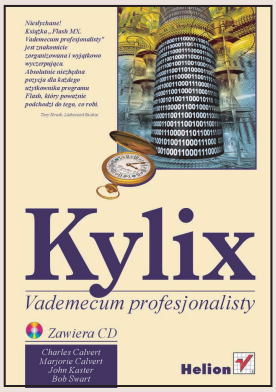
Kylix. Vademecum profesjonalisty



Autorzy: Charles Calvert, Marjorie Calvert, John Kaster, Bob Swart
Tłumaczenie: Jakub Mirek (rozdz. 1 – 10, 12 – 21), Andrzej Grażyński (rozdz. 11)
ISBN: 83-7197-701-8
Tytuł oryginału: Kylix Developer
Format: B5, stron: 808

Stworzenie przez firmę Borland Kyliksa, linuksowego odpowiednika popularnego Delphi, to jeden z kamieni milowych w rozwoju tego systemu operacyjnego. Programiści piszący aplikacje działające w systemie Linux otrzymali zintegrowane środowisko typu RAD, oparte na sprawdzonym języku ObjectPascal, umożliwiające „programowanie wizualne". Ich praca stała się odtąd szybsza i bardziej efektywna.

Książka „Kylix. Vademecum profesjonalisty" to kompletny przewodnik po Kyliksie zawierający zarówno informacje o środowisku programistycznym, jak też pełny opis języka ObjectPascal i bibliotek CLX. Nie zabrakło również prezentacji systemu Linux i jego specyficznych właściwości, która może przydać się osobom znającym już Delphi i rozpoczynającym pracę w nowym systemie operacyjnym.

Książka podzielona jest na pięć części:

1. Prezentacja Delphi i Linuksa — podstawy środowiska RAD, opis ObjectPascala, programowanie obiektowe, środowisko X-Window
2. Opis biblioteki CLX — architektura CLX, palety, tworzenie komponentów, programowanie grafiki w Kyliksie
3. Programowanie systemowe w Linuksie — aplikacje konsolowe, procesy i wątki
4. Programowanie bazodanowe w Kyliksie — DataCLX, sterowniki dbExpress, tworzenie przykładowej aplikacji
5. Tworzenie aplikacji WWW — serwer Apache, komponenty WebBroker

Niezależnie od tego, czy znasz już Delphi i chcesz przenieść swoje umiejętności do środowiska Linux, czy też przeciwnie, znasz dobrze Linuksa i pragniesz szybko tworzyć zaawansowane aplikacje, znalazłeś właśnie właściwy podręcznik.

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

Spis treści

Rozdział 1.

Programowanie wizualne

Charlie Calvert

Dzięki lekturze tego rozdziału zapoznasz się z narzędziami programistycznymi środowiska Kylix. Opisano tu IDE (zintegrowane środowisko programistyczne), narzędzie projektanta formularzy oraz kilka prostych sztuczek, które można wykorzystać w procesie wizualnego programowania.

Pod koniec tego rozdziału będziesz wiedział już co nieco o tym, jakiego rodzaju programy możesz tworzyć w Kyliksie, a także jak łatwo je tworzyć używając narzędzi zawartych w IDE.

Doświadczeni programiści Delphi dowiedzą się, jakie zmiany należy wykonać by przenieść ich aplikacje napisane w Object Pascalu ze środowiska Windows do Kyliksa. IDE Kyliksa jest właściwie po prostu IDE Delphi 5 przeniesionym do Linuksa. W rezultacie oba te środowiska programistyczne są bardzo podobne — chociaż nie identyczne. Przez cały czas będę podkreślał dzielące je różnice; podpowiem także Czytelnikom tworzącym dotychczas swoje programy na platformie Windows, co zrobić, by poczuli się komfortowo w świecie Linuksa.

IDE Kyliksa

IDE Kyliksa może być dostosowane do Twoich potrzeb. Możesz na nowo rozplanowywać, modyfikować, dodawać i usuwać oraz generalnie rekonfigurować paski narzędzi tak, aby odpowiadały Twojemu stylowi programowania.

Najbardziej oczywistą cechą IDE jest to, iż pozwala „przeciągać i upuszczać" wizualne obiekty (takie jak pola edycji czy listy wyboru) wprost na formularz. Obiekty te można potem modyfikować i dowolnie układać, tworząc tym samym potężny interfejs dla swojej aplikacji.

Sam edytor obsługuje wiele różnych układów klawiszy; możesz ponadto zmieniać ustawienia kolorów a także rozmiar i rodzaj czcionki. Konfiguracji można także poddać mnóstwo innych parametrów edytora, włącznie z ustawieniami tabulacji, grubością ramek, trybem wstawiania, wyróżnianiem składni i wieloma innymi cechami, zbyt licznymi, by je tu przytaczać.

Pracując z edytorem możesz otrzymać natychmiastową pomoc kontekstową na temat metod, których używasz. Przydatne informacje mogą pochodzić z samego systemu pomocy lub z narzędzia o nazwie Code Insight, które pokazuje parametry dla wywoływanej metody lub listę metod dostępnych dla konkretnego obiektu. Code Insight można wykorzystać nie tylko w stosunku do kodu CLX API, lecz także dla tego, który jest właśnie pisany.

Nawet system menu Kyliksa może być rozszerzony. Możliwe jest dodanie do menu nowych pozycji, które pozwalają uruchamiać inne aplikacje. Na przykład — jak przekonasz się jeszcze w tym rozdziale — można w łatwy sposób tak skonfigurować IDE, żeby źródło, nad którym pracujesz, otwierane było w emacs lub jakimś innym edytorze.

W tym rozdziale przedstawiono wszystkie wymienione dotychczas funkcje IDE. Z dalszej części książki dowiesz się więcej na temat cech IDE; niektóre z nich, bardziej zaawansowane, zostaną omówione znacznie dokładniej. W tym rozdziale na przykład omówię pokrótce, jak przeciągać komponenty na formularz. Jednak dla pełnego zrozumienia komponentów i ich relacji z IDE trzeba będzie poznać znacznie więcej (niekiedy bardzo złożonych) szczegółów, w związku z czym większość Części II, zatytułowanej „CLX", przeznaczę na nauczenie Cię tego, jak budować własne komponenty.

Całe IDE może zostać rozszerzone przez użycie szeregu obiektów nazwanych Tools API. Te procedury pozwalają właściwie na dodanie nowych funkcji do IDE i na znaczące poszerzenie już istniejących. Tools API omówię w rozdziale 8. — „Pakiety i kod współdzielony".

Używam IDE Delphi już od bardzo dawna; także IDE Kyliksa wykorzystuję niezwykle aktywnie znacznie dłużej, niż jest ono w ogóle użyteczne dla szerszej rzeszy użytkowników. Dzięki temu znam mnóstwo sztuczek pozwalających wykorzystać ich zalety. Postaram się przedstawić ich tutaj tak wiele, jak tylko to możliwe.

Plan ataku

Być może zaczynasz już przeczuwać, jak bardzo skomplikowane jest IDE Kyliksa. Musisz pamiętać, że jest to niezwykle potężne narzędzie. Nawet ci, którzy przez lata używali Delphi, mogą po prostu nie znać wszystkich jego funkcji.

Na szczęście, wiele cech IDE jest już względnie znanych doświadczonym użytkownikom. Stosowane w nim konwencje i metafory są po prostu takie same jak te, które można odnaleźć w programach StarOffice, Microsoft Office, Visual Basic, Visual C++, KDeveloper i w wielu innych powszechnie używanych narzędziach. Co więcej, nawet te bardziej zaawansowane a mniej znane funkcje są zwykle w miarę intuicyjne i łatwe do zrozumienia.

Spróbuję omówić tu wszystkie najważniejsze problemy, aby umożliwić Ci poznanie najważniejszych cech środowiska. Poświęcę również trochę czasu na wyszczególnienie kluczowych lub bardziej użytecznych własności, które mogłyby nie być oczywiste dla nowych użytkowników Kyliksa.

Zacznę od omówienia pasków narzędzi, pokazując, jak je przestawiać i konfigurować. Potem skupię się na podstawach wizualnego programowania, zwracając uwagę na techniki używania komponentów i projektowania interfejsów. Następnie skieruję uwagę na edytor i na to, jak go skonfigurować. Wykażę, jak bardzo może on być pomocny w przeglądaniu

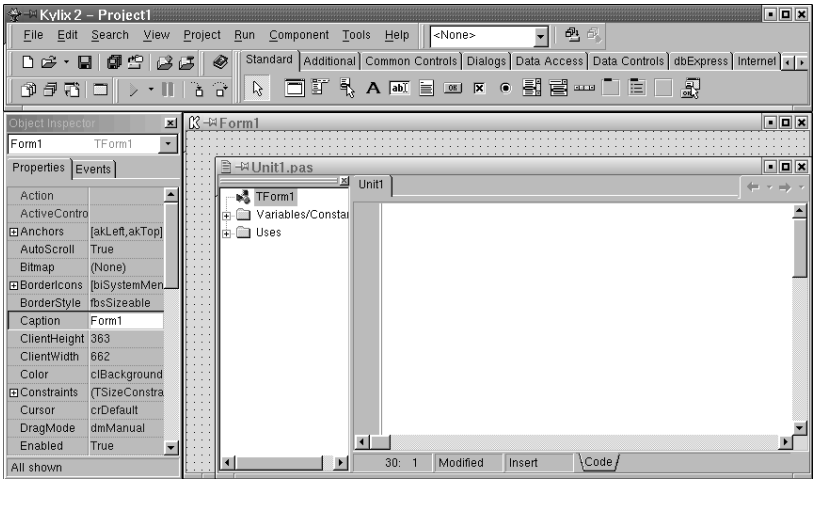
źródeł, w znajdowaniu metod i parametrów, których potrzebujesz; będziesz mógł również przekonać się, że zamienia on pisanie kodu w stosunkowo przyjemny i intuicyjny proces. Po omówieniu wszystkich ważniejszych tematów zajmę się jeszcze pokrótce scharakteryzowaniem niektórych z mniej istotnych, lecz dodających mocy cech IDE — kreatorów, ramek i wywołań zewnętrznych narzędzi; wspomnę także o systemie pomocy.

Choć nie wszystkie funkcje IDE są oczywiste dla początkujących, żadna z nich nie jest szczególnie trudna do zrozumienia. Z tego względu uporanie się ze wszystkimi zawartymi w tym rozdziale informacjami nie powinno być szczególnie wielkim wyzwaniem. Ja ze swej strony przez cały czas będę starał się dawać rady użyteczne dla programistów Javy, VB i Visual C++, którzy mogliby być zainteresowani wypróbowaniem nowej technologii. Poza tym będę także pamiętał o tych użytkownikach edytora emacs (czy vi), którzy chcieliby zapuścić się na to stosunkowo nowe i prawdopodobnie nieco podejrzane z ich punktu widzenia terytorium; postaram się zamieścić tak wiele kierowanych do nich wskazówek, ile to tylko możliwe. Mam nadzieję, że wszyscy nowi użytkownicy Kyliksa polubią to potężne i niezwykle użyteczne, a zarazem nie sprawiające wielkich kłopotów w obsłudze narzędzie.

Ten rozdział jest tematycznie bardzo zbliżony do rozdziału 5., zatytułowanego „Edytor i debugger". Razem przedstawiają one całe IDE Kyliksa. W połączeniu z pozostałymi rozdziałami z części I — „Poznanie Delphi i Linuksa" — pozwalają natomiast poznać funkcjonowanie Object Pascala.

Pierwsze spojrzenie na IDE

Aby ułatwić Ci orientację w nowym, nieznanym Ci jeszcze programie, na rysunku 1.1 przedstawiono główne części IDE. Są to menu, narzędzie *inspektora obiektów* (*Object Inspector*), *paleta komponentów* (*Component Palette*), edytor, narzędzie *projektanta formularzy* (*Form Designer*) oraz paski narzędzi.



Rysunek 1.1. *Główne składniki IDE*

W kolejnych partiach tej książki będę starał się przedstawić Ci każdy z tych składników, omawiając je wszystkie możliwie najbardziej szczegółowo. Jednakże może się zdarzyć, że jeszcze zanim nadejdzie czas na pełne wytłumaczenie jakiegoś zagadnienia, będę bez uprzedniego przygotowania nawiązywał do niektórych z nich. W takich przypadkach możesz spojrzeć właśnie na rysunek 1.1, aby zobaczyć, którą część IDE omawiam. Oczywiście, wkrótce potem otrzymasz także wszystkie niezbędne do pełnego zrozumienia problemu informacje.

Menu

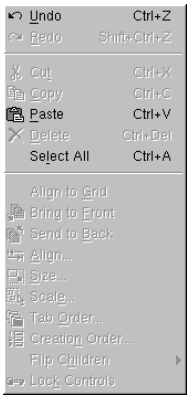
Wydaje się, że dość dobrym miejscem do rozpoczęcia naszej podróży po Kyliksie jest menu. Wprawdzie Ci, do których jest adresowana ta książka, prawdopodobnie są w stanie sami je rozpracować, jednak jest kilka rzeczy, o których warto tu wspomnieć; mogą być one przydatne przynajmniej niektórym Czytelnikom.

Menu służy do wykonywania szerokiej gamy zadań, włącznie z otwieraniem i zamykaniem plików, manipulowaniem debuggerem i dostosowywaniem środowiska. Domyślnie wszystkie funkcje IDE są dostępne poprzez menu.

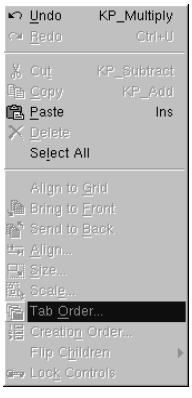
Klawisze skrótów w menu

Dzięki temu, że menu może być obsługiwane za pomocą konfigurowalnych skrótów klawiszowych, daje ono bardzo szybki dostęp do funkcji środowiska Kyliksa. Przy każdym poleceniu znajdziemy w nim odwołanie do skrótu klawiaturowego, którym można to polecenie wywołać; dzięki temu w łatwy sposób można wykonywać rutynowe zadania, takie jak zapis plików i projektów, czy kompilacja i uruchamianie aplikacji.

Po otworzeniu menu *Edit* zobaczysz to, co pokazano na rysunkach 1.2 i 1.3. Zatrzymaj się na chwilę i porównaj te dwa obrazki. Na pierwszy rzut oka wydają się identyczne. Jednak po bliższym przyjrzeniu się im zapewne zauważysz, że klawisze skrótów skojarzone z daną pozycją różnią się. Co jest tego przyczyną?



Rysunek 1.2.
Menu Edit w wersji default key mappings



Rysunek 1.3.
Menu Edit w wersji brief key mappings

Aby zrozumieć, o co tu chodzi, otwórz menu *Tools*, wybierz *Editor Options*, a następnie *Key Mappings*. Zmień *Key Mapping Module* z *Default* na *Brief*, następnie zaś na *Epsilon*. Po każdej zmianie kliknij *OK* i otwórz menu *Edit*. Jak sam widzisz, skróty klawiszowe dla każdej pozycji zmieniały się, w zależności od wybranego odwzorowania klawiszy.

Możesz stworzyć swoje własne ustawienia klawiszy i używać swoich własnych skrótów. W rozdziale 8. pokażę Ci jak to zrobić.

Linux

Jeśli jesteś użytkownikiem emacs lub vi, dla którego zintegrowane środowisko jest czymś nowym, zachęcam Cię do poświęcenia trochę czasu na zaznajomienie się z systemem menu. Standardowe ustawienia klawiszy pozwalają na otwieranie menu przez naciśnięcie klawisza *Alt* z pierwszą literą nazwy menu. Na przykład skrót *Alt+F* otwiera menu *File*; po jego otwarciu możesz wybrać pozycję z menu przez naciśnięcie klawisza z tą literą, która jest w tej pozycji podkreślona, np. *S* dla zapisu (*save*) czy *C* dla zamknięcia (*close*).

Menu podręczne

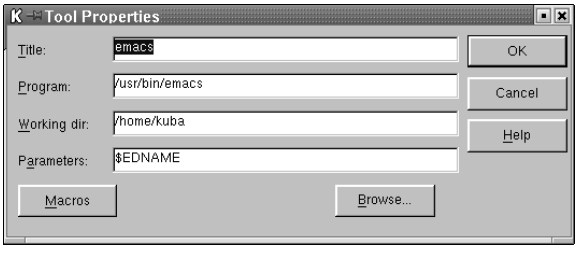
Kolejną kluczową cechą systemu menu Kyliksa są menu podręczne (ang. *pop-up menus*), wywoływane kliknięciem prawym przyciskiem myszy. Właściwie klikanie prawym przyciskiem myszy po to, żeby przywołać menu podręczne, jest częścią ideologii Borlanda, ponieważ to właśnie ta firma była pionierem na tym polu. Generalnie rzecz biorąc, gdy tylko jesteś w IDE Kyliksa i nie za bardzo wiesz, co dalej zrobić, spróbuj kliknąć prawym przyciskiem na czymkolwiek, a być może znajdziesz odpowiedź. Dla wszystkich ważniejszych okien w IDE opracowano różne i często całkiem złożone menu podręczne. Własne menu posiadają np. projektant formularzy, inspektor obiektów, edytor, paleta komponentów i pasek narzędzi. Nigdy zatem nie zapominaj zatem o klikaniu *prawym przyciskiem myszy*!

Dodawanie nowych narzędzi do menu

A oto kolejna niezwykle użyteczna właściwość systemu menu. Otwórz menu *Tools* i wybierz *Configure Tools*, po czym kliknij *Add*. W polu *Title* wpisz `emacs` (jeśli emacs nie jest zainstalowany w Twoim systemie, wybierz inny program, którego chcesz używać — np. `vi`, `kwrite` czy `joe`). W polu *Program* wpisz `/usr/bin/emacs` lub odpowiednią ścieżkę do innego, używanego przez Ciebie zamiast niego programu. W polu *Working dir*

wpisz nazwę swojego katalogu domowego, np. `/home/ccalvert`. Kliknij *Macros*, przewiń listę i wybierz *$EDNAME*. To makro przekaże do programu pełną ścieżkę pliku będącego na wierzchu w edytorze. Kliknij *OK* w oknie dialogowym *Tools Property*, a następnie przycisk *Close* w oknie *Tools Option*. Na rysunku 1.4 pokazano, jak powinno wyglądać poprawnie uzupełnione okno dla standardowej instalacji Red Hata.

Rysunek 1.4. *Okno dialogowe Tool Properties*



Otwórz menu *Tools*. Powinieneś teraz zobaczyć wpis *emacs* jako jedną z pozycji do wyboru w menu. Jeśli go tam nie ma lub jeśli jest zablokowany, wróć do menu *Tools* i powtórz czynności opisane w poprzednim akapicie.

Aby przetestować nową pozycję w menu, potrzebujesz jakiegoś pliku do edycji. Otwórz jakiś już istniejący projekt, żeby mieć pewność, że masz prawidłowy plik (możesz np. wejść do katalogu */Kylix/demos* i otworzyć projekt *BasicEd*. Otwórz *View*, wybierz *Units*, a następnie *BasicEd1*). Gdy zobaczysz w edytorze Kyliksa kod z zapisanego pliku, wybierz kolejno *Tools* i *emacs*. Powinien uruchomić się edytor emacs; możesz zatem zobaczyć swoje Pascalowe źródło w klasycznym edytorze GNU. Nowsze wersje edytora emacs powinny nawet automatycznie podświetlać składnię. Jeśli edytujesz w tym właśnie programie i zapiszesz zmiany, będą one zachowane, gdy wrócisz do IDE Kyliksa.

To wszystko — przynajmniej na razie — co chciałbym powiedzieć o systemie menu Kyliksa. Będę jeszcze wracał do tego tematu podczas omawiania innych niezwykle ważnych narzędzi, jak choćby debugger czy edytor. Skupię się wtedy na tych częściach menu, które będą związane z omawianym tematem.

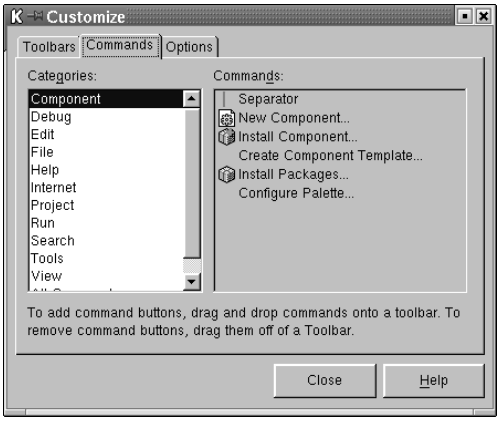
Paski narzędzi

Paski narzędzi standardowo pojawiają się bezpośrednio pod menu głównym oraz na prawo od niego. Klikając na nich, można zrealizować większość zadań, które da się wykonać przy użyciu menu i klawiatury. Paski są ponadto prostsze od menu w dostosowywaniu. Wyłącznie od Ciebie zależy, czy do wykonywania pewnych zadań będziesz używał menu, czy pasków narzędziowych.

Gdy klikniesz na pasku menu prawym przyciskiem myszki, pojawi się odpowiednie menu kontekstowe. Możesz ukryć niemal wszystkie paski narzędziowe, po prostu je odznaczając w wyświetlonym menu. Oprócz tego zauważysz pewnie, że z lewej strony każdego paska znajduje się uchwyt (właściwie wygląda on bardziej jak miniaturowy zderzak, niż uchwyt). Za jego pomocą możesz „złapać" pasek. Aby to zrobić, po prostu kliknij uchwyt; teraz, nie zwalniając klawisza myszki, przeciągnij pasek w miejsce, w którym chciałbyś go umieścić. Uchwyt może zostać użyty nie tylko do przemieszczenia paska, ale także do jego zupełnego usunięcia z pierwotnego położenia. Jeśli przeciągniesz pasek poza główne okno IDE, będzie się on unosił ponad pozostałymi oknami i nie zniknie za nimi.

Do paska narzędziowego można także dodawać poszczególne elementy. Kliknij na pasku prawym przyciskiem myszy. Wybierz pozycję *Customize* (Dostosuj), znajdującą się na samym dole wyświetlonego menu kontekstowego. W efekcie pojawi się okno dialogowe *Customize*. Będzie ono otwarte na zakładce *Toolbars*, z której możesz wybrać interesujące Cię paski narzędzi (możesz także ukryć te, których nie będziesz używał). Przenieś się teraz do zakładki *Commands*, klikając na niej. Jak widać na rysunku 1.5, jest ona podzielona na dwie części; po lewej stronie znajdują się kategorie komend, zaś po prawej same komendy. Gdy klikniesz jakąś kategorię, np. *Debug*, po prawej stronie pojawią się różne komendy należące do wybranej kategorii. Teraz możesz kliknąć dowolną z ikon komend i przeciągnąć ją na pasek. Dla wypróbowania tego przeciągnij w ten sposób ikonę *Breakpoints* z kategorii *Debug*; w efekcie ikona ta pojawi się na pasku narzędzi. Rysunki 1.5 i 1.6 ilustrują to, o czym była tu mowa.

Rysunek 1.5.
Zakładka Commands w oknie Customize



Rysunek 1.6. *Dwa obrazki tego samego paska narzędzi — pierwszy taki, jaki pojawia się standardowo, drugi zaś z kilkoma dodanymi do niego ikonami*

Możesz także usuwać to, co znajduje się na pasku. Ja na przykład rzadko używam przycisków *Step Into*, *Step Over* oraz *Pause* z paska *Debug*. W związku z tym zwykle usuwam je, umieszczając w ich miejscu ikony *Cut*, *Copy* i *Paste* z kategorii *Edit*. Lubię także mieć ikony *Compile* i *Build* z kategorii *Project* na pasku narzędzi Desktops, który znajduje się na prawo od menu. Oczywiście, nie musisz usuwać bądź dodawać dokładnie te same ikony co ja; piszę tu o tym jedynie po to, byś był świadomy otwartości środowiska na podobne działania.

Programowanie wizualne

Teraz, gdy mamy już za sobą wstępne informacje, nadszedł czas, by zgłębić tajniki programowania wizualnego. Temat ten stanowi oczywiście istotę programowania w Kyliksie. Techniki tworzenia wizualnego są największą korzyścią, jaką Kylix przynosi programistom i to właśnie one pozwalają zaoszczędzić mnóstwo czasu (prawdopodobnie drugą

tak istotną rzeczą jest debugger, który będzie omówiony w rozdziale 5.). Zanim zagłębimy się w szczegóły, niech będzie mi wolno dodać jeszcze parę słów wstępu. Programowanie wizualne jest wciąż nową ideą i podobnie jak wszystkie nowe idee spotyka się z oporem ludzi niechętnych zmianom. Chciałbym poświęcić jeszcze chwilę na obalenie kilku zarzutów, które stawia się najczęściej wobec tej technologii.

- **Zarzut pierwszy**: *Wizualne tworzenie programów może się wydawać czymś na kształt oszukiwania. Wydaje mi się, że coś jest nie tak ze sprowadzaniem skomplikowanych zadań do tak prostych czynności.* Prawdziwym celem programistów jest znalezienie jak najlepszego sposobu wykonania danego zadania. Wiadomo, że większość zdrowych ludzi, mając dość czasu, potrafiłaby przejść drogę z San Francisco do Nowego Jorku. Jednakże w większości przypadków — szczególnie wtedy, gdy dotyczy to biznesu — przelot samolotem okazuje się tańszy, bezpieczniejszy i bardziej wskazany, gdy chodzi o przemierzanie kontynentu. Podobnie programowanie wizualne dostarcza oszczędnych i wydajnych sposobów tworzenia interfejsu dla większości aplikacji. Nie ulega wątpliwości, że każdy może stworzyć ten sam interfejs pisząc kod w edytorze Kyliksa, ale łatwiejszym, prostszym i zwykle bezpieczniejszym sposobem jest zbudowanie go przy użyciu narzędzi wizualnych.

- **Zarzut drugi**: *Słyszałem, że wizualne komponenty są duże i nieporęczne, a ponadto nie mogą być włączone bezpośrednio do pliku wykonywalnego*. Zarzut ten jest uzasadniony, jeśli chodzi o niektóre narzędzia wizualne, lecz nie stosuje się do Kyliksa. Dzięki dziedziczeniu, komponenty zawierają bardzo niewielką porcję dodatkowego kodu w Object Pascalu, który pozwala nimi manipulować w czasie tworzenia aplikacji. Ów kod nie jest rozwlekły. Niektóre z wizualnych komponentów mogą być nieporęczne, lecz te, które wchodzą w skład Kyliksa, nie są większe (pod względem rozmiaru), niż jest to konieczne. Co więcej, można je — w zależności od potrzeb — włączać bezpośrednio do aplikacji. Ta część kodu, który sprawia, że komponent jest tym, czym jest, napisana jest dokładnie w Object Pascalu. Z tego powodu kod ten może być włączony bezpośrednio do programu w taki sam sposób, w jaki może być włączony każdy inny obiekt (niektóre części komponentów nie są napisane w Object Pascalu, lecz zamiast tego są częścią opartej na C/C++ biblioteki Qt. Ten temat poruszony został w drugiej części tej książki, traktującej o CLX). Jeśli mam być szczery, powinienem wspomnieć o tym, że cała biblioteka CLX jest dość duża, tak jak duże są MFC, Qt, GTK i inne biblioteki oparte na OOP. Na poziomie komponentów kod potrzebny do stworzenia kontrolki nie jest jednak szczególnie duży.

- **Zarzut trzeci**: *Każdy może programować wizualnie. Jestem prawdziwym programistą i chcę wykonywać pracę współmierną do moich umiejętności (poza tym, co z moją gwarancją pracy, jeśli programowanie staje się niczym więcej niż przeciąganiem myszą różnych elementów?)*. Jeśli naprawdę jesteś dobrym programistą, stratą czasu będzie dla Ciebie uprawianie nużącego i powtarzalnego programowania, jakie stanowi serce i duszę większości prac nad tworzeniem interfejsu. Narzędzia do programowania wizualnego pozwalają szybko uporać się z nudnymi czynnościami, tak byś mógł skoncentrować się na bardziej interesujących i ambitnych zadaniach. Po drugie, programowanie wizualne stwarza w rzeczywistości miejsca pracy na polu pisania komponentów interfejsu. Do opracowania dobrego komponentu potrzeba programisty z wielkimi umiejętnościami w kodowaniu,

dlatego dobrzy programiści mają bardzo duże szanse znalezienia pracy w tej dziedzinie. Poza tym, sednem dobrego programowania interfejsów są w końcu i tak prace estetyczne, a częściowo też szukanie rozwiązań czyniących program prostym dla użytkownika. Rzadko zdarza się znaleźć eksperta od zaawansowanych algorytmów, który byłby równocześnie utalentowanym projektantem interfejsów. Narzędzia programowania wizualnego, takie jak Kylix, pozwalają przekazać prace estetyczne ludziom odpowiednio utalentowanym, a ciężkie, intelektualne zadania — najzagorzalszym programistom.

- **Zarzut czwarty**: *Aplikacje stworzone przy pomocy programowania wizualnego są większe i bardziej złożone niż to, co mógłbym zrobić pisząc cały kod ręcznie*. To jest chyba najlepszy z klasycznych zarzutów wobec programowania wizualnego. Wierzę jednak, że ta skarga nie odnosi się do wizualnego programowania jako takiego, a do bibliotek OOP, leżących u jego podstaw. Niewizualne środowiska, takie jak Visual C++ Microsoftu również używają opartych na OOP bibliotek (niech nie zmyli Cię nazwa wspomnianego środowiska: Visual C++ nie jest narzędziem programowania wizualnego). Programiści VC są zależni od MFC, tak jak programiści GNOME od GTK, a programiści KDE od Qt. Wszystkie te biblioteki zwiększają objętość Twojego kodu. Co więcej, są one zastraszająco zawiłe. Niemniej jednak, zyskały dużą popularność, ponieważ wytworzyły standardy oraz uczyniły złożone zadania znacznie łatwiejszymi. Ponadto, podobnie jak programiści VC mogą pisać programy, które nie korzystają z MFC, tak programiści Kyliksa mogą tworzyć kod nie wykorzystujący CLX. Dla przykładu, w rozdziale 6., zatytułowanym „Poznanie środowiska Linuksa", nauczysz się jak pisać bezpośrednio do XLIB API tworząc programy dla X; nie używają one CLX w ogóle. W tej książce zobaczysz przykłady wywoływanych i sterowanych z linii poleceń małych aplikacji, które również nie używają CLX. Według dzisiejszych standardów programy te są całkiem małe, zajmując nie więcej niż 25 KB. Możesz tworzyć takie lub inne aplikacje Kyliksa nie używając nic poza emacs, kompilatorem Kyliksa wywoływanym z linii poleceń, `dcc` oraz `gdb`.
- **Zarzut piąty**: *Nie lubię programowania wizualnego, ponieważ niepokoi mnie fakt, iż wykorzystuje się w nim czarne skrzynki. Chcę mieć dostęp do całego kodu mojego projektu*. W programowaniu w Kyliksie nie ma nawet mowy o żadnych czarnych skrzynkach. Produkt dostarczony jest razem z wszystkimi źródłami do CLX. Dzięki narzędziom wizualnym otrzymujemy szybkie i sprawne metody napisania dodatkowego kodu. W każdym razie, podczas pracy z narzędziem projektanta formularzy zwykle możliwe jest zobaczenie całości kodu, który właśnie jest tworzony (wyjątkiem jest sytuacja, gdy korzysta się komponentów kupionych od osób trzecich, nie udostępniających źródeł). Kod wytwarzany podczas wizualnego programowania pojawia się w jednym z dwojga miejsc: bezpośrednio w pliku źródłowym aplikacji lub w pliku *xfm* — zobacz sekcja „Edytowanie pliku xfm" nieco dalej w niniejszym rozdziale.

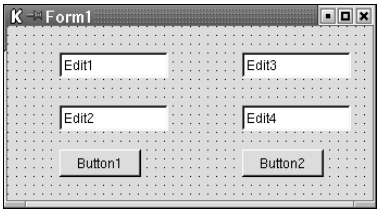
Na przestrzeni lat przyglądałem się jak programowanie zorientowane obiektowo z rzadko używanej technologii przeistacza się w jedną z najistotniejszych w świecie programistycznym. Wizualne programowanie nie ma jeszcze takiego prestiżu jak OOP, lecz spodziewam się, że w umysłach większości programistów z czasem dorówna, a może nawet przewyższy swoim znaczeniem OOP.

Paleta komponentów

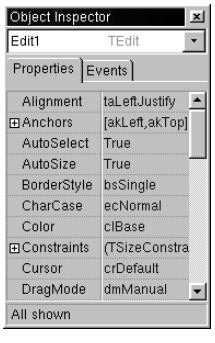
Pora przejść do konkretów. Zacznę od omówienia palety komponentów, a następnie przejdę do narzędzia projektanta formularzy, które zwykle jest używane łącznie z inspektorem obiektów. Na rysunku 1.7 przedstawiona została paleta komponentów, zaś na rysunku 1.8 zobaczyć można narzędzie projektanta formularzy. Inne blisko związane z nim narzędzie — inspektor obiektów — pokazane jest na rysunku 1.9.

Rysunek 1.7. *Paleta komponentów otwarta na zakładce Standard*

Rysunek 1.8. *Projektant formularzy z umieszczonymi różnymi komponentami*



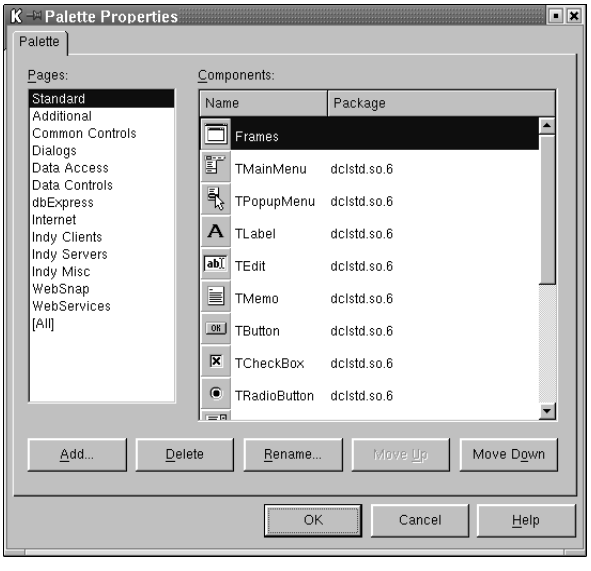
Rysunek 1.9. *Inspektor obiektów z wyświetlonymi parametrami kontrolki Edit*



Paleta komponentów pojawia się po prawej stronie pod paskiem menu. Składa się ona z zakładek, które zawierają komponenty dostarczone razem z produktem. Dosyć prosto można stworzyć nowe komponenty i dodać je do palety. Zagadnienie tworzenia komponentów zostanie szczegółowo omówione w części II.

Na zakładkach palety komponentów znaleźć można wiele użytecznych komponentów, na przykład komponenty pozwalające tworzyć aplikacje operujące na bazach danych oraz przeznaczone do tworzenia aplikacji WWW. Oczywiście, znajdziemy tam również standardowe elementy formularzy, takie jak kontrolki edycji czy etykiety.

Porządek, w jakim ułożone są zakładki na palecie komponentów, jest całkowicie konfigurowalny. Kliknij prawym przyciskiem na palecie komponentów i wybierz *Properties*. Pojawi się okno dialogowe nazwane *Palette Properties*, jak pokazano na rysunku 1.10. Po lewej stronie tego okna znajduje się lista zakładek, a po prawej — lista komponentów znajdujących się na danej zakładce. Kliknięciem określonej pozycji po lewej stronie zmieniasz listę komponentów po stronie prawej.



Rysunek 1.10. *Okno Palette Properties umożliwia zmianę wyglądu palety komponentów*

Możesz kliknąć lewym przyciskiem na tekście w oknie *Pages* (zakładki) i przeciągnąć go na nową pozycję. Przejdź dla przykładu na sam dół listy, znajdź pozycję znajdującą się tutaj na ostatnim miejscu i przeciągnij ją na samą górę. Gdy zamkniesz okno, zobaczysz, że ostatnia stała się pierwszą, a pierwsza — no cóż — nie ostatnią, lecz drugą.

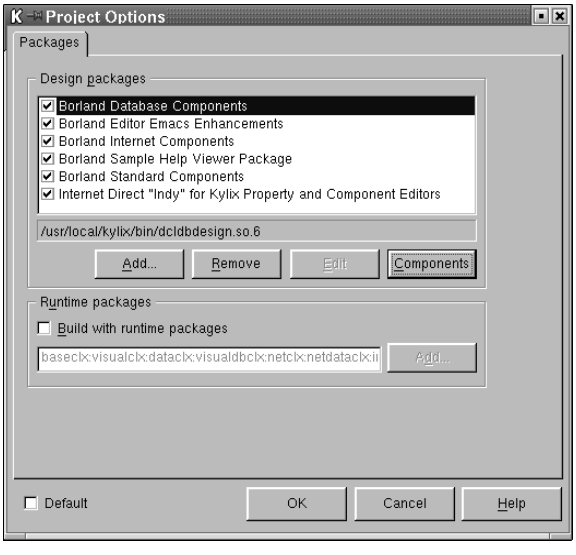
Dalsze zgłębianie okna *Palette Properties* ujawnia możliwość rearanżacji komponentów, usuwania ich z listy oraz ukrywania. Do wspomnianego okna można dostać się również poprzez menu *Component*, pod pozycją *Configure Palette*. Dodatkowe możliwości konfigurowania palety uzyskamy przez proste kliknięcie prawym przyciskiem myszy na samej palecie komponentów (pamiętaj, żeby podczas korzystania z IDE próbować klikać prawym przyciskiem myszy po prostu na wszystkim dookoła).

O pakietach

Komponenty na palecie komponentów są przechowywane w zbiorowych bibliotekach, znanych jako *pakiety* (ang. *packages*). Na dysku każda biblioteka jest po prostu specjalnym rodzajem współdzielonego obiektu — jaki programiści Windows nazwaliby DLL. Aby stworzyć komponent, piszesz obiekt, który pochodzi od jednego ze szczególnych zbiorów klas Object Pascala, a następnie kompilujesz swój komponent, który jest umieszczany w jednej z bibliotek. Jeśli wszystko zrobisz dobrze, IDE, które potrafi przeszukiwać biblioteki, pokaże ikonę Twojego obiektu na palecie komponentów.

Jak zobaczysz w części II, Istnieje jeszcze jeden, zupełnie inny sposób zmieniania palety komponentów. Wykorzystuje on zaawansowane techniki związane z bibliotekami.

Wybierając *Component*, i z menu *Install Packages* zobaczysz listę bibliotek, jak to pokazano na rysunku 1.11 (do pokazanego na nim okna możesz dostać się również wybierając kolejno *Project*, *Options*, *Packages*). Klikając przycisk *Components* przekonasz się, które komponenty znajdują się w danym pakiecie. Jak sam widzisz, większość pakietów dostarczonych z Kyliksem znajduje się w katalogu *kylix/bin*.



Rysunek 1.11. *Lista obecnie używanych pakietów, wyświetlona w oknie dialogowym Project Options*

Pole wyboru znajdujące się na liście *Design Packages* pozwala określić, czy komponenty z danego pakietu mają być wyświetlane, czy ukrywane. Przycisk *Remove* z okna *Project Options* pozwala całkowicie usunąć pakiet, wskutek czego Kylix nie będzie go więcej ładował. Jeśli zdecydujesz, by pewne większe pakiety nie były ładowane — zwłaszcza pakiety baz danych — całe środowisko Kyliksa będzie się uruchamiało szybciej. Może to także umożliwić stworzenie mniej „zagraconej", prostszej do ogarnięcia palety komponentów.

Na dole okna *Project Options* znajduje się opcja pozwalająca określić, czy pakiety wczytywane w czasie wykonywania programu będą używane. Dzięki niej możemy zadecydować, czy komponenty z określonych pakietów mają być włączone do programu wykonywalnego aplikacji. Jeśli zdecydujesz, że tak właśnie ma być, będziesz mógł łatwo dostać się do nich poprzez biblioteki, w których są przechowywane. Gotowe programy korzystające z pakietów dynamicznie wczytywanych są znacznie mniejsze od programów skonsolidowanych z wszystkimi swoimi obiektami wewnątrz pliku wykonywalnego. Z drugiej strony, jeśli zdecydujesz się na użycie zewnętrznych pakietów i będziesz chciał, by inni mogli korzystać z Twojej aplikacji, będziesz musiał rozprowadzać je razem ze swoim produktem.

To jednak wykracza już poza zakres tematyki tej części książki; wrócimy do tego w drugiej jej części. Chciałem tutaj jedynie wspomnieć o komponentach Kyliksa i dyskretnie podejrzeć ich architekturę. Są to zagadnienia absolutnie podstawowe i po prostu musisz je poznać, żeby zacząć rozumieć, jak to wszystko działa; później, w następnych rozdziałach, będziemy delikatnie rozsuwać kurtynę, aż do końca, odkrywając najcenniejsze zalety Kyliksa.

Używanie Projektanta formularzy

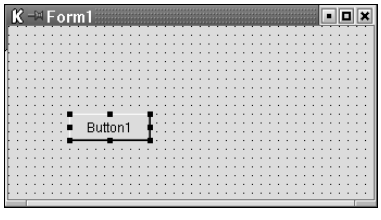
Projektant formularzy pozwala na zaprojektowanie interfejsu dla projektu. Narzędzie to ma formę okna, na którym możesz rozmieszczać różne obiekty z palety komponentów. Komponenty, które dodajesz do swojego projektu, mogą być widzialne lub nie. Bez względu na to, Kylix automatycznie tworzy instancję komponentu w momencie umieszczenia go

w projektancie formularzy. Możesz w trakcie projektowania nim manipulować. W czasie wykonywania Twój projekt będzie wyświetlał interfejs, który stworzyłeś w narzędziu projektanta formularzy.

Rozmieszczanie komponentów

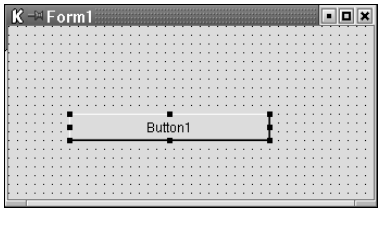
Paleta komponentów zawiera standardowe komponenty wyświetlane na zakładkach. Aby dodać komponent do projektanta formularzy, kliknij go na palecie, a następnie kliknij narzędzie projektanta. Możesz na przykład kliknąć komponent *button* (przycisk) i upuścić go w oknie projektanta, tworząc formularz taki jak na rysunku 1.12.

Rysunek 1.12. *Przycisk Button1 umieszczony w oknie projektanta formularzy Kyliksa*



Po umieszczeniu komponentu w jakimś określonym miejscu na oknie projektanta formularzy, można go przesuwać poprzez kliknięcie i przeciągnięcie na nowe miejsce. Można również zmienić jego rozmiar, używając uchwytów na jego krawędziach. Jeśli wolisz, możesz uzyskać ten sam efekt używając inspektora obiektów; powinieneś znaleźć w nim ustawienia `Width` (szerokość) i `Height` (wysokość) komponentu i zmienić ich wartości, jak pokazano na rysunku 1.13.

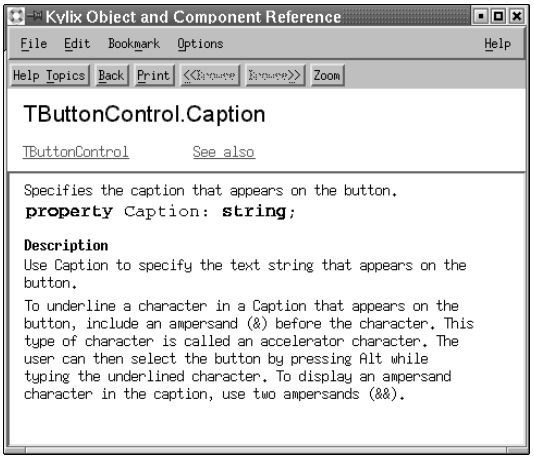
Rysunek 1.13. *Ustawienie własności Width przycisku na 160 spowodowało rozciągnięcie go wzdłuż formularza*



Zauważ, że możesz również przejść nieco wyżej, aż do właściwości *Caption* (podpis) i zmienić tekst pojawiający się na przycisku. Zwróć również uwagę na fakt, iż z utworzonym komponentem związanych jest wiele innych właściwości. Możesz wybrać dowolną z nich i nacisnąć *F1*, aby uzyskać pomoc na jej temat — patrz rysunek 1.14.

Kliknij przycisk, który umieściłeś w formularzu. Teraz przytrzymaj klawisz *Ctrl* i wciśnij kilkakrotnie klawisze kursora na klawiaturze. Zobaczysz, że przycisk przesuwa się w górę i w dół, w lewo i w prawo, za każdym razem po jednym pikselu. Zdejmij palec z klawisza *Ctrl* i zamiast niego wciśnij klawisz *Shift*. Wciskaj teraz jedną ze strzałek; odkryjesz, że można zwiększać i zmniejszać szerokość i wysokość komponentu po jednym pikselu. Są to małe drobiazgi, które podczas tworzenia formularza mogą okazać się bezcenne.

Gdy klikniesz umieszczony w formularzu komponent prawym przyciskiem myszy, w wyświetlonym w efekcie menu podręcznym znajdziesz opcje rozmieszczenia komponentów, takie jak *Align* (ułożenie), *Size* (rozmiar), *Scale* (skalowanie) i *Align to Grid* (dopasowanie



Rysunek 1.14.
Pomoc kontekstowa dla własności Caption obiektu TButton

do siatki). Opcje te są niesłychanie pomocne. Gdy masz np. cztery kontrolki i chcesz, żeby wszystkie miały ten sam rozmiar, naciskasz *Shift* i klikasz na nich po kolei, aż zaznaczysz wszystkie. Następnie prawym przyciskiem myszy klikasz na jednej z kontrolek, wybierasz *Size* i stosowną opcję, np. *Grow To Largest*.

Inspektor Obiektów

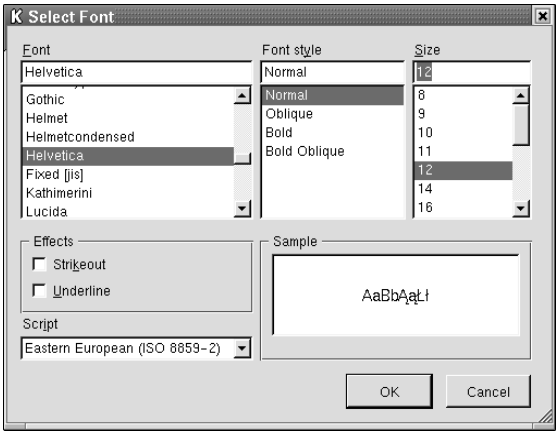
Inspektor obiektów jest jednym z najpotężniejszych narzędzi Kyliksa. Z jego pomocą możesz modyfikować własności komponentu lub generować związane z nim zdarzenia. Jeśli inspektor obiektów nie jest widoczny w IDE, a chcesz go wyświetlić, wybierz *View*, *Object Inspector*.

Gdy klikniesz komponent w projektancie formularzy, w inspektorze obiektów pojawi się jego nazwa. O tym, z którym komponentem chcesz pracować, możesz również zdecydować wybierając jego nazwę z rozwijanej listy inspektora.

Zakładka *Properties* inspektora obiektów ujawnia właściwości komponentu, które można modyfikować. Dla każdej właściwości po lewej stronie okna, w którym są one wyświetlane, znajduje się jej nazwa; po prawej można zobaczyć puste pole, tzw. *edytor ustawień właściwości* (ang. *property editor*). Niekiedy edytory właściwości są zwykłymi polami z tekstem, innym razem są to listy opcji, takich jak choćby pola `True` i `False`. Zdarzają się także bardziej skomplikowane jednostki. Aby zobaczyć przykład takiej skomplikowanej jednostki, umieść w formularzu kontrolkę `TEdit` z zakładki *Standard*. Kliknij dwukrotnie małą ikonę, która pojawi się w edytorze właściwości *Font* (czcionka) kontrolki `TEdit`. Pojawi się okno dialogowe *Font*, pokazane na rysunku 1.15. Wybierz w nim czcionkę, która ma się pojawić w kontrolce edycji.

Na ogół, gdy klikasz prawą stronę własności, wybierasz dla niej opcję. Komponent `TPanel` posiada np. właściwość `align`. Żeby zobaczyć, jak ona funkcjonuje, wybierz ten komponent z zakładki *Standard* i upuść go na formularzu. Następnie kliknij edytor właściwości; pojawi się rozwijana lista, z której możesz wybrać rodzaj ułożenia kontrolki.

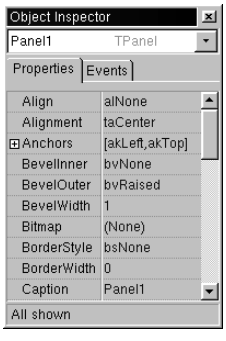
Rysunek 1.15.
Okno dialogowe Font



Dostrajanie Inspektora obiektów

Narzędzie inspektora obiektów jest konfigurowalne. Żeby wyświetlić pozycje według kategorii lub nazw, kliknij omawiane tu narzędzie prawym przyciskiem myszy i wybierz *Arrange*, *By Category* lub *Arange*, *By Name*. Dwa różne widoki, uzyskane w zależności od dokonanego wyboru, pokazane zostały na rysunkach 1.16 i 1.17.

Rysunek 1.16.
Inspektor obiektów z wartościami ułożonymi według nazw



Rysunek 1.17.
Inspektor obiektów z wartościami ułożonymi według kategorii

Możesz również filtrować zdarzenia. Umieść przycisk w formularzu, a następnie zmień element aktywny na narzędzie inspektora obiektów. Upewnij się, czy przycisk w formularzu jest wybrany i czy jest widoczny w inspektorze. Kliknij prawym przyciskiem myszy

na inspektorze i wybierz *View*, *None*. Wszystkie pozycje znikną. Kliknij ponownie i wybierz *View*, *Action*. W efekcie widoczny stanie się podzbiór własności dla obiektu `TButton`.

Edytowanie pliku xfm

Gdy upuszczasz przycisk na formularzu, w tle generowany jest kod. Część tego kodu umieszczona zostaje w edytorze. Reszta pojawia się w pliku *xfm*, który towarzyszy danemu formularzowi.

Odwołanie do pliku *xfm* znajduje się w źródle modułu (ang. *unit*) skojarzonego z danym formularzem, zaraz po słowie kluczowym `implementation`:

```
{$R *.xfm}
```

Ta dyrektywa informuje kompilator o tym, że część definicji dla formularza z tego modułu znajduje się w pliku *xfm*. Tę linię kodu znajdziemy we wszystkich modułach zawierających bezpośrednich potomków `TForm`, natomiast te źródła, które nie zawierają formularzy, zwykle jej nie potrzebują.

Delphi

Programiści Delphi powinni zauważyć, że *xfm* pisane jest małymi literami. Podczas przenoszenia programu z Delphi do Kyliksa, potrzebna będzie zamiana wszystkich `{$R *.DFM}` na `{$R *.xfm}`. W rzeczywistości zauważysz, że Kylix czasami przyjmuje pliki *dfm* Delphi z ich nie zmienionym rozszerzeniem. Prawdopodobnie wciąż jednak będziesz musiał zmienić w odwołaniach do nich zapisane wielkimi literami `DFM` na `dfm` pisane małymi.

Potrzebna będzie jeszcze jedna zmiana we wszystkich formularzach. Właściwość *Pixel Per Inch* normalnie ustawiona jest w Delphi na 96. Najlepszym ustawieniem w Kyliksie jest na ogół 75. Jeśli otworzysz w Kyliksie formularz z Delphi i kontrolki oraz tekst będą małe i ciasno rozstawione, spróbuj zmienić ustawienie *Pixel Per Inch* z 96 na 75.

Normalnie nie ma potrzeby edytowania kodu w plikach *xfm*; robi się to w wyjątkowych sytuacjach. Standardowo ich zawartość kontroluje się poprzez dokonywanie zmian w projektancie formularzy lub w inspektorze obiektów. A jednak kod ten może być edytowany i czasami okazuje się to przydatne.

Pliki *xfm* mogą mieć zarówno postać binarną, jak i tekstową, zależnie od ustawienia w menu *Tools*, *Environment*, *Preferences*, *Form Designer*, *New Form As Text* (ten sam efekt możesz uzyskać lokalnie, klikając formularz prawym przyciskiem myszy i wybierając *Text xfm*). Aby zobaczyć źródło formularza, możesz otworzyć tekstowy plik *xfm* w edytorze, ale możesz także w czasie jego projektowania kliknąć na nim prawym przyciskiem myszy i wybrać z menu *View As Text*.

Istnieje także program użytkowy wywoływany z linii poleceń, nazwany `convert`, który dokonuje konwersji z wersji tekstowej na binarną i vice versa. Aby go uruchomić, wpisz po prostu polecenie podobne do następującego:

```
convert MójFormularz.xfm
```

Zanim użyjesz tego polecenia, upewnij się, czy masz dobrze zdefiniowaną ścieżkę dostępu; niektóre dystrybucje Linuksa wyposażone są w programy o identycznej nazwie.

Kod w pliku *xfm* w dużej mierze jest prawdziwym kodem. Jeśli chcesz, możesz go ręcznie edytować, a nawet wstawić go do IDE. Aby zobaczyć jak to działa, zacznij nowy projekt i umieść przycisk w jego formularzu. Kliknij formularz prawym przyciskiem myszy i wybierz *View As Text*. W jego źródle znajdziesz następujący kod:

```
Object Button1: TButton
  Left = 264
  Top = 24
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder  = 0
end
```

Zmień teraz właściwość `Caption` tak, żeby zamiast `Button1` znalazło się `If there is any religion that would cope with modern scientific needs, it would be Buddhism. – Albert Einstein`. Kliknij prawym przyciskiem myszy i wybierz z menu *View As Form*. Teraz zmień rozmiary przycisku i obejrzyj rezultaty swojej ręcznie wykonanej pracy. Możesz dalej poeksperymentować podczas przeglądania formularza w formie tekstowej, zmieniając `Width`, `Height` i pozostałe własności.

Teraz przejdź do jakiegoś edytora tekstowego. Jeśli chcesz, możesz użyć edytora Kyliksa lub wybrać inny — jak KWrite lub KEdit — który dzieliłby schowek z Kyliksem. W tym edytorze napisz co następuje:

```
object Button2: TButton
  Left = 20
  Top = 10
  Width = 75
  Height = 25
    Caption =
      'Mądrość częstokroć jest bliżej, gdy się schylamy, aniżeli gdy się wznosimy. --
      ➥William Wordsworth'  TabOrder = 0
end
```

Użyj edytora, żeby skopiować cały ten blok tekstu. Teraz wróć do Kyliksa i zacznij nową aplikację, oglądając formularz w normalnym trybie, nie w tekstowym. Mając zaznaczony formularz wybierz *Edit* i *Paste* z menu. W lewym górnym rogu formularza powinien pojawić się przycisk, który zaprojektowałeś w edytorze. Możesz teraz za pomocą myszy zmienić jego rozmiar, tak żeby cały tekst był widoczny.

Technika wstawiania do projektanta formularzy kontrolek z edytora może być całkiem użyteczna. Szczególnie pomocna może się okazać podczas naprawiania zniszczonych formularzy lub podczas porządkowania formularza, który uległ całkowitemu zagmatwaniu. Przydaje się to również podczas przenoszenia aplikacji Delphi do Kyliksa.

Delphi

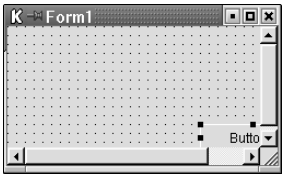
Format pliku *xfm*, zarówno binarny, jak i tekstowy, jest identyczny z formatem *dfm*, używanym przez Delphi pod Windows. Tak długo, jak wszystkie komponenty, właściwości i zdarzenia użyte w pliku *dfm* istnieją zarówno w VCL, jak i CLX, możesz używać plików *dfm* w aplikacji Kyliksa. Rozszerzenie *xfm* dla plików Kyliksa istnieje po to, byś mógł łatwo sprawdzić, że komponenty w pliku są raczej komponentami CLX niż VCL. Jak sam zobaczysz, można przyjąć zasadę, że możliwe jest używanie niezmienionych plików *dfm* w programach Kyliksa, jeżeli formularz, który definiują, jest stosunkowo prosty. W przeciwnym wypadku możesz mieć problemy. Ale nawet w najgorszych przypadkach wyedytowany plik *dfm* może pomóc ci w rozpoczęciu tworzenia formularza w Kyliksie.

Używający JBuildera programiści piszący w języku Java powinni znać już wiele technik programowania wizualnego używanych w Kyliksie. W JBuilderze jednak nie ma żadnych plików *xfm*. Zamiast tego, wszystkie wyniki Twoich akcji realizowanych za pomocą myszki komputerowej zostają zapamiętane w Twoim pliku źródłowym, zwykle w metodzie `jbInit`. To, czy bardziej podoba Ci się sposób programowania wizualnego w Kyliksie, czy w Javie, jest sprawą gustu; może też zależeć od określonego przypadku. Ja, gdy zacząłem używać technologii JBuildera, pomyślałem, że jest lepsza od ukochanej przeze mnie technologii Delphi. Jednak gdy zacząłem tworzyć większe i bardziej złożone programy w Javie, coraz bardziej nużące stawało się „przewijanie" metod `jbInit`, które składały się z wielu ekranów informacji wygenerowanych przez narzędzia do wizualnego projektowania (z pewnością lepiej było używać wizualnych narzędzi niż ręcznie pisać kod i zapewne kod ów był przynajmniej tak dobry — i w sumie na ogół taki sam — jak kod, który napisałbym ręcznie. W każdym razie było tego mnóstwo). Ostatecznie w konkurencji pomiędzy narzędziami Javy i Delphi przyznałbym pewnie remis. Java ma zadziwiającą cechę przechowywania całego kodu, napisanego w jednym języku, w jednym pliku źródłowym. Delphi oraz Kylix odizolowują cały ten nieestetyczny kod, umieszczając go w pliku *xfm*. Format tego pliku bardzo trafnie reprezentuje sposób przechowywania informacji przez tych programistów, którzy tworzą swe aplikacje korzystając z narzędzi do programowania wizualnego.

Ograniczenia i zakotwiczenia

Programiści używający języka Java stosują do kontroli rozmiarów komponentów w czasie uruchomienia programu technologie znane jako *layout managers* (zarządcy układu). Zarządcy ci pomagają określić sposób, w jaki ma się zachować formularz, gdy pojawia się w nieoczekiwanej rozdzielczości lub gdy użytkownik zmienia jego rozmiar. Kylix nie posiada tak złożonych narzędzi, oferuje za to właściwości nazwane `Constraints` (ograniczenia) i `Anchors` (zakotwiczenia), które w podobnych sytuacjach wykazują swoją zadziwiającą potęgę.

Umieść przycisk w prawym dolnym rogu formularza. Teraz chwyć ten róg i zacznij zmniejszać formularz, jak to jest pokazane na rysunku 1.18.



Rysunek 1.18.
Kiedy użytkownik zmienia rozmiar formularza, komponent staje się częściowo lub całkowicie niewidoczny

Jak sam widzisz, komponent może zostać zarówno częściowo, jak i całkowicie zakryty w trakcie projektowania lub w czasie uruchomienia. Jeżeli stanowi to dla Twojej aplikacji istotny problem, prostym rozwiązaniem okazuje się użycie własności `Anchors`.

Zwróć uwagę na to, iż w inspektorze obiektów obok własności `Anchors` występuje mały znak plusa. Kliknij go, aby otworzyć wszystkie pola tej własności. Zaznacz przycisk i ustaw pola `akLeft` i `akTop` na `False`, a `akRight` i `akBottom` na `True`. Teraz zmień znów rozmiar formularza (najlepiej zrób to kilkakrotnie), a zobaczysz, że podczas zmian przycisk zachowa swoje miejsce w prawym dolnym rogu.

Jeśli chcesz, możesz dodać do formularza dodatkowe przyciski. Umieść np. na nim cztery przyciski, po jednym w każdym jego rogu. Użyj właściwości `Anchors`, aby zakotwiczyć każdy z przycisków na właściwym mu miejscu. Spróbuj teraz zmienić rozmiar formularza.

Jeśli poprawnie wykonałeś zadanie, przyciski nie będą chowane (wykonując tą akcję możesz odkryć, że podwójnym kliknięciem na polu logicznym odwracasz jego ustawienie, tzn. z wartości `False` zmieniasz je na `True`, a z `True` na `False`). Obejrzyj program ButtonAnchor, aby zobaczyć, jak program powinien się zachowywać, jeśli te wartości zostały poprawnie ustawione.

Ci, dla których praca w Object Pascalu jest czymś zupełnie nowym, mogą być zainteresowani faktem, że stałe `True` i `False` pisane są wielką literą, podczas gdy dla pozostałych stałych używa się małych liter. Jest to rozwiązanie inne niż przyjęte w języku C, w którym w nazwach stałych występują tylko wielkie litery, oraz w Javie, w której używa się tylko małych liter. Co więcej, `True` i `False` są wartościami wbudowanymi w kompilator, więc nie są nigdy deklarowane w żadnym z plików źródłowych Pascala. Oczywiście Pascal nie jest językiem, który rozróżnia wielkość liter; nie ma dla niego znaczenia, czy napiszesz `False`, `false`, `fALSE`, czy `faLsE`. Dobrze jest jednak zgodnie z konwencją zaczynać wszystkie nazwy zmiennych i większość nazw typów wielką literą. Wyjątkiem od tej zasady są typy wyliczeniowe, których elementy zaczynają się małymi literami:

```
TNavigateBtn = (nbFirst, nrPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit,
➥nbPost, nbCancel, nbRefresh)
```

Jeśli nazwa zmiennej składa się z kilku słów, wszystkie powinny zaczynać się wielką literą, jak np. `MyVariable`. Słowa zastrzeżone, które są zawsze pogrubiane przez edytor, powinny być pisane małymi literami, z wyjątkiem typu `String`, którego pierwsza litera zwykle jest wielka (ten wyjątek dla słowa `String` sprawia, iż cechujący się ścisłą, nienaruszalną strukturą język wydaje się nieco mniej formalny). Wszystkie pozostałe słowa zastrzeżone, takie jak `begin`, `end`, `if`, `implementation` i `interface` są wedle konwencji pisane małymi literami.

Jak na razie powinieneś mieć cztery widzialne przyciski, każdy zakotwiczony w innym rogu formularza. Jeśli wszystko poszło dobrze, możesz zmieniać rozmiar okna, a każdy przycisk zostanie na swoim miejscu. Na pewno będziesz chciał sprawdzić, co się stanie, gdy uczynisz formularz tak małym, jak to tylko możliwe. Odpowiedź jest oczywista — nastąpi chaos. Aby zatem zabronić użytkownikom zbytniego zmniejszania formularza, możesz ustawić jego właściwość `Constraints` na odpowiednią wartość.

Zaznacz formularz kliknięciem na jego szarym tle. Zauważ, że do inspektora obiektów wypisane zostały właściwości aktualnie zaznaczonego obiektu, którym w tym wypadku jest formularz. Kliknij podwójnie właściwość `Constraints` w inspektorze obiektów. Otworzy się ona, umożliwiając Ci wprowadzenie wartości 200 i 250 — odpowiednio dla właściwości `MinWidth` i `MinHeight` formularza. Gdy spróbujesz zmienić jego rozmiar, zobaczysz, że zatrzymuje się teraz, nie pozwalając użytkownikowi spowodować bałaganu.

Konwencje w Pascalu są zwykle tak dosłowne, jak to tylko możliwe. Z tego powodu używanie skrótów w nazwach zmiennych nie jest tu tak powszechne, jak w innych językach. Generalnie rzecz biorąc, programiści Pascala przyjmą zmienne nazywające się `MinWidth` i `MinHeight`, ale `mWidth`, czy `mHght`. Mam zwyczaj doprowadzać tą dosłowność do granic i nazywać zmienne nawet `MinimumWidth` czy `MinimumHeight`. Kompilator nie umieści oczywiście tych nazw w kodzie skompilowanym z wyłączoną opcją *debug*, tak więc decydując się na użycie dłuższych nazw zmiennych nie zwiększasz rozmiaru finalnego pliku wykonywalnego.

Porządek tabulatora

W niektórych aplikacjach kolejność zmieniania elementów aktywnych klawiszem tabulacji (ang. *tab order*) może być istotna. Jeśli chcesz zmienić ją dla swoich komponentów, kliknij prawym przyciskiem jeden z nich i wybierz *Tab Order* z menu podręcznego. W oknie dialogowym *Edit Tab Order* zaznacz nazwę komponentu. Klikając strzałki w górę i w dół możesz zmienić miejsce komponentu w porządku tabulatora w swojej aplikacji.

Inspektor Obiektów i zdarzenia

Nadszedł czas, by omówić zdarzenia (ang. *events*). Zagadnienia z nimi związane znacznie przewyższają pod względem swej złożoności to wszystko, o czym była do tej pory mowa. Mamy tu wszak do czynienia z przejściem od zagadnienia wizualnych narzędzi do tematów związanych z edytorem. Omawianie zdarzeń związane jest z pisaniem kodu, zaś czynność ta wykonywana jest właśnie za pomocą edytora.

Na następnych kilku stronach zajmiemy się zagadnieniem wzajemnych powiązań pomiędzy inspektorem obiektów, edytorem, a regułami składni języka Object Pascal. W pewnym momencie dyskusji zostanie poddany nawet kompilator. Gdy to wszystko zostanie już wyjaśnione, zrozumienie kilku ostatnich punktów związanych z edytorem oraz z narzędziem *Code Insite* nie będzie stanowić kłopotu. Na końcu przyjrzymy się kilku programom napisanym w Object Pascalu.

Zakładka *Events* w inspektorze obiektów pokazuje zdarzenia skojarzone z komponentami. Jeśli klikniesz edytor właściwości dla zdarzenia, Kylix wygeneruje kod w *edytorze kodu źródłowego* (ang. *Source Code Editor*), a ten natychmiast zostanie pokazany, tak żebyś mógł zaprogramować w nim dalsze działania.

Żeby zobaczyć jak działają zdarzenia, zacznij nowy projekt, wybierając *File*, *New*. Teraz umieść w głównym formularzu projektu kontrolkę `TButton` z zakładki *Standard*. Zaznacz go przy pomocy myszki. Otwórz zakładkę *Events* w inspektorze obiektów. Kliknij dwukrotnie w polu edycji właściwości dla zdarzenia `OnClick`. Pokazany zostanie edytor, a do programu zostaną wstawione następujące linijki kodu:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Ten kod mógł zostać również wygenerowany poprzez podwójne kliknięcie samego przycisku. Sposób ten przynosi spodziewany efekt, ponieważ zdarzenie `OnClick` jest domyślnym zdarzeniem dla kontrolki `TButton`. Domyślne zdarzenie jest zwykle automatycznie przywoływane podwójnym kliknięciem kontrolki.

Zmodyfikuj kod metody kliknięcia przycisku w następujący sposób:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('This very place is the Lotus Land; this body, the buddha')
end;
```

Teraz uruchom opracowany przez Ciebie program i naciśnij przycisk. Pojawi się okno dialogowe, wyświetlające tekst, który podałeś metodzie `ShowMessage`.

Źródło modułu Kyliksa

Jeśli dokładnie śledziłeś treść poprzedniego podrozdziału i utworzysz domyślne zdarzenie dla formularza, do Twojego programu zostanie wstawione zdarzenie `FormCreate`:

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

Jeśli po uruchomieniu programu wrócisz do edytora, żeby zobaczyć źródło, odkryjesz, że nagłówek dla zdarzenia `FormCreate` zniknął. Gdzie się podział?

Metody generowane przez IDE, które nie zawierają kodu, są automatycznie usuwane, gdy zapisujesz źródło lub uruchamiasz program. Mówiąc ściślej, zasada ta jest prawdziwa dla wszystkich procedur obsługi zdarzeń (ang. *event handlers*) napisanych w standardowy sposób, bez względu na to, czy zostały wygenerowane przez IDE.

A co — możesz zapytać — oznacza „standardowy sposób" dla obsługi zdarzenia? Aby znaleźć poprawną odpowiedź na to pytanie, warto przyjrzeć się na wydruku 1.1 całemu źródłu modułu, który stworzyliśmy.

Wydruk 1.1. *Kod źródłowy prostego modułu z dwiema procedurami obsługi zdarzeń*

```
unit Unit1;

interface

uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

const
  Diamond = 'Wszystkie złożone rzeczy są jak sen, jak zjawa, ' +
    'kropla rosy i światło błyskawicy';

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.xfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(Diamond);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin

end;

end.
interface

uses
  Windows, Messages, SysUtils,
  Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,
  ExtCtrls;

const
  Diamond = 'All composed things are like a dream, ' +
    'a phantom, a drop of dew, a flash of lightening';

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(Diamond);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin

end;

end.
```

W następnych kilku sekcjach znajdziesz szczegółowe opisy różnych elementów tego pliku źródłowego. W szczególności skoncentrujemy się na różnicy pomiędzy interfejsem (`interface`) a implementacją (`implementation`).

Interfejs modułu Kyliksa

Na wydruku 1.1 przedstawiono deklarację typowego formularza w Kyliksie. Kod podzielony jest na trzy sekcje. Pierwszą z nich stanowi nazwa modułu, która w tym wypadku wygląda tak:

```
unit Unit1;
```

Sekcja druga zaczyna się słowem `interface` i kończy dokładnie przed słowem `implementation`:

```
interface

uses
  SysUtils, Types, Classes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls;
const
  Diamond = 'Wszystkie złożone rzeczy są jak sen, jak zjawa, ' +
    'kropla rosy i światło błyskawicy';

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Rzecz jasna, ta część kodu nazywana jest interfejsem programu. Pod nią znajduje się część implementacji. Rozciąga się ona od początku słowa `implementation` aż do końca źródła, który zawsze oznaczony jest słowem kluczowym `end`, po którym następuje kropka.

W sekcji interfejsu zawarte są deklaracje dla modułu, jednak bez właściwego kodu. Sekcja ta przypomina nieco plik nagłówkowy dla programu w C lub C++, z tą tylko różnicą, że jest zintegrowana z implementacją w jednym pliku źródłowym. A zatem, w Pascalu wybrano drogę pośredniczącą pomiędzy Javą a C/C++. Inaczej niż w C/C++, cały kod danego modułu znajduje się w pojedynczym pliku. Inaczej niż w Javie, część deklaracyjna obiektu jest wyraźnie odseparowana od jego implementacji.

Sekcja interfejsu składa się z trzech części:

- **Pole** `uses` — tutaj znajdziesz listę innych modułów, od których ten jest zależny. Jest to ekwiwalent dyrektywy `#include` w C++ i `import` w Javie. Choć nie jest

to nigdzie wyraźnie zadeklarowane, wszystkie moduły Pascala używają modułu `System`. Borland dostarcza kompletne źródła do tego modułu.

- **Sekcja** `const` — tu możesz zadeklarować stałe. Jeśli zadeklarujesz stałą bez zadeklarowania jej typu, jej zmiana w programie nie będzie możliwa; jeśli natomiast zadeklarujesz ją z podaniem typu, będziesz mógł zmieniać jej wartość także wewnątrz programu. Ten drugi przypadek jest w rzeczywistości sposobem na zadeklarowanie z określeniem wartości obiektu poza blokiem implementacji. Oto deklaracja zmiennej, która zawiera deklarację typu:

  ```
  MyNumber: Integer = 2;
  ```

  Jeśli spróbujesz zmienić stałą, którą zadeklarowałem jako `Diamond` w metodzie `Button1Click`, kompilator zgłosi błąd. Nie stanie się tak natomiast w przypadku zmiany wartości `MyNumber`.

- **Sekcja** `type` — w tym miejscu definiujesz typy danych, których będziesz używał w swoim programie. W tym przypadku moduł definiuje typ nazwany `TForm1`, który pochodzi od obiektu `TForm`. Ten ostatni jest częścią CLX. Obiekt `TForm` jest w stanie utworzyć instancję formularza. `TForm1` jest Twoją odmianą tego szczególnego obiektu. W naszym przypadku `TForm1` wykorzystuje obiekt podstawowy `TForm`, dodając doń kontrolkę przycisku oraz dwie metody — obie zostaną omówione bardziej szczegółowo nieco dalej w tym samym podrozdziale.

- **Sekcja** `var` — tutaj deklarujesz wszystkie zmienne globalne, które chcesz dodać do swojego programu. Zmienne te będą miały zasięg globalny w całym programie. Jak wszyscy doświadczeni programiści wiedzą, zmienna globalna zwykle oceniana jest źle. W związku z tym, dla implementacji modułu na ogół nie powinieneś dodawać niczego do sekcji `var`, chyba że jesteś absolutnie pewien tego, co chcesz zrobić.

Każda z sekcji, która pojawia się w interfejsie, może również pojawić się w implementacji. W rzeczywistości wszystkie sekcje, z wyjątkiem pola `uses`, mogą pojawić się nawet wewnątrz pojedynczej metody. Miejsce dla deklaracji powinieneś wybrać zgodnie z zasięgiem, jaki chcesz nadać swojej zmiennej. Jeśli chcesz, aby była ona globalna, umieść ją w interfejsie modułu. Jeśli chcesz, by była dostępna dla całego modułu, ale nie dla programu, umieść ją na początku części implementacyjnej. Jeśli chcesz, by typ, stała lub zmienna były widoczne tylko dla pojedynczej metody, umieść je w tej metodzie, jak w następującym fragmencie kodu:

```
procedure Foo;
const
  S = 'Moja stała łańcuchowa';
type
  TMyArray = array[0..5] of Integer;
var
  MyArray: TMyArray;
  i: Integer;
begin
  ShowMessage(S);
  for i := 0 to 5 do
    MyArray[i] := i;
  ... // Code omitted here
end;
```

Implementacja modułu w Kyliksie

W sekcji definiowania typów, gdzie zadeklarowany został obiekt `TForm`, znajdziesz następującą linię kodu:

```
procedure Button1Click(Sender: TObject);
```

Jest to deklaracja metody `Button1Click`. Jak zapewne pamiętasz, metoda ta jest wywoływana za każdym razem, kiedy użytkownik kliknie przycisk `Button1`. A oto jej implementacja:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(Diamond);
end;
```

Jeśli umieścisz kursor w ciele metody `Button1Click` i trzymając wciśnięte klawisze *Shift* i *Ctrl* naciśniesz strzałkę w górę, zostaniesz przeniesiony do miejsca deklaracji tej metody. Jeśli naciśniesz teraz strzałkę w dół, wrócisz do jej implementacji. Jednym słowem, IDE jest w pełni świadomy relacji pomiędzy tymi dwoma blokami kodu, nawet jeszcze przed kompilacją. Zagadnienie to omówię bardziej szczegółowo w podrozdziale „Eksplorator kodu" rozdziału 5.

Jeśli znasz się na programowaniu, na pewno rozumiesz relacje pomiędzy deklaracją a implementacją wspomnianej tu metody. Jednak związek pomiędzy obiektem `TButton` i tą metodą może wydać się trudny do zrozumienia. Innymi słowy, skąd obiekt `TButton` „wie", że ma wywołać tę metodę, gdy zostanie kliknięty?

Programiści korzystający na co dzień z języka Java znają już jedną metodę tworzącą związek pomiędzy obiektem a jego procedurami obsługi zdarzeń. Programiści Windows API znają zapewne jeszcze inną. W Object Pascalu natomiast znów obrano trzecią drogę, która na szczęście jest całkiem prosta i jej zrozumienie nie przysparza problemów.

Wszystkie obiekty `TButton` pochodzą od obiektu nazwanego `TControl`, który w definicji swojego typu zawiera pole `TNotifyEvent`:

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;

  TControl = class(TComponent)
  private
    FParent: TWinControl;
    FWindowProc: TWndMethod;
    ... // Pominięty kod
    FOnEndDrag: TEndDragEvent;
    FOnClick: TNotifyEvent;
    ... // Pominięty kod
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick stored
    IsOnClickStored;
 end;
```

Właściwość `OnClick` obiektu `TButton` jest domyślnie ustawiana na `nil`. Wartość `nil` odpowiada 0; jest ona odpowiednikiem tego, co programiści C++ nazywają `NULL` i jest bardzo podobna do tego, co programiści Javy znają pod nazwą `null`. Jeśli `OnClick` jest

ustawiona na nil, użytkownik może klikać przycisk cały dzień, a i tak nic się nie stanie. Gdy jednak właściwości tej jest przypisana metoda taka jak Button1Click, właśnie ona zostanie wykonana.

Klikając dwukrotnie przycisk w projektancie formularzy lub edytor właściwości dla zdarzenia OnClick w inspektorze obiektów, przydzielasz konkretną metodę do właściwości OnClick obiektu TButton.

Zwróć uwagę na to, że do właściwości OnClick nie możesz przypisać jakiejkolwiek metody; powinna mieć ona typ TNotifyEvent:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

TNotifyEvent jest deklaracją metody z następującym podpisem:

```
procedure TForm1.Button1Click(Sender: TObject);
```

Pomijając frazę of object z końca deklaracji TNotifyEvent, deklarowałbyś wskaźnik do procedury, zamiast do metody. Procedura oczywiście nie jest częścią obiektu. Ta deklaracja na przykład jest wskaźnikiem kierującym do procedury, nie metody:

```
TMyNotifyEvent = procedure(Sender: TObject);
```

A tu mamy implementację procedury tego typu:

```
procedure Button1Click(Sender: TObject);
```

Różnica polega na tym, że procedura typu TNotifyEvent jest częścią obiektu, podczas gdy procedura typu TMyNotifyEvent nią nie jest.

Ponadto, funkcja zwraca wartość, procedura nie.

W programach napisanych w C/C++ i Javie metoda, która w rzeczywistości nic nie zwraca, w deklaracji oddaje void:

```
void Button1Click(TObject Sender);
```

C++

Oto jak wyglądałaby deklaracja dla TNotifyEvent w C++:

```
typedef void __fastcall (__closure *TNotidyEvent)(System::TObject* Sender);
```

Java

Nie można zadeklarować wskaźnika do metody w Javie, ponieważ język ten nie posiada typu wskaźnikowego. To jedna z przyczyn, dla których Java posiada inny niż Object Pascal mechanizm zdarzeń. Wielką zaletą systemu zdarzeń Javy jest to, że zdarzenie OnClick w Javie (w jej dialekcie znane jako actionPerformed) może wskazywać na więcej niż jedną metodę w tym samym czasie. W Object Pascalu pojedyncza instancja TButton może posiadać tylko jedną obsługę zdarzenia OnClick na raz. Z drugiej strony, wielką zaletą systemu Pascal jest to, że jest on o wiele łatwiejszy do zrozumienia niż system Java, a także to, że nie wymaga tworzenia żadnych pośrednich obiektów, takich jak EventListeners.

Teatr tajemnic: Gdzie podziała się metoda FormCreate?

Jak zapewne pamiętasz, cała ta dyskusja zaczęła się w momencie, gdy zauważyliśmy, że zniknęła metoda `FormCreate`:

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

Jak już wiesz, metoda ta ma swoją deklarację w części interfejsu i swoją implementację w części implementacyjnej Twojego kodu. Chcąc ją ręcznie usunąć, musiałbyś wymazać kod w dwóch miejscach. W zwykłym edytorze tekstowym nie stanowi to żadnego problemu, ponieważ prawdopodobnie nie wstawiłbyś kodu, gdybyś nie miał takiego zamiaru. Jednak w środowisku programowania wizualnego, takim jak Kylix, można po prostu klikać sobie tu i tam; w takim przypadku zanim się zorientujesz, Twój moduł będzie wypełniony procedurami obsługi zdarzeń, których niekoniecznie potrzebowałeś. Aby jakoś temu zapobiec, w Kyliksie wprowadzona została zasada, na mocy której każda procedura obsługi zdarzenia, która nie zawiera kodu, powinna być całkowicie usuwana w momencie zapisu lub uruchomienia. Nie dotyczy to wszystkich metod — tylko obsługi zdarzeń.

Jeśli Kylix usunie coś, czego nie chciałeś usunąć, odtworzenie tego nie będzie trudne. Wracasz po prostu do inspektora obiektów, na zakładce *Events* odnajdujesz stosowną pozycję i tworzysz na nowo kod.

Z drugiej strony, jeśli kod obsługi jakiegoś zdarzenia nagle zaczął sprawiać wrażenie martwego, możesz po prostu usunąć kod, który dodałeś do metody i wcisnąć *Save* — jak widać, metoda zniknęła.

Jeśli nie chcesz implementować metody od razu, ale nie chcesz też, by zniknęła, możesz po prostu dodać do niej komentarz:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // foobar
end;
```

Zresztą wystarczy nawet umieszczenie samych dwu ukośników bez tekstu; poinformujesz w ten sposób IDE Kyliksa, że ma zachować tę metodę.

Pliki źródłowe Pascala

Kod napisany w Pascalu umieszcza się w trzech rodzajach plików. Oto one:

1. Główny plik programu, czasami nazywany także plikiem projektu Delphi
2. Moduł
3. Plik włączany (ang. *include file*)

Pliki projektu Delphi

Główny plik źródłowy projektu zawiera główny blok programu. Blok ten zaczyna się słowem zastrzeżonym `begin` i kończy słowem zastrzeżonym `end`. Oto bardzo prosty program w Pascalu:

```
program SimpleProgram;
begin
  WriteLn('Simple Program');
end.
```

Na program ten składają się zaledwie główny blok i nagłówek programu. Jak widzisz, blok programu zaczyna się słowem `begin` i kończy słowem `end`. Po ostatnim `end` zapisano kropkę, ponieważ oznacza ono w tym przypadku koniec programu. Jest to w tym pliku jedyne miejsce, w którym po `end` występuje kropka.

Jeśli projekt składa się z jednego pliku, główny blok programu jest jego punktem wejścia. Jako taki jest odpowiednikiem funkcji `main` lub `WinMain` w programach w C++.

Główny plik programu może mieć jedno lub dwa rozszerzenia. W większości programów w Delphi, główny plik powinien mieć rozszerzenie *.dpr* (od *Delphi Project*), jak w *SimpleProgram.dpr*. Dla głównego pliku programu poprawne jest rozszerzenie *.pas*, jednakże w większości przypadków nie używałbym go, ponieważ IDE nie zapewnia tego samego zestawu usług dla plików *pas*, co dla plików *dpr*.

Moduły

Moduły (ang. *units*) to pliki w Pascalu, które należą do projektu. Plik główny, a także plik projektu, posiadają zwykle wiele modułów, które są ich własnością.

Moduły mają rozszerzenie *.pas*. A zatem w typowej aplikacji Delphi plik główny ma rozszerzenie *.dpr*, a pozostałe pliki *.pas*.

Moduły podzielone są na dwie części. Część początkowa znana jest jako *interfejs* (ang. *interface*), a część główna nazywana jest *implementacją* (ang. *implementation*). Obydwie pojawiają się w jednym pliku. Programiści C lub C++ mogą myśleć o interfejsie jak o odpowiedniku tego, co nazywają plikiem nagłówkowym.

Wydruki 1.2 i 1.3 przedstawiają przykład bardzo prostego programu składającego się z pliku projektu i prostego modułu.

Wydruk 1.2. *Plik główny projektu SimpleUnit*

```
program SimpleUnit;

uses
  MyUnit;

begin
  DoSomething;
end.
```

Wydruk 1.3. *Moduł należący do programu SimpleUnit*

```
unit MyUnit;

interface

procedure DoSomething;

implementation

procedure DoSomething;
begin
  WriteLn('Jestem poetą ciała i jestem poetą duszy -Whitman');
end;

end.
```

Moduły zostały stworzone z myślą o rozbijaniu kodu na logiczne segmenty. Np. w jednym można umieścić cały kod zajmujący się wejściem-wyjściem, w drugim natomiast możemy trzymać kod obsługujący bazę danych, itd.

W Delphi moduły służą jeszcze drugiemu celowi. Jeśli chcesz edytować formularze przy użyciu projektanta formularzy, każdy z nich musi mieć swój własny moduł. Można umieścić kilka formularzy w jednym module, lecz projektant formularzy nie będzie wtedy działał poprawnie.

Śledząc wydruk 1.3 możesz zobaczyć, że interfejs modułu może zawierać deklaracje. Z drugiej strony, implementacja zawiera wykonywalny kod modułu. Moduł pokazany na wydruku 1.3 jest bardzo prosty. W większości programów moduł będzie jednak zawierał wiele różnych elementów, takich jak definicje stałych, typów, klas, czy funkcji.

Punkty wejścia programów w Pascalu

Nie zawsze główny blok programu w Pascalu jest jego punktem wejścia. Wydruk 1.4 zawiera zmienioną wersję modułu przedstawionego na wydruku 1.3.

Wydruk 1.4. *Zmieniony moduł należący do programu SimpleUnit*

```
unit MyUnit;

interface

procedure DoSomething;

implementation

var
  MyString: string;

procedure DoSomething;
begin
  WriteLn(MyString);
end;
```

```
initialization

  MyString:= 'Obiecuję Ci, że ten, który przyjmie moje słowo, nie ujrzy nigdy śmierci';

finalization

  MyString:= '';

end.
```

Ta wersja modułu pokazuje, że nie zawsze główny blok programu będzie jego punktem wejścia. Sekcja implementacji modułu `MyUnit` zawiera klauzulę `initialization` oraz `finalization`. Kod z sekcji `initialization` jest wywoływany w momencie, gdy moduł jest pierwszy raz ładowany do pamięci. Dzieje się to jeszcze zanim główny blok programu zostanie wykonany. Gdy tylko program zostanie wczytany do pamięci, kod wykonuje skok do pierwszego modułu wypisanego w bloku `uses`. Jeśli będzie w nim znajdować się sekcja inicjalizacji, zostanie ona wykonana.

Jeśli projekt zawiera wiele modułów, wszystkie ich sekcje inicjalizacji zostaną wykonane zanim kod w ogóle dojdzie do głównego bloku programu. Mówiąc dość oględnie, jest to duża różnica w stosunku do tego, co dzieje się w programie C/C++.

Object Pascal wyposażony jest w bloki inicjalizacji, by dać twórcom możliwość inicjalizacji zmiennych zadeklarowanych w danym module. Aby mogli oni wyczyścić kod, który mógł zostać wykonany, dodano sekcję zakończenia (`finalization`).

Klauzula uses i błędne koła odwołań

Na samym początku wydruku 1.2 możesz zauważyć element składni znany jako klauzula `uses`. W tej opcjonalnej klauzuli możesz wyliczyć wszystkie moduły, od których Twój plik zależy. W naszym przypadku główny plik programu `SimpleUnit` zależy od `MyUnit`.

W poprzedniej sekcji opisałem, co się dzieje podczas wykonywania programu. Ciekawe, że coś bardzo podobnego dzieje się podczas przetwarzania plików w czasie kompilacji. Pierwsza klauzula z głównego pliku programu jest podawana do kompilatora. W przykładzie, który omawialiśmy, kompilator najpierw otworzy *SimpleUnit.dpr* i odkryje, że `MyUnit` został użyty w jego klauzuli `uses`. Zaczyna więc przetwarzać `MyUnit`. Gdy skończy, wraca do pliku głównego, napotyka następny moduł w polu `uses` i zaczyna go przetwarzać. Jeśli jakikolwiek moduł z klauzuli `uses` programu głównego zawiera pole `uses`, które wskazuje na kolejne moduły, zostaną one przetworzone przed modułami wyliczonymi w programie głównym.

Moduły mogą zawierać swoje własne pola `uses`. Pierwsze takie pole w module znajduje się bezpośrednio pod słowem `interface`; drugie za słowem `implementation`:

```
unit Foo;

interface

uses
  unit1, unit2;
```

```
implementation

uses
  unit0, unit3;

end.
```

Jeżeli unit1 zawiera unit2 w swoim pierwszym uses, to unit2 w swoim pierwszym uses nie może zawierać unit1. Jeśli spróbujesz utworzyć taką, niewłaściwą zależność, zgłoszony zostanie błąd circular unit reference. Aby temu zapobiec, przenieś odwołanie do modułu unit1 w module unit2 niżej, do pola uses sekcji implementacji; lub odwrotnie — w module unit1 przenieś do tejże sekcji odwołanie do modułu unit2.

W większości przypadków to rozwiąże problem. W kilku jednak takie działanie nie wystarczy. Żeby zrozumieć dlaczego takie rozwiązanie może nie działać, musisz wiedzieć, że moduł unit1 nie dostrzeże żadnych elementów w unit2, dopóki ten nie zostanie zapisany w jego polu uses. Tak więc, jeśli w interfejsie modułu unit1 potrzebujesz odwołać się do czegoś, co zostało zadeklarowane w interfejsie modułu unit2, musisz wpisać unit2 na listę pierwszego uses modułu unit1. Jeżeli oprócz tego potrzebujesz wykorzystać w interfejsie modułu unit2 coś zdefiniowanego w interfejsie modułu unit1, to masz pecha. Rozważ przykłady przedstawione na wydrukach 1.5 i 1.6.

Wydruk 1.5. *Moduł deklarujący typ TMyArray i używający modułu unit2 w swojej klauzuli uses*

```
unit unit1;

interface

uses
  unit2;

type
  TMyArray = array[1..0] of TUnit2Type;

var
  Unit2Type: TUnit2Type;

implementation

end.
```

Wydruk 1.6. *Moduł wykorzystywany przez unit1, używający typu TMyArray z modułu unit1 oraz definiujący typ TUnit2Type, używany przez moduł unit1*

```
unit unit2;

interface

uses
  unit1;

type
  TUnit2Type: Integer;

var
  TFooArray: TMyArray;
```

```
implementation

end.
```

W przykładach tych można wskazać zakorzeniony bardzo głęboko istotny problem, którego rozwiązanie wymaga dużej ostrożności. Obydwa moduły odwołują się do siebie nawzajem w swoich pierwszych klauzulach `uses`. Nie jest to dozwolone, ponieważ powoduje efekt błędnego koła odwołań. Nie można tego też rozwiązać standardową techniką przenoszenia odwołania z sekcji interfejsu do sekcji implementacji. Nie da to pożądanego efektu, ponieważ każdy z modułów wykorzystuje elementy zadeklarowane w interfejsie drugiego. Mówiąc ściślej, `unit2` odwołuje się do definicji typu `TMyArray`, podczas gdy `unit1` wykorzystuje `TUnit2Type`.

Jeśli kiedykolwiek napotkasz tego typu problem, będziesz mógł go rozwiązać albo poprzez połączenie obu plików w jeden, albo utworzenie trzeciego. W niektórych przypadkach łatwiej jest po prostu złączyć dwa pliki w jeden. W innych prościej będzie przenieść niektóre elementy z `unit1` do `unit2` i vice versa. Jeszcze w innych, najlepszym rozwiązaniem będzie utworzenie trzeciego modułu i umieszczenie w nim odpowiednich deklaracji z pozostałych dwu. To, które rozwiązanie uznasz za najlepsze, zależy przede wszystkim od Twoich upodobań, a także od samych określonych przypadków.

Problem błędnego koła odwołań nie nęka programistów C/C++. Trzeba zatem uznać, że w tym jednym przypadku, język C/C++ ma przewagę nad Object Pascalem. Warto jednak zauważyć, że tego typu problemy powstają, gdy program jest źle zorganizowany. Np. konstrukcja modułów przedstawionych na wydrukach 1.5 i 1.6 jest wyraźnie przypadkowa, wręcz nonsensowna. Jeśli poprawnie konstruujesz swoje programy, tego typu problemy nie powinny często powstawać. Ujmując rzecz z drugiej strony; gdy napotkasz tego typu błąd, powinieneś zastanowić się, czy nie jest on po prostu sposobem, w jaki kompilator próbuje poinformować Cię o tym, że powinieneś jeszcze raz przemyśleć konstrukcję swojego programu, ponieważ ma ona słaby punkt.
Po napisaniu tego wszystkiego, przyznam teraz, że w niektórych bardzo rzadkich przypadkach konstruowanie modułów wzajemnie się odwołujących ma sens; wówczas omawiany błąd może być irytujący. Niemniej jednak, jeśli dostaniesz tego typu błąd, musisz zaimplementować jedno z wspomnianych wyżej rozwiązań.

Na koniec jeszcze jedna zasada, która powinna utkwić w pamięci podczas konstruowania modułów: kiedy deklarujesz poszczególne sekcje w interfejsie, nie musisz zwracać uwagi na ich porządek. Zwykle sekcja `const` jest pierwsza, po niej następuje sekcja `type`, a następnie sekcja `var`. Nic jednak nie stoi na przeszkodzie, aby odwrócić ten porządek. Co więcej, możesz mieć w interfejsie wiele sekcji danego typu. Np. po sekcji `const` może następować sekcja `type`, po której może znajdować się znów sekcja `const`.

Kompilacja w Kyliksie

Może wydaje Ci się nieco zaskakujące to, że Kylix zna tworzony przez Ciebie kod, nawet jeśli nie został on jeszcze skompilowany. Okazuje się, że kod, który wpisujesz, jest nieustannie, na bieżąco, kompilowany przez IDE. Jego binarna postać nie jest zapisywana na dysk, lecz trzymana w pamięci i używana do wykrywania składników kodu. IDE „wie" np., czy pracujesz nad obsługą zdarzenia, czy nad inną metodą, którą sam utworzyłeś. Informacja ta jest wykorzystywana przez IDE na zakładce *Events* w inspektorze obiektów,

a także do automatycznego wymazywania pustych procedur obsługi zdarzeń podczas zapisywania kodu. Wykorzystuje ją również narzędzie o nazwie Code Insight, omówione w rozdziale 5.

Jak już zapewne zauważyłeś, aplikacje w Kyliksie kompilują się w bardzo krótkim czasie. Małe projekty zdają się kompilować natychmiast, a te wielkie w zaledwie kilka sekund. Taki stan rzeczy ma dwie przyczyny:

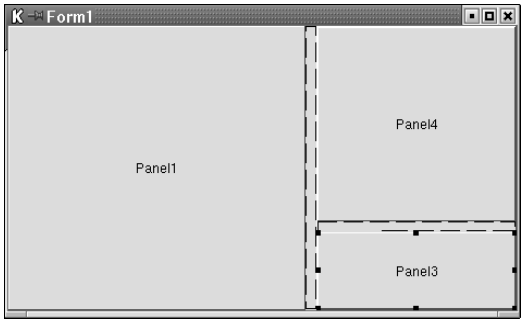
- Object Pascal jest dosyć prostym językiem. Unikano w nim elementów składniowych o skomplikowanym i wielorakim znaczeniu, takich jak przeciążony operator[1] czy wielokrotne dziedziczenie. W rezultacie kod, który piszesz, może być nie tylko bez trudu zrozumiany przez użytkownika, lecz także nie sprawia kłopotu kompilatorowi podczas przetwarzania. Czasy kompilacji programów są naprawdę bardzo krótkie.
- Dzięki temu, że Object Pascal używa swojego własnego formatu binarnego, który jest często czymś więcej, niż tylko utworzoną na dysku prostą kopią struktur znajdujących się w pamięci, cykl łączenia jest przyspieszony. W szczególności, kompilator przetwarza kod i zapisuje z trudem zdobyte w tym procesie informacje na dysk. Format plików *o* i *obj* generowanych przez programy C++ jest bardzo złożony i ma mało wspólnego z obrazem, który powstaje w pamięci w wyniku kompilacji. Odpowiednikiem plików *o* i *obj* w Object Pascalu jest plik *dcu*. Z punktu widzenia kompilatora te pliki są stosunkowo proste w zapisie i ponownym odczytaniu do pamięci. Jednym słowem, są one dla kompilatora prostsze w przetwarzaniu.

Prędkość działania aplikacji Kyliksa skompilowanej bez załączania informacji dla debugera, uruchomionej poza IDE, powinna być w zasadzie porównywalna z prędkością działania aplikacji C lub C++. W chwili, gdy piszę te słowa, nie ma jeszcze ostatecznych danych dla Kyliksa. W przeszłości jednak, kod skompilowany w Object Pascalu Borlanda był tak szybki, jak kod C/C++, lub przynajmniej zaledwie o kilka procent wolniejszy. W odróżnieniu od Javy i Visual Basica, skompilowany kod Object Pascala nigdy nie był trzy lub nawet cztery razy wolniejszy od kodu w C/C++.

Praca z pojemnikami (Containers)

Kolejne zadanie wykonywane w projektancie formularzy związane jest z przydzielaniem przestrzeni poprzez pracę z kontrolkami paneli i zakładek. Możesz np. chcieć stworzyć aplikację z listą wyboru po lewej stronie. Gdy użytkownik kliknie listę, w podwójnej przestrzeni po jej prawej stronie będą pojawiały się dodatkowe informacje, jak pokazano na rysunku 1.19. Co więcej, możesz chcieć, by użytkownik mógł zmieniać rozmiar poszczególnych elementów okna.

[1] W Kyliksie i Delphi 6 istnieje specyficzny sposób przeciążania operatorów z wykorzystaniem typu Variant — *przyp. red.*

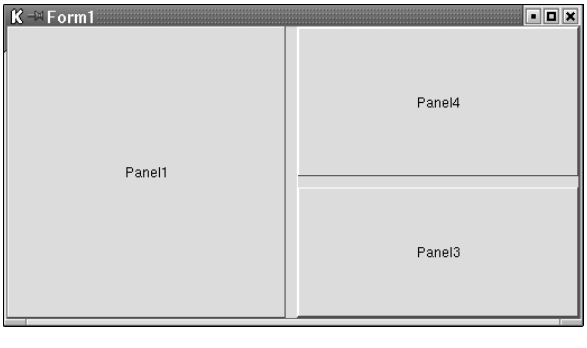


Rysunek 1.19. *Złożona forma z wieloma powierzchniami — kontrolka rozdzielacza (ang. splitter control) pozwala na zmianę rozmiarów każdej z nich*

Aby stworzyć taki formularz, zrób co następuje:

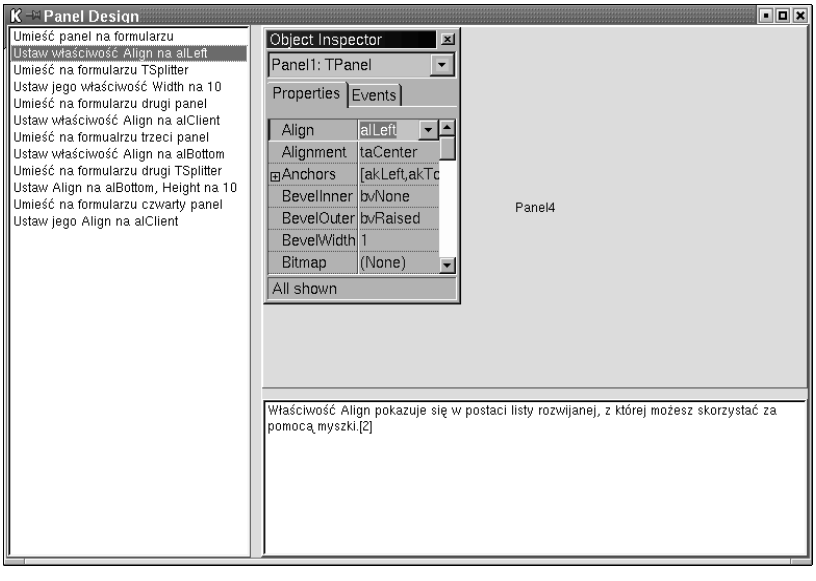
1. Upuść na formularzu panel. Ustaw jego właściwość `Align` na `alLeft`.
2. Zmień zakładkę na *Additional* i umieść kontrolkę `TSplitter` na prawo od panelu. Ustaw jego szerokość (`Width`) na `10`, tak żeby użytkownik mógł go łatwo chwycić podczas działania programu.
3. W pustej przestrzeni po prawej stronie umieść drugi panel i ustaw jego właściwość `Align` na `alClient`.
4. W górnej części drugiego panelu umieść trzeci, z właściwością `Align` ustawioną na `alBottom`.
5. Upuść drugi rozdzielacz na drugim panelu. Ustaw jego właściwość `Align` na `alBottom`, a wysokość (`Height`) na `10`.
6. Na górze drugiego panelu, ponad drugim rozdzielaczem, umieść czwarty panel. Ustaw jego właściwość `Align` na `alClient`.

Uruchom program i zmień rozmiary okna, tak by wyglądało jak na rysunku 1.20. Zauważ, że takie rozstawienie daje Ci trzy w pełni regulowane przestrzenie, które — jeśli tego zażądamy — będą zachowywać się jak formularze zagnieżdżone w większym formularzu.



Rysunek 1.20. *Możesz użyć rozdzielaczy do stworzenia paneli o zmiennych rozmiarach; mogą Ci się one przydać w Twoich programach*

Możesz użyć tego podstawowego wzorca do różnych aplikacji, takich jak ta w programie `PanelDesign`, który znajdziesz w katalogu *Chap01* w źródłach dostarczonych razem z tą książką. Program ten pokazany jest na rysunku 1.21; omówiony zostanie w następnej sekcji, zatytułowanej „Program PanelDesign”.



Rysunek 1.21. *Program powstały na bazie paneli o zmiennych rozmiarach, instruujący jak tworzyć takie właśnie panele*

Program PanelDesign

Program `PanelDesign` łączy w sobie wiele idei, które jak dotąd zostały omówione w tym rozdziale. Pokazano go na wydrukach od 1.7 do 1.9.

Wydruk 1.7. *Kod programu PanelDesign*

```
unit Main;

interface

uses
  SysUtil, Types, Classes,
  QGraphics, QControls, QForms,
  QDialogs, QExtCtrls, QStdCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Splitter1: TSplitter;
    Panel2: TPanel;
    Panel3: TPanel;
    Splitter2: TSplitter;
    Panel4: TPanel;
    ListBox1: TListBox;
    Image1: TImage;
    Memo1: TMemo;
    procedure FormShow(Sender: TObject);
    procedure ListBox1Click(Sender: TObject);
  private
    FList: TStringList;
    function GetSelNum: String;
    procedure ParseListRes(var TempList: TStringList; StartNum: Integer);
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses
  LCodeBox;

{$R *.xfm}

procedure TForm1.FormShow(Sender: TObject);
var
  TempList: TStringList;
begin
  ListBox1.Items.LoadFromFile('instructions.txt');
  FList := TStringList.Create;
  TempList := TStringList.Create;
  TempList.LoadFromFile('descriptions.txt');
  ParseListRes(TempList);
  FList.SaveToFile('Foo.txt');
  TempList.Free;
end;

function TForm1.GetSelNum(): String;
var
  ListNum: Integer;
begin
  ListNum := ListBox1.ItemIndex + 1;
  Result := Int2StrPad0(ListNum, 2);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
var
  PicName: string;
begin
  PicName := GetStartDir() + 'step' + getSelNum + '.bmp';
  if FileExists(PicName) then
    Image1.Picture.LoadFromFile(PicName);
  if ListBox1.ItemIndex < FList.Count then
    Memo1.Text := FList.Strings[ListBox1.ItemIndex];
end;

procedure TForm1.ParseListRes(var TempList: TStringList);
var
  i: Integer;
  S, WholeLine: string;
  NewPar: Boolean;
begin
  WholeLine := '';

  for i := 0 to 2 do      // Sprawdzam TempList.Strings[i + 1] dlatego
    TempList.Append('');  // te linie muszą być tutaj!
```

```
  NewPar := False;
  for i := 0 to TempList.Count - 1 do begin
    S := TempList.Strings[i];
    S := StripEndChars(S, #32);
    if (S <> '') then
      S := S + #32;
    if (i = 0) or (NewPar and (S <> '')) then begin
      WholeLine := S;
      if TempList.Strings[i + 1] <> '' then begin
        NewPar := False;
      end else begin
        NewPar := True;
        FList.Add(WholeLine);
        WholeLine := '';
      end;
    end else if S = '' then
      // Nic nie rób
    else if (i < (TempList.Count - 1)) and (TempList.Strings[i + 1] = '') then
    begin
      NewPar := True;
      WholeLine := WholeLine + S;
      FList.Add(WholeLine);
      WholeLine := '';
    end else begin
      S := StripBlanks(S);
      S := S + #32;
      WholeLine := WholeLine + S
    end;
  end;
end;

end.
```

Wydruk 1.8. *Plik Instructions.txt, który określa kolejne kroki budowania w Kyliksie multipanelowej aplikacji*

```
Umieść panel na formularzu.
Ustaw jego właściwość align na alLeft.
Umieść na formularzu TSplitter.
Ustaw jego właściwość Width na 10
Umieść na formularzu drugi panel
Ustaw jego właściwość Align na alClient.
Umieść na formualrzu trzeci panel
Ustaw jego właściwość Align na alBottom.
Umieść na formularzu drugi komponent TSplitter
Ustaw Align na alBottom, wysokość na 10.
Umieść na formularzu czwarty panel
Przypisz jego właściwości Align wartość alClient.
```

Wydruk 1.9. *Plik directions.txt, w którym znajdują się komentarze na temat każdego kroku potrzebnego do zbudowania w Kyliksie multipanelowej aplikacji*

```
Panel znajduje się na zakładce Standard.[1]

Właściwość Align pokazuje się w postaci listy rozwijanej,
z której możesz skorzystać za pomocą myszki.[2]
```

```
Komponent TSplitter znajduje się na zakładce
Additional palety komponentów.[3]

Dwiema najważniejszymi właściwościami kontrolki
rozdzielacza są jej ustawienie (Align) oraz rozmiary
(Width i Height). Domyślną wartością właściwości
Align jest alLeft, co jest dokładnie tym, czego
chcemy, dlatego zmodyfikować musimy jedynie
właściwość Width.[4]

Upuszczając drugi panel, upewnij się, że znajdzie
się on na pustej części formularza po prawej
stronie. To, gdzie umieszczasz komponent, jest bardzo
istotne. Jeśli swój panel umiejscowiłbyś na górze
komponentu Panel1, ciężko byłoby go potem przesunąć
poza jego obszar.

Ustawienie właściwości Align komponentu na alClient
powoduje, że wypełnia on całe dostępne miejsce
na swoim obiekcie nadrzędnym (rodzicu). W naszym przypadku
obiektem nadrzędnym jest formularz.

Ponownie, pamiętaj, żeby umieścić ten panel po właściwej
stronie formularza.

Po ustawieniu właściwości Align na alBottom, nowy
panel zajmie cały spód swojego rodzica, który w tym
przypadku nie jest formularzem, ale panelem.

Nie przejmuj się tym, w jaki sposób po upuszczeniu
ustawia się kontrolka rozdzielacza. Umieść ją po prostu
na komponencie mającym być jej obiektem nadrzędnym.

Określając wartość właściwości Align rozdzielacza
i dostosowując jego wysokość, możesz umieścić go
dokładnie w miejscu, w którym sobie zażyczysz.

I znowu, pamiętaj o tym, aby umieścić go dokładnie
w pustej przestrzeni na panelu Panel2. Nie pozwól, żeby
podczas upuszczania dotykał on komponentów Panel1 i Panel3.

Ustaw właściwość Align panelu, żeby umieścić go we
właściwym miejscu. Nie trać czasu próbując ręcznie
umiejscowić ten komponent.
```

Interfejs programu `PanelDesign` posiada po lewej stronie listę wyboru do przechowywania pojedynczych linii tekstu. Na dole, po prawej stronie, znajduje się kontrolka `TMemo`, która przechowuje wiele linii tekstu. Na górze widać kontrolkę `TImage`, która pozwala programiście w prosty sposób wyświetlić użytkownikowi obrazek.

Lista wyboru zawiera plik *instructions.txt*, który wylicza kolejne etapy potrzebne do stworzenia multipanelowego programu. Kontrolka `image` (obraz) pokazuje obrazki przedstawiające, co użytkownik ma zrobić w każdym kroku. Kontrolka `memo` dodaje komentarz do każdego kroku i przedstawia szczegóły, które mogą się przydać nowym w środowisku Kylix.

Program pozwala użytkownikowi klikać pojedyncze linie tekstu w liście. Po każdym kliknięciu, możesz zobaczyć towarzyszący rysunek w kontrolce `image` i komentarz w kontrolce `memo`.

Teraz, gdy rozumiesz, o czym jest program, dowiedz się, jak to wszystko zostało razem zebrane. Mam nadzieję, że przekonasz się, iż Kylix bardzo upraszcza tworzenie tego typu — stosunkowo skomplikowanych — programów. Nie będę objaśniał co zrobić, żeby stworzyć taki interfejs, ponieważ program robi to sam. W zamian pokażę Ci kod potrzebny do tego, by program zachowywał się właściwie w czasie działania.

Inicjalizacja formularza programu PanelDesign

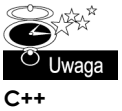
Zdarzenie `OnShow` komponentu jest wywoływane tuż przed wyświetleniem go użytkownikowi. W momencie, gdy wywoływane jest to zdarzenie, wszystkie istotne elementy Twojej aplikacji zostały już zainicjalizowane. Inne zdarzenie, `OnCreate`, jest wywoływane przed `OnShow`. `OnCreate` jest dobrym miejscem na umieszczenie kodu, który nie wpływa na wygląd interfejsu programu. Jednakże, jeśli chcesz poustawiać właściwości formularza, zanim ten zostanie wyświetlony użytkownikowi, najlepiej zrób to w `OnShow`, nie w `OnCreate`.

Oto zdarzenie `OnShow` dla głównego formularza programu `PanelDesign`:

```
procedure TForm1.FormShow(Sender: TObject);
var
  TempList: TStringList;
begin
  ListBox1.Items.LoadFromFile('instructions.txt');
  FList := TStringList.Create;
  TempList := TStringList.Create;
  TempList.LoadFromFile('descriptions.txt');
  ParseListRes(TempList);
  FList.SaveToFile('Foo.txt');
  TempList.Free;
end;
```

Zauważ, że tylko jedną linijkę zajmuje wczytanie do listy wyboru zawartości pliku *instructions.txt*. Właściwość `Items` listy jest typu `TStrings`. Obiekt `TStrings` został zaprojektowany z myślą o przechowywaniu listy łańcuchów znakowych. Jeśli umieścisz takowy we właściwości `Items` obiektu `TListBox`, łańcuch ów zostanie wyświetlony na liście wyboru.

Obiekt `TStringList` pochodzi od obiektu `TStrings`. `TStringList` jest przeznaczony do użytku jako pojemnik (ang. *container*) łańcuchów znakowych. Posiada wszystkie właściwości obiektu `TStrings`, nie jest jednak przywiązany do żadnej kontrolki. Reprezentuje samego siebie.



C++

Obiekt `TStringList` posiada analogiczny obiekt, nazwany `TList`. Obiekt `Tlist` został opracowany po to, by przechowywać dowolne obiekty, podczas gdy `TStringList` utworzono po to, by przechowywać łańcuchy znaków. Te dwa obiekty odgrywają razem rolę podobną do tej, jaką odgrywają kluczowe klasy w bibliotece STL w programach w C++.

Java

Klasy `TStringList` i `TList` w Object Pascalu odgrywają tę samą rolę, jaką może odgrywać w Javie tablica obiektów `String` lub obiekt `ArrayList`.

Oto jak tworzy się instancję obiektu TStringList:

```
FList := TStringList.Create;
```

A oto kod, który najpierw tworzy obiekt TStringList, potem wczytuje do niego plik *descriptions.txt* i w końcu przekazuje listę obiektów następnej procedurze:

```
TempList := TStringList.Create;
TempList.LoadFromFile('descriptions.txt');
ParseListRes(TempList);
```

Funkcja ParseListRes rozbija zawartość pliku *descriptions.txt* w taki sposób, że każdy akapit traktowany jest jako pojedynczy łańcuch znaków. Algorytm zaczyna od początku pliku, wczytuje pierwszą linię, wyrzuca z jej początku i końca wszystkie białe znaki i zapisuje wynik do zmiennej WholeLine. Potem w pętli znajduje następną linię w pliku, obcina jej końcowe bity i dodaje do WholeLine. Kontynuuje te działania do momentu napotkania pustej linii. Jest ona traktowana jako znacznik końca akapitu i początek nowego. Tak więc, gdy ma już cały akapit, dołącza jego tekst do obiektu TStringList:

```
FList.Add(WholeLine);
```

Algorytm kontynuuje swoją drogę przez cały plik, dodając akapit za akapitem do TStringList. Gdy skończy, wychodzi z pętli i zwraca sterowanie z powrotem do metody FormShow.

Obsługa wejścia użytkownika w programie PanelDesign

Obiekt ListBox posiada zdarzenie nazwane OnClick, które wywoływane jest za każdym razem, gdy użytkownik kliknie listę. Implementacja tego zdarzenia w naszym programie, razem z pomocniczą metodą GetSelNum, wygląda następująco:

```
function TForm1.GetSelNum(): String;
var
  ListNum: Integer;
begin
  ListNum := ListBox1.ItemIndex + 1;
  Result := Int2StrPad0(ListNum, 2);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
var
  PicName: string;
begin
  PicName := GetStartDir() + 'step' + getSelNum + '.bmp';
  if FileExists(PicName) then
    Image1.Picture.LoadFromFile(PicName);
  if ListBox1.ItemIndex < FList.Count then
    Memo1.Text := FList.Strings[ListBox1.ItemIndex];
end;
```

Parametr Sender przekazany metodzie ListBox1Click jest kopią obiektu, który utworzył zdarzenie. W tym wypadku zawsze będzie to lista wyboru. Aby dostać się do obiektu, skorzystam z zasad polimorfizmu i napiszę kod, który wygląda tak:

```
procedure TForm1.ListBox1Click(Sender: TObject);
var
  MyListBox: TListBox;
```

```
begin
  if Sender if TListBox then begin
    MyListBox := TListBox(Sender);
    MyListBox.Items.Add('Staś');
  end;
end;
```

Kod ten jest wprawdzie poprawny, jednak w programie PanelDesign nie ma z niego pożytku. Nie ma rzeczywistej potrzeby używania obiektu Sender, więc po prostu go ignoruję.

W metodzie ListBox1Click chciałbym przede wszystkim wyznaczyć nazwę obrazka towarzyszącego wyborowi użytkownika. Aby utworzyć tę nazwę, wywołuję metodę GetSelNum. Ożywa ona właściwości TListBox.ItemIndex do wyznaczenia pozycji na tej liście, na której kliknął użytkownik:

```
ListNum := ListBox1.ItemIndex + 1;
```

Następnie wywołuję metodę Int2StrPad0, która zapewnia, że wynik zwrócony przez funkcję będzie miał dwa znaki, nie tylko jeden. Będziemy chcieli otrzymać 01, a nie tylko 1, jeśli użytkownik kliknął np. pierwszą pozycję z listy:

```
Result := Int2StrPad0(ListNum, 2);
```

Funkcje i wbudowana zmienna Result

Int2StrPad0 można znaleźć w module LCodeBox.pas. Moduł ten nie jest dostarczony razem z Kyliksem. Znajdziesz go jednak wśród materiałów dołączonych do tej książki; jest także dostępny na mojej stronie WWW, o adresie *www.elvenware.com*.

```
function Int2StrPad0(N: LongInt; Len: Integer): string;
begin
  FmtStr(Result, '%d', [N]);
  while Length(Result) < Len do
    Result := '0' + Result;
end;
```

Zmienna Result jest predeklarowana dla wszystkich funkcji w Kyliksie. Zawsze jest typu zwracanego przez daną funkcję. W tym wypadku Result jest łańcuchem.

C/C++, Java

W C/C++ i Javie wartość zwracana przez funkcję wyznaczana jest przez użycie instrukcji return:

```
int result;
result = 2 + 2;
return result;
```

Można tu dostrzec dwie istotne różnice w sposobie wykonywania takich rzeczy pomiędzy Object Pascalem a Javą i C++. Po pierwsze istotne jest to, czy zwracana zmienna jest deklarowana przez Ciebie, czy przez kompilator. Po drugie, w Pascalu użycie instrukcji return jest ukryte, podczas gdy w C i w Javie jest ono jasno określone. Moim zdaniem, obie techniki są intuicyjne i proste w użyciu.

Int2StrPad0 wywołuje Pascalową procedurę FmtStr, która jest odpowiednikiem funkcji sprintf w C/C++. W swoich pierwszych dwu parametrach pobiera ona łańcuch, zaś jako trzeci parametr przyjmuje tablicę wartości:

```
procedure FmtStr(
  var StrResult: string;         // Wartość zwracana przez procedurę
  const Format: string;          // Łańcuch z osadzonymi znakami formatującymi
  const Args: array of const );  // Wartości do umieszczenia w łańcuchu
```

Łańcuch formatu zawiera te same sekwencje sterujące, które używane są w funkcji `sprintf`. Do wartości, które możesz przekazać, należą: `%d` (liczba całkowita), `%u` (reprezentacja dziesiętna), `%e` (zmiennoprzecinkowe, w postaci naukowej), `%f` (jak wcześniej, lecz w prostszej postaci), `%g` (odpowiednik `%e` lub `%f`), `%m` (pieniądze), `%n` (liczba), `%p` (wskaźnik), `%s` (łańcuch znaków), `%x` (heksadecymalnie). Więcej szczegółów znajdziesz w podręcznej pomocy Kyliksa, pod hasłem `Format Strings`.

Parametr `Args` zawiera tablicę argumentów. Rozważ np. ten fragment kodu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S, Temp: string;
begin
  S := 'Zuzia';
  Temp := 'Zmienna S zawiera łańcuch %s, ma adres $%p, a jej długość wynosi %d
znaków.';
  FmtStr(S, Temp, [S, @S, Length(S)]);
  Edit.Text := S;
end;
```

Można tutaj zobaczyć, że przez parametr `Args` zostały przekazane trzy wartości. Pierwszą jest łańcuch, drugą adres tego łańcucha, a trzecią jego długość. Po wywołaniu procedury, `StrResult` wygląda następująco: `Zmienna S zawiera łańcuch Zuzia, ma adres $0012F5E0, a jej długość wynosi 5 znaków`.

C/C++

W C/C++ adres zmiennej pobierasz za pomocą symbolu `&`, w Pascalu zaś używasz symbolu `@`. W C/C++ wyznaczasz wartość szesnastkową poprzez `0x`, jak w `0xFFFF`; w Pascalu używasz do tego symbolu `$`, np. `$FFFF`. Zachowanie tych symboli jest w obydwu językach identyczne, różnią się jedynie same symbole.

Funkcja `Int2StrPad0` jest zakończona następującą pętlą `while`:

```
while Length(Result) < Len do
  Result := '0' + Result;
```

Ten fragment kodu mówi, że `'0'` ma być wstawiane na początek łańcucha `Result`, dopóki nie osiągnie on długości określonej w zmiennej `Len`. Np. jeśli `Result` wynosi 2, a `Len` jest równe 3, po pierwszym przejściu przez pętlę `Result` będzie miało wartość 02, a po drugim przejściu osiągnie 002. W tym momencie spełniony zostanie warunek wyjścia z pętli, a zatem zakończy się ona.

Nie będę w tej książce objaśniał jak działa w Object Pascalu pętla `while`. Doświadczeni programiści znajdą wszystkie istotne informacje o niej, wpisując słowo `while`, w edytorze Kyliksa, umieszczając na nim kursor i naciskając klawisz *F1*. Pojawi się okno pomocy, wyświetlające wszystkie informacje niezbędne do skonstruowania takiej pętli. W gruncie rzeczy pętla `while` jest po prostu pętlą `while`, bez względu na to, czy pisana jest w C, Object Pascalu, Javie czy BASIC-u.

Istnieje jeszcze drugi sposób na przetłumaczenie w Object Pascalu liczby całkowitej na łańcuch:

```
var
  S: string;
  i: Integer;
begin
  i := 2;
  S := IntToStr(i);
end;
```

W bardzo dużym stopniu intuicyjną funkcję `IntToStr` będę w tej książce wykorzystywał jeszcze nie raz. Towarzyszy jej funkcja `StrToInt`, która działa w taki oto sposób:

```
var
  S: string;
  i: Integer;
begin
  S := '2';
  i := StrToInt(S);
end;
```

Jeśli któraś z tych funkcji nie powiedzie się, to spowoduje wyjątek (ang. *exception*). Obydwie są w bardzo dużym stopniu intuicyjne i znakomicie zaprojektowane. Jak wszystkie dobrze zaprojektowane techniki, są one tak proste i oczywiste, że można się tylko zastanawiać, dlaczego ludzie w ogóle próbują wykonać to zadanie innymi sposobami.

Wczytywanie obrazu z dysku

Na tym etapie powinieneś rozumieć już pierwsze dwie linie metody `ListBox1Click`:

```
PicName := GetStartDir() + 'step' + GetSelNum + '.bmp';
if FileExists(PicName) then
  Image1.Picture.LoadFromFile(PicName);
```

Łańcuch `PicName` zawiera słowo `'step'` oraz tekstową wersję numeru pozycji zaznaczonej na liście wyboru i rozszerzenie `'.bmp'`. Jeśli użytkownik kliknął np. trzecią pozycję z listy, wartość `PicName` będzie wynosiła `step03.bmp`.

Funkcja `GetStartDir`, pochodząca z modułu `LCodeBox`, wygląda następująco:

```
function GetStartDir: string;
begin
  Result := ExtractFilePath(ParamStr(0));
  Result := IncludeTrailingPathDelimiter(Result)
end;
```

Wszystkie funkcje wywoływane przez `GetStartDir` są wbudowanymi funkcjami Pascala. `ParamStr(0)` zwraca pełną ścieżkę z nazwą aktualnie uruchomionego programu. `ExtractFilePath` wyciąga z tego jedynie ścieżkę. Jeśli np. `ParamStr(0)` jest ustawione na `/home/ccalvert/src/srcpas/PanelDesign/PanelDesign`, to `ExtractFilePath` wyrzuci z tego nazwę pliku wykonywalnego i zwróci wynik `/home/ccalvert/src/srcpas/PanelDesign/`. Dzięki `IncludeTrailingPathDelimiter` z pewnością otrzymamy wynik `/home/ccalvert/src/srcpas/PanelDesign/`, a nie `/home/ccalvert/src/srcpas/PanelDesign` lub `/home/ccalvert/`

src/srcpas/PanelDesigisn\. Jeśli użyjesz tej funkcji w programie pod Windows, zapewni ona, że do łańcucha ścieżki dołączony będzie lewy (ang. *backslash*), a nie zwykły ukośnik (ang. *slash*).

C/C++

Zarówno w C, jak i w Javie do wykonania tego samego zadania, które spełnia `ParamStr(0)`, służy pierwsza wartość parametru `Args` podanego w punkcie wejścia.

Po skomponowaniu poprawnej ścieżki do pliku z bitmapą, kod wywołuje wbudowaną funkcję `FileExists`, by upewnić się, że plik *step03.bmp* rzeczywiście znajduje się w katalogu, w którym program jest wykonywany. Upewniwszy się co do tego, wywołuje `Image1.Picture.LoadFromFile(PicName);`, aby wczytać plik do pamięci i wyświetlić go użytkownikowi.

Wyświetlanie wielu linii tekstu

Jak zapewne pamiętasz, kod przedstawiony nieco wcześniej w tym rozdziale przedstawiał jak przetworzyć plik *instructions.txt* i umieścić go w obiekcie `TStringList` nazwanym `FList`. Następujący kod pobiera zawartość jednego z elementów `TStringList` i wyświetla w kontrolce `Memo`:

```
if ListBox1.ItemIndex < FList.Count then
  Memo1.Text := FList.Strings[ListBox1.ItemIndex];
```

Kontrolka `Memo` automatycznie zawija wyrazy i wstawia suwaki (ang. *scrollbars*), jeśli są one potrzebne. Właściwość `Strings` obiektu `TStringList` pobiera z listy indeks tego łańcucha, który chcesz wyświetlić i zwraca skojarzony łańcuch znaków. Jeśli np. lista łańcuchów zawiera `"Ojciec"`, `"Syn"`, i `"Duch Święty"`, to `FList.Strings[0]` zwróci `"Ojciec"`, a `FList.Strings[2]` — `"Duch Święty"`.

Właściwość `Count` obiektu `TStringList` mówi ile elementów znajduje się na liście. Bądź ostrożny z używaniem tej zmiennej: jej numeracja zaczyna się od 1, podczas gdy właściwość `Strings` obiektu `TStringList` od 0. Jeśli np. w `TStringList` znajduje się jeden element, `Count` będzie równe 1, ale do elementu dostać się będzie można poprzez `MyStringList.Strings[0]`.

Jedyną częścią programu `PanelDesign`, której nie omówiłem, jest tkwiący u podstaw kod CLX. Nie ma w Kyliksie „czarnych skrzynek”. Jeśli chcesz zobaczyć, jak jest zaimplementowany obiekt `TStringList`, uruchom system Kylix, umieść w edytorze kursor na słowie `TStringList`, kliknij prawym przyciskiem myszy i wybierz *FindDeclaration*. Zostaniesz przeniesiony do miejsca deklaracji tego obiektu. Następnie możesz wcisnąć *Ctrl+Shift+Strzałka w dół*, żeby przenieść się do implementacji. Ale tu wybiegam już poza nakreślone ramy wykładu. Zagadnienie poruszania się w edytorze jest samo w sobie na tyle istotne, że poświęcę mu więcej miejsca w rozdziale 5.

Doszliśmy do końca dyskusji na temat programu `PanelDesign`. Mam nadzieję, że to, iż wspomnieliśmy o nim w tym miejscu, pomogło Ci zgłębić tajemnicę tego, jak skonstruować w Kyliksie prosty program.