);
    }

    public static FunWezel createWezel(FunWezelWewnetrzny wezel) {
        return new FunWezel(wezel, WEZEL_WEWNETRZNY);
    }

    public boolean isLisc(){
        return type == LISC;
    }

    public boolean isWezel(){
        return type == WEZEL_WEWNETRZNY;
    }

    public FunWezelWewnetrzny asWezel(){
        return (FunWezelWewnetrzny)val;
    }
}

/**
 *
 * FunWezelWewnetrzny = {
 *    lewy : FunWezel
 *    prawy : FunWezel
 *    zawartosc : int
 * }
 *
 */
public final class FunWezelWewnetrzny {

    private final FunWezel lewy;
    private final FunWezel prawy;
    private final int zawartosc;

    private FunWezelWewnetrzny(FunWezel lewy, FunWezel prawy, int zawartosc) {
        this.lewy = lewy;
        this.prawy = prawy;
        this.zawartosc = zawartosc;
    }

    public static FunWezelWewnetrzny create(FunWezel lewy, FunWezel prawy,
            int zawartosc) {
        return new FunWezelWewnetrzny(lewy, prawy, zawartosc);
    }

    public FunWezel getLewy() {
        return lewy;
```

```
    }

    public FunWezelWewnetrzny setLewy(FunWezel lewy) {
        return new FunWezelWewnetrzny(lewy, prawy, zawartosc);
    }

    public FunWezel getPrawy() {
        return prawy;
    }

    public FunWezelWewnetrzny setPrawy(FunWezel prawy) {
        return new FunWezelWewnetrzny(lewy, prawy, zawartosc);
    }

    public int getZawartosc() {
        return zawartosc;
    }

    public FunWezelWewnetrzny setZawartosc(int zawartosc) {
        return new FunWezelWewnetrzny(lewy, prawy, zawartosc);
    }
}

public class Operacje {

    public static int rozmiar(FunWezel wezel) {
        if (wezel.isLisc())
            return 0;
        else if (wezel.isWezel()) {
            FunWezelWewnetrzny wezelWew = wezel.asWezel();
            return rozmiar(wezelWew.getLewy()) + 1
                    + rozmiar(wezelWew.getPrawy());
        } else
            throw new RuntimeException();
    }
}
```

Nietrudno zauważyć, że teraz dodanie kolejnych operacji wymaga tylko dodawania funkcji do klasy `Operacje`. Warto zwrócić uwagę, że funkcje te są statyczne, a więc nie są związane z żadnym obiektem klasy. Ich wynik zależy tylko i wyłącznie od wartości argumentów, które są klasami niezmienialnymi.

Rzuca się w oczy fakt, że zamiast osiągnąć prostotę, otrzymujemy klasy, których kod jest znacznie obszerniejszy niż w przypadku standardowego podejścia, stosowanego w językach obiektowych, nie mówiąc już o kodzie w języku funkcyjnym. Nietrudno też zauważyć, że klasy te są schematyczne (oprócz oczywiście klasy `Operacje`). Aż się prosi, aby takie klasy były generowane przez automat na podstawie zwięzłego opisu. Okazuje się, że istnieją narzędzia umożliwiające wykonanie tego zadania.

Jedno z nich potrafi generować klasy niezmienialne, typy wariantowe, niezmienialne kolekcje (o czym później), a także użyteczne metody przydatne przy używaniu tych klas (jak

np. szkielet konstrukcji *match*, który w połączeniu z dostępną w środowisku Eclipse funkcją wklejania kodu wołanej metody znacznie ułatwia korzystanie z tej konstrukcji). Do generowania klas wykorzystuje bardzo prosty język opisu (fragmenty dotyczące konkretnych typów są umieszczane na początku klasy w formie komentarza). Tak więc aby zdefiniować typy `FunWezel` i `FunWezelWewnetrzny` wystarczy następujący opis:

```
FunWezel = variant {
    lisc
  | wezel of FunWezelWewnetrzny
}

FunWezelWewnetrzny = {
    lewy : FunWezel
    prawy : FunWezel
    zawartosc : int
 }
```

Teraz już zdecydowanie podejście funkcyjne jest prostsze do zaimplementowania niż obiektowe. Jedyną jego wadą jest dostępność. Otóż nie jest powszechnie dostępne, więc nie mogło zostać użyte do wersji ostatecznej edytora. Konieczne było ręczne napisanie wszystkich klas niezmienialnych.

3.2.2. Niezmienialne kolekcje

O ile zdefiniowanie niezmienialnych typów nie stanowi implementacyjnie i wydajnościowo większego problemu, to korzystanie z niezmienialnych kolekcji przysparza problemów. Wynika to m.in. z tego, że w języku Java tablice o niezerowej wielkości zawsze są obiektami zmienialnymi. W klasycznym podejściu do tego problemu korzysta się z klonowania [Goetz03]. Rozwiązanie to ma oczywiste wady:

- Klonowanie nie jest szeroko stosowanym mechanizmem, bywa trudne do poprawnego zaimplementowania i należy raczej go unikać [Hanley05].
- Jest kosztowne czasowo. Aby uzyskać kolekcję niezmienialną, która różni się od drugiej jednym elementem, trzeba przekopiować ją całą podmieniając ten jeden element. Tak więc złożoność operacji wstawienia elementu do kolekcji niezmienialnej (a dokładniej uzyskiwania drugiej kolekcji o zawartości zmienionej w ten sposób, że zmieniony jest jeden element) jest liniowa!

Na szczęście nie jest to jedyne możliwe podejście. Prostszym przypadkiem są słowniki. W ich przypadku bez problemu można osiągną czas wstawiania, usuwania i podmieniania (usunięcie połączone ze wstawieniem) elementu na poziomie $O(\log n)$. Wystarczy użyć do tego celu zrównoważonego drzewa poszukiwań binarnych (np. AVL [BDR96]) modyfikując operacje na nim w taki sposób, aby wszystkie węzły drzewa były niezmienialne. Wynika z tego, że każda operacja na słowniku powoduje powstanie $O(\log n)$ nowych obiektów, co w porównaniu z czasem $O(n)$ przy klonowaniu stanowi istotny postęp. Technika ta została zaproponowana w [CLR98] jako sposób implementacji trwałych zbiorów dynamicznych. Przy implementacji niezmienialnego drzewa istotne jest, aby nie trzymać w węzłach referencji do

rodziców danego węzła. Jeśli taka informacja miałaby być przechowywana, to każda zmiana węzła powodowałaby konieczność skopiowania wszystkich węzłów w drzewie[5].

Tablice stanowią większy problem, ponieważ idealnie by było, gdyby czas wstawiania i pobierania elementu był na poziomie $O(1)$. Warto się w tym momencie zastanowić jakie operacje na tablicach będą wykonywane najczęściej oraz jakie cechy tablicy nas interesują. Charakterystyczne cechy tablicy to:

- Możliwość wstawienia elementu na dowolną pozycję tablicy w czasie $O(1)$, co pociąga za sobą to, że kolejność elementów w tablicy ma znaczenie.
- Możliwość przeglądania wszystkich elementów tablicy po kolei w czasie $O(n)$.
- Opcjonalnie wstawienie elementu pomiędzy pewne elementy w czasie $O(n)$[6].
- Opcjonalnie usuwanie elementu tablicy czasie $O(n)$(z tych samych powodów co powyżej).

Okazuje się, że tablice najczęściej używane są do trzymania elementów, które ułożone są w konkretnej kolejności, a następnie wielokrotnego ich przeglądania sekwencyjnego. Operacje bezpośredniego dostępu do konkretnego elementu są rzadsze, a nieraz zachodzi konieczność wstawiania elementu pomiędzy inne lub usuwania go z tablicy, co prowadzi do jej kopiowania.

Dlatego sensownym wydaje się zaimplementowanie tablicy niezmienialnej jako drzewa binarnego. Otrzymujemy wtedy czas dostępu do elementu, a także podmiany elementu na poziomie $O(\log n)$. Jedyną niedogodnością takiego podejścia jest potencjalny czas przeglądnięcia wszystkich elementów takiej tablicy (wykonując operacje pobrania elementu po kolei dla każdego elementu), który wynosi $O(n \log n)$. Tę niedogodność można zniwelować stosując odpowiednio zaimplementowany iterator, zapewniając czas przeglądania całej kolekcji $O(n)$.

Aby drzewo mogło spełniać swoją rolę jako tablica, należy ustalić na nim określony porządek. Najlepszym do tego celu wydaje się porządek infiksowy[7] z tego względu, że jest zgodny z porządkiem w standardowych drzewach BST[8]. Dzięki temu można stworzyć ogólną implementację drzewa BST. Do ustalania ścieżki, którą należy podążać w drzewie zamiast wartości z węzłów, należy wykorzystać implementację interfejsu[9], która będzie potrafiła odpowiadać na pytania o kierunek dalszego przeszukiwania drzewa.

W przypadku zwykłych słowników decyzja ta będzie oczywiście zależała od wartości przechowywanych w węzłach obiektów, zaś w przypadku tablic od liczby elementów w poddrzewach.

3.2.3. Obiekty funkcyjne

Istnienie obiektów funkcyjnych w językach funkcyjnych jest istotnym czynnikiem zwiększającym z jednej strony modularyzację programów pisanych w językach funkcyjnych, a z drugiej umożliwiającym lepsze wykorzystanie już istniejącego kodu. Jako przykład niech posłuży popularna a zarazem prosta funkcja `foldl`. Jej zadaniem jest iteracyjne aplikowanie danej funkcji do elementów listy. Oto jej definicja w języku SML:

[5]Dla każdego węzła na ścieżce do danego węzła należałoby poprawić jego dzieci, co w konsekwencji prowadzi do poprawienia wszystkich potomków węzłów na ścieżce do korzenia, czyli wszystkich węzłów.

[6]W przypadku zwykłych tablic nie jest możliwe wstawianie elementu bez wykonania kopiowania, więc przyjmujemy czas $O(n)$.

[7]W porządku infiksowym dzieci na lewo od danego węzła są przed nim, a na prawo po nim.

[8]W drzewach BST dzieci na lewo od danego węzła przechowują wartości mniejsze, a na prawo większe od wartości w węźle.

[9]Z funkcyjnego punktu widzenia jest to funktor.

```
fun foldl f zero [] = zero
  | foldl f zero (a::b) = f a (foldl f zero b);
```

Przykładowe użycie:

```
fun plus a b = a + b;
fun razy a b = a * b;
fun suma = foldl plus 0;
fun iloczyn = foldl razy 1;
```

W łatwy sposób otrzymujemy nowe funkcje, które operują na liście licząc jej sumę i iloczyn.

Bezpośrednie uzyskanie takiego efektu w Javie jest niemożliwe, gdyż tym języku nie można przekazywać funkcji bezpośrednio ani aplikować funkcji bez podania wszystkich jej argumentów. Można jednak uzyskać taką funkcjonalność dzięki odpowiedniemu zastosowaniu interfejsów. Przekazywanie interfejsów jest podobne do przekazywania funkcji. Nie wiemy jak wygląda klasa implementująca dany interfejs. Znane są jedynie deklaracje metod. Dzięki temu możemy używać interfejsu jako bytu całkowicie abstrakcyjnego, zupełnie jak przy przekazywaniu funkcji w językach funkcyjnych.

Pozostała jeszcze kwestia przekazywania funkcji jako wyniku działania innej funkcji. I tu znów z pomocą przychodzą interfejsy oraz mechanizm klas anonimowych. Klasy anonimowe to takie, które nie posiadają nazwy i których definicja znajduje się w miejscu użycia (tzn. powstania obiektu danej klasy). Tak więc przekazywanie funkcji można zaimplementować jako przekazywanie obiektu klasy anonimowej implementującej pewien interfejs.

Częściowa aplikacja funkcji powstaje z połączenia tych dwóch technik. Wystarczy w pewnej funkcji jako argument przyjąć interfejs z funkcją, której częściową aplikację chcemy przeprowadzać, oraz argumenty do zaaplikowania i przekazać interfejs z funkcjami, których listy parametrów są pomniejszone o zaaplikowane argumenty.

Przykładowa implementacja funkcji `foldl` w Javie oraz przykład jej użycia wygląda następująco (dla uproszczenia w tablicy są przechowywane wartości całkowitoliczbowe).

```
public interface IOper {
    public int oper(int a, int b);
}

public class Plus implements IOper {

    public int oper(int a, int b) {
        return a + b;
    }

}

public class Razy implements IOper {

    public int oper(int a, int b) {
        return a * b;
    }
```

```
}

public interface IFoldl {

    public int foldl(IOper oper, int zero, int[] tab);

}

public interface IFoldlProsty {

    public int foldl(int[] tab);

}

public class Foldl implements IFoldl {

    public int foldl(IOper oper, int zero, int[] tab) {
        int ret = zero;
        for (int i = tab.length - 1; i >= 0; i--) {
            ret = oper.oper(tab[i], ret);
        }
        return ret;
    }

}

public class FoldUtil {

    private IFoldlProsty zaaplikuj(final IFoldl foldl, final IOper oper,
            final int zero) {
        return new IFoldlProsty() {
            public int foldl(int[] tab) {
                return foldl.foldl(oper, zero, tab);
            }
        };
    }

    public IFoldlProsty dajSume(int[] tab) {
        return zaaplikuj(new Foldl(), new Plus(), 0);
    }

    public IFoldlProsty dajIloczyn(int[] tab) {
        return zaaplikuj(new Foldl(), new Razy(), 1);
    }
}
```

Opisane techniki są często stosowane przez programistów Javy, którzy nierzadko nie zdają sobie sprawy ich funkcyjnego rodowodu. Klasycznym przykładem ich użycia jest wzorzec projektowy *Visitor*. Zastosowanie go polega na stworzeniu pewnej klasy, której zadaniem jest przeglądanie kolekcji oraz interfejsu, którego instancje są używane do odwiedzania elementów tej kolekcji. Klasa, która przegląda kolekcję nie wie nic na temat implementacji klasy odwiedzającej elementy kolekcji. Cała logika operacji jest zamknięta w klasie odwiedzającej.

3.3. Podsumowanie

Technika programowania funkcyjnego w języku Java nie jest niczym nowym. Opisy różnych jej aspektów można znaleźć zarówno w Internecie, jak i literaturze. Dotyczą one głównie obiektów niezmienialnych [Hanley05, Bloch01] i przekazywania funkcji [Belap04]. Koncepcję konstrukcji *match* zaczerpnąłem z własnego doświadczenia programistycznego w firmie „Comarch S.A.", gdzie wykorzystuje się techniki programowania funkcyjnego w języku Java. Z tego też źródła pochodzi koncepcja użycia drzew w charakterze tablic niezmienialnych. Opis pomysłu niezmienialnych kolekcji można też znaleźć w [CLR98] jako jedno z ćwiczeń.

Dzięki programowaniu funkcyjnemu można stworzyć kod, który jest łatwiejszy w utrzymaniu, rozwijaniu, a także mniej podatny na błędy. Łatwiejsze jest uzyskanie modularności programu, a co za tym idzie poprawienie jego czytelności. Większość głównych mechanizmów języków funkcyjnych da się przenieść do języka Java. Niektóre z nich wymagają pewnego wstępnego wysiłku implementacyjnego (jak np. stworzenie niezmienialnych, generycznych kolekcji), ale wysiłek ten procentuje wyższą jakością i mniejszą podatnością na błędy powstałego kodu.

Z tych powodów zdecydowałem się zastosować ten styl programowania przy tworzeniu *Wizualnego Edytora Skryptów programu Ant* (w skrócie WESA). W następnym rozdziale opiszę funkcjonalność edytora, jego architekturę oraz problemy, jakie napotkałem przy jego tworzeniu i przyjęte rozwiązania.

Rozdział 4

WESA – Wizualny Edytor Skryptów programu Ant

4.1. Główne cechy narzędzia

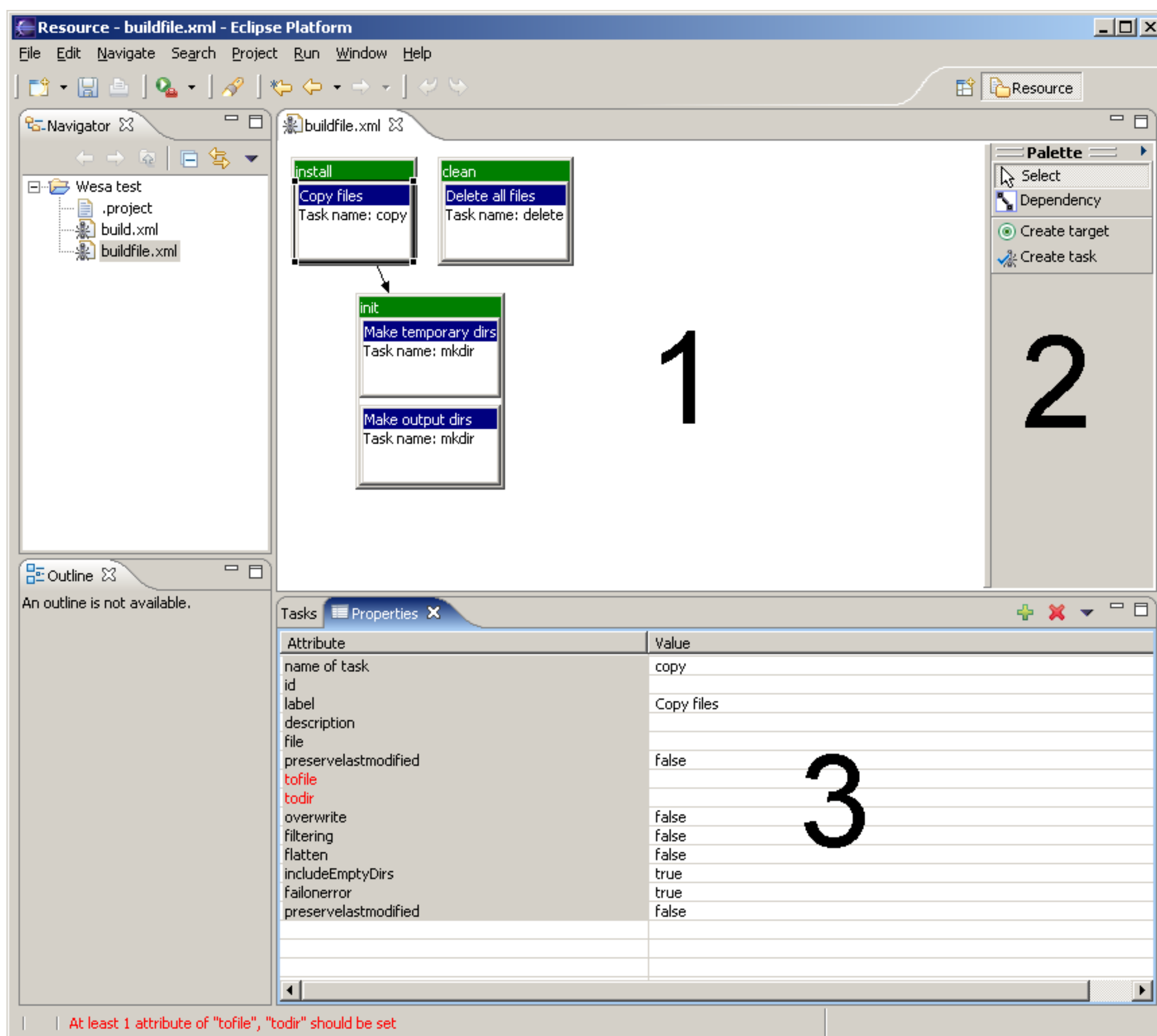
Motywacją do stworzenia edytora WESA były niedostatki istniejących narzędzi (a w zasadzie brak działających narzędzi), umożliwiających graficzną edycję. Jak zostało to przedstawione w rozdziale 1, większość istniejących narzędzi ułatwiających pracę z Antem umożliwia edycję w sposób czysto testowy, nie pozwalając na bieżąco śledzić powstających zależności. Drugą ich wadą jest często wbudowanie składni Anta. Gdy zmienia się składnia (zazwyczaj poprzez dodanie nowych zadań, bądź atrybutów do już istniejących) trzeba uaktualnić całe narzędzie. Inną jest niewystarczająca kontrola błędów. Wiele zadań Anta ma atrybuty, które zależą od siebie nawzajem. Zależności te są sprawdzane przez narzędzia (a właściwie przez Anta) dopiero w momencie wykonania skryptu. Z powyższych powodów powstał edytor WESA.

Edytor oferuje następujące udogodnienia:

- Edycja skryptów Anta w sposób graficzny, a co się z tym wiąże:
  - natychmiastowy podgląd zależności,
  - bezpośredni dostęp do wszystkich atrybutów edytowanych obiektów,
  - standardowe operacje dla edytorów graficznych, takie jak funkcja wycofania zmian undo i redo, dodawanie elementów przy pomocy palety, usuwanie elementów, przemieszczanie ich i kopiowanie przy pomocy techniki „przeciągnij i upuść”, odczyt i zapis skryptów Anta.
- Możliwość definiowania składni dodatkowych zadań (poprzez łatwo dostępny plik konfiguracyjny), a także generyczna obsługa zadań, których składnia nie została zdefiniowana.
- Informacja o błędach edycji wraz z podświetleniem błędnych obiektów.

Główny ekran edytora prezentuje rysunek 4.1. Można w nim wyróżnić następujące części:

1. Obszar edycyjny. Zawiera obiekty reprezentujące cele, zadania i zależności pomiędzy celami. Użytkownik przy pomocy myszy może zaznaczać poszczególne obiekty, przemieszczać je, a także usuwać. Działanie kursora myszy zależy od wybranego z palety narzędzia.



Rysunek 4.1: Ekran główny edytora

2. Paleta. Z niej użytkownik dokonuje wyboru narzędzia, którego będzie później używał. Do dyspozycji jest zaznaczanie i wiążące się z nim operacje wykorzystujące technikę „przeciągnij i upuść”, tworzenie połączeń, tworzenie celów oraz tworzenie zadań.

3. Właściwości obiektów. W tym oknie użytkownik może ustawiać właściwości obiektów takie jak: rodzaj zadania i jego konfiguracja[1], opis, który pojawia się w okienkach podpowiedzi (ang. *tooltip*) i inne. W przypadku, gdy nic nie jest zaznaczone, wyświetlane są właściwości projektu.

Typowy cykl pracy składa się z następujących kroków:

1. Umieszczenie w głównym obszarze edycyjnym zadań, będących definicjami zmiennych używanych w projekcie.

2. Umieszczenie celów i zdefiniowanie zależności pomiędzy nimi.

[1] Jeśli pole dopuszcza tylko określone wartości, to użytkownik wybiera je z listy.

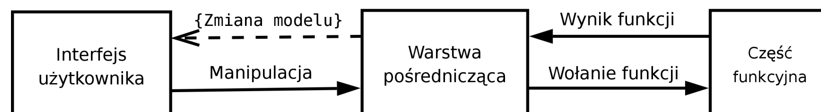
3. Umieszczenie w celach zadań i ewentualnych podzadań oraz określenie ich rodzaju (w oknie właściwości).

4. Zdefiniowanie dodatkowych właściwości zadań.

5. Zapisanie skryptu na dysk.

4.2. Architektura

Edytor WESA został zrealizowany jako wtyczka działająca w środowisku Eclipse. Do jego powstania został wykorzystany zestaw bibliotek GEF, a także techniki opisane w rozdziale 2. Jego architektura jest zgodna ze wzorcem MVC. Rysunek 4.2 przedstawia ogólną architekturę edytora.

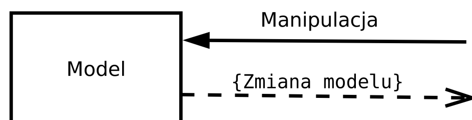


Rysunek 4.2: Architektura edytora

Istotnym założeniem architektonicznym jest brak bezpośredniej możliwości użycia przez interfejs użytkownika (co obejmuje również elementy edycyjne, pełniące funkcje kontrolerów w architekturze MVC) funkcji realizujących logikę edytora. Wszystkie akcje użytkownika są zamieniane na komendy (proces ten został opisany w rozdziale 2). Jednak w edytorze jest tylko jedna klasa dla komend. Jej implementacja jest trywialna:

```
public class ProjectCommand extends Command {

    private final AppCmd.MMIntf appMM;
    private final ProjectCmd cmd;

    public ProjectCommand(AppCmd.MMIntf appMM, ProjectCmd cmd) {
        this.appMM = appMM;
        this.cmd = cmd;
    }

    public void execute() {
        appMM.projectCmd(cmd);
    }

    public void redo() {
        appMM.redo();
    }

    public void undo() {
        appMM.undo();
    }
}
```

Nie trudno zauważyć, że komendy nie zawierają żadnych bezpośrednich odniesień do obiektów modelu[2]. Wykonanie komend odbywa się w części funkcyjnej z udziałem warstwy pośredniczącej. Został tu zastosowany schemat, który przedstawia rysunek 4.3.



Rysunek 4.3: Schemat interakcji z modelem

Jedyną drogą do zmiany modelu, jest użycie specjalnej klasy manipulatora, która pobiera stan modelu, zleca wykonanie operacji, a następnie ustawia stan modelu. Model przy każdej zmianie wysyła powiadomienia do zarejestrowanych w nim obiektów nasłuchu. Wykonanie operacji na modelu nie powoduje przekazania żadnego wyniku. Nie jest do tego potrzebna wiedza o modelu, ani dostęp do niego, czego przykładem jest klasa `ProjectCommand`. Z drugiej strony, aby reagować na zmiany modelu nie trzeba mieć możliwości wykonywania na nim operacji.

4.2.1. Interfejs użytkownika

Cały interfejs został zbudowany przy pomocy zestawu bibliotek GEF. Jednakże GEF nie jest przystosowany do współpracy z obiektami modelu, który jest obiektem. Każda jego zmiana powoduje w istocie powstanie nowego modelu z nową wartością. Kłóci się to z podejściem stosowanym w GEF-ie, gdzie istnieje odwzorowanie z obiektów stanowiących część modelu w elementy edycyjne. Aby temu zaradzić, a przy okazji rozwiązać problem z powiadamianiem o zmianach modelu, należało opakować niezmienialny obiekt w zmienialną klasę. Istotne przy tym jest zachowanie założenia o modelmanipulacji, czyli zapewnienie, że jedyną możliwością zmiany modelu jest skorzystanie z obiektu manipulatora. Zastosowane rozwiązanie zostanie opisane szerzej w punkcie 4.3.

4.2.2. Warstwa pośrednicząca

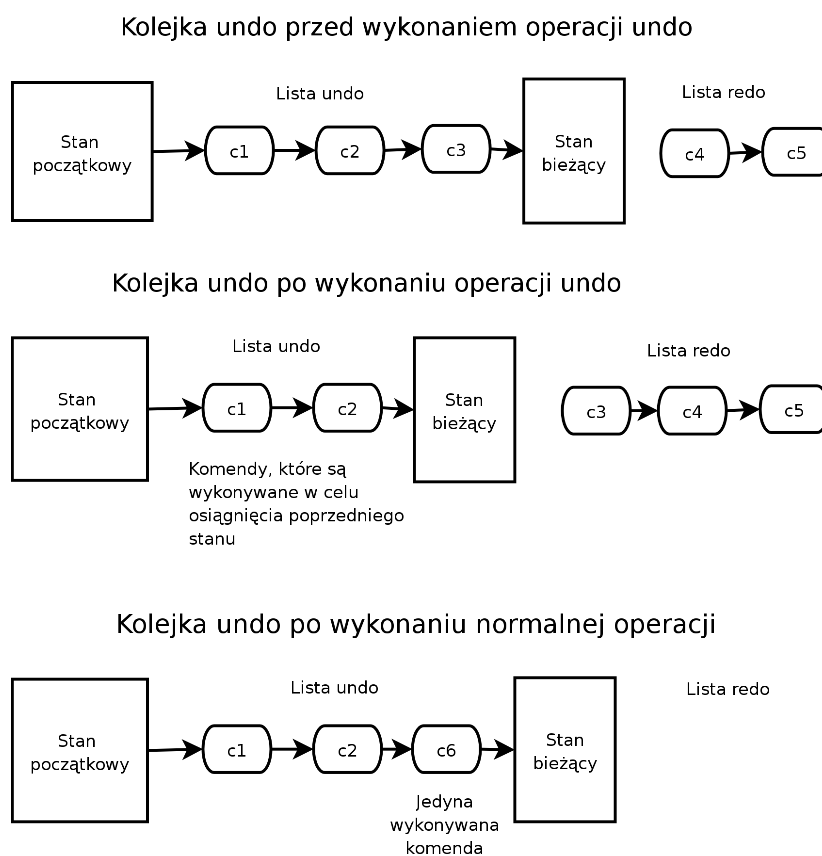
Kluczową rolę w warstwie pośredniczącej odgrywa klasa `wesa.logic.appliers.SyncApplier`. Odpowiada ona za synchronizowanie dostępu do modelu, udostępnianie go, wykonywanie komend i aplikowanie ich wyników do modelu. Tylko ta klasa może faktycznie zmieniać model, co w konsekwencji powoduje rozesłanie powiadomień o zmianie. Dostęp do niej odbywa się poprzez klasę pośredniczącą `wesa.logic.appliers.ProxyMM`, która otrzymuje z zewnątrz początkowy stan modelu i udostępnia model i modelmanipulator oraz interfejs funkcji pomocniczych, które pozwalają wykonać na modelu funkcje i przekazać bezpośrednio ich wynik, jednak nie zmieniają stanu modelu (więc nie są to manipulacje). Używane są one do obsługi okna z właściwościami obiektów.

Wykonywanie komend odbywa się za pomocą mechanizmu aplikatorów. Aplikator to obiekt, który służy do przeprowadzania operacji na innym obiekcie (może ich być kilka). Po stworzeniu aplikatora wykonuje się na nim zestaw metod (zwykle jedną) za pomocą udostępnianego przez niego interfejsu, a następnie pobiera się przechowywaną wartość. Interfejs udostępniany przez aplikator można łatwo konwertować do postaci, w której jedyną jego metodą jest `invoke`. Jej zadaniem jest wykonanie podanej jako argument komendy. Konwersja

[2]Przy stosowaniu GEF-a popularna jest praktyka przechowywania w komendach odniesień do modelu.

jest możliwa w dwie strony, więc aplikator może przekazywać interfejs służący do wykonywania komend w dwóch postaciach: prostej (metoda `invoke`) i złożonej (zestaw funkcji, z których każda jest związana z wykonaniem pewnej manipulacji). Przykładami aplikatorów są klasy `wesa.logic.appliers.AppAplier`, `wesa.logic.appliers.ProjectApplier` i `wesa.logic.appliers.ProjectUndoApplier`.

W warstwie pośredniczącej znajduje się również implementacja kolejki undo. Cechą charakterystyczną tej implementacji jest niezależność od wykonywanych komend. Kolejka undo znajduje się w stanie aplikacji. Zbudowana jest z początkowego stanu, listy komend, które można wycofać (*lista undo*), listy komend, które można powtórzyć (*lista redo*) oraz stanu bieżącego. Wycofanie komendy polega na przeniesieniu ostatniej komendy z listy undo do listy redo, wykonaniu wszystkich komend z listy undo przyjmując jako stan początkowy stan początkowy zapisany w kolejce undo oraz na uaktualnieniu stanu bieżącego. Powtórzenie komendy wygląda podobnie z tym, że komenda wędruje z listy redo do listy undo. Dzięki takiemu mechanizmowi możliwe jest wycofanie dowolnej manipulacji. Proces ten przedstawiono na rysunku 4.4.



Rysunek 4.4: Mechanizm wycofywania operacji

Po przekroczeniu pewnej ustalonej liczby należy razem z uaktualnianiem stanu końcowego uaktualniać stan początkowy, aby kolejka zbytnio się nie rozrosła. Kolejkę undo realizuje klasa `ProjectUndoApplier`, która jest aplikatorem. Używa się jej dokładnie tak, jak innych aplikatorów. Przykładowo wykonanie komendy dotyczącej projektu wygląda następująco (fragment klasy `AppAplier`).

```
public void projectCmd(ProjectCmd cmd) {
   ProjectUndoApplier applier = new ProjectUndoApplier(appState
              .getProject(), appState.getSyntax(), appState
              .getUndoQueue());
   applier.getMM().invoke(cmd);
   appState = appState.setProject(applier.getProjectState());
   appState = appState.setUndoQueue(applier.getQueue());
}
```

4.2.3. Część funkcyjna

Ta część jest odpowiedzialna za realizację logiki edytora. Klasy ją tworzące znajdują się w pakietach `wesa.logic.functions`, `wesa.logic.manip` i `wesa.logic.utils`. Jej cechą charakterystyczną jest to, że wszystkie tworzące ją klasy mają tylko metody i niezmienialne pola statyczne, a więc są klasami użytkowymi (ang. *utility class* [Bloch01]). Drugą ich cechą jest operowanie na klasach niezmienialnych, które reprezentują typy używane do reprezentacji modelu aplikacji i danych pomocniczych, powstałe zgodnie z zasadami opisanymi w rozdziale 3. Z tych powodów są bezstanowe i wyniki metod zależą tylko i wyłącznie od danych wejściowych.

W tej części edytora znajdują się wiele klas, których implementacji nie będę przytaczał, gdyż zajęłoby to zbyt wiele miejsca (jak chociażby klasy odpowiedzialne za wykrywanie błędów w strukturze skryptu w stosunku do odgraniczeń nałożonych przez zadaną składnię).

Warto zwrócić jedynie uwagę na sposób dostępu do poszczególnych elementów. Otóż położenie każdego elementu opisuje wskaźnik, który jest tablicą identyfikatorów obiektów napotykanych na ścieżce do niego. Każdy obiekt (projekt, cel lub zadanie) zawiera tablicę podobiektów (jedną lub dwie w przypadku projektu). Ułożenie obiektów w tablicy jest istotne. Z uwagi na fakt, że potrzebny jest szybki dostęp do konkretnego elementu w tablicy na podstawie jego identyfikatora (przy wyciąganiu obiektu pod podanym wskaźnikiem) z każdą tablicą związana jest mapa, która odwzorowuje identyfikatory w indeksy w tablicy.

4.3. Modele

Jak zostało wspomniane w punkcie 4.2.1 idea niezmienialnych obiektów kłóci się z GEF-em, gdyż ten wymaga, aby modele zmieniały swój stan i o tych zmianach powiadamiały. Dlatego konieczne było opakowanie niezmienialnych obiektów w zmienialne modele. Każdy model implementuje interfejs `wesa.models.Model`, który wygląda następująco:

```
public interface Model {

    public Disposable addListener(ModelListener listener);

    public Object get();
}
```

Nietrudno zauważyć, że nie ma możliwości bezpośredniej zmiany takiego modelu, więc elementy edycyjne nie będą mogły wprowadzać zmian inaczej, niż przez modelmanipulacje. Jakieś zmiany modelu są jednak potrzebne. Dlatego model pełny, który może być zmieniany, implementuje interfejs `FullModel`, będący rozszerzeniem interfejsu `Model`:

```
public interface FullModel extends Model {

    public void set(Object value);

}
```

Klasa implementująca ten interfejs to `wesa.models.FullModelImpl`. Jedyny obiekt tej klasy występuję wewnątrz klasy `SyncApplier`, dzięki czemu tylko tam może nastąpić właściwa zmiana modelu.

Z każdym modelem związana jest lista obiektów nasłuchujących. Dodanie nasłuchu do modelu odbywa się poprzez metodę addListener, która przekazuje obiekt implementujący interfejs `wesa.models.Disposable`, wyglądający następująco:

```
public interface Disposable {

    public void dispose();

}
```

Wygląda on identycznie jak interfejs `org.eclipse.gef.Disposable`. Dzięki powtórzeniu definicji interfejsu wszystkie klasy i interfejsy związane z obsługą modeli znajdują się w jednym pakiecie, dzięki czemu mogą zostać użyte w innym projekcie, niekoniecznie korzystającym z bibliotek GEF.

Wywołanie metody `dispose` usuwa obiekt nasłuchu z listy obiektów nasłuchujących.

Zaprezentowane rozwiązanie nie jest jeszcze wystarczające. Elementy edycyjne nie są przecież zainteresowane modelem całej aplikacji, tylko jego fragmentami. Tutaj z pomocą przychodzi mechanizm filtrowania modeli. Polega on na tym, że tworzymy nowy model w oparciu o pewien model bazowy i zadaną funkcję filtrującą. Wtedy każda operacja `get` na modelu przekaże rezultat przefiltrowany przez funkcję filtrującą. Klasa realizująca tę ideę to `wesa.models.FilteredModel`. Jest na tyle prosta, że przytoczę ją w całości.

```
public class FilteredModel implements Model {

    private final Model base;
    private final ModelFilter filter;
    private final Object lock = new Object();

    public FilteredModel(Model base, ModelFilter filter) {
        this.base = base;
        this.filter = filter;
    }

    public Disposable addListener(ModelListener listener) {
        synchronized (lock) {
            return base.addListener(listener);
        }
    }
```

```
    public Object get() {
        synchronized (lock) {
            Object val = base.get();
            return filter.filter(val);
        }
    }
}
```

Jako argument dostaje model bazowy i funkcję filtrującą, a dokładniej prosty interfejs. Jest to oczywiście realizacja idei przekazywania funkcji, zaprezentowanej w rozdziale 3. Interfejs `wesa.models.ModelFilter` wygląda następująco:

```
public interface ModelFilter {

    public Object filter(Object value);

}
```

Dzięki takiemu podejściu każdy element edycyjny może otrzymać interesujący go fragment modelu i dzięki tworzeniu odpowiednich funkcji filtrujących budować modele dla elementów edycyjnych odpowiedzialnych za obiekty składowe.

Mechanizm filtrowania modeli wystarcza, aby pogodzić idee programowania funkcyjnego z GEF-em. Mimo to można do tego mechanizmu wprowadzić pewną optymalizację. Zauważmy, że przy każdej zmianie modelu dowiadują się o niej wszystkie elementy edycyjne bez względu na to, czy ta zmiana ich dotyczy, czy nie. Oczywiście można by przerzucić na elementy edycyjne obowiązek sprawdzania, czy nowy stan modelu różni się od starego, ale lepiej jest stworzyć po prostu nowy rodzaj modelu – model leniwy. Model leniwy to taki, który informuje o zmianach tylko wtedy, gdy są one istotne. Funkcję użyteczności zmian dostaje z zewnątrz. Oczywiście najprostsza i najszybsza jest funkcja oparta na równości referencyjnej. W naszym przypadku jest ona wystarczająca, gdyż zawartość modelu to obiekt niezmienialny. Jeśli referencje są takie same, to na pewno są takie same wnętrza. Aby jeszcze poprawić ten mechanizm można by przy każdej funkcji `setXXX` w obiektach niezmienialnych sprawdzać, czy nowa wartość jest taka sama jak stara (równość referencyjna) i jeśli tak, to przekazywać `this`. Dzięki temu zmniejszy się liczba „fałszywych powiadomień”.

Klasa realizująca idee modelu leniwego to `wesa.models.LazyModel`. Jako argument przyjmuje model bazowy i funkcję użyteczności (implementację interfejsu `wesa.models.ModelComparator`). Zawiera też własną listę obiektów, które nasłuchują na nim, a sam nasłuchuje na modelu bazowym. W przypadku nadejścia powiadomienia od modelu bazowego ustala przy pomocy funkcji użyteczności, czy dana zmiana była istotna. Jeśli tak, to powiadamia obiekty, które się w nim zarejestrowały jako nasłuch. W przeciwnym przypadku aktualizuje tylko poprzedni stan modelu.

4.4. Budowa prostego skryptu przy pomocy edytora WESA

Na zakończenie tego rozdziału przedstawię krok po kroku jak przy pomocy edytora WESA przygotować prosty skrypt, którego zadaniem będzie stworzenie kilku katalogów i przeniesie tam wybranych plików z katalogu odczytanego ze zmiennej środowiskowej. Skrypt będzie też umożliwiał zrobienie porządków, tzn. usunięcie stworzonych katalogów wraz z zawartością. Cel ten można osiągnąć wykonując następujące kroki:

1. Aby rozpocząć budowę skryptu, należy stworzyć pusty dokument. W tym celu w otwartym projekcie (jeśli nie ma żadnego otwartego należy go stworzyć) wybieramy polecenie `File|New|File`. Jako nazwę pliku podajemy `build.xml` i potwierdzamy przyciskiem `Finish`. Ilustruje to rysunek 4.5. Plik zostanie stworzony i otworzy się standardowy edytor Anta z Eclipsa. Aby móc edytować plik w sposób graficzny, zamykamy standardowe okno edytora.

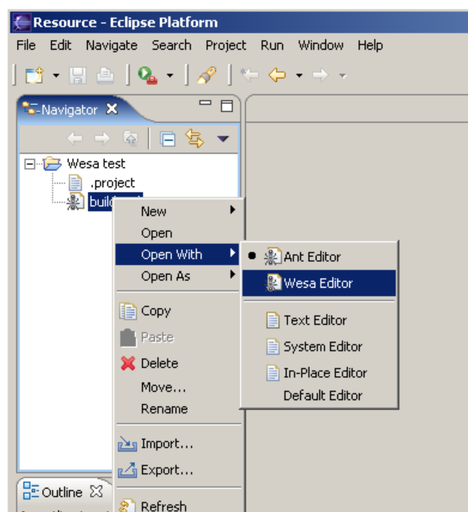
Rysunek 4.5: Tworzenie nowego pliku

2. W celu otwarcia skryptu do edycji klikamy na niego prawym przyciskiem myszy, a następnie wybieramy polecenie `Open with|Wesa editor` jak na rysunku 4.6. Okno otwartego edytora z pustym skryptem przedstawia rysunek 4.7.

3. Klikając na strzałce u góry rozwijamy ją. Dostępne są następujące polecenia (licząc od góry):

   - zaznaczanie,
   - definiowanie zależności między celami,
   - tworzenie celów,
   - tworzenie zadań.

   W naszym skrypcie będą potrzebne trzy cele. Klikamy więc na polecenie tworzenia celów, a następnie trzy razy klikamy w obszarze edytora. Powstaną trzy prostokąty reprezentujące cele, co przedstawia rysunek 4.8.



Rysunek 4.6: Otwieranie edytora

4. Wybieramy teraz na narzędzie służące do zaznaczania. Gdy klikniemy na cel, w dolnej części ekranu ukażą się jego właściwości[3]. W pole `name` wpisujemy „init". Kolejne cele nazywamy „install" i „clean". Możemy też opisać cele wypełniając pole `description` odpowiednim opisem, który będzie później ukazywał się jako podpowiedź po wskazaniu celu myszą, a także stanowił dokumentację skryptu. Sytuację po wykonaniu powyższych czynności prezentuje rysunek 4.9.

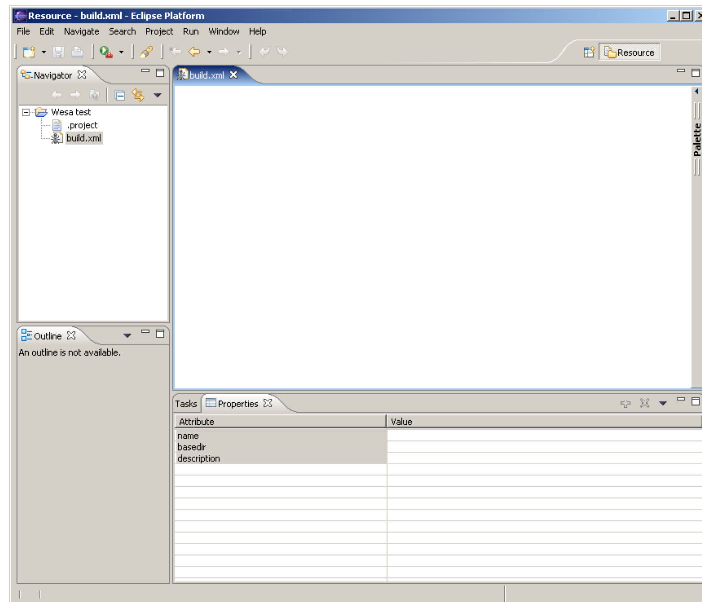
5. Ponieważ kopiowanie plików (wykonywane wewnątrz celu „install") musi odbyć się po stworzeniu katalogów (wykonywane wewnątrz celu „'init'), musimy zdefiniować zależność pomiędzy zadaniami „init" i „install". W tym celu wybieramy narzędzie do tworzenia zależności. Następnie klikamy na zadanie podrzędne (w naszym przypadku „install"), a później na nadrzędne (tym razem „init"). Sytuację po wykonaniu tych czynności przedstawia rysunek 4.10.

6. Teraz przyszedł czas na zdefiniowanie kilku pomocniczych zmiennych, które będą używane w dalszych częściach skryptu. Ponieważ do definicji zmiennej służy zadanie `property`, wybieramy narzędzie do tworzenia zadań, a następnie przy jego pomocy w głównym obszarze edycyjnym tworzymy trzy zadania. We właściwościach tych zadań jako atrybut `name of task` wpisujemy „property"[4]. Aby wizualnie odróżnić zmienne, nadajemy im też etykiety (atrybut `label`). Niech będą to „environment property", „output mp3" i „output avi". Dla „environment property" ustawiamy atrybut `environment` na wartość „env"[5]. Dla „output mp3" ustawiamy atrybut `name` na „mp3dir", a `value` na „./mp3" (dla „output avi" odpowiednio „avidir" i „./avi"). Powinniśmy otrzymać sytuację przedstawioną na rysunku 4.11.

[3]Jeśli okno właściwości (ang. *properties*) nie jest widoczne, to należy wykonać polecenie `Window|Show view|Properties`.)

[4]Warto zauważyć, że po wypełnieniu atrybutu `name of task`, w oknie właściwości pojawiają się nowe atrybuty, a niektóre z nich mają kolor czerwony. Jest to związane z tym, że edytor rozpoznał zadanie i ma zdefiniowaną dla niego składnię, zgodnie z którą atrybuty są źle wypełnione. Po prawidłowym wypełnieniu (wskazówki co do tego znajdują się w pasku stanu) wszystkie atrybuty będą miały kolor czarny.

[5]Jak zostało to opisane w punkcie 1.3, aby móc się odwoływać do zmiennych środowiskowych, należy zdefiniować przedrostek, który umożliwi rozróżnienie odwołań do zmiennych środowiskowych od odwołań do zmiennych ze skryptu.



Rysunek 4.7: Otwarty edytor z pustym dokumentem

7. Kolejną czynnością, jaką należy wykonać, jest zdefiniowanie zadań w ramach celów. W tym celu wybieramy narzędzie służące do tworzenia zadań, a następnie klikamy w obrębie celu. Zacznijmy od celu „init". Tworzymy w nim dwa zadania `mkdir` (jako wartość atrybutu `name of task` wpisujemy „mkdir"). Ważne jest przy tym, aby zadania te były na jednym poziomie. Jeśli przypadkowo zagnieździliśmy zadania w sobie, to można to łatwo naprawić przeciągając zadanie ze środka na zewnątrz. Nadajemy zadaniom wybrane przez siebie etykiety (np. „make avi dir" i „make mp3 dir"), a następnie wypełniamy atrybut `dir` wartościami „${avidir}" i „${mp3dir}" (oczywiście po jednej wartości dla każdego z zadań).

8. Teraz możemy zdefiniować zadania, które wykonają kopiowanie. W celu „install" tworzymy dwa zadania `copy`. W każdym z nich wypełniamy atrybut `todir` wartościami odpowiednio „${avidir}" i „${mp3dir}".

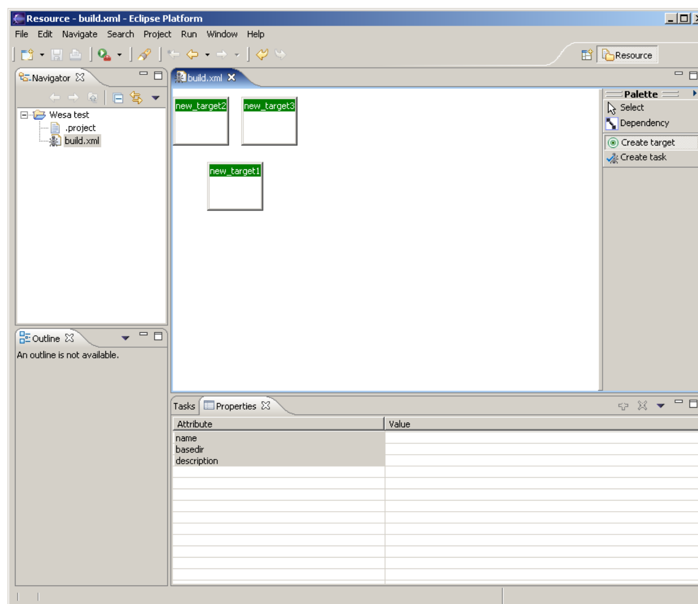
9. Zadania kopiujące wymagają zagnieżdżenia w sobie podzadania, które definiuje zbiór plików do skopiowania[6]. W tym celu w każdym z zadań kopiujących tworzymy zadanie `fileset`. Dla obu wypełniamy atrybut `dir` wartością „${env.INPUT_DIR}". Wypełniamy też atrybut `includes` wartościami odpowiednio „*.avi" i „*.mp3".

10. Pozostał jeszcze do wypełnienia cel „clean". Tworzymy w nim zadanie `delete`. Atrybut `inludeEmptyDirs` ustawiamy na „true"[7]. W zadaniu `delete` tworzymy dwa podzadania `fileset`. W jednym z nich ustawiamy atrybut `dir` na „${avidir}", a w drugim na „${mp3dir}".

11. W tym momencie mamy już gotowy skrypt. Pozostała nam jeszcze ostatnia czynność, a mianowicie zapisanie skryptu na dysk. Z menu `File` wybieramy `Save`. Możemy też

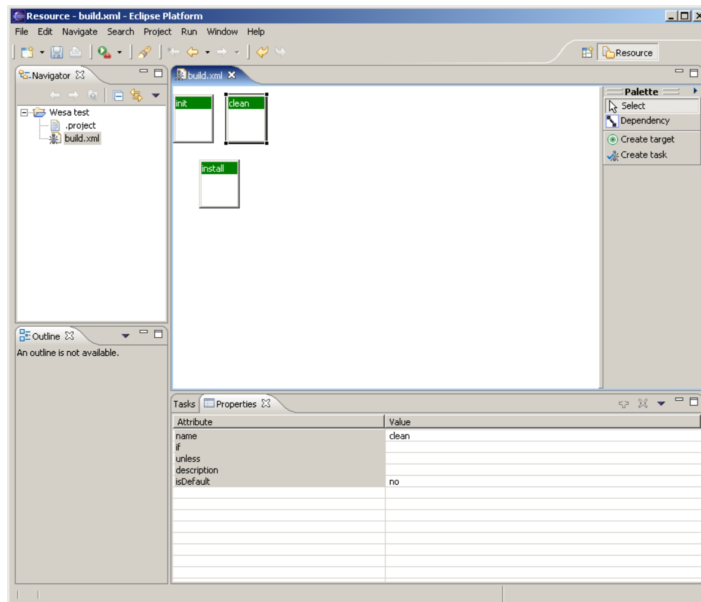
[6] W przypadku kopiowania jednego pliku można wykorzystać atrybut `file`.

[7] Atrybut `inludeEmptyDirs` zadania `delete` ma z góry zdefiniowany zbiór wartości, więc użytkownik nie może wpisywać dowolnych wartości jak to ma miejsce w przypadku innych atrybutów.

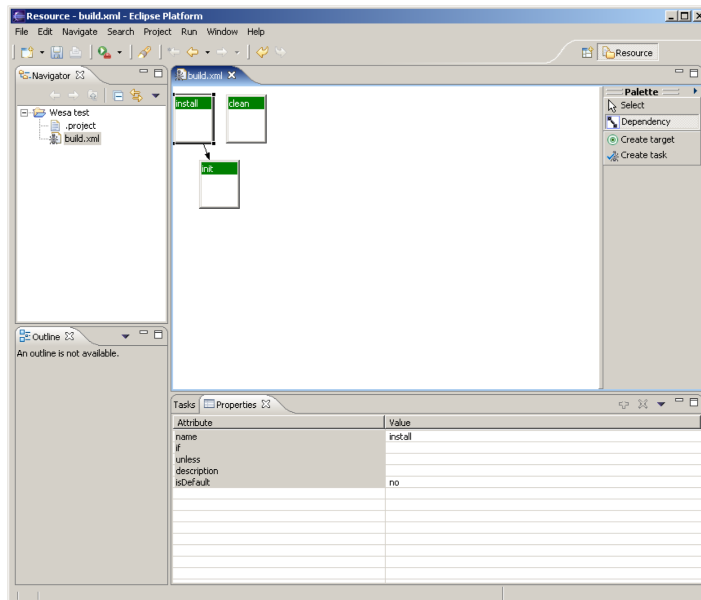


Rysunek 4.8: Skrypt składający się tylko z celów

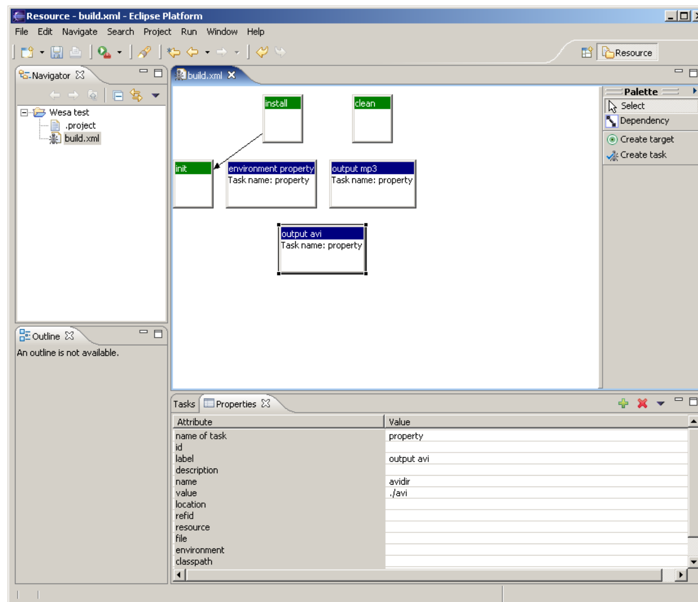
użyć skrótu klawiszowego *Ctrl+S*. Gotowy skrypt (w wersji graficznej) wygląda tak jak na rysunku 4.12.



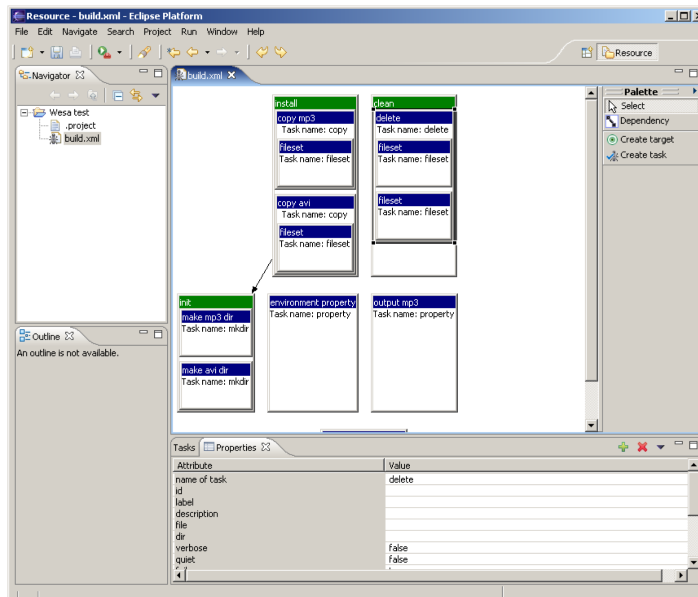
Rysunek 4.9: Nadawanie nazw celom



Rysunek 4.10: Definiowanie zależności pomiędzy celami



Rysunek 4.11: Definiowanie zmiennych



Rysunek 4.12: Gotowy skrypt

Rozdział 5

Zakończenie

Z niniejszej pracy płynie kilka wniosków, ale chyba najważniejszym jest następujący: nawet w obszarach pozornie dobrze zbadanych, w których zdawać by się mogło nie można wiele dodać, jest pole na nowe podejście. Mimo, że Ant jest tak popularny i powstało tak wiele narzędzi wspierających pracę z nim, to żadno z nich nie oferowało możliwości graficznej edycji skryptów. Edytor WESA jest właśnie takim narzędziem. Ma on oczywiście wiele niedociągnięć, takich jak niezbyt bogata baza zdefiniowanych zadań, wynikających chociażby z ograniczeń narzuconych przez ramy pracy magisterskiej (m.in. czasowych), mimo to pokazuje, że idea graficznej edycji skryptów Anta może być z powodzeniem realizowana. Zalety edytora nie kończą się jednak na graficznej edycji. Rzeczą, która odróżnia go od innych, jest możliwość zdefiniowania własnych zadań (oczywiście wraz z opisem restrykcji nakładanych na atrybuty i podelementy, co nie jest spotykane w innych narzędziach), a następnie używania ich w edytorze.

Bardzo pomocny przy tworzeniu edytora był GEF, dzięki któremu możliwe było powstanie działającego edytora w rozsądnym czasie. Sam GEF stanowi również ciekawy temat jako taki i z powodzeniem głębsza jego analiza mogłaby stać się tematem pracy magisterskiej, bądź całkiem pokaźnej książki.

Logika edytora została zrealizowana w duchu funkcyjnym, dzięki czemu większość błędów jakie powstawały wynikały raczej z błędów w zrozumieniu pewnych aspektów GEF-a (dokumentacja do GEF-a nie jest zbyt obszerna, przez co konieczne było zaglądanie do kodu źródłowego w celu zrozumienia pewnych aspektów). Jedynym problemem przy stosowaniu tego podejścia była konieczność ręcznego pisania klas niezmienialnych. I tu otwiera się pole na kolejny temat pracy magisterskiej: stworzenie narzędzia wspomagającego proces generacji klas niezmienialnych. Oczywiście najlepiej, aby takie narzędzie było powszechnie dostępne, aby programiści z całego świata mogli korzystać z dobrodziejstw podejścia funkcyjnego do programowania w Javie.

Dodatek A

Zawartość płyty CD

jpasternak_praca_mgr_2005.pdf Praca magisterska w formacie PDF.

wesa_1.0.0.zip Spakowany edytor WESA wraz ze źródłami.

gef-sdk-3.0.1.zip Zestaw bibliotek GEF wymagany przez edytor WESA.

install.txt Opis sposobu instalowania edytora WESA.

Bibliografia

[Antidote05] *Antidote*, http://ant.apache.org/projects/antidote/, 2005

[ANTMan05] *ANT Manual*, http://ant.apache.org/manual/index.html, The Apache Software Foundation, 2005

[AntRunner04] *AntRunner*, http://antrunner.sourceforge.net/ 2004

[BDR96] Lech Banchowski, Krzysztof Diks, Wojciech Rytter, *Algorytmy i struktury danych*, WNT, 1996

[Belap04] Abhijit Belapurkar, *Functional programming in the Java language*, http://www-106.ibm.com/developerworks/java/library/j-fp.html, 2004

[Bloch01] Joshua Bloch, *Effective Java: Programming Lanuage Guide*, Addisan Wesley, 2001

[CLR98] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Wprowadzenie do algorytmów*, WNT, 1998

[Deacon00] John Deacon, *Model-View-Controler* (MVC) Architecture, 2000

[Eclipse05] *Eclipse*, http://www.eclipse.org/, 2005

[GEFRed04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Phillippe Vanderheyden, *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, IBM, 2004

[GHJV98] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1998

[Goetz03] Brian Goetz, *Java theory and practice: To mutate or not to mutate?*, http://www-106.ibm.com/developerworks/java/library/j-jtp02183.html, 2003

[Grand05] *Grand: Graphical Represantation of Ant Dependencies*, http://www.ggtools.net/grand/

[GraphViz05] *GraphViz*, http://www.graphviz.org/, 2005

[Hanley05] John O'Hanley, *Avoid clone*, http://www.javapractices.com/Topic71.cjp, 2005

[Hudson03] Randy Hudson, *Create an Eclipse-based application using the Graphical Editing Framework*, http://www-106.ibm.com/developerworks/opensource/library/os-gef/, 2003

[JBuilder05] *JBuilder*, http://www.borland.com/jbuilder/index.html, 2005

[Lee03] Daniel Lee, *Display a UML Diagram using Draw2D*,
http://eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html, 2003

[Make00] Free Software Foundation, *GNU Make*, http://www.gnu.org/software/make/, 2000

[Metsker02] Steven John Metsker, *Design Patterns Java Workbook*, Addison Wesley, 2002

[Mills02] Ashkey J.S Mills, *Ant Tutorial*,
http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/ant/ant.html, 2002

[Nazar05] Tomasz Nazar, *Krótki kurs Anta*, http://www.ii.uni.wroc.pl/ nthx/java/ant_pl.htm, 2005

[Neill03] Conor MacNeill, *What is Ant?*, http://codefeed.com/tutorial/ant_intro.html, 2003

[PluginGuide05] IBM, *Platform Plug-in Developer Guide*,
http://download.eclipse.org/eclipse/downloads/drops/S-3.1M6-200504011645/org.eclipse.platform.doc.isv.3.1M6.pdf.zip, 2005 Mode

[rlemaigr04] *GEF Description*,
http://eclipsewiki.editme.com/GefDescription,
http://eclipsewiki.editme.com/GefDescription2, 2004

[Roy03] Peter van Roy, Seif Haridi, *Concepts, Techniques, and Models of Computer Programming*, 2003

[SHNM05] Matthew Scarpino, Stephen Holder, Stanford Ng, Laurent Mihalkovic, *SWT/JFace in Action*, Manning, 2005

[VAE05] Vicken Krissian, Amine Chadly, *Visual Ant Editor*, http://vae.berlios.de/,
http://openfacts.berlios.de/index-en.phtml?title=Visual_Ant_Editor,
http://developer.berlios.de/projects/vae/, 2005

[Vizant02] *Vizant – Ant task to visualize buildfile*, http://vizant.sourceforge.net/, 2002

[Wiki04] *Model View Controller As An Aggregate Design Pattern*,
http://c2.com/cgi/wiki?ModelViewControllerAsAnAggregateDesignPattern, 2004

[Zoio04] Phil Zoio, *Building a Database Schema Diagram Editor with GEF*,
http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html, 2004