Wykorzystanie narzędzia CASE do modelowania projektu RAD

Rafał Biedrzycki
Instytut Systemów Elektronicznych
Politechnika Warszawska
Email: rbiedrzy@elka.pw.edu.pl

Streszczenie. *W artykule przedstawiono koncepcję stanowiącą próbę połączenia zalet i właściwości udostępnianych przez narzędzia CASE z zaletami środowisk wspierających metodologię RAD. Zaproponowano techniki wykorzystujące język UML a pozwalające na modelowanie aplikacji RAD. Dzięki nim możliwe stało się modelowanie interfejsu użytkownika, czyli wyglądu aplikacji. W zasadniczej części artykułu przedstawiono zasady tworzenia modelu, a następnie pokazano proces modelowania aplikacji przykładowej.*

1. Wprowadzenie. Użytkownicy końcowi aplikacji oczekują interfejsu łatwego w użyciu, tak bardzo intuicyjnego, aby możliwa była natychmiastowa praca z programem, bez konieczności tracenia czasu na czytanie instrukcji. Niestety takie interfejsy jest trudno zaprojektować i zaimplementować. Wiele badań wykazuje, iż średnio na obsługę interfejsu użytkownika przeznaczone jest 48% kodu aplikacji i około 50% czasu programistów [2]. Aplikacje stają się coraz prostsze w obsłudze a jednocześnie coraz trudniejsze do utworzenia. Aby umożliwić powstawanie takich aplikacji należy opracować odpowiednie narzędzia wspomagające pracę programistów. Prowadzone są prace badawcze nad tego typu narzędziami. Krótki przegląd powstałych rozwiązań oraz ich możliwości przedstawił w swoim artykule Egbert Schlungbaum [2]. Prowadzone są również badania nad optymalnym doborem elementów interfejsu użytkownika do modelu biznesowego [3]. Prace te mają charakter teoretyczny. Trudno było by zachęcić programistów do zastosowania dowolnego z proponowanych tam rozwiązań. Istnieją również komercyjne rozwiązania np. „Bold for Delphi" firmy Borland [1]. System ten umożliwia połączenie narzędzia RAD z modelem w UML, jednak nie umożliwia modelowania wyglądu.

Wśród tych wszystkich rozwiązań brak jest takiego, które pozwala w prosty sposób modelować zarówno część biznesową aplikacji jak i interfejs użytkownika. Na podstawie takiego modelu powinna powstawać aplikacja gotowa do pracy w maksymalnie możliwym stopniu. Nawet drastyczne zmiany wyglądu interfejsu użytkownika powinny być możliwe dzięki niewielkim zmianom w modelu.

W dalszej części przedstawię próbę opracowania rozwiązania spełniającego postawione wyżej wymagania.

2. Założenia. Do modelowania wykorzystano język UML, ponieważ jest to znany i uznawany standard. Nadaje się on doskonale do modelowania części biznesowej aplikacji. Posiada także wbudowane mechanizmy rozszerzeń, dzięki którym daje się go również zastosować do bardziej zaawansowanych celów. Jako obiekt modelowany obrano projekt w środowisku RAD. Programowanie w tego typu środowiskach polega na odpowiednim doborze komponentów, ustawieniu ich właściwości, zapisaniu w zadanym języku logiki aplikacji. Dzięki takiemu podejściu aplikacja powstaje szybko, jest stosunkowo niezawodna, wymaga wpisania niewielkiej ilości kodu. Połączenie zalet obu tych środowisk dało by bardzo silne i dobre narzędzie. Aby tego dokonać konieczne było opracowanie techniki modelowania. W tym celu poczyniono następujące założenia:

- proces modelowania musi przebiegać szybko, tak aby chcieli go używać projektanci przyzwyczajeni do prostoty i szybkości w osiąganiu wyników jaką dają narzędzia RAD;
- model powinien określać wygląd aplikacji;
- dokładne położenie i rozmiary komponen-

tów powinny zostać ustalone automatycznie na podstawie modelu;

- modelowanie powinno być możliwe w standardowych narzędziach CASE;
- istnieją typowe rodzaje wyglądu aplikacji;
- funkcje biznesowe muszą zostać rozdzielone od funkcji interfejsu użytkownika;
- powinna istnieć możliwość szybkiej zmiany wyglądu aplikacji;
- niektóre komponenty współwystępują z innymi.

3. Podstawy modelowania. W celu zwiększenia możliwości modelowania w UML wykorzystano wbudowane mechanizmy rozszerzeń. Wykorzystano mechanizm definiowania stereotypów oraz mechanizm wartości etykietowanych.
Dzięki użyciu mechanizmu stereotypów wyróżniono dwa rodzaje klas:

- klasy biznesowe – służą do modelowania części biznesowej aplikacji;
- klasy użytkownika – służą do modelowania wyglądu aplikacji.

Dzięki wykorzystaniu mechanizmu wartości etykietowanych zdefiniowano kilka dodatkowych właściwości dla klas. Dla klas biznesowych zdefiniowano dodatkową właściwość o nazwie *View*. Od jej wartości zależy zachowanie systemu generacji. Jedynie klasy, w których nadano wartość tej właściwości znajdą odzwierciedlenie w kontrolkach interfejsu użytkownika. Reguły modelowania oraz generacji kodu dla części biznesowej aplikacji są takie same dla wszystkich generatorów kodu, wobec czego pominę ich opis.

4. Modelowanie interfejsu użytkownika. Aby umożliwić modelowanie interfejsu użytkownika wyróżniono trzy podstawowe rodzaje klas:

- klasy definicji wyglądu;
- klasy definicji typu;
- klasy definicji grupy.

Wykorzystanie ich wszystkich do modelowania nie jest często konieczne. Zależy to od rodzaju interfejsu, jaki chcemy uzyskać. Nie jest również konieczne podawanie wartości dla wszystkich właściwości tych klas. Dzięki wyodrębnieniu typowych układów komponentów udaje się często zaproponować odpowiednie wartości domyślne, co dodatkowo przyspiesza proces modelowania.

4.1. Klasa definicji wyglądu. Klasa definicji wyglądu odpowiada za wybór rodzaju interfejsu użytkownika, sprzężenie z odpowiednią klasą biznesową oraz określenie, które atrybuty znajdą odzwierciedlenie w kontrolkach interfejsu użytkownika. Aby sprostać tym potrzebom zdefiniowano następujące właściwości dla tej klasy:

- *Layout* – określa sposób rozmieszczenia komponentów. Jako wartość przyjmuje napis w formacie: *nazwa rozkładu*(*lista parametrów rozdzielanych przecinkami*). W chwili obecnej dostępne są rozkłady: *Grid, Wizard, MultiWindow.*
- *ShortDesc* – właściwość ta definiuje napis, który pojawi się w pasku tytułowym okna głównego. Jej niezdefiniowanie spowoduje pobranie tytułu okna z nazwy klasy.

Zdefiniowano również dodatkowe właściwości dla atrybutów tej klasy:

- *GroupName* – jest używana dla bardziej zaawansowanych rozkładów komponentów. Pozwala na grupowanie atrybutów w klasie dzięki możliwości nadania każdemu dodatkowej nazwy.
- *ShortDesc* – definiuje napis, który pojawi się jako statyczny tekst opisujący znaczenie pola aktywnego, np. tekst wypełniający komponent *TLabel* a opisujący znaczenie komponentu *TEdit*. Jej niezdefiniowanie spowoduje pobranie opisów z nazwy atrybutu.

Klasa definicji wyglądu powinna zostać sprzężona z odpowiadającą jej klasą biznesową. Aby tego dokonać wystarczy podać jej nazwę we właściwości *View* wybranej klasy biznesowej. Wyboru interesujących nas atrybutów dokonujemy dzięki powtórzeniu ich nazw w omawianej klasie. Dla każdego tak wybranego atrybutu wymagane jest podanie jego typu. Nazwa tego typu może być dowolna, ważne jest jednak aby kojarzyła się ona z jego funkcją oraz aby zdefiniować ten typ na diagramie za pomocą *klasy definicji typu.*

4.2. Dostępne rozkłady. Przewidziano i zdefiniowano trzy podstawowe rodzaje rozkładu komponentów: *Grid, Wizard, MultiWindow.*

Koncepcja typu *Grid* jest analogiczna do odpowiedniego zarządcy rozkładu z języka Java. Pozwala on na rozłożenie komponentów w siatce jednakowych komórek. Jego zdefiniowanie wymaga podania następujących parametrów: liczby kolumn, liczby wierszy, odstępu między komponentami w poziomie, odstępu między komponentami w pionie. Przykładowa definicja *GRID(2,8,100,8)* oznacza, iż komponenty zostaną rozłożone w dwu kolumnach. Będzie osiem wierszy, odstęp poziomy między komponentami wyniesie sto pikseli, a pionowy osiem. Komponenty są rozkładane od strony lewej do prawej, z góry na dół. Dostępna płaszczyzna (forma) jest dzielona na wymaganą liczbę jednakowych komórek (uwzględniając podane odstępy). Do każdej komórki wstawiany jest komponent i jeśli może – przyjmuje jej rozmiary. Kolejność wstawiania komponentów odpowiada kolejności występowania odpowiadających im atrybutów. Dla potrzeb wprowadzenia odstępu zdefiniowano specjalny typ atrybutu: *Skip*. Atrybut o dowolnej nazwie, lecz tego typu zostanie odwzorowany jako puste miejsce w siatce.

Dzięki użyciu typu *Wizard* możemy utworzyć interfejs użytkownika w ostatnio popularnej formie Wizarda. Komponenty zostaną rozmieszczone na odpowiednich zakładkach, powstaną stosowne przyciski nawigacyjne. Za podział komponentów między poszczególne zakładki jest odpowiedzialna właściwość *GroupName*, natomiast wygląd tych zakładek określają *klasy definicji grupy.* Rozkład tego typu wymaga wyspecyfikowania odpowiednich parametrów. Pierwszym parametrem jest żądana liczba zakładek, następnie występują nazwy grup atrybutów, które zostaną przyporządkowane do kolejnych zakładek. Przykładowa definicja *Wizard(3,basic,additional,advanced)* oznacza, iż powstały wizard będzie składał się z trzech zakładek – przy czym na pierwszej z nich znajdą się atrybuty z grupy *basic*, a wygląd tej grupy definiuje klasa definicji grupy o nazwie *basic*, na drugiej zakładce znajdą się atrybuty z grupy *additional*, a wygląd tej grupy definiuje klasa definicji grupy o nazwie *additional* itd.

Typ *MultiWindow* pozwala na tworzenie aplikacji z wieloma oknami. Podobnie jak dla typu *Wizard*, odbywa się grupowanie, ale tym razem pod kątem przydziału do poszczególnych okien. Zaraz po uruchomieniu aplikacji pojawia się okno, którego grupa została zdefiniowana jako pierwsza. Wszystkie grupy i typy należy zdefiniować analogicznie jak dla rozkładu *Wizard*. Przykładowa definicja *MultiWindow(3,basic,additional,advanced)* oznacza, iż powstaną trzy okna. W pierwszym z nich pojawią się reprezentacje atrybutów z grupy *basic*, w drugim pojawią się reprezentacje atrybutów z grupy *additional*, itd. W rozkładzie *MultiWindow* pasek tytułowy każdego z okien wypełniany jest na podstawie nazwy grupy, którą reprezentuje.

4.3. Klasa definicji typu. Klasa definicji typu służy do zdefiniowania użytego wcześniej w klasie definicji wyglądu typu abstrakcyjnego atrybutów. Jest ona również odpowiedzialna za zdefiniowanie rozkładu komponentów dla nie trywialnych typów. Takie podejście pozwala na grupowanie logicznie pewnych atrybutów, nadawanie im jednakowego wyglądu, z możliwością bardzo szybkich zmian jednocześnie dla całej grupy. Aby klasa mogła spełnić swoje zadanie zdefiniowano dla niej następujące właściwości:

- *Look* – określa rodzaj komponentów mających stanowić odwzorowanie definiowanego typu. Obecnie dostępne rodzaje to: *Editable, EditableWithDesc*;

- *Layout* – Definiuje ona wzajemne położenie komponentów służących do graficznej prezentacji danego typu atrybutów.
  Dla typu *EditableWithDesc* własność ta może przyjmować następujące wartości:
  - *LeftLabelled* – tekst z opisem pojawi się z lewej strony komponentu *TEdit*;
  - *UpLabelled* – tekst z opisem pojawi się nad komponentem *TEdit.*

Właściwość *Look* może obecnie przyjmować następujące wartości:

- *Editable* – nastąpi sprzężenie danego atrybutu z komponentem *TEdit*. Oczywiście wszelkie konwersje typów zachodzą automatycznie (np. konieczne, gdy atrybut w klasie biznesowej jest typu *Integer*).
- *EditableWithDesc* – typ podobny do *Editable*, dodatkowo dla każdego atrybutu generowana jest etykieta (label). Tekst w niej widoczny pobierany jest z właściwości *ShortDesc* odpowiedniego atrybutu *klasy definicji wyglądu*. W przypadku nie nadania wartości tej właściwości, opisem staje się nazwa atrybutu.

4.4. Klasa definicji grupy. Klasa definicji grupy jest używana w rozkładach wymagających podziału atrybutów na zbiory rozłączne. Służy ona do definiowania rozkładu komponentów dla grupy atrybutów. Jej nazwa musi odpowiadać nazwie grupy. Podobnie jak dla klasy definicji wyglądu zdefiniowano dla niej następujące właściwości:

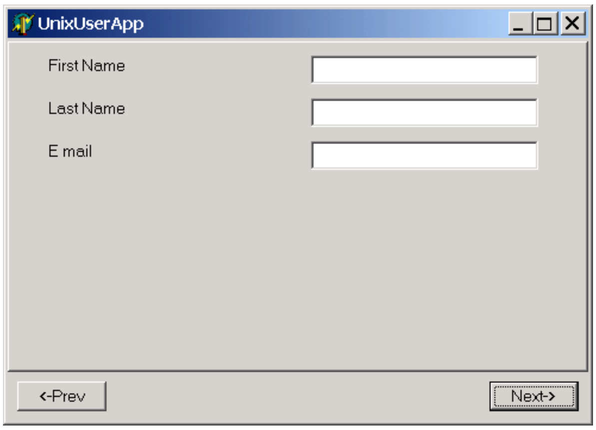
- *Layout* – właściwość ta informuje jaki rodzaj rozkładu komponentów należy zastosować dla tej grupy. Dostępne są wszystkie wcześniej omawiane rozkłady komponentów;
- *ShortDesc* – właściwość ta definiuje napis, który stanowi opis grupy. Jej niezdefiniowanie spowoduje pobranie tego opisu z nazwy grupy.

5. Przykład. Aby lepiej zrozumieć sposób modelowania przedstawię w piętnastu punktach proces tworzenia aplikacji testowej służącej do zakładania konta w systemie unixowym:

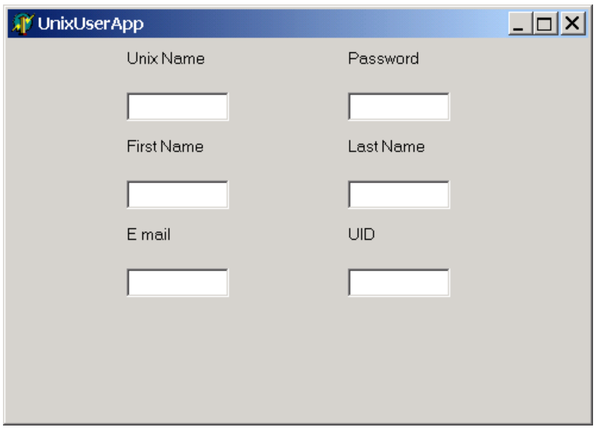
1. Utworzyć nowy diagram klas.
2. Umieścić na diagramie klasę typu *business*, nadać jej nazwę, np. *New Unix User*.
3. Utworzyć następujące atrybuty:
    - *Unix Name* – identyfikator użytkownika, np. rbi;
    - *Password* – hasło;
    - *First Name* – imię;
    - *Last Name* – nazwisko;
    - *E-mail* – adres poczty elektronicznej;
    - *UID* – unikalny, liczbowy identyfikator użytkownika.
4. Dla atrybutu *UID* ustawić typ atrybutu na *Integer*. Dla pozostałych atrybutów, dla uproszczenia, ustawić typ *String*.
5. Utworzyć klasę typu *user*, nadać jej nazwę – np. *UnixUserApp*;
6. Ustawić właściwość *View* klasy biznesowej na *UnixUserApp*;
7. Do pustej jeszcze *klasy definicji wyglądu*, należy dodać atrybuty o nazwach odpowiadających atrybutom z *klasy biznesowej*.
8. Dla atrybutu *UID* ustawić typ *Identyfikatory*, dla pozostałych atrybutów typ *Napisy*.
9. Zdefiniować na diagramie *klasy definicji typu* o nazwach: *Identyfikatory, Napisy*.
10. W klasach definicji typu należy ustawić właściwość *Look* na *EditableWithDesc* oraz właściwość *Layout* na *LeftLabelled*.
11. Zgrupować atrybuty według ich przynależności do poszczególnych zakładek *wizarda*. W tym celu atrybutom *Unix Name, Password* ustawić właściwość *GroupName* na np. *Basic*. W analogiczny sposób atrybutowi *UID* przyporządkować grupę o nazwie *advanced*, a pozostałym grupę *additional*.
12. W *klasie definicji wyglądu* zdefiniować właściwość *Layout*. Ponieważ chcemy uzyskać wygląd typu *wizard* z trzema zakładkami, to jako wartość tej właściwości należy podać: *Wizard(3,basic,additional,advanced)*.
13. Zdefiniować zachowanie komponentów na zakładkach wizarda. W tym celu umieścić na diagramie *klasy definicji grupy*, nadając im nazwy odpowiadające nazwom zakładek. Nazwy te wystąpiły w poprzednim punkcie podczas definiowania rozkładu komponentów. Tworzymy klasy *basic, additional, advanced*.
14. Ustawić w ostatnio utworzonych klasach właściwość *Layout* na *Grid(2,8,100,8)*.

15. Uruchomić proces generacji kodu.

Wykonanie wielu z pośród wymienionych wyżej czynności nie było konieczne, ponieważ system zaproponowałby odpowiednie ustawienia jako wartości domyślne. Wygląd tak powstałej aplikacji zobaczyć można na rysunku 1. Na rysunku 2 zobaczyć można inny wariant wyglądu przy zachowaniu tej samej funkcjonalności. Tak duże zmiany oblicza aplikacji powstały dzięki niewielkim zmianom na diagramie.



Rysunek 1: Wygląd aplikacji przykładowej.



Rysunek 2: Inny wariant wyglądu aplikacji przykładowej

6. Zalety i wady. Każdy pomysł ma swoje mocne i słabe strony. W tym rozdziale podjęto próbę analizy zalet i wad koncepcji, jak również możliwości przyszłych ulepszeń.

Do wad koncepcji niewątpliwie zaliczyć należy ograniczony podzbiór możliwych do uzyskania wyglądów – przewidziano jedynie część ze zbioru możliwych rozkładów komponentów. Niestety nigdy nie uda się odgadnąć wszystkich przydatnych ich kombinacji. Można jednak, z powodzeniem, przewidzieć najczęściej stosowane i zalecane. Wprowadza to pewną standaryzację interfejsów użytkownika – co jest zaletą, ponieważ użytkownik w każdym programie znajdzie podobny układ komponentów, dzięki czemu skróci się czas potrzebny na opanowanie nowego programu.

Do zalet koncepcji zaliczyć można rozdzielenie funkcji biznesowych generowanej aplikacji od funkcji interfejsu użytkownika. Żadna z danych używanych w aplikacji nie jest przechowywana w kontrolce graficznej – ma własną reprezentację w odpowiedniej klasie biznesowej, a w razie potrzeby informacja ta jest uzgadniana z kontrolkami interfejsu użytkownika. Najważniejszą zaletą pomysłu jest możliwość modelowania wyglądu aplikacji. Kolejną zaletą pomysłu jest niewątpliwie duża szybkość modelowania. Dzięki wyróżnieniu najczęstszych układów komponentów modelowanie przebiega bardzo szybko. W związku z tym bardzo szybko otrzymujemy działający i niezawodny szkielet aplikacji. Należy także podkreślić łatwość wprowadzania zmian. Zmiana charakteru wyglądu najczęściej odbywa się dzięki zmianie tylko jednej wartości w modelu.

Nie bez znaczenia dla łatwości użycia i czytelności diagramów jest to, jakiego narzędzia CASE użyjemy. W tym projekcie użyto Select Enterprise w wersji 6. Narzędzie to wyświetla bezpośrednio na diagramie tylko klasy oraz nazwy atrybutów i metod. Dużo lepiej w tym względzie zachowuje się RationalRose. Należałoby również zastanowić się nad opracowaniem własnego narzędzia CASE bardziej eksponującego mechanizmy rozszerzeń UML, np. wartości etykietowane. Nie jest to wbrew pozorom takie trudne, tym bardziej, że można znaleźć w Internecie narzędzia CASE, o otwartym kodzie źródłowym. Należałoby również zastanowić się nad wzbogaceniem listy dostępnych typów i rozkładów. Dobrym pomysłem wydaje się utworzenie edytora rozkładów za pomocą którego każdy użytkownik mógłby wzbogacać możliwości systemu. Innym ciekawym problemem godnym zastanowienia jest podjęcie próby generacji odwrotnej, czyli tworzenia modelu na podstawie gotowej aplikacji.

7. Podsumowanie. Opracowano sposób reprezentacji projektu RAD w UML, z uwzględnieniem możliwości modelowania interfejsu użytkownika. Dzięki, mającemu charakter pionierski, użyciu języka UML do modelowania aplikacji RAD, udało się opracować system charakteryzujący się prostotą w modelowaniu, dużą elastycznością tego procesu, natychmiastową gotowością do uruchomienia kodu zaraz po generacji oraz możliwością bardzo szybkiego przeprowadzenia zmian w wyglądzie docelowej aplikacji.

Modelowanie połączone z językami wysokiej generacji daje nowe możliwości, zwiększa stabilność rozwiązania, skraca czas między pomysłem a jego implementacją, a dzięki zastosowaniu diagramów i notacji graficznej rozwiązania stają się samodokumentujące i zrozumiałe. Zalety te oraz nowe możliwości zostały już zauważone przez czołowych światowych producentów narzędzi dla programistów, takich jak TogetherSoft czy Borland. Wprowadzili oni możliwość modelowania do swoich narzędzi, zdefiniowali sposoby przejścia od modelu do działającej aplikacji. Użycie tych mechanizmów jest obecnie jeszcze opcjonalne, jednak zyskują one coraz większą popularność i mogą stać się standardem.

Rozwiązania takie mają przed sobą przyszłość i dlatego należy prowadzić w tym kierunku dalsze badania.

Literatura

[1] *„Starting with Bold for Delphi/Bold for C++"*, Anthony Richardson, *http://www.viewpointsa.com*

[2] Egbert Schlungbaum *„Model-based User Interface Software Tools Current state of declarative models"*, November 1996

[3] François Bodart, Jean Vanderdonckt *„On the Problem of Selecting Interaction Objects"*, August 1994