Perl. Zaawansowane programowanie

Autor: Sriram Srinivasan
Tłumaczenie: Adam Podstawczyński
ISBN: 83-7197-999-1
Tytuł oryginału: Advanced Perl Programming
Format: B5, stron: 446

Umiesz programować w Perlu, lecz czujesz pewien niedosyt? Pracujesz nad większym projektem niż zazwyczaj i jesteś zagubiony? A może chciałbyś dodać do swojej aplikacji efektowny interfejs użytkownika, bardziej zaawansowany mechanizm przechwytywania błędów lub obsługę sieci i nie wiesz jak to zrobić?

Ta książka pomoże Ci stać się lepszym programistą bez względu na to, czy Twoja znajomość Perla jest powierzchowna, czy dogłębna. Nauczysz się zaawansowanych technik przygotowywania programów w Perlu o jakości produkcyjnej. Poznasz metody przetwarzania danych i używania obiektów, które wcześniej mogły wydawać Ci się czarną magią. Książka przedstawia szerokie zastosowania Perla: od sieci, baz danych, po interfejsy użytkownika. Znajdziesz w niej także opis wewnętrznych mechanizmów języka umożliwiających tworzenie wydajniejszych aplikacji oraz łączenie Perla z językiem C.

Do najważniejszych tematów poruszanych w książce należą:

- Praktyczne zastosowania pakietów i klas (programowanie obiektowe)
- Złożone struktury danych
- Trwałość danych (np. bazy danych)
- Sieci
- Interfejsy graficzne budowane za pomocą pakietu Tk
- Interakcja z funkcjami języka C
- Osadzanie i rozszerzanie interpretera Perla
- Wybrane aspekty wewnętrznych mechanizmów Perla

W książce przystępnie wytłumaczono wszystkie zagadnienia związane z Perlem, o których zapewne chciałbyś wiedzieć więcej: odwołania, przechwytywanie błędów operatorem eval, nieblokujące operacje wejścia/wyjścia, zasadność stosowania domknięć oraz dowiązania z użyciem mechanizmu tie. Jej lektura spowoduje, że poczujesz się prawdziwym hakerem – mistrzem Perla.

„Nieprzeciętny tekst i najbardziej zaawansowana książka o Perlu, jaką napisano. Autor – specjalista – objaśnia trudne koncepcje w sposób klarowny i kompletny.”
Jon Orwant, redaktor The Perl Journal

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

Spis treści

14

Interfejsy użytkownika oparte na Tk

W tym rozdziale nauczymy się budować graficzne interfejsy użytkownika (*Graphical User Interface — GUI*) za pomocą jednego z najbardziej funkcjonalnych i najlepiej prezentujących się pakietów narzędzi (*toolkit*): pakietu Tk [1]. Zaczniemy od zwięzłego przeglądu większości widgetów Tk oraz niektórych rozszerzeń. Dowiemy się także czegoś o *zarządzaniu geometrią* (jak układać widgety na formularzu). Następnie pokrótce przeanalizujemy obsługę liczników czasu w Perlu — będziemy z nich intensywnie korzystać w rozdziale 15., *Przykład graficznego interfejsu użytkownika: Tetris*. Później omówimy dowiązania zdarzeń, za pomocą których będziemy odwzorowywali dowolne kombinacje zdarzeń związanych z myszą i klawiaturą na wywołania funkcji. Wreszcie powiemy o problemach związanych z pętlą obsługi zdarzeń, podobnych do tych, z jakimi mieliśmy do czynienia w rozdziale 12., *Rozwiązania sieciowe oparte na gniazdach*.

W celu uproszczenia opisu w tym rozdziale przedstawimy małe i odizolowane porcje kodu, ilustrujące sposób użycia poszczególnych widgetów oraz funkcji Tk. Dopiero w następnych dwóch rozdziałach połączymy te narzędzia i rozwiążemy za ich pomocą praktyczne problemy.

Skoro jesteśmy już przy temacie interfejsów użytkownika, warto aby Czytelnik (oraz użytkownicy korzystający z jego lub jej programów) zapoznali się ze świetną, prezentującą zupełnie nowy punkt widzenia książką Alana Coopera[1] *About Face: The Essentials of User Interface Design* [3].

[1] Zwanego „ojcem języka Visual Basic".

Wprowadzenie do graficznych interfejsów użytkownika, Tk oraz modułu Perl/Tk

Na poziomie najbardziej podstawowym wszystkie platformy okienkowe (Apple Macintosh, X Window oraz Microsoft Windows) są bardzo proste. Udostępniany przez nie niskopoziomowy interfejs programowania służy do tworzenia i zarządzania oknami, zgłaszania interesujących zdarzeń (np. związanych z myszą i klawiaturą) oraz do rysowania elementów graficznych, takich jak: linie, okręgi i mapy bitowe. Problem polega na tym, że narysowanie nawet prostego formularza wymaga napisania znacznych ilości kodu i odczytania tysięcy (dosłownie) stron dokumentacji.

Często wykorzystywane fragmenty kodu interfejsów graficznych ewoluowały do postaci *widgetów* (w środowisku Microsoft Windows określanych mianem „kontrolek"); przykłady widgetów to: przyciski, paski przewijania i listy. Budowanie interfejsu graficznego sprowadza się teraz do uruchomienia interaktywnego „projektanta formularzy" i przeciągnięcia w odpowiednie miejsce takich gotowych do użycia elementów. Programowanie obiektowe jest teraz o wiele łatwiejsze!

Okazuje się, że widgety i języki skryptowe doskonale ze sobą współpracują. Widgety mają proste interfejsy, zaś interfejsy graficzne oparte na formularzach nie muszą być maksymalnie wydajne. Cechy te sprawiają, że skrypty świetnie spisują się przy tworzeniu interfejsów GUI. Jeśli dodamy do tego fakt, że interfejsy graficzne muszą umożliwiać elastyczne konfigurowanie (ponieważ to właśnie z tą częścią aplikacji ma do czynienia użytkownik — a w ogóle interfejs graficzny to najczęściej po prostu *jest* aplikacja), łatwo zrozumieć, skąd wzięła się ogromna popularność takich narzędzi, jak: Visual Basic, PowerBuilder czy Hypercard.

W systemach uniksowych rolę interfejsu okienkowego pełni X Window. Na jego bazie zbudowano różne pakiety narzędzi: Athena, InterViews, Motif i Tk. Jeśli chodzi o profesjonalny wygląd, łatwość użytkowania i dostępną dokumentację, Tk nie ma sobie równych. No i na dodatek jest darmowy!

W przeciwieństwie do innych pakietów narzędzi, Tk został opracowany specjalnie pod kątem języka skryptowego Tcl.[2] Można wręcz zastanawiać się, czy Tk nie jest główną przyczyną popularności Tcl. Wiele osób nie przepada za językiem skryptowym Tcl, ale uwielbia Tk — i stara się dostosowywać Tk do swojego ulubionego języka skryptowego, czy to Scheme, Pythona, Guile'a, czy oczywiście Perla. Pierwszą próbę połączenia Perla z Tk podjął Malcolm Beattie, który wewnętrznie wykorzystał interpreter Tcl do uzyskania dostępu do biblioteki Tk.

Nick Ing-Simmons podjął się zadania ambitniejszego: oczyścił Tk z osadzonego kodu Tcl i nadał mu ogólną, przenośną warstwę, ułatwiającą dodawanie Tk do innych języków skryptowych; rozwiązanie to określa się nazwą pTk (*portable Tk* — przenośny Tk). Następnie

[2] Zarówno Tcl, jak i Tk zostały opracowane przez dr. Johna Ousterhouta, wówczas pracownika Uniwersytetu Kalifornijskiego w Berkeley, a obecnie firmy Sun Microsystems. Patrz: *http://www.sunlabs.com/research/Tcl*.

do tego mechanizmu dodał „opakowanie" dla Perla5 („opakowania" dla innych języków mają zostać dołączone w przyszłości). To połączenie pTk i modułu odpakowującego Perla *Tk.pm* określa się wspólną nazwą Perl/Tk i to właśnie ono jest przedmiotem niniejszego rozdziału.

W tym samym czasie dr Ousterhout wraz ze swoim zespołem w firmie Sun przeniósł Tcl i Tk na platformy Windows i Mac; wkrótce to samo zrobiono z połączeniem Perl/Tk. Inne połączenia umożliwiające realizację interfejsu graficznego to oczywiście Tcl/Tk oraz Python/Tk (ten ostatni nie jest oparty na pTk). Microsoft przenosi swoje rozwiązania ActiveX (dawniej OLE) i VBA (Visual Basic for Applications) do środowiska uniksowego, mogą więc one wkrótce stanowić liczącą się konkurencję dla Tk. Samemu pakietowi narzędzi VB daleko jest do funkcjonalności Perla w połączeniu z Tk, za to środowisko programistyczne i obsługa ze strony firm niezależnych są dużo lepsze. Żyjemy w ciekawych czasach!

Do Tk dodano szereg nowych, profesjonalnie prezentujących się widgetów — udostępniono je w postaci biblioteki rozszerzeń Tix. Autorem biblioteki jest Ioi Kim Lam. Rozszerzenia obejmują dymki (do wyświetlania tekstu pomocy), notatniki i siatki, takie jak te stosowane w arkuszach kalkulacyjnych. Dystrybucja Perl/Tk obejmuje również mechanizmy dostępu z Perla do tych nowych widgetów.

Pierwszy program w Perl/Tk

Wszystkie interfejsy użytkownika budowane z użyciem Perl/Tk budowane są według takiej samej, ogólnej struktury:

1. Tworzymy okno główne *main* (określane także oknem najwyższego poziomu — *top-level*).
2. Tworzymy egzemplarze jednego lub większej liczby widgetów. Konfigurujemy je i umieszczamy wewnątrz okna głównego. Widget jest po prostu zbiorem danych i metod, które w efekcie dają pewien element graficzny interfejsu — np. przycisk lub listę — i powodują, że zachowuje się on w określony sposób po kliknięciu lub wykonaniu na nim innej czynności.
3. Uruchamiamy pętlę obsługi zdarzeń. Od tej pory o działaniu programu decydują czynności wykonywane przez użytkownika (zdarzenia).

W listingu 14.1 przedstawiono wszystkie te kroki w kolejności takiej, jak opisana powyżej. W wyniku wykonania kodu wyświetlany jest prosty interfejs graficzny[3], taki jak na rysunku 14.1.

Listing 14.1. Prosty kod realizujący interfejs graficzny

```
use Tk;                                          # Dołączamy moduł
# ------------------------------------------------------
# Tworzymy okno główne (main)
```

[3] Nie ma w nim nic interaktywnego — graficznie też jest raczej ubogi.

```
# ---------------------------------------------------------
$glowne = MainWindow->new();
$glowne->title ("Proste");
# ---------------------------------------------------------
# Tworzymy egzemplarze widgetów i konfigurujemy je
# ---------------------------------------------------------
$l = $glowne->Label(text   => 'witaj',          # etykieta
                anchor => 'n',                  # tekst zakotwiczamy
                                                # "na północy" (north)
                relief => 'groove',             # styl ramki
                width  =>  10, height => 3);    # szerokie na 10 znaków,
                                                # wysokie na 3


$l->pack();      # Domyślne miejsce w oknie głównym
# ---------------------------------------------------------
# Uruchamiamy nieskończoną pętlę służącą do oddelegowywania zdarzeń.
# ---------------------------------------------------------
MainLoop();
```

Rysunek 14.1. Nasz pierwszy ekran Perl/Tk

Ten przykład ilustruje wiele ważnych koncepcji zastosowanych w Tk (a także tych związanych z interfejsami graficznymi w ogóle).

Okno główne jest elementem najbardziej zewnętrznym; otaczają je tylko uchwyty zmiany wymiarów okna, menu systemowe oraz przyciski minimalizacji, maksymalizacji i zamknięcia okna (określane również mianem *dekoracji*). Aplikacja może wykorzystywać dowolną liczbę okien głównych.

Od okna głównego wymagamy następnie wyświetlenia widgetu etykiety o określonych właściwościach. Jeśli właściwości widgetu chcemy zmienić, wywołujemy na nim metodę `configure`:

```
$l->configure (text => 'foobar', foreground => 'red');
```

Niektóre widgety, np. Frame czy Notebook, same mogą zawierać inne widgety, a więc w hierarchii widgetów dozwolone jest dowolne zagnieżdżanie. Na szczycie hierarchii jest zawsze okno główne.

Metoda `pack` jest następnie wywoływana w celu przeprowadzenia *zarządzania geometrią*, tj. przydzielenia pozycji, szerokości i wysokości widgetu. Ta funkcja jest po prostu oddelegowana do *pojemnika* widgetu, tj. głównego okna, który oblicza jakie „działki ekranu" przydzielić poszczególnym widgetom podrzędnym. To tak jakby napisać „`skarpetki->pakuj`" i pozwolić walizce samej określić, gdzie skarpetki mają zostać umieszczone i ile można ich tam upchać.

„Pakowanie" to tylko jeden ze schematów zarządzania geometrią, służących do lokowania widgetów. Tk udostępnia menedżer geometrii *grid* oraz mechanizm *placer*; oba te elementy omówimy w punkcie „Zarządzanie geometrią" w dalszej części rozdziału.

Możliwe jest utworzenie i upakowanie widgetu za jednym zamachem:

```
$l = $glowne->Label (text => 'Ojej')->pack();
```

W większości przypadków nie ma nawet potrzeby przechwycenia wartości zwracanej z tego wyrażenia, chyba że planujemy na takim widgecie wykonywać potem metody. Typowe podejście polega na określeniu wszystkich parametrów w chwili tworzenia widgetów, a potem wywołaniu funkcji `MainLoop`. Tak będziemy postępować bardzo często w przykładach zamieszczonych w niniejszej książce.

Koncepcję pętli zdarzeń omówiliśmy już w rozdziale 12., ale powiemy więcej na ten temat w punkcie „Pętle zdarzeń" w dalszej części rozdziału. Tymczasem wystarczy widzieć, że `MainLoop` zajmuje się „oddelegowywaniem zdarzeń" i kończy działanie dopiero wtedy, gdy użytkownik zamknie okno przez dwukrotne kliknięcie menu systemowego. Funkcja ta musi zostać wywołana — inaczej utworzony formularz nigdy nie pojawi się na ekranie (dla widgetów trzeba zaś, z tego samego powodu, wywoływać `pack`).

I to tyle! Teraz wystarczy wiedzieć, jakie widgety mamy do dyspozycji, jakie właściwości one obsługują i jak je łączyć. Spójrzmy więc.

Formularze interfejsów graficznych: sposób łatwiejszy

Po co pisać kod tworzący statyczne ekrany, kiedy ekrany takie można po prostu narysować? Stephen Uhler z zespołu Tcl/Tk w Sun Microsystems napisał graficzny kreator pracujący w trybie WYSIWIG o nazwie SpecTcl (wym. jak angielskie słowo *spectactle*), obsługujący różne języki. Narzędzie dostosowano do interfejsów Perl/Tk, Java/Tk i Python/ Tk — odpowiednie odmiany noszą nazwy SpecPerl, SpecJava i SpecPython. Przeniesienie na interfejs Perl/Tk zrealizował Mark Kvale i program można pobrać z jego strony prywatnej:[4] *http://www.keck.ucsf.edu/~kvale/specPerl/*.

Za pomocą oprogramowania SpecPerl można rozmieszczać widgety w sposób wizualny, ustawiać ich właściwości w odpowiednich formularzach i wybierać kolory oraz czcionki z dostępnych palet — wszystko to jest bardzo wygodne.

[4] Sun rozpoczął sprzedawanie SpecTcl, a więc SpecPerl jest oparty na starszym (i darmowym) kodzie SpecTcl.

Jednak w tym rozdziale (i następnych dwóch) kod interfejsu graficznego będziemy pisali ręcznie, a nie za pomocą opisanego wyżej oprogramowania. Jest ku temu kilka powodów. Po pierwsze, nie będziemy budowali bardzo wyszukanych formularzy. Po drugie, w większości przypadków koncentrujemy się na dynamicznych aspektach Tk, a kreator graficzny umożliwia tylko budowanie statycznych formularzy. Po trzecie, kiedy już Czytelnik zrozumie niniejszy rozdział, będzie także rozumiał kod generowany przez SpecPerl.

Przegląd widgetów

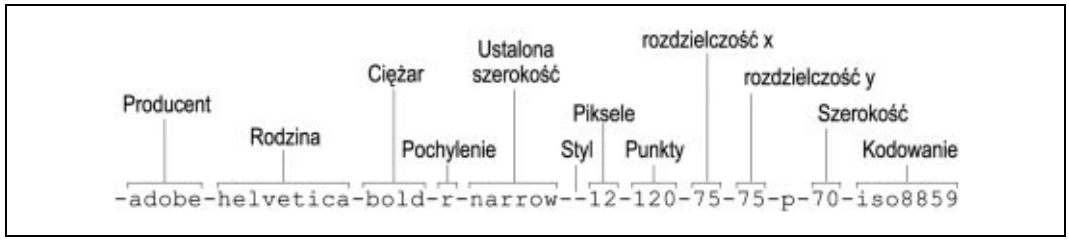
W tej części opiszemy klasy najciekawszych widgetów zaimplementowanych w Tk i Tix oraz powiemy o ich najczęściej stosowanych opcjach konfiguracyjnych i metodach. W celu zachowania porządku i szybszego wyszukiwania w przyszłości (gdy Czytelnik będzie już wiedział, czego szukać) ten szeroki zestaw właściwości i metod opisano w oddzielnym dodatku A, *Spis widgetów Tk*. Należy pamiętać, że ten rozdział — choć obszerny — traktuje tylko o podzbiorze (choć znacznym) wszystkich widgetów Tk. Nie są opisane wszystkie widgety dostępne w Tk i Tix. Do oprogramowania Perl/Tk dołączana jest oryginalna, przejrzyście napisana i kompleksowa dokumentacja interfejsu Tk.

Właściwości widgetów

Aby oswoić się z właściwościami, które można skonfigurować w przypadku wszystkich widgetów, warto zerknąć na tabelę A.1. Większość tych właściwości ma postać zwykłych łańcuchów tekstowych lub liczb; zanim przejdziemy do opisywania właściwych widgetów, warto jednak bliżej przyjrzeć się trzem właściwościom, związanym z czcionkami, grafiką i kolorami.

Czcionki

Czcionki określa się w formacie *XLFD* (*X Logical Font Description*), który składa się z 14 pól rozdzielonych łącznikami, tak jak to przedstawiono na rysunku 14.2.



Rysunek 14.2. Pola definicji czcionki

Na szczęście nie musimy pamiętać wszystkich tych pól. Każde z nich można zastąpić znakiem wieloznacznym „*” lub „?”; trzeba jednak pamiętać, aby łączników było tyle, ile trzeba. W systemie X Window dostępne są dwa narzędzia — graficzne *fontsel* i wsadowe *xlsfonts* — wyświetlające wszystkie dostępne w systemie kombinacje tych pól; za pomocą tych narzędzi można po prostu wybrać wymaganą czcionkę. Przede wszystkim należy

określić producenta, rodzinę, ciężar, pochylenie i liczbę punktów; pozostałe pola można zignorować (aby uzyskać polskie „ogonki" trzeba również wybrać odpowiednie kodowanie, `iso8859-2` — *przyp. tłum.*). Punkty określa się jako liczbę dziesiątych części punktu, a więc 120 oznacza czcionkę 12-punktową. Pochylenie można określić jako „i" (czcionka pochyła) lub „r" (zwykła). Czcionkę widgetu ustawiamy za pomocą odpowiedniej właściwości:

```
$etykieta->configure (
   font => '-adobe-helvetica-medium-r-normal--8-80-75-75-p-46-*-1');
```

W systemach Windows i Mac wartości fontów można określać albo w formacie XLFD, albo w prostszy, „windowsowy" sposób: `Helvetica 24 bold`. Pierwszy format będzie cały czas obsługiwany na wszystkich platformach.

Grafika

Niektóre widgety, np. przyciski lub etykiety, mogą zawierać dwukolorowe *mapy bitowe* lub wielokolorowe *mapy pikseli*. Ponieważ ta sama mapa bitowa lub obraz mogą służyć do dekorowania wielu widgetów, Tk traktuje je jako obiekty, które można wyświetlić w wielu miejscach. To znaczy, obiekt obrazu zawiera dane, a widgety wiedzą, jak wyświetlać te dane we własnej przestrzeni. A więc wyświetlenie mapy bitowej lub mapy pikseli wymaga wykonania dwóch kroków: utworzenia obiektu obrazu przez podanie nazwy pliku obrazu oraz skonfigurowania właściwości „bitmap" lub „pixmap" widgetu przez podanie nazwy tego obiektu.

W zależności od typu posiadanego pliku obrazu, utworzenie obiektu wymaga zastosowania odpowiedniej funkcji:

```
# Tylko format XBM (X Bitmap)
$obraz = $etykieta->Bitmap(file => 'twarz.xbm');


# Tylko format XPM (X Pixmap)
$obraz = $etykieta->Pixmap(file => 'buzia.xpm');


# Konstruktor Photo wymagany jest przy formatach GIF lub PPM (Portable pixmap)
$obraz = $etykieta->Photo(file => 'grymas.gif');
```

Teraz zmiana podkładu graficznego etykiety jest już prosta:

```
$etykieta->configure (image => $obraz);
```

Jeśli plik graficzny jest mapą bitową, używamy opcji „bitmap", a jeśli plikiem XPM lub GIF — właściwości „image". W przypadku map bitowych o tym, jakie kolory zostaną zastosowane, decydują opcje `foreground` i `background`; w przypadku innych obrazów kolory opisane są w samym pliku graficznym.

Kolory

Kolory można określać za pomocą nazw symbolicznych, np. „red" lub „yellow". W katalogu bibliotek systemu X znajduje się plik o nazwie *rgb.txt*, w którym wymienione są wszystkie dostępne nazwy symboliczne. Kolory można również podawać za pomocą

wartości RGB w postaci #RGB, #RRGGBB, #RRRGGGBBB lub #RRRRGGGGBBBB, gdzie R, G i B oznaczają jedną cyfrę szesnastkową, określającą odpowiednio natężenie koloru czerwonego, zielonego i niebieskiego.

To tyle krótkiego omówienia. Spójrzmy teraz na same widgety Tk i Tix.

Etykiety i przyciski

W tabeli A.1 zawarto większość informacji koniecznych do korzystania ze standardowych właściwości etykiet. Już po samych nazwach tych właściwości łatwo rozpoznać, do czego służą, więc nie będziemy im tutaj poświęcać więcej miejsca.

Przyciski to po prostu etykiety z jedną dodatkową właściwością: opcją `command`, która umożliwia przypisanie funkcji zdarzeniu polegającemu na kliknięciu przycisku. W poniższym przykładzie pokazano procedurę `zmien_napis`, która zmienia etykietę widgetu:

```
use Tk;
$glowne = MainWindow->new();
$przycisk = $glowne->Button(text    => 'Start',
                      command => \&zmien_napis);
$przycisk->pack();
MainLoop();
sub zmien_napis {                       # Tworzymy podprocedurę
    $przycisk->cget('text') eq "Start"         ?
        $przycisk->configure(text => 'Stop') :
        $przycisk->configure(text => 'Start');
}
```

Metoda `cget` pobiera wartość właściwości dającej się konfigurować.

Zamiast funkcji można użyć domknięcia, tak jak to pokazano poniżej (pominięto resztę szablonowego kodu):

```
$przycisk = $glowne->Button(
              text    => 'Start',
              command => sub {
                            $przycisk->cget('text') eq "Start"    ?
                            $przycisk->configure(text => 'Stop') :
                            $przycisk->configure(text => 'Start')
                         }
);
```

Trzeci sposób wykorzystania właściwości `command` polega na przekazaniu jej anonimowej tablicy, której pierwszym elementem będzie procedura; inne elementy tablicy przekazywane są do tej procedury w trakcie jej wykonywania:

```
$przycisk->configure (command => [\&zmien_napis, "nowy napis"]);
```

Takie podejście zastosujemy w dalszej części rozdziału do realizacji przewijania definiowanego z poziomu aplikacji.

Przełączniki i pola wyboru

Przełącznik (*radio button*) jest widgetem obejmującym łańcuch tekstowy, mapę bitową lub obraz oraz element graficzny w kształcie rombu, nazywany „guzikiem" (*indicator*) — patrz rysunek 14.3. Podobnie jak przyciski, przełączniki obsługują opcję „command". Jednak w przeciwieństwie do przycisków przełączniki zazwyczaj stosowane są grupami — użytkownik może wybrać jedną spośród kilku możliwości. Dlatego w przełącznikach dostępne są dwie właściwości — `variable` (zmienna) i `value` (wartość) — służące do synchronizacji z innymi przełącznikami grupy; chodzi o to, żeby zawsze mógł być włączony tylko jeden guzik. Kiedy klikniemy dany przełącznik, guzik jest włączany, a odpowiednia wartość zmiennej ustawiana jest na jego własną wartość. Jeśli wartość tej zmiennej zostanie zmodyfikowana, przełącznik sprawdza, czy jest ona identyczna z jego właściwością `value`; jeśli jest — widget włącza własny guzik. Jak łatwo się domyślić, do wewnętrznego monitorowania zmian wartości wykorzystano mechanizm tie.

Rysunek 14.3. Przykład przełącznika

W poniższym przykładzie tworzymy grupę przełączników. Rolę zmiennej synchronizującej odgrywa `$napoj`.

```
$napoj = "kawa";                        # Wartość początkowa
$kawa = $glowne->Radiobutton (
                                variable => \$napoj,
                                text     => 'Kawa',
                                value    => 'kawa');


$herbata    = $glowne->Radiobutton (
                                variable => \$napoj,
                                text     => 'Herbata',
                                value    => 'herbata');


$mleko   = $glowne->Radiobutton (
                                variable => \$napoj,
                                text     => 'Mleko',
                                value    => 'mleko');

# Układamy przełączniki na ekranie
$kawa->pack (side => 'left');
$herbata->pack    (side => 'left');
$mleko->pack   (side => 'left');
```

Ponieważ przełączniki mają różne wartości, ale korzystają z tej samej zmiennej, mamy zagwarantowane, że w danej chwili włączony jest tylko jeden guzik.

Więcej właściwości i metod związanych z przełącznikami można znaleźć w tabeli A.3.

Pole wyboru (*checkbox*) jest bardzo podobne do przełącznika. Kwadratowy guzik powiązany jest z odpowiednią wartością zmiennej. W przeciwieństwie do przełącznika, zmiana wartości nie musi powodować zmiany wartości innego pola wyboru (choć i takie zachowanie można w łatwy sposób osiągnąć). Pola wyboru stosuje się tam, gdzie użytkownik ma mieć możliwość wybrania wielu spośród różnych dostępnych opcji.

Pole graficzne

Widget pola graficznego (*canvas*) umożliwia wyświetlanie ustrukturyzowanych elementów graficznych. Udostępnia metody do przetwarzania *elementów* graficznych, takich jak okręgi, prostokąty, łuki, odcinki, mapy bitowe, elementy składające się z wielu odcinków i tekst. Umożliwia nawet osadzanie innych widgetów i traktowanie ich jak zwykłe elementy pola graficznego.

W przeciwieństwie do pól graficznych w pakiecie Abstract Windowing Toolkit Javy (a także w zasadzie w każdym innym pakiecie narzędzi GUI jaki znam), pola graficzne Tk są same w sobie obiektami: obsługują właściwości podobnie jak inne widgety i zezwalają na stosowanie tych właściwości albo do poszczególnych elementów graficznych, albo do całych nazwanych grup tych elementów. Można im również przypisać funkcje (tj. tak jakby powiedzieć „jeśli wskaźnik myszy znajdzie się nad tym okręgiem, wywołaj procedurę `foo`").

Elementy pola graficznego tym różnią się od widgetów, że każdy widget uzyskuje od serwera X własne okno; nie jest tak w przypadku elementów pola graficznego. Ponadto w przeciwieństwie do widgetów elementy pola graficznego nie są objęte funkcjami zarządzania geometrią (nie można zmienić ich wielkości za pomocą pojemnika). Pozostaje dla mnie niejasne, dlaczego w Tk zdecydowano się nie ukrywać tej różnicy przed użytkownikiem. W pakiecie InterViews (biblioteka X Window oparta na C++, później dostępna pod nazwą „Fresco") na przykład widgety i ustrukturyzowane elementy graficzne dziedziczą po ogólnym obiekcie graficznym — *glifie*. Takie podejście wydaje się bardziej przejrzyste. Z drugiej strony wdzięczny jestem, że tak precyzyjna implementacja ustrukturyzowanych elementów graficznych jest dostępna za darmo, wraz ze świetną dokumentacją — a więc moja uwaga w szerszym kontekście ma naprawdę małe znaczenie.

Aby narysować odcinek w widgecie pola graficznego, wywołujemy metodę `Canvas::create`:

```
$glowne = MainWindow->new();
# najpierw tworzymy widget pola graficznego
$pole = $glowne->Canvas(width => 200, height => 100)->pack();

# w tym polu rysujemy odcinek

$id = $pole->create ('line',
                     10, 10, 100, 100,  # od x0,y0 do x1, y1
                     fill => 'red');    # kolor wypełnienia obiektu
```

Pierwszym parametrem polecenia `create` jest typ elementu pola graficznego; pozostałe parametry zależą od pierwszego. `create` zwraca identyfikator, którego można potem użyć do odwołania się do danego obiektu. Na przykład możemy uaktualnić współrzędne obiektu metodą `coords`:

```
$pole->coords ($id, 10, 100);
```

Wszystkie współrzędne w Tk są liczone od lewego górnego roku. Współrzędna `x` rośnie od lewej do prawej, a `y` — od góry w dół.

Można przesunąć obiekt względem jego bieżącej pozycji. Służy do tego metoda `move`:

```
$pole->move ($id, 15, 23); # 15 i 23 to przesunięcia x i y
```

Elementy pola graficznego można konfigurować metodą `itemconfigure`; w tabeli A.5 wymieniono właściwości i metody dla elementów każdego typu oraz widgetu pola graficznego jako całości.

Jedną z najwygodniejszych funkcji obsługiwanych przez pole tekstowe jest możliwość oznaczania jednego lub wielu elementów łańcuchem-identyfikatorem. Obiekt można oznaczać dowolną liczbą takich etykiet. Słowo `all` w identyfikatorze oznacza wszystkie obiekty pola graficznego. Można oznaczyć obiekt w trakcie tworzenia lub za pomocą metody `addtag`. Identyfikator `current` oznacza element, nad którym właśnie znajduje się wskaźnik myszy. Wszystkie metody pola graficznego, które przyjmują jako argument identyfikator elementu, mogą także zamiast niego obsłużyć łańcuch. Na przykład aby przenieść wszystkie obiekty z etykietą „grupa” o 10 pikseli w prawo, napiszemy tak:

```
$pole->move('grupa', 10, 0); # przesunięcie x = 10, przesunięcie y = 0
```

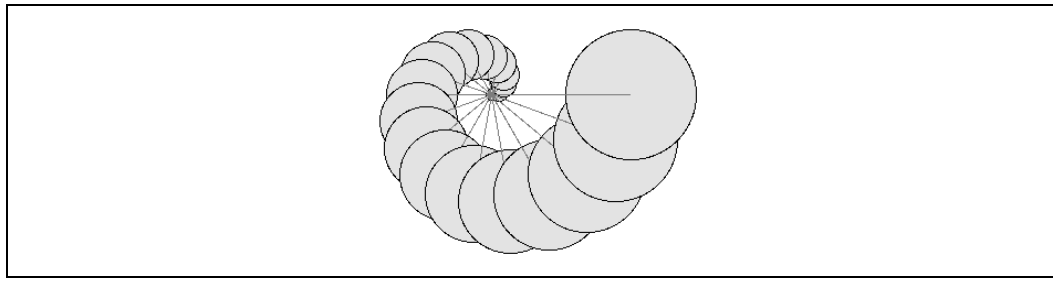
Z właściwości tej będziemy intensywnie korzystać w rozdziale 15.

W listingu 14.2 przedstawiono program rysujący zestaw okręgów, których środki umieszczone są wzdłuż spirali Archimedesa. Wzór na spiralę Archimedesa to $r = a\Theta$, gdzie r, tzn. promień (oznaczany na rysunku odcinkami), proporcjonalnie zależy od kąta Θ. Dodatkowy efekt wizualny osiągnięto przez uzależnienie również promieni okręgów od tego kąta.

Listing 14.2. Rysowanie spirali Archimedesa

```
use Tk;

$glowne = MainWindow->new();
$pole = $glowne->Canvas(width => 300, height => 245)->pack();
# Rysuje koła wzdłuż spirali Archimedesa
# Środki tych kół umieszczane są wzdłuż spirali
# (promień spirali = stała * theta)


$oryg_x = 110; $oryg_y = 70;                # miejsce początkowe
$PI = 3.1415926535;
$promien_okregu = 5;                        # promień pierwszego okręgu
$promien_spirali = 0;
```

```
for ($kat = 0; $kat <= 180;
     $promien_spirali += 7, $promien_okregu += 3, $kat += 10)
{

    # przesunięcie współrzędnych spirali: r.cos( [theta] ) i r.sin( [theta] )

    # sin() i cos() wolą kąty w radianach (stopnie* [pi] /90)
    $spir_x = $oryg_x + $promien_spirali * cos ($kat * $PI / 90);
    $spir_y = $oryg_y - $promien_spirali * sin ($kat * $PI / 90);
    # spir_x oraz spir_y są współrzędnymi środka nowego koła
    # Canvas::create wymaga podania górnego lewego i prawego dolnego rogu
    $pole->create ('oval',
             $spir_x - $promien_okregu,
             $spir_y - $promien_okregu,
             $spir_x + $promien_okregu,
             $spir_y + $promien_okregu,
             -fill => 'yellow');
    $pole->create ('line',
             $oryg_x, $oryg_y,
             $spir_x, $spir_y,
             fill => 'slategray');


}

MainLoop();
```



Rysunek 14.4. Kompozycja z ustrukturyzowanych elementów narysowana w polu graficznym

Tekst i wprowadzanie danych

Widget tekstowy (*text*) wyświetla jeden lub wiele wierszy tekstu oraz umożliwia jego edycję (domyślne skróty klawiszowe są takie jak w Emacsie — męczarnia dla tych, którzy wciąż korzystają z *vi*...). Widget ten jest na tyle funkcjonalny, że może służyć do wyświetlania stron WWW podobnie jak w przeglądarce; zresztą realizowano już takie projekty — dystrybucja Perl/Tk zawiera implementację przeglądarki internetowej *tkweb*, a Guido van Rossum, autor Pythona, napisał za pomocą tego języka i Tk przeglądarkę internetową *Grail*, potrafiącą wykonywać aplety Pythona.

W tym punkcie przyjrzymy się pokrótce możliwościom widgetu tekstowego, zaś szczegółowo omówimy go — i na jego bazie zbudujemy aplikację — w rozdziale 16., *Przykład graficznego interfejsu użytkownika: Przeglądarka podręczników man*.

Wstawianie tekstu na pozycjach bezwzględnych

Kiedy chcemy programowo wstawić tekst w pewnym określonym miejscu lub w pewnym zakresie miejsc, musimy określić przynajmniej jeden indeks. Indeks to łańcuch, np. „2.5", oznaczający wiersz 2., kolumnę 6. (numery wierszy liczone są od 1, a kolumn od 0). Poniższy kod tworzy widget tekstowy i wstawia łańcuch na wybranej pozycji:

```
$t = $glowne->Text(width => 80, height => 10)->pack();
$t->insert('2.5', 'Próbka');
```

Wstawianie tekstu na pozycjach logicznych

Widget tekstowy obsługuje koncepcję *znacznika* (*mark*) — nazwy jednej pozycji w widgecie tekstowym, nadanej przez użytkownika. Pozycja ta liczona jest w odstępach między dwoma znakami, a nie definiowana jako para wiersz-kolumna. Ułatwia to wstawianie znaku na pozycji znacznika. Zaznaczona w ten sposób pozycja jest obiektem logicznym; nie zmienia się po wprowadzeniu lub usunięciu tekstu. Widget obsługuje wbudowane znaczniki, np. `insert` (miejsce kursora tekstowego), `current` (znak najbliższy wskaźnikowi myszy), `wordend` (koniec słowa, nad którym znajduje się wskaźnik myszy), `end` (koniec wiersza zawierającego kursor tekstowy) itd. Znaczniki można wstawiać zamiast opisanych wcześniej numerów wierszy i kolumn:

```
$t->insert("end", "Próbka");              # Insert text at end
```

Szczegółową listę określników indeksów można znaleźć w tekście znajdującym się przed tabelą A.6. W przykładach zamieszczonych w poniższych punktach tworzymy widget tekstowy i wstawiamy łańcuchy w różnych miejscach, stosując różne typy identyfikatorów.

Wstawianie z indeksowaniem względnym

Indeksy mogą być także definiowane względem indeksu podstawowego. Na przykład:

```
$t->insert('insert +5,
          'Próbka'); # 5 znaków za pozycją kursora tekstowego
$t->insert('insert linestart', 'Próbka'); # przejdź do pozycji kursora
                                          # tekstowego a następnie do początku
                                          # wiersza
```

Zmiana właściwości zakresów tekstu za pomocą znaczników

Opisywany widget obsługuje koncepcję *znaczników* lub znacznikowanych stylów — łańcuchów definiowanych przez użytkownika i odpowiadających listom właściwości tekstu (czcionka, kolor, wzór punktowy itp.). Spójrzmy:

```
$tekst->tagConfigure('foo',
                     foreground => 'yellow', background => 'red');
```

Teraz łańcuch „foo" można zastosować do jednego lub więcej zakresów tekstu w tym widgecie, np.:

```
$tekst->tagAdd('foo', '3.5', '3.7');
```

Spowoduje to podświetlenie fragmentu tekstu w wierszu 3., indeksy od 5. do 7. Indeksy określające zakres mogą być także bezwzględnymi lub względnymi pozycjami znacznika. Na przykład poniższy fragment kodu zmienia właściwości wiersza, w którym znajduje się kursor tekstowy:

```
$text->tagAdd('foo', 'insert linestart', 'insert lineend');
```

Można określać wiele znaczników opisujących zachodzące na siebie porcje tekstu i przeciwnie, jeden znacznik można zastosować dla wielu zakresów. Wszystkie widgety tekstowe obsługują specjalny znacznik `sel`, który odzwierciedla bieżący zakres zaznaczonego tekstu. Można wstawić nowy tekst i określić dla niego formatowanie — wystarczy dołączyć jako trzeci parametr `insert` nazwę znacznika:

```
$t->insert('Próbka', '3.5','foo');
```

Widget tekstowy umożliwia osadzanie dowolnego innego widgetu i traktowanie go jak jednego znaku — możemy więc umieścić w tekście przyciski i listy, które będą „poruszały się" wraz z całym tekstem przy wprowadzaniu kolejnych znaków.

Widget wprowadzania danych

Do wprowadzania pojedynczych wierszy tekstu w Perl/Tk służy widget Entery, który nie obsługuje znaczników ani osadzonych okien; jest to po prostu taka „lżejsza" wersja widgetu Text.

Perl/Tk (nie oryginalny Tk) udostępnia również widget TextUndo, stanowiący podklasę widgetu Text. Mechanizm ten pozwala na wycofanie nieograniczonej liczby zmian w tekście (ale nie można automatycznie wprowadzić ponownie tych zmian — nie można „wycofać cofania") oraz obsługuje metody pobierania i zapisywania tekstu z i do plików. Widget ten nie jest obecny w oryginalnej dystrybucji Tcl/Tk.

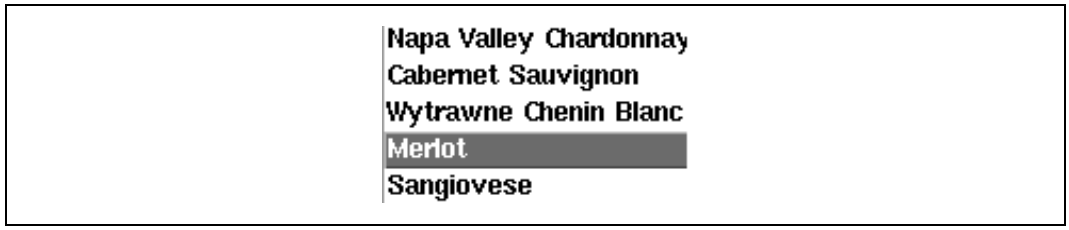
Widget tekstowy i dowiązania

Widget tekstowy obsługuje metody `TIEHANDLE` oraz `print`, co umożliwia wykorzystanie go jako modułu symulującego uchwyty plików. Oto jak można wykorzystać ten mechanizm do przekierowania operacji na uchwycie pliku do widgetu tekstowego:

```
use Tk;
my $mw = MainWindow->new;            # Tworzymy główne okno
my $t  = $mw->Scrolled('Text');      # Tworzymy przewijane okno tekstowe
$t->pack(-expand => 1,
         -fill => 'both');           # Konfigurujemy je
tie (*TEXT, 'Tk::Text',$t);          # dowiązujemy uchwyt pliku do widgetu
print TEXT "Hej tam\n";              # To zostanie wyświetlone w widgecie
```

Lista

Widget listy (*list*) służy do wyświetlania listy łańcuchów, wiersz po wierszu (rysunek 14.5). Wszystkie łańcuchy wyświetlane są z takimi samymi właściwościami. Jeśli chcielibyśmy użyć różnych czcionek i kolorów, możemy napisać prostą metodę „opakowującą" ten widget i tak zasymulować taką urozmaiconą listę.



Rysunek 14.5. Lista z nazwami win

Domyślnie zdarzenia w liście polegają na wybieraniu i anulowaniu wyboru elementów. Jeśli wymagamy dodatkowych funkcji — na przykład specjalnej obsługi dwukrotnego kliknięcia myszą — musimy sami dowiązać to zdarzenie do wybranej funkcji. Tryby wybierania elementów to: single, browse (domyślny), multiple oraz extended. W trybie single lub browse można w danej chwili wybrać tylko jeden element. W trybie browse można również przeciągnąć wybór, przytrzymując przycisk 1. W trybie multiple można wybrać dowolną liczbę elementów; wybieranie i anulowanie wyboru nie wpływa na fakt wybrania innych elementów. W trybie extended możliwe jest wybieranie wielu elementów przez klikanie i przeciąganie; jednak pojedyncze kliknięcie, przed wybraniem elementu bieżącego, powoduje anulowanie poprzedniego wyboru.

Podobnie jak w widgecie tekstowym, można tutaj w różny sposób określać pozycje na liście — nie tylko za pomocą indeksu. Na przykład można użyć słów `end` oraz `active` (tam, gdzie znajduje się kursor). Indeksy, właściwości i metody listy wymieniono w tabeli A.8. Poniżej, w listingu 14.3, przedstawiono kod, w wyniku którego powstała lista na rysunku 14.5.

Listing 14.3. Lista z przypisanymi funkcjami

```
use Tk;
$glowne = MainWindow->new();
$lista_win = $glowne->Listbox("width" => 20, "height" => 5
                          )->pack();
$lista_win->insert('end', # Wstawiamy na końcu taką listę
                   "Napa Valley Chardonnay", "Cabernet Sauvignon",
                   "Wytrawne Chenin Blanc", "Merlot", "Sangiovese" );
$lista_win->bind('<Double-1>', \&kup_wino);
sub kup_wino {
    my $wino = $lista_win->get('active');
    return if (!$wino);  # Zakończ, jeśli żaden element listy nie jest aktywny
    print "Ach, '$wino'. Cóż za wspaniały gust!\n";
    # Usuń wino z listy
    $lista_win->delete('active');
}
MainLoop();
```

Lista nie udostępnia takiej właściwości jak `command`, a więc aby stworzyć dowiązanie między dwukrotnym kliknięciem myszą a zdefiniowaną przez nas podprocedurą, musimy korzystać z bardziej ogólnej metody `bind`. Więcej informacji o tej technice przedstawimy w punkcie „Dowiązywanie zdarzeń”.

Ramki

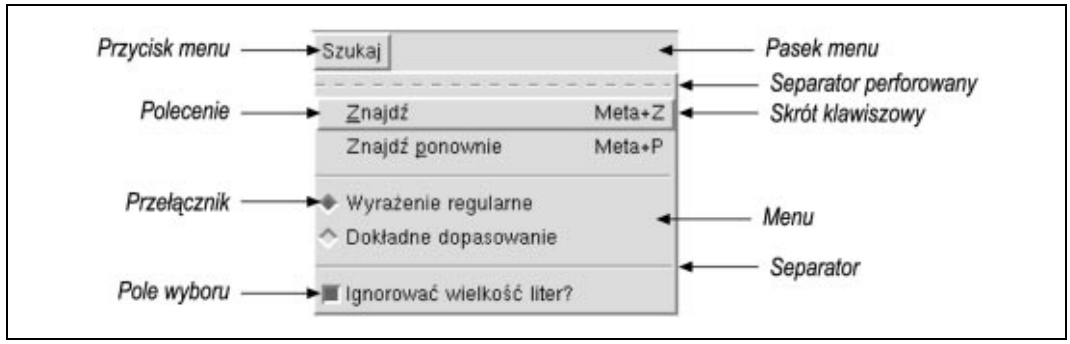
Widgety ramek (*Frame*) nie są zbyt interesujące; przydają się, gdy chcemy budować bardziej zaawansowane zestawy widgetów lub tworzymy widgety kompozytowe.

Kiedy konstruujemy złożony formularz GUI, najlepiej jest podzielić ekran na części, z których każda ma specyficzną funkcję; części te umieszczane są właśnie w ramkach. Ramki takie możemy potem „na wyższym poziomie" dowolnie układać. Więcej na ten temat powiemy w punkcie „Zarządzanie geometrią" w dalszej części rozdziału. Widgety ramek są pojemnikami, więc można zażądać od nich utworzenia innych, „podstawowych" widgetów, np. przycisków, pól tekstowych czy pasków przewijania.

Menu

Termin „menu" powszechnie stosuje się do określania obiektu pojawiającego się po kliknięciu myszy i zawierającego zbiór widgetów etykiet lub przycisków. Istnieją trzy typy menu: menu rozwijane (*pulldown menu*), menu opcji (*option menu*) oraz menu podręczne (*popup menu*).

Tk udostępnia widget MenuButton, którego kliknięcie może powodować wyświetlenie podręcznego widgetu Menu. Widget Menu jest po prostu pojemnikiem widgetów stanowiących elementy wyświetlonego menu, nie określa zaś samego układu tego elementu. Do rozróżnienia między koncepcją menu a widgetem Menu użyjemy różnych stylów czcionek. Elementy zaprezentowano na rysunku 14.6.



Rysunek 14.6. Menu rozwijane i przycisk menu

Aby skonstruować menu, musimy wykonać następujące kroki:

1. Utworzyć *pasek menu*, w którym zostanie umieszczony widget MenuButton. Pasek menu to po prostu widget Frame.
2. Utworzyć przynajmniej jeden widget MenuButton i umieścić go w pasku menu.
3. Zażądać od widgetów MenuButton utworzenia i zarządzania widgetami-wpisami menu.

Właściwości i interfejs programowania widgetów MenuButton i Menu opisano w tabelach odpowiednio A.9 i A.10. W listingu 14.4 pokazano sposób zaprogramowania menu widocznego na rysunku 14.6.[5]

Listing 14.4. Menu rozwijane „Szukaj”

```
use Tk;
$glowne = MainWindow->new();
# Widget Frame pełni rolę pojemnika przycisków menu
$pasek_menu = $glowne->Frame()->pack(side => 'top');



# Przycisk menu "szukaj"
$przycisk_szukaj = $pasek_menu->Menubutton(text         => 'Szukaj',
                                           relief       => 'raised',
                                           borderwidth  => 2,
                                           )->pack(side => 'left',
                                                   padx => 2
                                                   );



# Przycisk menu "znajdź"
$przycisk_szukaj->command(-label       => 'Znajdź',
                          accelerator => 'Meta+Z',
                          underline   => 0,
                          command     => sub {print "znajdź\n"}
                    );
# Przycisk menu "znajdź ponownie"
$przycisk_szukaj->command(-label       => 'Znajdź ponownie',
                         accelerator  => 'Meta+P',
                         underline    => 7,
                         command      => sub {print "znajdź ponownie\n"}
                    );



$przycisk_szukaj->separator();
$typ_dopasowania = 'regexp';    # Domyślny typ wyszukiwania: wg wyrażenia
                                # regularnego
$wielk_lit = 1;                 # Domyślnie ignorujemy wielkość liter
                                # (włączamy pole wyboru)
                                # Dopasowanie wg wyrażenia regularnego
$przycisk_szukaj->radiobutton(-label    => 'Wyrażenie regularne',
                              value    => 'regexp',
                              variable => \$typ_dopasowania);
                               # Dopasowanie dokładnie podanego łańcucha
$przycisk_szukaj->radiobutton(-label    => 'Dokładne dopasowanie',
                              value    => 'exact',
                              variable => \$typ_dopasowania);
$przycisk_szukaj->separator();
                               # Ignorujemy wielkość liter
$przycisk_szukaj->checkbutton(-label    => 'Ignorować wielkość liter?',
                              variable => \$wielk_lit);

MainLoop();
```

[5] W celu uzyskania poprawnie wyświetlanych polskich „ogonków” użyłem dla poszczególnych widgetów dodatkowej właściwości, `font => '-adobe-helvetica-medium-r-*-*-12-*-*-*-*-*-iso8859-2'` — *przyp. tłum.*

W przykładzie na widgecie MenuButton (`$przycisk_szukaj`) wywołujemy takie metody, jak: `command`, `separator`, `checkbutton` i `cascade`. Co ciekawe, są to akurat metody interfejsu widgetu Menu, a nie MenuButton (patrz tabele A.9 i A.10). Ze względu na wygodę programisty, polecenia te obsługuje widget Perl/Tk MenuButton — po prostu „po cichu" oddelegowuje je do powiązanego widgetu Menu.

Zazwyczaj wpisy menu układane są w tej kolejności, w jakiej zostały utworzone, ale można jawnie określić ich pozycję za pomocą metody `add`. Składnia indeksowania jest podobna do tej znanej z widgetu listy i została opisana w dodatku A. Za pomocą tej metody będziemy tworzyli dynamiczne menu w rozdziale 16.

Paski przewijania i samo przewijanie

Choć paski przewijania (*scrollbar*) są „pełnoprawnymi" widgetami, rzadko używane są same — zawsze sterują skojarzonymi z nimi innymi widgetami. Ze względu na to ścisłe powiązanie Perl/Tk udostępnia funkcję pomocniczą `Scrolled`, która dołącza paski przewijania do wybranego widgetu — nie musimy więc jawnie tworzyć, określać wielkości i „pakować" takich pasków. Poniżej pokazano, jak tworzyć przewijaną listę:

```
$przewijana_lista = $glowne->Scrolled('Listbox', opcje listy,
                                      scrollbars => 'se');
```

Wewnętrznie tworzony jest widget Frame, paski przewijania w poziomie i pionie (o ile są konieczne) oraz lista; następnie wszystko to jest „pakowane", a widget Frame (pojemnik) otrzymuje zwrócone odwołanie. Czyż nie pięknie? Perl/Tk upraszcza sprawę jeszcze bardziej: dla najbardziej typowych obiektów — przewijanych list i pól tekstowych — udostępnia metody pomocnicze `ScrlListBox` oraz `ScrlText`:

```
$przewijana_lista = $glowne->ScrlListBox(opcje listy);
```

Właściwie tyle trzeba wiedzieć o przewijaniu — Czytelnik może teraz przejść od razu do punktu „Skala" bez utraty ciągłości wywodu.

Niestandardowe przewijanie

Czasem jednak chcemy mieć pełniejszą kontrolę nad przewijaniem. Na przykład tworzymy trzy listy i chcemy zsynchronizować ich przewijanie. Musimy zrobić tak, aby pasek przewijania informował wszystkie trzy widgety o fakcie przesunięcia. Jest z tym pewien drobny problem: mechanizm ten powinien działać również w przeciwną stronę, tj. widgety także powinny informować, że zostały przewinięte z innych przyczyn. Na przykład klikamy jedną listę i przeciągamy wskaźnik myszy — lista zostanie przewinięta; trzeba więc zapewnić, aby pasek przewijania oraz dwie pozostałe listy zachowały się podobnie. Innymi słowy, elementem sterującym nie zawsze jest tutaj pasek przewijania — jest to raczej relacja obustronna.

Jak widać w tabeli A.11, nie istnieje żadna jawna właściwość dowiązująca pasek przewijania do widgetu, ale istnieje właściwość `command` umożliwiająca powiadomienie pewnej funkcji o fakcie przesunięcia suwaka. Ponadto wszystkie widgety umożliwiające przewijanie

(listy, pola tekstowe, ramki i pola graficzne) obsługują dwie metody `xview` i `yview` (tabela A.12), które informują przewijany widget, która część jego treści ma zostać wyświetlona w oknie. A więc żeby pasek przewijania powodował przewinięcie widgetu, skonfigurujemy właściwość `command` takiego paska w następujący sposób:

```
$pasek->configure (command => [N$widget]);
```

Pasek przewijania automatycznie wywołuje określoną metodę (`xview` lub `yview`) na widgecie. Skąd widget wie, o ile ma zostać przewinięty? Otóż, o czym jeszcze nie wspominaliśmy, pasek dołącza do wywołania `yview` argumenty — wewnętrznie komunikat wysłany z paska przewijania do widgetu wygląda więc mniej więcej tak:

```
$widget->yview('moveto', 30);
```

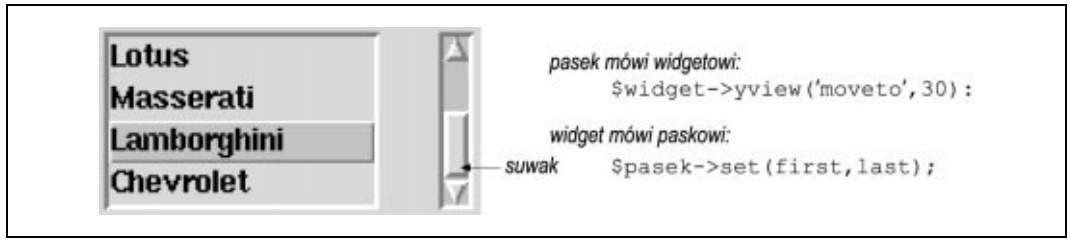
Takie polecenie powoduje, że widget tak przemieszcza swoją zawartość, że górny wiersz lub piksel ustawiony jest na 30 procentach wysokości.

A teraz spójrzmy, jak ten mechanizm działa w drugą stronę, gdy to widget informuje pasek przewijania.

Wszystkie przewijane widgety obsługują metody `xscrollcommand` oraz `yscrollcommand`, które powinniśmy skonfigurować tak, aby wywoływały metodę `set` paska przewijania:

```
$lista->configure ('yscrollcommand', [$pasek]);
```

Tę symbiotyczną relację przedstawiono na rysunku 14.7. Szczegółowe informacje o wymienionych wyżej poleceniach i właściwościach można znaleźć w tabelach A.11 i A.12.



Rysunek 14.7. Interakcja między paskiem przewijania i skojarzonym z nim widgetem (listą)

W listingu 14.5 pokazano, jak współpracują ze sobą poszczególne polecenia konfiguracyjne w połączeniu z jedną listą.

Listing 14.5. Konfigurowanie paska przewijania i listy, tak aby wzajemnie powodowały przewijanie

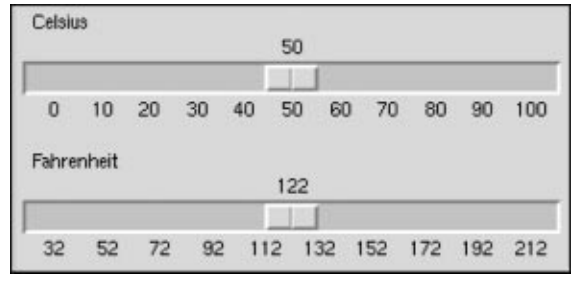
```
use Tk;
$glowne = MainWindow->new();
$lista_aut = $glowne->Listbox("width" => 15, "height" => 4,
                              )->pack(side => 'left',
                                      padx => 10);
```

```
$lista_aut->insert('end', # Dopisujemy następujące elementy na końcu listy
                   "Acura", "BMW", "Ferrari", "Lotus", "Maserati",
                   "Lamborghini", "Chevrolet"
                  );

# Rysujemy pasek przewijania i informujemy go o liście
$pasek = $glowne->Scrollbar(orient  => 'vertical',
                            width   => 10,
                            command => ['yview', $lista_aut]
                            )->pack(side => 'left',
                                    fill => 'y',
                                    padx => 10);

# Informujemy też listę o pasku
$lista_aut->configure(yscrollcommand => ['set', $pasek]);
MainLoop();
```

Skala

Widget skali (*scale*) przypomina termometr. Wzdłuż poziomego lub pionowego „korytka" wyświetlane są odpowiednie oznaczenia; wewnątrz korytka znajduje się suwak, którym można poruszać programowo lub ręcznie (za pomocą myszy lub klawiatury). Właściwości i metody skali wymieniono w tabeli A.13.

Na rysunku 14.8 przedstawiono dwie skale z podziałkami Celsjusza i Fahrenheita (bazowa jest skala Celsjusza od 0 do 100 stopni). Skale są skoordynowane — przesunięcie jednego suwaka wpływa na drugi suwak.



Rysunek 14.8. Skoordynowane skale z podziałkami Celsjusza i Fahrenheita

W listingu 14.6 pokazano, jak można zaimplementować taki program.

Listing 14.6. Konwersja stopni Celsjusza na Fahrenheita z użyciem widgetów skali

```
use Tk;
# Przelicznik stopni Celsjusza na Fahrenheita z wykorzystaniem skal
$glowne = MainWindow->new();


$wart_celsjusza = 50;
oblicz_fahrenheita();
```

```
#---------------------- Skala CELSJUSZA-----------------------------
$glowne->Scale(orient       => 'horizontal',
        from         => 0,                  # Od 0 stopni C
        to           => 100,                # do 100 stopni C
        tickinterval => 10,
        label        => 'Celsius',
        font         => '-adobe-helvetica-medium-r-normal'
        . '--10-100-75-75-p-56-iso8859-1',
        length       => 300,                # w pikselach
        variable     => \$wart_celsjusza,   # zmienna globalna
        command      => \&oblicz_fahrenheita  # zmieniamy Fahr.
       )->pack(side => 'top',
               fill => 'x');
#---------------------- Skala FAHRENHEITA---------------------------
$glowne->Scale(orient       => 'horizontal',
        from         => 32,                 # Od 32 stopni F
        to           => 212,                # do 212 stopni F
        tickinterval => 20,                 # co 20 stopni
        label        => 'Fahrenheit',
        font         => '-adobe-helvetica-medium-r-normal'
        . '--10-100-75-75-p-56-iso8859-1',
        length       => 300,                 # w pikselach
        variable     => \$wart_fahrenheita,  # zmienna globalna
        command      => \&oblicz_celsjusza   # zmieniamy Cels.
       )->pack(side => 'top',
               fill => 'x',
               pady => '5');


sub oblicz_celsjusza {
    # Suwak skali Celsjusza automatycznie się porusza po zmianie
    # zmiennej $wart_celsjusza
    $wart_celsjusza = ($wart_fahrenheita - 32)*5/9;
}


sub oblicz_fahrenheita {
    $wart_fahrenheita = ($wart_celsjusza * 9 / 5) + 32;
}

MainLoop();
```

W przykładzie tym, w wyniku przesunięcia suwaka skali Celsjusza, wywoływana jest funkcja `oblicz_fahrenheita()`. Funkcja ta zmienia wartość `$wart_fahrenheita`, skojarzoną ze skalą Fahrenheita. Jak widać, obsługa skali zazwyczaj wymaga tylko użycia właściwości `command` i `variable`. Nie trzeba jawnie wywoływać metody `set()`.

Lista hierarchiczna

Dane hierarchiczne, np. struktury systemu plików lub wykresy obrazujące organizację, można przedstawić za pomocą widgetu listy hierarchicznej HList. Każdy wpis jest wcinany o jeden poziom względem wpisu nadrzędnego. Opcjonalnie HList może rysować rozgałęzienia; z wpisami można również skojarzyć ikony oraz inne widgety. Wpis identyfikowany jest nie przez indeks (jak w liście), lecz przez „ścieżkę wpisu" przypominającą ścieżkę do pliku (rolę separatora pełni wybrany przez użytkownika znak). Niektóre interesujące właściwości i metody widgetu HList wymieniono w tabeli A.14.

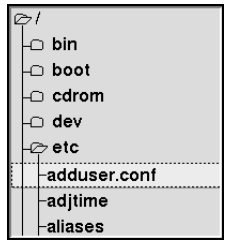
W listingu 14.7 budujemy opartą na widgecie HList przeglądarkę systemu plików. Dwukrotne kliknięcie katalogu powoduje rozwinięcie lub zwinięcie jego zawartości oraz odpowiednią zmianę ikony.

W listingu 14.7 przedstawiono jeden ze sposobów uzyskania wyniku zilustrowanego na rysunku 14.9. Warto zwrócić szczególną uwagę na kod otwierający i ustawiający mapy bitowe, a także na część zmieniającą kształt kursora po wykonaniu zadania.

Listing 14.7. Przeglądarka systemu plików oparta na widgecie HList

```
use Tk;
require Tk::HList;
$glowne = MainWindow->new();
$h = $glowne->Scrolled('HList',
                       '-drawbranch'     => 1,
                       '-separator'      => '/',
                       '-indent'         => 15,
                       '-command'        => \&pokaz_ukryj_katalog,
                       )->pack('-fill'   => 'both',
                               '-expand' => 'y');

$ikony{"otwarty"}   = $glowne->Bitmap(-file => './otwarty_folder.xbm');
$ikony{"zamkniety"} = $glowne->Bitmap(-file => './folder.xbm');

pokaz_ukryj_katalog("/");
MainLoop();

#-----------------------------------------------------------------------
sub pokaz_ukryj_katalog {       # Wywoływana po dwukrotnym kliknięciu wpisu
    my $sciezka = $_[0];
    return if (! -d $sciezka);  # To nie katalog
    if ($h->info('exists', $sciezka)) {
        # Przełączamy status katalogu.
        # Poznajemy, czy jest otwarty, po sprawdzeniu następnego wpisu:
        # otwarty jest wtedy, gdy jest to podłańcuch bieżącej ścieżki
        $nast_wpis = $h->info('next', $sciezka);
        if (!$nast_wpis || (index ($nast_wpis, "$sciezka/") == -1)) {
            # Nie jest otwarty. Otwieramy.
            $h->entryconfigure($sciezka, '-image' => $ikony{"otwarty"});
            dodaj_zaw_katalogu($sciezka);
        } else {
            # Jest otwarty. Zamykamy, zmieniamy ikonę i usuwamy podrzędny węzeł.
            $h->entryconfigure($sciezka, '-image' => $ikony{"zamkniety"});
            $h->delete('offsprings', $sciezka);
        }
    } else {
        die "'$sciezka' nie jest katalogiem\n" if (! -d $sciezka);
        $h->add($sciezka, '-itemtype' => 'imagetext',
                '-image' => $ikony{"otwarty"}, '-text' => $sciezka);
        dodaj_zaw_katalogu($sciezka);
    }
}

sub dodaj_zaw_katalogu {
    my $sciezka = $_[0];
    my $starykursor = $glowne->cget('-cursor');
    $glowne->configure('-cursor' => 'watch');
    $glowne->update();
    my @pliki = <$sciezka/*>;
```

```
        foreach $plik (@pliki) {
            $plik =~ s|//|/|g;
            ($tekst = $plik) =~ s|^.*/||g;
            if (-d $plik) {
                $h->add($plik, '-itemtype' => 'imagetext',
                        '-image' => $ikony{"zamkniety"}, '-text' => $tekst);
            } else {
                $h->add($plik, -itemtype => 'text',
                        '-text' => $tekst);
            }
        }
        $glowne->configure('-cursor' => $starykursor);
    }
```



Rysunek 14.9. Widget HList w przeglądarce systemu plików

Na tym kończy się nasza przechadzka po galerii widgetów Tk i Tix. Opis innych widgetów można znaleźć w dokumentacji Tk; w katalogu *contrib* dystrybucji Tk można natomiast znaleźć widgety dostarczone przez niezależnych programistów. Spójrzmy teraz na pozostałe mechanizmy udostępniane przez pakiet Tk: zarządzanie geometrią, liczniki czasu oraz dowiązania i pętle zdarzeń.

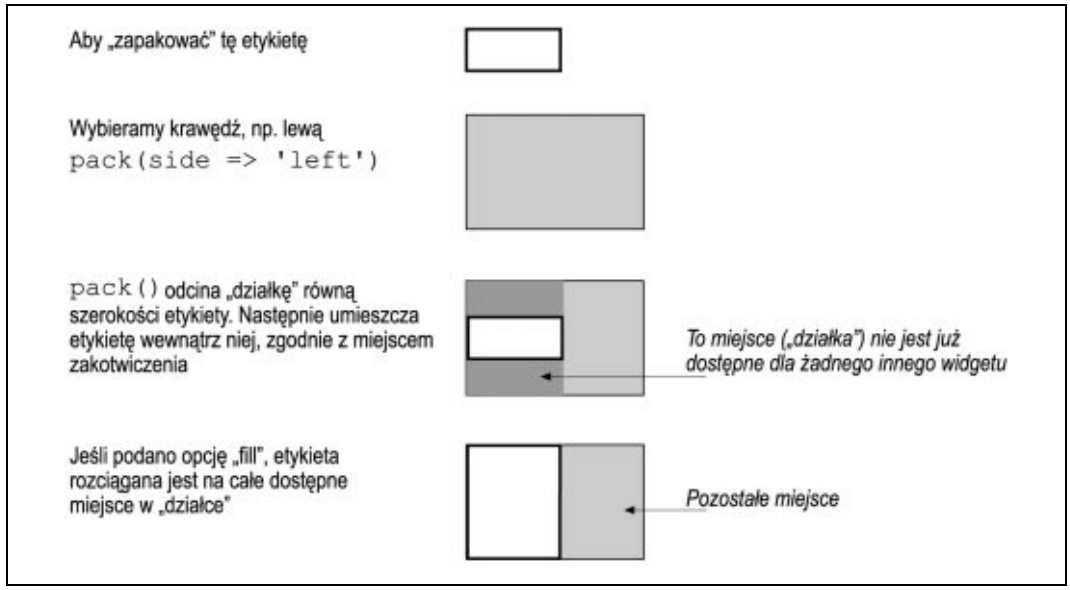
Zarządzanie geometrią

Zobaczyliśmy już, do czego służy metoda `pack`. Chodzi o koncepcję określaną mianem „zarządzania geometrią" — układania widgetów na ekranie i określania sposobu ich zachowania, gdy wymiary ekranu ulegną zmianie. Tk obsługuje trzy typy menedżerów geometrii: *placer*, *packer* oraz *grid*. Najprostszy jest menedżer placer; podobnie jak w przypadku widgetu Bulletin Board pakietu Motif albo w przypadku reguł zarządzania geometrią Visual Basic, wystarczy określić tutaj współrzędne *x* i *y* każdego widgetu. Więcej informacji o menedżerze placer można znaleźć w dokumentacji Tk.

Packer

Packer, podobnie jak widget Form pakietu Motif, jest dobrym menedżerem geometrii, obsługującym koncepcję dostępnego (pozostałego) pola, na którym widget można umieścić. Packer nie jest obiektem; to po prostu algorytm zaimplementowany za pomocą metody `pack()`. Innymi słowy, wywołanie `$widget->pack()` oznacza, że polecamy widgetowi „upakować się" w najbliższym dostępnym miejscu wewnątrz widgetu nadrzędnego.

Kiedy pakujemy walizkę, zazwyczaj zaczynamy od jednego końca i każdy następny przedmiot umieszczamy w najbliższym wolnym miejscu. Dokładnie tak działa packer; jest jednak jedna ważna różnica. Kiedy już widget dołączony jest do krawędzi widgetu nadrzędnego, cała ta krawędź jest „odcinana" z pozostałej dostępnej przestrzeni. Algorytm ten zilustrowano na rysunku 14.10.



Rysunek 14.10. Algorytm „pakowania"

Gdyby w zilustrowanym wyżej przykładzie parametrowi `side` przekazano wartość `top` lub `bottom`, wysokość odciętej „działki" zależna byłaby od wysokości etykiety.

Funkcja `pack` służy do wykonywania trzech czynności:

- Określania kolejności pakowania widgetów.

  Kolejność wywołania `pack` wpływa na kolejność umieszczania widgetów w oknie; to zaś wpływa na ilość miejsca dostępnego dla widgetu. Kiedy wymiary pojemnika ulegają zmianie, algorytm pakowania wykonywany jest ponownie, przy zastosowaniu tej samej kolejności.

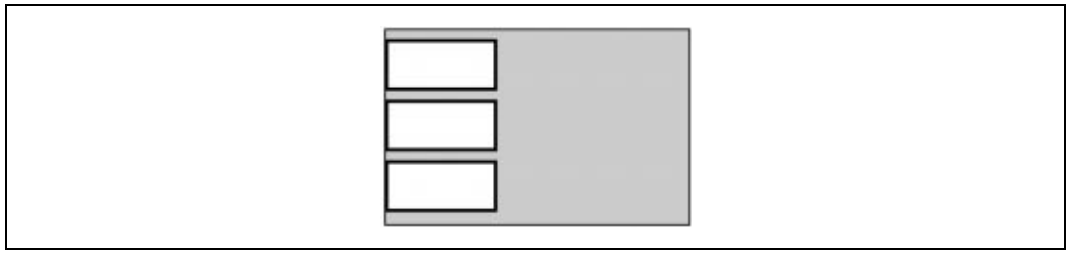
- Określania, jak wypełniana jest dostępna „działka".

  Tym ustawieniem steruje wartość `fill`: `x` (rozciągnij widget wzdłuż osi *x* do pełnej szerokości „działki"); `y` (rozciągnij widget wzdłuż osi *y* do pełnej wysokości „działki"); `both` (wzdłuż obu osi); `none` (nie rozciągaj). Opcje `ipadx` i `ipady` powodują, że wokół widgetu pozostawiane jest trochę miejsca (konieczne jest wtedy przydzielenie odpowiednio większej „działki"). Opcja `anchor` określa krawędź lub róg „działki", do którego przylega widget. Domyślnie opcja ta ma wartość `center`.

- Określania, co ma się stać z pozostałym miejscem widgetu nadrzędnego po wstawieniu wszystkich wymaganych widgetów potomnych.

  Służy do tego parametr `expand` („rozciągnij"). Zazwyczaj ostatni wstawiany widget zajmuje całe pozostałe miejsce — „wypełnia" je w całości. Jeśli jednak inne widgety były pakowane z wartością `expand` ustawioną na `y` (tak), wtedy dodatkowe miejsce w poziomie dzielone jest równo między wszystkie widgety z taką wartością — ale te, których wartość `side` jest równa `left` bądź `right`. Podobnie dodatkowe miejsce w pionie dzielone jest między wszystkimi widgetami z wartościami `top` lub `bottom`. Algorytm pakowania nigdy nie dopuszcza do nakładania się widgetów.

A jak umieścić trzy widgety po lewej stronie (tak jak na rysunku 14.11), jeśli pierwszy ma zająć całą „pionową działkę"?



Rysunek 14.11. Trzy widgety „upakowane" po lewej stronie

Jedynym sposobem rozwiązania tego problemu jest utworzenie widgetu ramki i umieszczenie go po lewej stronie okna głównego. Ponieważ ramka jest pojemnikiem innych widgetów, te trzy widgety można umieścić wewnątrz ramki — można to uzyskać, pisząc następujący kod:

```
$ramka->pack(-side => 'left', -fill => 'y', -expand => 'y');
# Teraz tworzymy przyciski p1, p2 i p3 jako elementy potomne ramki
# i pakujemy je od góry do dołu
$b1 = $ramka->Button (-text => 'Hej  ')->pack();
$b2 = $ramka->Button (-text => 'cześć')->pack();
$b3 = $ramka->Button (-text => 'wam  ')->pack();
```

`pack` domyślnie umieszcza widgety, zaczynając od góry.

Alternatywnym sposobem — i być może prostszym — jest skorzystanie z menedżera geometrii grid.

Grid

Metoda `grid` informuje widget, że ma skorzystać z usług menedżera geometrii `grid` (siatka). Wszystkie widgety potomne danego widgetu macierzystego muszą używać tego samego menedżera geometrii; jednak dla dowolnej kombinacji widgetów macierzystego i potomnych oraz dla widgetów osadzonych wewnątrz tych potomnych możemy użyć dowolnego menedżera geometrii.

Menedżer geometrii `grid` umożliwia układanie widgetów w wierszach i kolumnach, podobnie jak to się robi z użyciem znaczników HTML-a do tworzenia tabel. Szerokość kolumny zależy od najszerszego widgetu w niej umieszczonego, a wysokość — od najwyższego. Siatkę taką uzyskujemy w następujący sposób:

```
$przycisk->grid (row =>0, column =>0);
```

To polecenie umieszcza przycisk w górnym lewym rogu.

Podobnie jak w przypadku tabel HTML, widget może zajmować dowolną liczbę wierszy i kolumn — służą do tego opcje `rowspan` i `columnspan`. Widget wciąż jednak należy do wiersza i kolumny określonych opcjami `row` i `column`:

```
$przycisk->grid(row => 1, col => 2,
             columnspan => 2, sticky => 'ns');
```

Utworzony powyżej przycisk zajmuje dwie kolumny, ale nie wykorzystuje całego dostępnego miejsca. Opcja `sticky` powoduje, że widget „przylepiony" jest do ścianek północnej i południowej swojej komórki. Gdyby wartość tej opcji wynosiła `nsew` (*north* — północ; *south* — południe; *east* — wschód; *west* — zachód), widget zostałby rozciągnięty na całą komórkę. Domyślnie widget umieszczany jest w centrum „działki" i zajmuje tylko tyle miejsca, ile musi zajmować. Podobnie jak packer, menedżer `grid` także obsługuje opcje `ipadx` oraz `ipady`.

Liczniki czasu

Tk udostępnia niezajmujące wiele zasobów liczniki czasu: procedura może zostać wywołana po upłynięciu czasu określonego w milisekundach. Jednorazowe liczniki tworzymy metodą `after` — można ją zastosować w dowolnym widgecie; liczniki powtórzeniowe uzyskuje się metodą `repeat`. W poniższym przykładzie przycisk zmienia etykietę po wciśnięciu, a po 300 milisekundach etykieta powraca do poprzedniego stanu.

```
$przycisk->configure (text => 'A kuku.',
                      command => \&zmien_nazwe);
sub zmien_nazwe {
    my ($stara_nazwa) = $przycisk->cget('text');
    $przycisk->configure (text => '...ojej');
    $przycisk->after (300,
                      sub {$przycisk->configure(text => $stara_nazwa)});
}
```

Z metody `after` będziemy intensywnie korzystali w rozdziale 15. do tworzenia animacji.

Obie metody, `after` i `repeat`, zwracają obiekty w postaci liczników czasu. Ponieważ mechanizmy te są dość wydajne i mało wymagające (w przeciwieństwie do funkcji `alarm()` i sygnału `SIGALRM`), możemy tworzyć wiele liczników bez obawy o szybkość działania programu. Aby anulować licznik, używamy metody `cancel`:

```
$licznik = $przycisk->after(100, sub {print "foo"});
$licznik->cancel();
```

Mechanizm liczników w Tk, w przeciwieństwie do `SIGALRM`, nie powoduje wywłaszczenia; aby było możliwe sprawdzenie, czy licznik skończył odliczanie, kontrola musi zostać przekazana do pętli zdarzeń. Długo działająca podprocedura może więc opóźnić odliczanie.

Dowiązywanie zdarzeń

Dowiązanie zdarzenia jest skojarzeniem pewnej funkcji z dowolnym typem zdarzenia. Przykłady już widzieliśmy — na przykład właściwość `command` widgetu przycisku powoduje, że kliknięcie przycisku myszy wywołuje procedurę zdefiniowaną przez użytkownika. Polecenie `bind()` zapewnia bardziej ogólny (a więc niskopoziomowy) dostęp do tych najbardziej podstawowych zdarzeń, takich jak wciśnięcie lub zwolnienie klawisza na klawiaturze czy przycisku myszy (kliknięcie myszą składa się z wciśnięcia i puszczenia przycisku myszy — jak widać, zajmujemy się tutaj naprawdę niskopoziomowymi zdarzeniami). Inne „interesujące" zdarzenia to ruchy myszą, wejście wskaźnika myszy w obszar okna lub wyjście z niego, przeniesienie lub zmiana wielkości okna. Same widgety do działania i tak wykorzystują metodę `bind()`, ale użytkownik może określić własne dowiązania. Dowiązana procedura wykonywana jest wtedy, gdy śledzone zdarzenie ma miejsce wewnątrz widgetu lub jest z nim związane (np. zmiana wielkości okna).

Składnia funkcji `bind` jest następująca:

```
$widget->bind(sekwencja zdarzeń, podprocedura);
```

Sekwencja zdarzeń to łańcuch zawierający nazwy podstawowych zdarzeń; każde podstawowe zdarzenie umieszczane jest w nawiasach kątowych. Poniżej przedstawiono przykłady sekwencji zdarzeń:

```
"<a>"                   # Wciśnięto klawisz "a" (nie wciśnięto klawisza
                        # Control/shift/meta
"<Control-a>            # Wciśnięto Control i a
"<Escape> <Control-a>"  # Sekwencja dwóch zdarzeń
"<Button1>"             # Wciśnięto przycisk 1 myszy
"<Button1-Motion>"      # Poruszono myszą przy wciśniętym przycisku 1
```

Jedno zdarzenie (to umieszczane w nawiasach kątowych) ma następującą składnię:

```
"<modyfikator-modyfikator...-modyfikator-typ-składnik>"
```

Przykłady *modyfikatorów* to: `Control`, `Meta`, `Alt`, `Shift`, `Button1` (lub `B1`), `Button2`, `Double` (dwukrotne kliknięcie) i `Triple` (kliknięcie potrójne). Modyfikator `Any` oznacza wszystkie możliwe modyfikatory (a także brak modyfikatora).

Typ zdarzenia to `KeyPress` (wciśnięcie klawisza), `KeyRelease` (puszczenie klawisza), `ButtonPress` lub `Button` (wciśnięcie przycisku myszy), `ButtonRelease` (puszczenie przycisku myszy), `Enter` (wejście w obszar okna), `Leave` (wyjście z obszaru okna) oraz `Motion` (ruch).

Przy określaniu zdarzeń związanych z klawiaturą *składnik* jest łańcuchem określającym konkretny klawisz. W przypadku znaków ASCII jest to po prostu sam znak; inne symbole to: `Enter`, `Right`, `Pickup`, `Delete`, `BackSpace`, `Escape`, `Help`, `F1` itd.

Najczęściej obsługiwane zdarzenia to wciśnięcia klawiszy i kliknięcia myszą. Dlatego w Tk wprowadzono skrótowy zapis: zamiast pisać `<KeyPress-a>`, wystarczy `<a>`; a zamiast `<Button1-ButtonPress>`, wystarczy `<1>`.

Widgety pól tekstowego i graficznego umożliwiają precyzyjniejsze określanie zdarzeń. Oprócz zdarzeń związanych z samym widgetem, można tutaj dowiązać procedury do funkcji związanych ze znacznikami. Jako pierwszy argument `bind` można podać nazwę znacznika, a jako drugi i trzeci — odpowiednio sekwencję zdarzeń i nazwę podprocedury:

```
$tekst->bind('odsylacz', '<1>', \&otworz_strone);
```

Po wprowadzeniu takiego kodu każda porcja tekstu oznaczona jako „odsylacz" będzie reagowała na kliknięcie przyciskiem myszy przez wywołanie podprocedury `otworz_strone`.

Dowiązywanie wielu podprocedur

Do tego samego zdarzenia można dowiązać kilka podprocedur. Na przykład wciśnięcie przycisku myszy może być interpretowane zarówno jako `<Button1>`, jak i jako `<Double--Button1>`. Jeśli dla danego widgetu (lub znacznika) zachodzi konflikt zdarzeń, wywoływana jest procedura określona dla zdarzenia bardziej specyficznego. Tutaj bardziej specyficznym zdarzeniem jest `<Double-Button1>` (zdarzenie to jest dokładniej opisane niż `<Button1>`).

Oprócz dopasowania najbardziej specyficznego dowiązania na poziomie widgetu, Tk podobnie postępuje na poziomie klasy (np. klasy „wszystkie przyciski), potem na poziomie głównym widgetu, wreszcie na poziomie o nazwie „all" (wszystkie). Wykonywane są dowiązania wszystkich czterech kategorii. Tę kolejność można zmienić metodą `bindtags()`, jednak nie zalecam takiego postępowania.

Choć Tk umożliwia zmianę domyślnych dowiązań widgetów, zalecam pozostawienie ich niezmienionych. Użytkownicy przyzwyczajają się do pewnego sposobu działania programów. Na przykład dwukrotne kliknięcie wewnątrz widgetu pola tekstowego zazwyczaj powoduje zaznaczenie słowa znajdującego się pod wskaźnikiem myszy; użytkownik mógłby odczuć zmniejszenie wygody korzystania z programu, gdybyśmy dowiązanie to zmienili. Z drugiej strony istnieje wiele miejsc, w których można — i powinno się — dodawać własne dowiązania. Najczęściej robi się to w polach tekstowym i graficznym, o czym przekonamy się w następnych dwóch rozdziałach.

Składniki

Wiemy już, jak precyzyjnie określić zdarzenie. Są jednak sytuacje, w których musimy postąpić dokładnie odwrotnie: chcemy zdefiniować zdarzenie możliwie ogólnie, np. jako `<Any-KeyPress>` (wciśnięcie dowolnego klawisza). Przecież nie będziemy określali oddzielnych dowiązań dla każdego klawisza. Jednak kiedy już klawisz zostanie wciśnięty, być może przydałoby się, aby podprocedura uzyskiwała informację o tym, który to był klawisz. Tu właśnie dochodzą do głosu składniki zdarzenia.

Każde zdarzenie zawiera informację o składnikach; do uzyskania tych informacji służy funkcja `Ev()`. Parametrem tej funkcji jest jeden znak określający tę część rekordu zdarzenia, jaka nas interesuje. `Ev('k')` określa kod znaku, `Ev('x')` i `Ev('y')` — współrzędne *x* i *y* wskaźnika myszy, a `Ev('t')` — czas zdarzenia. Funkcja `Ev` obsługuje ponad 30 takich parametrów. Poniżej pokazano, jak korzystać z tego mechanizmu:

```
$etykieta->bind("<Any-KeyPress>" => [\&porusz, Ev('k')]);
sub porusz {
    my $klawisz = shift;
    if ($klawisz eq 'k') {
       porusz_w_lewo();
    } elsif ($klawisz eq 'l') {
       porusz_w_prawo();
    }
}
```

W tym przykładzie funkcja `bind` jest tak uruchomiona, że rejestruje zdarzenia związane z wciskaniem klawiszy; do określonej przez użytkownika podprocedury przesyłane są przy każdym wywołaniu kody klawiszy.

Pętle zdarzeń

`MainLoop` uruchamia pętlę zdarzeń. Przechwytuje ona zdarzenia związane z systemem okien i przekazuje je odpowiednim widgetom. Kiedy w odpowiedzi na zdarzenie wywoływana jest pewna procedura, oczekuje się, że jej wykonywanie zostanie zakończone możliwie szybko; w przeciwnym razie wstrzymuje ona wszystkie kolejne zdarzenia, które zaszły od chwili jej wywołania.

Jeśli konieczne jest wykonanie długiej czynności, intensywnie korzystającej z procesora, programista powinien podzielić ją na mniejsze fragmenty i ustawić licznik, który będzie to zadanie wykonywał w regularnych odstępach czasu. Dzięki temu pętla zdarzeń ma szansę obsłużyć oczekujące zdarzenia. Takie dzielenie czasu procesora określa się mianem „wielozadaniowości równoległej". Wczesne wersje Microsoft Windows (do wersji 3.1) wymagały, aby aplikacje zachowywały się właśnie w taki, poprawny sposób. W przeciwnym razie „zamrażały" one cały system operacyjny.

W zadaniach takich jak generacja grafiki trójwymiarowej czy animacje — czyli intensywnie korzystających zarówno z procesora, jak i z interfejsu graficznego — można użyć metody `$widget->update`, która powoduje przetworzenie wszystkich zdarzeń. Kończy ona działanie dopiero wtedy, gdy wszystkie oczekujące w kolejce komunikaty (w tym żądania odświeżenia ekranu) zostaną przetworzone.

W środowiskach ukierunkowanych na zdarzenia nie powinno się stosować blokujących funkcji systemowych, o czym powiedzieliśmy już w rozdziale 12. Najbardziej znane funkcje tego typu to `read` i `write`, szczególnie jeśli wykonywane są na potokach i gniazdach. Na przykład operator <> blokuje wykonywanie programu dopóty, dopóki nie uzyska kolejnego wiersza tekstu. Zamiast bezpośrednio wykonywać operację wejścia-wyjścia, trzeba spowodować, aby Tk poinformował nas, że taką operację można już

przeprowadzić — wiemy, że wtedy nie zablokuje ona programu. Tk udostępnia procedurę `fileevent`, która powiadamia procedurę, kiedy deskryptor pliku jest gotowy do odczytu lub zapisu. Korzystamy z niej w następujący sposób:

```
open (F, "/tmp/foo");
$przycisk->fileevent(F, "readable", \&odczytaj_plik);
sub odczytaj_plik {
    if (eof(F)) {
        $przycisk->fileevent(F, "readable", undef); # anuluj dowiązanie
            return ;
    }
    if (sysread (F, $buf, 1024)) {
        $tekst->insert('end', $buf);                # dopisz odczytane dane
    } else {
        # sysread zwrócił undef. Problem z plikiem.
        $tekst->insert('end', "Błąd !!!";
        $przycisk->fileevent(F, "readable", undef); # anuluj dowiązanie
    }
}
```

Kiedy wywołana jest taka procedura, Tk (który w Uniksie wykorzystuje wewnętrznie funkcję `select`) gwarantuje, że do odczytu lub zapisu jest gotowy najwyżej jeden znak. Nie gwarantuje się gotowości większej ilości danych — dalszy odczyt lub zapis spowoduje lub nie spowoduje blokowania. Funkcja wywoływana jest również w przypadku napotkania znaku końca pliku lub wystąpienia błędu, trzeba więc sprawdzać, czy nie zaszła któraś z tych okoliczności. W przeciwnym razie procedura wykonywana jest zaraz po tym, jak zwróciła dane — czyli tworzona jest nieskończona pętla. Jak powiedzieliśmy przy okazji poruszania tematyki sieciowej, najlepiej stosować nieblokujące funkcje wejścia-wyjścia, o ile tylko są obsługiwane przez system.

W tym rozdziale omówiliśmy widgety, pętle zdarzeń, liczniki czasu i dowiązania zdarzeń. W następnych dwóch połączymy opisane koncepcje i zastosujemy je do rozwiązania praktycznych problemów. Zobaczymy również, jak w naprawdę ciekawy sposób można wykorzystać dwa przydatne widgety Tk: pola graficzne i tekstowe.

Źródła informacji

1. Biblioteki i teksty o Tcl/Tk dostępne pod adresami *http://www.sunlabs.com/research/Tcl* (strony firmy Sun) i *http://www.neosoft.com* (wszystko o Tcl/Tk).

2. Dokumentacja *Perl/Tk*.

3. *About Face: The Principles of User-Interface Design.* Alan Cooper. IDG Books Worldwide, 1995.

   Książka forsująca nowy sposób postrzegania tematyki projektowania interfejsów graficznych.

4. *Bringing Design to Software*. Terry Winograd. Addison-Wesley, 1996.

   Specjaliści różnych dziedzin piszą o projektowaniu dobrych programów, szczególnie interfejsów użytkownika.