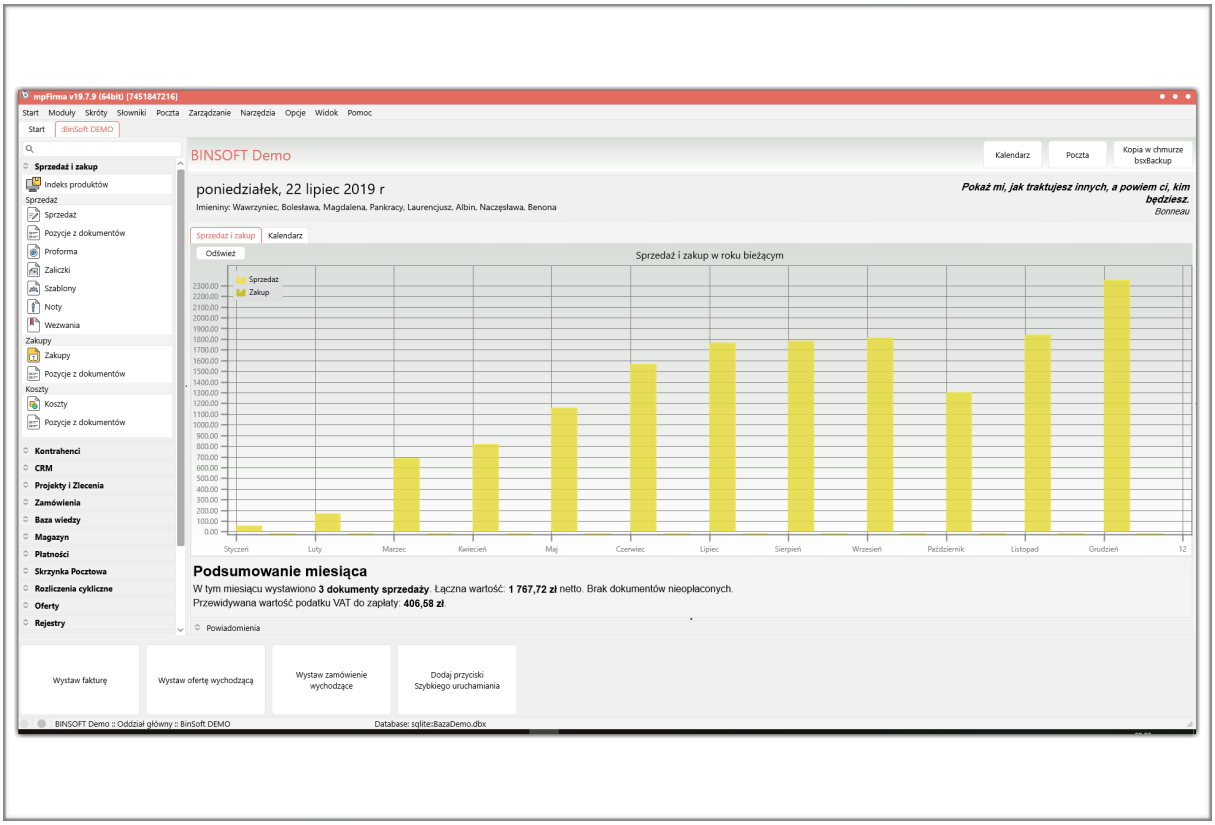
Instrukcja programisty



Oprogramowanie serii MP

www.binsoft.pl

Lipiec 2019

Wstęp

Firma BinSoft jest producentem oprogramowania dla firm serii MP i ABC. Dostępnych jest kilka wariantów tego oprogramowania dostosowanych do różnych branży. W chwili pisania tego podręcznika publiczne były następujące aplikacje:

- **mpCRM** - program do zarządzania relacjami z klientami,
- **mpPOS** - program do prowadzenia sprzedaży detalicznej;
- **mpFaktura** - program do prowadzenia sprzedaży,
- **mpFirma** - kompletne rozwiązanie do obsługi firm;
- **mpSekretariat** - rozwiązanie dla sekretariatów;
- **mpGabinet Lekarski** - rozwiązanie dla gabinetów i przychodni lekarskich;
- **abcFaktury**, **abcMagazynu**, **abcGabinetu** - najprostsze wersje naszego oprogramowania dostosowane do różnych branży;

Wszystkie te aplikacje bazują na silniku zwanym **BSX**. Dzięki temu są ze sobą kompatybilne i różnią się jedynie dostępnością określonych modułów oraz funkcji.

Aplikacje dostępne są dla systemów Windows (w wersji 32- i 64-bitowej) oraz macOS. Dzięki zachowaniu kompatybilności pomiędzy tymi wersjami, istnieje możliwość pracowania na wspólnych danych z poziomu różnych systemów operacyjnych.

Silnik **BSX** został stworzony w taki sposób, aby można było rozszerzać funkcjonalności aplikacji stworzonych przy jego pomocy, za pośrednictwem tzw. wtyczek (ang. *plugins*). Niniejszy podręcznik wyjaśnia zasady tworzenia takich rozszerzeń. Dzięki temu mechanizmowi istnieje możliwość rozbudowywania

każdego z programów MP/ABC o dodatkowe funkcjonalności, w tym integracje z innymi aplikacjami i systemami, dostosowywanie i wdrażanie do indywidualnych potrzeb klienta itp.

Za pośrednictwem mechanizmu wtyczek istnieje możliwość:

- tworzenia własnych struktur w bazie danych (tabel),
- tworzenia tzw. widoków na te tabele,
- tworzenia formularzy,
- tworzenia elementów menu i panelu modułów,
- tworzenia funkcjonalności działających w tle;
- rozszerzania funkcjonalności istniejących modułów;

API jakie oferuje BSX pozwala programiście m.in. na:

- bezpośredni dostęp do bazy danych z użyciem dialektu BSX SQL (niezależnie od systemu bazodanowego, z którym połączony jest BSX);
- dostęp do API umożliwiających komunikację za pomocą HTTP, FTP, SFTP, REST;
- dostęp do API umożliwiającego obsługę drukarek fiskalnych, terminali płatniczych;
- i wiele więcej.

Aby rozpocząć tworzenie własnych rozszerzeń i integracji z systemem BSX (programami MP/ABC) nie jest potrzebne żadne specjalistyczne oprogramowanie. Wystarczy dowolny edytor tekstowy (ASCII). Rozszerzenia można tworzyć pod dowolnym systemem operacyjnym.

Oprócz zasady tworzenia wtyczek dla systemu BSX, podręcznik ten wyjaśnia różne inne, bardziej zaawansowane informacje związane z obsługą, konfiguracją i zarządzaniem programami z serii MP/ABC.

Modele współpracy

Dostępne są trzy modele współpracy z firmą BinSoft w zakresie tworzenia własnych rozszerzeń. Poniżej przedstawiono krótkie charakterystyki każdego z nich.

Wariant 1 - Podstawowy

Partner ma możliwość tworzenia własnych rozszerzeń do oprogramowania MP /ABC bez ponoszenia żadnych dodatkowych opłat ani zgłaszania takiej chęci. Ograniczona jest jednak dostępność API i funkcji jakie ono oferuje. Ten wariant charakteryzuje się również jedynie podstawową pomocą techniczną świadczoną za pośrednictwem poczty e-mail. Tworzone wtyczki są „jawne", co oznacza, że ich kod źródłowy dostępny jest dla każdego, kto skorzysta z takiego rozwiązania. Głównym przeznaczeniem tego modelu współpracy jest rozbudowa i dostosowywanie aplikacji do własnych potrzeb.

Niniejsza instrukcja wyjaśnia zasady tworzenia wtyczek tylko w tym modelu współpracy, a więc dostępnym dla wszystkich użytkowników.

Wariant 2 - Rozszerzony

Ten poziom współpracy wymaga zgłoszenia do firmy BinSoft i uiszczenia stosownej opłaty licencyjnej. W jej ramach Partner uzyskuje narzędzia pozwalające na kompilację tworzonych wtyczek, dzięki czemu dystrybuując je nie udostępnia swoich kodów źródłowych. Użytkownik uzyskuje pełny dostęp do API oraz priorytetowej pomocy technicznej. Wariant ten przeznaczony jest dla firm, które pragną zajmować się wdrażaniem aplikacji MP u klientów końcowych, tworzeniem komercyjnych rozszerzeń i integracji itp.

Wariant 3 - Partnerski

Najwyższy z poziomów współpracy z firmą BinSoft. Wymaga podpisania umowy i dokonania opłaty licencyjnej. W ramach tego modelu współpracy Partner otrzymuje pełną dokumentację do systemu BSX oraz niezbędne

narzędzia do tworzenia własnych rozszerzeń i własnych aplikacji. Partner uzyskuje także pełne, priorytetowe wsparcie techniczne. Ten model współpracy przeznaczony jest dla firm, które pragną tworzyć pełne, indywidualne wdrożenia systemów bazodanowych dla firm. Umożliwia dowolne dostosowywanie aplikacji MP, dodawanie własnych modułów i rozszerzeń, a nawet tworzenie własnych, unikalnych aplikacji obrandowanych własną firmą. W tym wariancie istnieje również możliwość uzyskania pełnego kodu źródłowego wszystkich składowych modułów oraz zgodę na ich modyfikację i dostosowywanie do własnych potrzeb.

Podstawy tworzenia wtyczek

Tworzenie wtyczek do programów MP/ABC polega na przygotowywaniu odpowiednich plików XML oraz PAS (Pascal). Nie są do tego celu potrzebne zatem żadne dodatkowe narzędzia, wystarczy ulubiony edytor kodu ASCII (bez formatowania). Za pośrednictwem plików XML opisuje się strukturę danych w bazie danych, wygląd formularzy, widoków, elementów menu itp. Za pomocą języka Pascal i plików PAS można dodać logikę do danej wtyczki.

Każde rozszerzenie (plugin) składa się z reguły z jednego lub kilku plików. Wszystkie powinny być umieszczone w jednym folderze nazwanym według schematu `Plugin.FIRMA.NazwaPluginu`,
np. `Plugin.MySoft.WtyczkaTestowa`. Wszystkie tego typu foldery umieszcza się w katalogu `plugins` (Domyślnie: `Dokumenty/BinSoft/mpFirma/Core/plugins` lub `Dokumenty/mpFirma/Core/plugins` - zależnie od wersji programu).

W momencie uruchamiania aplikacji MP/ABC, system BSX automatycznie przeszukuje folder `plugins` w poszukiwaniu wszystkich podfolderów wtyczek, następnie przetwarza wszystkie pliki XML znajdujące się w tych podfolderach. Ignorowane są foldery rozpoczynające się znakiem `@`. Dzięki temu chcąc „wyłączyć” daną wtyczkę nie ma potrzeby jej usuwania. Wystarczy poprzedzić jej nazwę wspomnianym znakiem.

Z uwagi na fakt, że BSX analizuje wszystkie pliki XML znajdujące się w folderze z daną wtyczką, nie ma znaczenia jak taki plik zostanie nazwany.

Struktura tego pliku powinna wyglądać następująco:

```
<plugin name="Nazwa.Wtyczki" caption="Tytuł wtyczki" >
…
</plugin>
```

Wewnątrz znacznika <plugin> znajdują się znaczniki opisujące różne elementy, np. elementy menu, panelu zakładek, widoków czy formularzy.

Nazwa wtyczki (atrybut name) powinna mieć formę ścieżki - analogiczną no nazwy folderu, w którym dany plik się znajduje.

Spójrzmy na prosty przykład (**P1**):

wtyczka.xml

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka">
 <system>
   <param name="autorun" src="skrypt.pas" />
 </system>
</plugin>
```

skrypt.pas

```
begin
  ShowMessage('Witaj Świecie!');
end.
```

Po uruchomieniu aplikacji MP/ABC system BSX przeanalizuje plik wtyczka.xml. W sekcji <system> dostępny jest tzw. parametr <param> o nazwie autostart. Za jego pośrednictwem możemy wskazać plik ze skryptem Pascala, który ma zostać zainicjowany w momencie uruchamiania aplikacji. W efekcie powyższego zapisu w XML-u, po uruchomieniu programu MP/ABC „uruchomiony" zostanie skrypt.pas, czego efektem będzie wyświetlenie komunikatu „Witaj Świecie".

Zmieniając parametr na (**P2**):

```
<param name="initDatabase" src="skrypt.pas" />
```

moglibyśmy zainicjować skrypt nie po uruchomieniu programu, lecz dopiero po otwarciu bazy danych.

Spójrzmy na kolejny przykład (**P3**):

wtyczka.xml

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka" >
 <menu>
  <item name="MainMenu">
    <item name="mnStart">
      <item caption="-" />
      <item caption="Moja opcja" onclick="[skrypt.pas]Klik" />
    </item>
  </item>
 </menu>
</plugin>
```

skrypt.pas

```
procedure Klik;
begin
  ShowMessage('Kliknięto w menu');
end;
```

Tym razem, z poziomu pliku XML stworzyliśmy nową pozycję w menu *Start* zatytułowaną *Moja opcja*. Kiedy opcję tę klikniemy wywoła się procedura `Klik` z pliku `skrypt.pas`.

W ten sposób możemy tworzyć dowolną strukturę elementów w menu oraz podpinać pod te elementy dowolne funkcje z plików PAS.

Wreszcie spójrzmy na jeszcze jeden przykład (**P4**).

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka" >
 <menu>
  <item name="MainMenu">
    <item name="mnStart">
```

```
      <item caption="-" />
      <item caption="Moja opcja"
          onclick="showform:Plugin.MyCompany.Wtyczka.Okienko" />
    </item>
  </item>
 </menu>

 <forms>
   <form name="Okienko" caption="Formularz wtyczki"
         width="500" height="400">
    <edit name="imie" caption="Imię" width="100" />
    <edit name="nazwisko" caption="Nazwisko" width="200" />
   </form>
 </forms>

</plugin>
```

Tym razem po kliknięciu w pozycję *Moja opcja* w menu Start pojawi się nowe okno zatytułowane *Formularz wtyczki*. Zawiera ono dwa pola edycyjne, jedno pod drugim, pozwalające na wprowadzenie imienia i nazwiska, oraz dwa przyciski *OK* i *Anuluj*. Wypełnienie tych pól nie wpływa oczywiście na działanie programu (nie jest nigdzie zapamiętywane). Klikniecie w dowolny z tych przycisków powoduje jedynie zamknięcie okna. W dalszej części podręcznika dowiemy się jednak jak budować bardziej rozbudowane formularze tego typu, jak przechwytywać różnego rodzaju zdarzenia, jak zapamiętywać dane w bazie danych i wiele więcej.

Przeładowanie plików i monitor Developera

System BSX ładuje kod wtyczki (tj. pliki XML i/lub PAS) w momencie uruchamiania programu lub w momencie pierwszego wyświetlania określonego formularza lub widoku. Oznacza to, że wprowadzanie zmian w plikach XML i/lub PAS nie będzie skutkowało zmianami w działającej aplikacji, do czasu jej

ponownego uruchomienia. Przygotowywanie wtyczek i własnych rozszerzeń sprowadza się jednak do wprowadzania wielokrotnych zmian w kodzie, poprawiania wyglądu formularzy itd. Dlatego system BSX oferuje przydatny skrót klawiszowy: **CTRL+F12**. Powoduje on ponowne przeładowanie wszystkich plików XML i PAS, oraz wyczyszczenie bufora wygenerowanych okien. Dlatego po każdej zmianie w plikach XML/PAS, aby zobaczyć ich efekt, należy skorzystać ze wspomnianego skrótu.

System BSX posiada również wbudowany tzw. *Monitor Developera*. Pozwala on na podgląd wszystkich wykonywanych zapytań do bazy danych, analizę wszystkich wątków aplikacji, bezpośredni dostęp do serwera SQL oraz serwera BSX, dostęp do tzw. konsoli BSX i wiele więcej. Aby wywołać okno developera należy posłużyć się skrótem **SHIFT+CTRL+ALT+F12**.

Podsumowanie:

- **`CTRL+F12`** - przeładowanie plików XML i wyczyszczenie zbuforowanych okien;
- **`SHIFT+CTRL+ALT+F12`** - uruchomienie *Monitora Developera*;
- **`SHIFT+CTRL+ALT+F11`** - uruchomienie *Konsoli BSX*;

Chcąc wyświetlić okno developera poproszeni będziemy o podanie hasła serwisowego. Aby je uzyskać należy się zgłosić do firmy BinSoft.

Elementy „wtyczki”

Aby zrozumieć zasadę działania i tworzenia własnych rozszerzeń w systemie BSX należy uprzednio poznać stosowane w tym zagadnieniu pojęcia oraz ich znaczenie. W kolejnych akapitach wyjaśnione zostaną wszystkie tego typu elementy, wraz z informacjami o tym jak należy je definiować w plikach XML.

Tabele

Tabele definiują struktury w bazie danych. Każda tabela posiada kolumny określające rodzaj przechowywanych danych oraz wiersze reprezentujące te dane. Każda tabela posiada swoją nazwę, po czym jest identyfikowana. Przyjęło się nadawać tabelom przedrostki związane z naszą firmą. Dzięki temu będzie duże prawdopodobieństwo, że nie użyjemy nazwy użytej już przez kogoś innego. Wszystkie tabele utworzone przez aplikacje MP i zaprojektowane przez firmę BinSoft mają przedrostek **bs_**.

Tworząc tabelę definiujemy jej kolumny. Każda kolumna posiada swoją nazwę oraz typ. Typ definiuje jakiego rodzaju dane będą w niej przechowywane. Obsługiwane typy są następujące:

- **varchar** - typ tekstowy (do 250 znaków),
- **text** - typ tekstowy (niemal dowolnie długie teksty),
- **int** - typ liczb całkowitych,
- **double** - typ liczb zmiennoprzecinkowych,
- **datetime** - typ daty i czasu,
- **date** - typ służący do przechowywania samej daty,
- **blob** - typ danych binarnych;

System BSX obsługuje wiele różnych systemów bazodanowych. W zależności od tego, do jakiego systemu jest podłączony - tworzy w nim odpowiednie typy danych dla wybranych kolumn - na podstawie typów obsługiwanych przez niego samego (z listy powyżej).

W pliku XML definiującym wtyczkę można umieścić parę znaczników `<tables>...</tables>` określającą, że nastąpi definicja tabel w bazie danych. Następnie w obrębie tych znaczników umieszcza się dowolnie wiele sekcji `<ta-`

ble>...</table> - definiujących poszczególne tabele. Za każdym razem, kiedy użytkownik „podłączy" się do jakiejś bazy danych, system analizuje owe sekcje <tables> i porównuje je z tabelami w bazie danych. Jeśli wykryje, że brakuje jakiejś tabeli lub tabela istnieje, ale brakuje w niej odpowiednich kolumn, automatycznie utworzy brakujące elementy.

W różnych wtyczkach może istnieć definicja tej samej tabeli. System BSX przeanalizuje wszystkie te definicje razem. Dzięki temu tworząc własne rozszerzenie możemy rozbudować istniejące tabele o dodatkowe pola. Możemy np. w tabli `bs_invoices` (faktury) dodać potrzebne nam pole. Aby rozszerzyć istniejącą tabelę należy w znaczniku <table> dodać dodatkowy atrybut `extend="true"`.

Kolejne kolumny w sekcji <table> definiuje się znacznikiem <field>.

Znaczniki:

- **<tables>...</tables>** - sekcja opisująca tabele w bazie danych;
- **<table>...</table>** - opis pojedynczej tabeli;

  Atrybuty:

  - **name (string)** - nazwa tabeli;
  - **caption (string)** - tytuł tabeli;
  - **extend (string)** - wartość `true` oznacza, rozszerzamy istniejącą tabelę;
- **<field>** - opis kolumny tabeli;

  Atrybuty:

  - **name (string)** - nazwa kolumny,
  - **type (string)** - typ kolumny,

- **mtype (string)** - typ wewnętrzny,
- **caption (string)** - tytuł kolumny,
- **default (string) -** wartość domyślna,
- **createindex (bool)** - czy utworzyć indeks dla tej kolumny,
- **relation (string)** - nazwa tabeli, z którą dana kolumna jest w relacji,
- **foreign (string)** - wartość `true` oznacza, że ma dana kolumna jest kluczem obcym do tabeli opisanej w atrybucie `relation`. Wartość `#DELETE` tworzy również klucz obcy z ustawionym parametrem kasowania kaskadowego, w przypadku usunięcia danego wpisu;
- **size (int)** - wielkość pola (dotyczy pól typu `varchar`);

Tytuł kolumny (`caption`) nie jest obowiązkowy i jedynie opisuje kolumnę w pliku XML. Nie jest wyświetlany w żadnym miejscu w programie. Oprócz atrybutu `name` - wszystkie pozostałe atrybuty nie są obowiązkowe.

Definiując kolumnę określamy jej typ (`type`). Możemy także zdefiniować typ wewnętrzny (`mtype`). Jako wartość atrybutu `type` możemy użyć słów: `varchar`, `text`, `int`, `double`, `date`, `datetime` oraz `blob`. Typem wewnętrznym możemy zmienić sposób interpretacji danego pola przez system BSX.

Uwaga! Opisując tabelę nie definiujemy pola klucza głównego. Jest nim zawsze automatycznie dodawana kolumna **`id`**. Zalecane jest również dodanie w każdej tabeli kolumn: `add_id_user`, `add_time`, `modyf_id_user`, `modyf_time` - są to predefiniowane pola opisujące kto i kiedy utworzył dany rekord (wiersz tabeli) oraz kto i kiedy go zmodyfikował.

Jeśli nie chcemy aby system BSX automatycznie utworzył kolumnę `id` należy dodać atrybut `primaryid=""` do znacznika definiującego tę tabelę.

Oto przykładowa definicja tabeli (**P5**):

```
 <tables>

  <table name="mc_kontakty" caption="Kontakty">

   <field name="add_id_user" type="int" mtype="add_idusers"
          createindex="true" />

   <field name="add_time" type="datetime" mtype="add_time" />

   <field name="modyf_id_user" type="int" mtype="modyf_idusers"
          createindex="true" />

   <field name="modyf_time" type="datetime" mtype="modyf_time" />


   <field name="imie" type="varchar" size="50" />

   <field name="nazwisko" type="varchar" size="100" />

   <field name="wiek" type="int" default="0" />

   <field name="stanowisko" type="int" default="0" />

  </table>


</tables>
```

System BSX pozwala na definiowanie znacznie więcej informacji związanych z tabelami, m.in.: tworzenie triggerów (wyzwalaczy), automatycznie dodawanych rekordów w momencie tworzenia tabeli, tworzenie indeksów, wykonywanie dowolnego kodu Pascala w momencie tworzenia danej tabeli itp. Informacje te są jednak dostępne w wyższych wariantach współpracy z BinSoft niż wariant *Podstawowy*.

Widoki

Widok (ang. *view*) - to „podgląd" tabeli w bazie danych. Z punktu widzenia programu jest to obiekt tabelki zawierającej kolumny i wiersze, wyświetlający zawartość jakiejś tabeli z bazy danych. Definiując widok ustalamy, na którą tabelę z bazy danych ma on wskazywać, oraz które kolumny z tej tabeli ma wyświetlać. Widoki pozwalają na niemal dowolną konfigurację. Możemy wyświetlać wprost zawartość jakiejś tabeli, możemy także ustalić zapytanie SQL poprzez które uzyskamy tę tabelę. Mamy możliwość dowolnej konfiguracji wyświetlanych kolumn, określania ich wyrównywania, wymiarów, sposobu interpretowania wyświetlanych w nich danych itp. Możemy tworzyć filtry, definiować kolumny, po których mają być automatycznie wykonywane różne obliczenia, przechwytywać różnego rodzaju zdarzenia i wiele więcej.

Wszystkie widoki opisuje się znacznikiem `<view>...</view>` w obrębie pary znaczników `<views>...</views>`. Podczas definicji widoku może wystąpić szereg innych znaczników opisujących różne cechy danego widoku. Najważniejsze z nich to: `<query>` oraz `<field>`.

Znaczniki:

- **`<views>`...`</views>`** - sekcja opisująca widoki;
- **`<view>`...`</view>`** - opis pojedynczego widoku;

  Atrybuty:

  - **`name`** (**string**) - nazwa widoku;
  - **`caption`** (**string**) - tytuł widoku;
  - **`image`** (**string**) - ikona widoku;
  - **`canresizecolumns`** (**bool**) - czy można zmieniać rozmiar kolumn;

- **autostretchcolumn (string)** - nazwa kolumny, która ma być automatycznie rozciągana;
- **defaultrowheight (int)** - domyślna wysokość wierszy; np. dla wartości 2 - każdy wiersz tabeli będzie wysoki na dwa wiersze;
- **inherited (string)** - nazwa innego widoku, po którym ma nastąpić dziedziczenie;

• **<field>** - opis kolumny wyświetlanej w widoku;

Atrybuty:

- **name (string)** - nazwa kolumny w bazie danych,
- **caption (string)** - tytuł kolumny,
- **type (string)** - typ kolumny (fizyczny w bazie danych),
- **mtype (string)** - typ wewnętrzny (sposób interpretacji danych),
- **width (int)** - domyślna szerokość kolumny,
- **align (string)** - sposób wyrównywania zawartości danych w kolumnie; dostępne wartości: `left`, `center`, `right`;
- **search (bool)** - czy ma być dostępne wyszukiwanie po danej kolumnie,
- **sum (bool)** - czy zwartość kolumny ma być automatycznie sumowana;
- **template (string)** - szablon zawartości;
- **orderby (string)** - nazwa kolumny, po której ma odbywać się sortowanie (podajemy ten atrybut wówczas, jeśli chcemy by sortowanie odbywało się po innej kolumnie, niż ta której zawartość wyświetlamy);

• **<query>** - opis źródła pobieranych danych oraz powiązanego formularza;

Atrybuty:

- **`table` (string)** - nazwa tabeli, z której mają być czerpane dane,
- **`where` (string)** - dodatkowy warunek, którym możemy ograniczyć wyświetlane rekordy,
- **`orderby` (string)** - domyślna kolumna, po której dane mają być posortowane,
- **`getfields` (string)** - dodatkowe kolumny, które mają być pobrane w zapytaniu (kolumny podajemy wymieniając je po przecinku);
- **`form` (string)** - nazwa powiązanego formularza;
- **`leftjoin, leftjoin1, leftjoin2, leftjoin3 (string)`** - dodatkowe połączenia `LEFT JOIN` z innymi tabelami; Wartość powinna mieć postać: `tabela#warunek`;

Spójrzmy na przykład (**P6**):

```
<views>
 <view name="KontaktyView" caption="Lista kontaktów">
   <field name="imie" caption="Imię" type="varchar"
          width="110" align="left" search="true" />
   <field name="nazwisko" caption="Nazwisko" type="varchar"
          width="120" align="center" search="true" />
   <field name="wiek" caption="Wiek" type="int" width="100"
          align="left" search="true" />


   <field name="stanowisko" caption="Stanowisko" type="int"
          mtype="select" align="center" width="100">
    <option id="0">Nieznane</option>
    <option id="1">Dyrektor</option>
    <option id="2">Prezes</option>
   </field>
   <query table="mc_kontakty" form="KontaktyForm" />
 </view>
</views>
```

W przykładzie utworzony został widok o nazwie (`name`) *KontaktyView* prezentujący dane pochodzące z tabeli (`table`) *mc_kontakty*. Ponieważ w znaczniku `<query>` nie podano atrybutów `where`, `orderby` itp. - pobrane zostaną wszystkie wiersze, a z nich tylko te kolumny które zostały opisane znacznikami `<field>`. Dane będą domyślne posortowane po kolumnie `id`.

Wyświetlone zostaną kolumny: *imie*, *nazwisko*, *wiek* oraz *stanowisko*. Kolumna *stanowisko* jest typu `int`, a zatem zawiera jako wartości liczby całkowite. Jako atrybut `mtype` nadana została mu wartość `select`, co oznacza, że dane zapisane w tej kolumnie mają być interpretowane jako dane typu „wybór". W tym wariancie, w obrębie znacznika `<field>` opisano możliwe wartości znacznikiem `<option>`. Kiedy zatem w wyświetlanym rekordzie danych *stanowisko* przyjmie wartość 1 - wyświetli się w tej kolumnie napis *Dyrektor*. Gdy przyjmie ono wartość 2 - wyświetli się napis *Prezes* itd.

Dla znacznika `<option>` można również nadać atrybut `table` wskazujący inną tabelę w bazie danych oraz atrybut `id` wskazujący kolumnę, która ma być kluczem. Wówczas system BSX pobierze rekordy z tej tabeli i potraktuje je jako możliwe wartości dla danego pola. Oprócz atrybutu `table` można podać `orderby`, `groupby` oraz `where` - ograniczając pobierane rekordy oraz je sortując. Jako tekst objęty znacznikiem `<option>` podajemy szablon pobieranych danych. W ten sposób można tworzyć relacje pomiędzy tabelami.
Przykład: `<option table="tabela" id="id">{nazwa}</option>`

Poprzez atrybut `form` znacznika `<query>` definiuje się formularz, który ma być powiązany z danym widokiem. Formularz ten będzie wywoływany w momencie próby dodania nowego rekordu do danej tabli, lub podczas próby edycji rekordu istniejącego.

Kolumna niepowiązana z bazą danych

Jeśli chcemy umieścić w widoku kolumnę, która nie ma swojego odpowiednika w bazie danych, wówczas jej nazwę należy poprzedzić symbolem `@`, np.:

```
<field name="@kolumna" caption="Kolumna" width="200" />
```

Kolumny tego typu mogą być przydatne, gdy pod dany widok podepniemy skrypt i z jego poziomu będziemy modyfikować zawartość owej kolumny.

Szablony zawartości kolumny

Czasami zachodzi potrzeba wyświetlenia w jednej kolumnie widoku zawartości kilku kolumn z bazy danych. Aby tego dokonać możemy skorzystać z mechanizmu tzw. szablonów. Polega on na tym, że do opisu danej kolumny dodajemy atrybut `template`, a w nim szablon zawartości kolumny. W szablonie tym możemy odwoływać się do innych kolumn pobieranych w zapytaniu. Aby tego dokonać umieszczamy ich nazwy w nawiasach klamrowych. Szablony pozwalają również na używanie prostych znaczników HTML, tj. `<b>`, `<i>` oraz `<u>`, przy czym symbole < i > zapisujemy jako `[[` oraz `]]`. Szablon pozwala także na użycie symbolu `|`, który oznacza „nowa linia". Przykład:

```
<field name="pname" caption="Nazwa kontrahenta" template="[[b]]{pname}[[/
b]]|{pstreet} {pstreet_n1}, {ppostcode} {ppost}" width="380" type="varchar"
search="true" />
```

Powyższy przykład spowoduje wyświetlenie kolumny zatytułowanej „*Nazwa kontrahenta*", w której wyświetli się imię i nazwisko kontrahenta oraz jego adres (w drugiej linijce). Próba posortowania po tej kolumnie będzie skutkowała sortowaniem po kolumnie `pname`. Podobnie wyszukiwanie będzie działało tylko po tej kolumnie.

Formularze

Formularze to okna składające się z różnego rodzaju kontrolek (np. pól edycyjnych, wielowierszowych pól tekstowych, kontrolek wprowadzania daty, godziny, wstawiania obrazków itp.) - pozwalające na dodawanie i modyfikowanie danych w tabelach bazy danych. Można również tworzyć formularze niepowiązane z tabelami w bazie danych i służące do innych celów, potrzebnych programiście.

Wszystkie formularze definiuje się w obrębie znaczników `<forms>...</forms>`. Każdy formularz opisuje się znacznikiem `<form>`.

Znaczniki:

- **`<forms>`**...**`</forms>`** - sekcja opisująca formularze;
- **`<form>`**...**`</form>`** - opis pojedynczego formularza;

  Atrybuty:

  - **`name`** (**string**) - nazwa formularza;
  - **`caption`** (**string**) - tytuł formularza;
  - **`width`** (**int**) - domyślna szerokość okna formularza,
  - **`height`** (**int**) - domyślna wysokość okna formularza,
  - **`table`** (**string**) - powiązana tabela danych w bazie danych,
  - **`image`** (**string**) - powiązana ikona dla formularza,
  - **`btnokvisible`** (**bool**) - czy ma być widoczny przycisk *OK*,
  - **`btnsavevisible`** (**bool**) - czy ma być widoczny przycisk *Zapisz*,
  - **`btncancelvisible`** (**bool**) - czy ma być widoczny przycisk *Anuluj*,
  - **`btnprintvisible`** (**bool**) - czy ma być widoczny przycisk *Drukuj*,

    - **buffers** (**bool**) - czy okno ma być buforowane;

- **<field>** - ukryte pole z danymi;
- **<edit>** - pole tekstowe,
- **<editbtn>** - pole tekstowe z przyciskiem;
- **<memo>** - wielowierszowe pole tekstowe,
- **<combobox>** - pole wyboru,
- **<comboedit>** - pole wyboru z możliwością wprowadzenia wartości,
- **<checkbox>** - pole typu *checkbox*,
- **<calendaredit>** - pole z datą,
- **<dateedit>** - pole z datą;
- **<layout>** - „prostokąt" na dane,
- **<groupbox>** - pole grupujące inne kontrolki;
- **<tabs>** - panel z zakładkami,
- **<tab>** - pojedyncza zakładka,
- **<shape>** - obiekt graficzny (np. pozioma linia, prostokąt itp.);
- **<hint>** - pole podpowiedzi;
- **<button>** - przycisk;
- …

Istnieje kilkadziesiąt znaczników, z których możemy skorzystać w sekcji `<form>` definiując tym samym różnego rodzaju kontrolki. Powyżej wymieniono tylko niektóre z nich.

Większość znaczników posiada atrybuty:

- **name** (**string**) - opisujący nazwę danej kontrolki,

- **`caption` (string)** - opisujący tytuł kontrolki,
- **`width` (int)** - szerokość kontrolki,
- **`height` (int)** - wysokość kontrolki,
- **`text` (string)** - zawartość kontrolki,
- **`left` (int)** - pozycja „od lewej”,
- **`top` (int)** - pozycja „od góry”,
- **`align` (string)** - sposób automatycznego „przyciągania” lub „rozciągania” kontrolki (możliwe wartości: `left`, `top`, `right`, `bottom`, `client`),
- **`margins` (int)** - marginesy;

Jeśli definiujemy kolejne kontrolki nie określając ich położenia (`left`, `top`) ani „przyciągania” - wówczas automatycznie umieszczane są one jedna pod drugą. Zapis zatem:

```
<edit name="imie" caption="Imię" />
<edit name="nazwisko" caption="Nazwisko" />
```

spowoduje wyświetlenie dwóch kontrolek - pól tekstowych zatytułowanych *Imię* i *Nazwisko* - jedno pod drugim.

Często zachodzi potrzeba wyświetlenia dwóch kontrolek obok siebie. Wówczas jako wartość atrybutu `top` można wprowadzić liczbę „-1”. Spowoduje to, że dane pole automatycznie przesunie się do góry na wysokość pola poprzedniego, oraz przesunie się w prawo - aby znajdowało się obok poprzedniego pola. Można zatem zapisać to tak:

```
<edit name="imie" caption="Imię" />
<edit name="nazwisko" caption="Nazwisko" top="-1" />
```

Jeśli dany formularz ma powiązaną tabelę w bazie danych (`table`) - podczas jego wyświetlania domyślnie pokażą się przyciski *OK*, *Zapisz* oraz *Anuluj*. W tym wariancie, kliknięcie przycisku *Zapisz* spowoduje automatycznie zapisanie rekordu w bazie danych. Wypełniane będą te pola w powiązanej tabli bazy danych, dla których istnieją kontrolki o tych samych nazwach. Jeśli zatem w tabeli mamy kolumny *imie*, *nazwisko*, *wiek*, to jeśli na formularzu znajdą się kontrolki o takich samych nazwach, zostaną one powiązane z tymi kolumnami.

Na formularzu może istnieć znacznie więcej kontrolek niż pól w bazie danych. Wówczas nie będą one po prostu powiązane z danymi w bazie. Mogą być wykorzystywane do innych celów.

Atrybut `name` nie jest obowiązkowy. Szczególnie w przypadku kontrolek, które służą tylko celom „wizualnym”. Na przykład, kontrolki `<tabs>`, `<layout>`, `<tab>` - wszystkie one służą celom jedynie wizualnym. Stąd nie ma potrzeby nadawania im nazw. System BSX przydzieli te nazwy automatycznie. Jeśli jednak będziemy chcieli się odwoływać do tych kontrolek z poziomu powiązanego kodu Pascala, wówczas nazwy muszą być im nadane.

Jeśli powiązana tabela ma zdefiniowane pola: `add_id_user`, `add_time`, `modyf_id_user`, `modyf_time` - wówczas będą one automatycznie uzupełniane przez system BSX w momencie zapisywania danych.

Spójrzmy na przykład formularza (**P7**):

```
<forms>

  <form name="KontaktyForm" table=„mc_kontakty" caption="Kontakt"
    width="500" height="400">

   <edit name="imie" caption="Imię" width="100" />

   <edit name="nazwisko" caption="Nazwisko" width="200" top="-1" />

   <edit name="wiek" caption="Wiek" width="80"
         textalign="center" mtype="int" />

   <combobox name="stanowisko" caption="Stanowisko" width="150">

    <option id="0">Nieznane</option>
```

```
    <option id="1">Dyrektor</option>
    <option id="2">Prezes</option>
   </combobox>
  </form>
</forms>
```

Niektóre kontrolki służą do grupowania innych. Najważniejsze z nich to:

- **<layout>** - niewidoczny prostokąt;
- **<panel>** - widoczny prostokąt;
- **<groupbox>** - pole (ramka) grupujące inne opcje;
- **<tabs>** - panel z zakładkami; wewnątrz tego znacznika można użyć tylko znacznika `<tab>` - zakładka; dopiero w nim możemy stosować już dowolne inne znaczniki.

Chcąc zbudować interfejs użytkownika należy odpowiednio użyć powyższych kontrolek - zagnieżdżając jedne w drugich.

Wszystkie kontrolki posiadają atrybut `align`, który określa sposób wyświetlania danej kontrolki. Jest to szczególnie istotne w przypadku kontrolek grupujących (wymienionych wyżej). Możliwe wartości dla `align` to:

- `left` - dany obiekt przyciągnięty jest do lewej krawędzi; ma zatem zawsze wysokość „maksymalną" obiektu w którym się znajduje. My powinniśmy określić jedynie szerokość (`width`);
- `right` - analogicznie - do prawej strony;
- `top` - przyciągnięcie „do góry". Obiekt ma zatem szerokość kontrolki, w której się znajduje, a my powinniśmy ustawić jedynie wysokość poprzez atrybut `height`. Możemy także zamiast atrybutu `height` użyć `autoheight="true"`, a wówczas BSX dobierze wysokość tak by wszystkie wewnętrzne kontrolki się zmieściły;

• `bottom` - analogicznie - przyciągnięcie do dołu;

• `client` - „wypełnienie” wolnego obszaru.

Budowanie interfejsu z kontrolek używających atrybut `align` jest zalecane, gdyż wówczas okna dostosowują swoją zawartość do rozmiaru.

Chcąc uzyskać „odstępy” pomiędzy kontrolkami - powinniśmy używać atrybutu `margins`, np. `margins="5"` - oznacza marginesy z każdej strony po 5 px. Jeśli chcemy określić jeden margines, wówczas używamy: `marginleft`, `margintop`, `marginbottom` lub `marginright`.

Menu

Oprogramowanie MP posiada menu, zapewniające dostęp do wielu różnych funkcji np. tworzenie bazy danych, tworzenie kopii bezpieczeństwa, dostęp do określonych modułów, pomoc itd. Mówimy tutaj o menu głównym aplikacji. Użytkownik ma możliwość tworzenia własnych wpisów w owym menu.

Oprócz menu głównego istnieją menu kontekstowe wyświetlane w różnych miejscach programu. Na przykład menu pod prawym przyciskiem myszy po kliknięciu w tabelę w jakimś widoku lub w jakąś kontrolkę w formularzu.

Definicję elementów menu dokonuje się w sekcji `<menu>...</menu>`. W obrębie tej sekcji, poszczególne elementy definiuje się znacznikiem `<item>`.

Znaczniki `<item>` można w sobie dowolnie zagnieżdżać, tzn. w obrębie danego znacznika `<item>` można ponownie tworzyć kolejne elementy tego typu. W ten sposób tworzymy hierarchiczną strukturę menu.

Znacznik `<item>` może przyjąć następujące atrybuty:

- **`name` (string)** - nazwa elementu menu,
- **`caption` (string)** - tytuł elementu menu,
- **`onclick` (string)** - powiązana instrukcja, jaka ma zostać wykonana po wybraniu danego elementu menu;

Atrybut `name` nie jest obowiązkowy. Jeśli jednak zostanie podany wówczas system BSX sprawdzi, czy istnieje już jakiś element o tej nazwie. Jeśli tak, nie utworzy go, lecz „podepnie" się pod niego. Jeśli taki element nie istnieje, wówczas zostanie utworzony. Dzięki temu mechanizmowi możemy zmodyfikować opis i zasadę działania dowolnego istniejącego elementu menu, dodawać własne opcje do istniejących elementów, czy też tworzyć własne.

Menu główne programu nosi nazwę `MainMenu`. W jego obrębie istnieją zawsze elementy o predefiniowanych nazwach:

- **mnStart** - menu *Start*,
- **mnModules** - menu *Moduły*,
- **mnDictionaries** - menu *Słowniki*,
- **mnManagment** - menu *Zarządzanie*,
- **mnTools** - menu *Narzędzia*,
- **mnOptions** - menu *Opcje*,
- **mnView** - menu *Widok*,
- **mnHelp** - menu *Pomoc*;

Dzięki powyższym nazwom możemy „wpiąć" się w istniejące elementy. Oto przykład (**P8**).

```
<menu>
 <item name="MainMenu">
   <item caption="Wtyczka">
     <item caption="Lista kontaktów A"
           onclick="showview:Plugin.MyCompany.Wtyczka.KontaktyView" />
     <item caption="Lista kontaktów B"
           onclick="showhomeview:Plugin.MyCompany.Wtyczka.KontaktyView" />
   </item>
 </item>
</menu>
```

Powyższy przykład spowoduje wyświetlenie w menu głównym nowej pozycji *Wtyczka*, a w niej dwóch elementów: *Lista kontaktów A* i *Lista kontaktów B*. Kliknięcie w te elementy spowoduje wyświetlenie widoku *Plugin.MyCompany.Wtyczka.KontaktyView*. Pierwszy z przycisków wyświetli ten widok jako odrębne okno, natomiast drugi z elementów menu wyświetli widok w obszarze roboczym progra-

mu. Różnica wynika z komendy: `showview` i `showhomeview`. Dla obu tych komend przekazujemy parametr - nazwę widoku - po znaku „:”. Jako nazwę widoku należy zawsze podawać pełną jego ścieżkę.

Panel modułów

Po uruchomieniu programu MP i otwarciu dowolnej bazy danych, po lewej stronie widoczny będzie tzw. panel modułów. Zapewnia on dostęp do wszystkich modułów jakie oferuje program. Z poziomu naszych wtyczek również mamy możliwość tworzenia tam własnych kategorii, a w nich własnych przycisków. Dokonuje się tego w sekcji `<categories>...</categories>`.

W obrębie tej sekcji tworzymy tzw. kategorie. Opisujemy je znacznikiem `<category>`. W obrębie każdej kategorii tworzymy elementy znacznikiem `<item>`.

Znaczniki:

- **`<categories>`...`</categories>`** - sekcja opisująca kategorie;
- **`<category>`...`</category>`** - opis pojedynczej kategorii;

  Atrybuty:

  - **`caption`** (**string**) - tytuł kategorii;
- **`<item>`** - element kategorii;

  Atrybuty:

  - **`caption`** (**string**) - tytuł elementu,
  - **`image`** (**string**) - powiązana ikona,
  - **`onclick`** (**string**) - instrukcja, która ma zostać wykonana po kliknięciu w ten element;

Spójrzmy na przykład (**P9**).

```
<categories>
  <category caption="Wtyczka">
     <item caption="Lista kontaktów A" image="db"
           onclick="showview:Plugin.MyCompany.Wtyczka.KontaktyView" />
```

```
      <item caption="Lista kontaktów B" image="db"
            onclick="showhomeview:Plugin.MyCompany.Wtyczka.KontaktyView" />

  </category>

</categories>
```

Przykładowa wtyczka

W poprzednich podpunktach opisane zostały wszystkie podstawowe elementy, z jakich buduje się wtyczki w systemie BSX. Po zebraniu wszystkich zaprezentowanych przykładów powstała prosta wtyczka. Zawiera ona elementy menu oraz własną kategorię w bocznym menu kategorii. Pozwala na wyświetlenie widoku listy *Kontaktów* - na dwa sposoby. Jako odrębne okno lub wewnątrz obszaru roboczego. Umożliwia dodawanie własnych kontaktów oraz ich edycję i usuwanie.

Oto kod źródłowy owej wtyczki (**P10**):

wtyczka.xml

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka" >


 <tables>

  <table name="mc_kontakty" caption="Kontakty">

    <field name="add_id_user" type="int" mtype="add_idusers"
           createindex="true" />

    <field name="add_time" type="datetime" mtype="add_time" />

    <field name="modyf_id_user" type="int" mtype="modyf_idusers"
           createindex="true" />

    <field name="modyf_time" type="datetime" mtype="modyf_time" />


    <field name="imie" type="varchar" size="50" />

    <field name="nazwisko" type="varchar" size="100" />

    <field name="wiek" type="int" default="0" />

    <field name="stanowisko" type="int" default="0" />

  </table>

 </tables>


 <views>

  <view name="KontaktyView" caption="Lista kontaktów">
```

```
    <field name="imie" caption="Imię" type="varchar" width="110"
           align="left" search="true" />

    <field name="nazwisko" caption="Nazwisko" type="varchar" width="120"
           align="center" search="true" />

    <field name="wiek" caption="Wiek" type="int" width="100" align="left"
           search="true" />



    <field name="stanowisko" caption="Stanowisko" type="int"
           mtype="select" align="center" width="100">

     <option id="0">Nieznane</option>

     <option id="1">Dyrektor</option>

     <option id="2">Prezes</option>

    </field>



    <query table="mc_kontakty" form="KontaktyForm" />


  </view>

</views>



<forms>

   <form name="KontaktyForm" table="mc_kontakty" caption="Kontakt"
         width="500" height="400">

    <edit name="imie" caption="Imię" width="100" />

    <edit name="nazwisko" caption="Nazwisko" width="200" top="-1" />

    <edit name="wiek" caption="Wiek" width="80" textalign="center"
          mtype="int" />

    <combobox name="stanowisko" caption="Stanowisko" width="150">

     <option id="0">Nieznane</option>

     <option id="1">Dyrektor</option>

     <option id="2">Prezes</option>

    </combobox>

   </form>

</forms>



<menu>

  <item name="MainMenu">

    <item caption="Wtyczka">
```

```
      <item caption="Lista kontaktów A"
            onclick="showview:Plugin.MyCompany.Wtyczka.KontaktyView" />

      <item caption="Lista kontaktów B"
            onclick="showhomeview:Plugin.MyCompany.Wtyczka.KontaktyView" />

    </item>

  </item>

 </menu>


 <categories>

   <category caption="Wtyczka">

      <item caption="Lista kontaktów A" image="db"
            onclick="showview:Plugin.MyCompany.Wtyczka.KontaktyView" />

      <item caption="Lista kontaktów B" image="db"
            onclick="showhomeview:Plugin.MyCompany.Wtyczka.KontaktyView" />

   </category>

 </categories>


</plugin>
```

Jeśli folder z daną wtyczką spakujemy programem ZIP - powstanie *paczka*, którą następnie możemy przekazać innej osobie posiadającej oprogramowanie MP. Aby taką *paczkę* zainstalować, wystarczy wybrać z menu *Start* opcję *Uruchom lub zainstaluj skrypt* i wskazać owy plik ZIP. Oprogramowanie MP automatycznie rozpakuje wtyczkę do folderu **plugins**.

Zainstalowanie rozszerzenia w opisany wyżej sposób jest metodą najprostszą. Posiada jednak szereg wad np. wtyczka nie zostanie zainicjowana do czasu ponownego uruchomienia programu. Istnieje bardziej zaawansowana metoda tworzenia paczek z wtyczkami, pozwalająca na ich automatyczną inicjalizację oraz wykonanie dowolnych czynności startowych. Zostanie to opisane w odrębnych rozdziałach tego podręcznika.

Przekazując ową wtyczkę innej osobie lub firmie, uzyska ona tym samym jej kod źródłowy. Nawiązując głębszą współpracę z firmą BinSoft, możemy uzyskać specjalne narzędzie - kompilator - którym będziemy mogli zabezpieczyć nasz kod źródłowy. Aby uzyskać więcej informacji w tym zakresie zapraszamy do kontaktu.

Język PascalBSX

Korzystając z plików XML możemy opisywać strukturę danych w bazie danych, widoki tabel, formularze, elementy menu i panelu kategorii. Aby dodać logikę do naszych wtyczek lub rozszerzeń BSX, należy posłużyć się językiem programowania PascalBSX.

PascalBSX to pewna odmiana klasycznego języka programowania Pascal, dostosowana do potrzeb systemu BSX. Składnia tego języka jest identyczna jak Object Pascal. Zmianie uległ jedynie sposób interpretowania typów danych.

Dla każdego tworzenia przez nas formularza oraz dla każdego widoku, możemy przygotować odpowiedni plik z kodem PascalBSX i w ten sposób dodać logikę do danego elementu. Możemy na przykład:

- wykonać określone czynności w momencie otwierania formularza, jego zamykania, drukowania itp.;
- możemy przechwytywać moment wyboru różnych elementów formularza, zmiany zawartości pól, wprowadzania danych z klawiatury itp;
- możemy przechwytywać moment wyświetlania danych w widoku, modyfikacji ich itp.;
- i wiele więcej.

Przegląd elementów Pascala

W tym podrozdziale przedstawiony został bardzo krótki przegląd elementów Pascala. Opisana została składnia instrukcji iteracyjnych, warunków, klas, procedur i funkcji itp. Założeniem jest, że osoba czytająca zna jakiś z języków programowania i przedstawione informacje omówią jedynie składnię danego elementu w dialekcie PascalBSX. Nie będą one omawiane i wyjaśniane szczegółowo, gdyż wykraczałoby to poza zakres niniejszego podręcznika.

Ogólna budowa

Każdy skrypt w języku PascalBSX powinien być zapisany jako czysty tekst (bez formatowania) w pliku z rozszerzeniem PAS. Kodowaniem znaków takiego pliku powinno być UTF8.

Skrypt analizowany jest od góry do dołu i składa się z instrukcji oraz słów kluczowych. Jeśli wystąpi w nim sekcja:

```
begin
  //ciąg instrukcji
end;
```

zostanie ona automatycznie wykonana, w momencie załadowania tego pliku. Sekcja ta nie jest jednak konieczna.

Słowa kluczowe

Słowa kluczowe to słowa mające swoje określone znacznie w składni danego języka programowania. Na przykład słowo `begin` rozpoczyna instrukcję złożoną, `end` - kończy instrukcję złożoną, `for` - rozpoczyna pętlę itd. Takich słów należy używać w ściśle określony sposób - w taki w jaki przewiduje ich składnia.

W przykładach prezentowanych w tym podręczniku, słowa kluczowe oznaczone zostały czcionką pogrubioną.

Instrukcje i wyrażenia

Instrukcje są to identyfikatory jakiś procedur lub funkcji, które mają zostać wykonane lub obliczone. Każda instrukcja powinna być zakończona znakiem średnika „`;`”.

Wyrażenia to ciągi instrukcji połączone operatorami. Dostępne są następujące operatory arytmetyczne:

- `+` - dodawanie (również łączenie ciągów znaków),
- `−` - odejmowanie,
- `*` - mnożenie,
- `/` - dzielenie (dotyczy tylko liczb zmiennoprzecinkowych),
- `mod` - reszta z dzielenia (dotyczy tylko liczb całkowitych),
- `div` - dzielenie całkowite.

Dostępne są także operatory logiczne:

- = - czy równe,
- <> - czy różne,
- < - czy mniejsze,
- > - czy większe,
- <= - czy mniejsze bądź równe,
- >= - czy większe bądź równe;

Wynikiem wyrażeń połączonych operatorami logicznymi jest wartość logiczna. Może ona przyjąć wartość `true` („prawda”) bądź `false` („fałsz”). Jeśli wartość wyrażenie przyjmie wartość logiczną `true`, mówimy że „jest spełnione”.

Wyrażenia logiczne możemy łączyć operatorami:

- `and` - iloczyn logiczny,

- `or` - suma logiczna,
- `not` - negacja logiczna;

Komentarze

Komentarze można wstawiać na trzy różne sposoby:

- komentarze jednolinijkowe rozpoczynają się dwuznakiem `//` i obowiązują do końca danej linii;
- komentarze wielolinijkowe objęte znakami `{` i `}`;
- komentarze wielolinijkowe objęte dwuznakami: `(*` i `*)`;

Instrukcje złożone

Instrukcja złożona do dowolny ciąg znaków objęty słowami kluczowymi: `begin` oraz `end`. Wszędzie gdzie składnia przewiduje użycie pojedynczej instrukcji, można w jej miejscu użyć instrukcji złożonej.

Instrukcje warunkowe

Pascal przewiduje dwie instrukcje warunkowe.

Instrukcja if

Dostępne są dwa warianty:

```
if warunek then instrukcja1;
if warunek then instrukcja1 else instrukcja2;
```

Jako `warunek` należy umieścić wyrażenie, które przyjmie wartość logiczną. Jeśli warunek jest spełniony (wyrażenie przyjmie wartość `true` - „prawda”) wykonana zostanie `instrukcja1`. W przeciwnym wypadku wykona się `instrukcja2` (jeśli jest podana).

Chcąc wykonać więcej instrukcji przy spełnieniu lub niespełnieniu warunku, należy skorzystać z instrukcji złożonej. Wówczas składnia może przyjąć postać:

```
if warunek then
begin
```

```
   //instrukcja1;
   //instrukcja2;
   //…
end;
```

Instrukcja case

Składnia:

```
case wyrażenie of
wartość1: instrukcja1;
wartość2: instrukcja2;
…
else instrukcja3;
end;
```

Obliczana jest wartość `wyrażenia`, a następnie wykonana zostanie instrukcja z etykietą o danej wartość. Wyrażenie musi przyjąć wartość całkowitą. Jeśli na liście wartości nie znajdzie się otrzymana wartość, wykona się instrukcja znajdująca się po słowie kluczowym `else` (jeśli ją podano).

Instrukcje iteracyjne

Dostępne są trzy formy instrukcji iteracyjnych (pętli):

Pętla for

Oto jej składnia:

```
for licznik:=wartość_początkowa to wartość_końcowa do instrukcja;
for licznik:=wartość_końcowa downto wartość_początkowa do instrukcja;
```

Zmienna `licznik` musi być zmienną całkowitoliczbową i powinna być wcześniej zadeklarowana.

Pętla while

Oto składnia:

```
while warunek do instrukcja;
```

Instrukcja wykonywana jest tak długo, dopóki warunek jest spełniony (przyjmuje wartość logiczną `true` - „prawda”).

Pętla repeat

Oto składnia:

```
repeat
  instrukcja1;
  instrukcja2;
  …
until warunek;
```

Ciąg instrukcji znajdujący się pomiędzy słowami kluczowymi `repeat` oraz `until` wykonywany jest tak długo, aż `warunek` zostanie spełniony.

Przerywanie pętli

Podczas korzystania z pętli przydane są dwa specjalne słowa kluczowe wpływające na ich przebieg:

- `break` - powoduje przerwanie wykonywania pętli;
- `continue` - powoduje skok do warunku danej pętli;

Procedury i funkcje

Procedury i funkcje to podprogramy opatrzone własną nazwą (identyfikatorem), z których możemy korzystać wielokrotnie. Funkcja różni się od procedury tym, że może zwrócić jakąś wartość. Oto ich składnie.

Składnia procedury:

```
procedure nazwa;
begin
  //ciąg instrukcji
end;
```

Składnia funkcji:

```
function nazwa:typ_zwracanej_wartości;
begin
  //ciąg instrukcji
  Result:=wartość;
end;
```

Funkcje w przeciwieństwie do procedur, posiadają „wewnętrzną zmienną” `Result`, poprzez którą określamy zwracaną wartość.

Jeśli procedura lub funkcja powinna przyjmować jakieś argumenty, wymieniamy je w nawiasach okrągłych po nazwie danego podprogramu. Oto przykład:

```
procedure sprawdz(a,b,c);
begin
  //ciąg instrukcji;
end;


function wieksza(a,b):Boolean;
begin
  Result:=a>b;
end;
```

Chcąc przerwać wykonywanie procedury lub funkcji, należy skorzystać z instrukcji `exit;`

Zmienne

Przed użyciem zmiennej powinna być ona zawsze zadeklarowana. Deklaracja zmiennej polega na wymienieniu jej nazwy po słowie kluczowym `var`. Deklarując kilka zmiennych wystarczy raz użyć słowo kluczowe `var`, a następnie kolejne nazwy zmiennych wymienić po przecinku.

Zmienne można zadeklarować na samym początku pliku z programem - wówczas są to zmienne globalne, widoczne w obrębie całego pliku, lub przed słowem kluczowym `begin` rozpoczynającym procedurę lub funkcję. Wówczas będą to zmienne lokalne widoczne tylko w obrębie danego podprogramu.

Przykład:

```
var imie, nazwisko; //zmienne globalne
function oblicz(x,y) : Integer;
var a,b; //zmienne lokalne
begin
```

```
  a:=x+y;
  b:=x-y;
  Result:=a*b;
end;
```

Typy danych

Każda zmienna w programie posiada określony typ. Wpływa on na sposób traktowania tej zmiennej, a więc zarządzania jej pamięcią, sposobem działania operatorów z nią związanych itp. Dostępne są następujące podstawowe typy danych:

- `integer` - liczby całkowite,
- `double` - liczby zmiennoprzecinkowe,
- `string` - ciągi tekstowe,
- `boolean` - wartości logiczne;
- `TDateTime` - data i czas;

Interpreter PascalBSX automatycznie określa typ danej zmiennej w momencie przypisania jej pierwszej wartości. Jeśli zatem do zmiennej przypisana zostanie liczba całkowita, przyjmie ona typ `Integer`. Jeśli zmiennej przypisany zostanie ciąg tekstowy (objęty apostrofami) - przyjmie ona typ `String`.

Operatorem przypisania w języku Pascal jest dwuznak: `:=`

Podczas deklaracji zmiennych można podać typ danych jaki one powinny posiadać. Możliwość ta została dodana po to, aby osoby programujące od lat w języku Pascal nie musiały zmieniać swoich przyzwyczajeń. Z punktu widzenia interpretera PascalBSX, typ ten i tak jest ignorowany.

Oto kilka przydanych funkcji, związanych z konwersją typów danych:

- `I2S(x)` - funkcja zamienia liczbę całkowitą na ciąg tekstowy;

- `S2I(s)` - funkcja zamienia ciąg tekstowy na liczbę całkowitą;
- `CorrectD2S(x)` - konwersja „ceny” (liczby `double`) do ciągu tekstowego;
- `CorrectS2D(s)` - konwersja ciągu tekstowego do „ceny” (liczby `double`);
- `CorrectS2S(s)` - konwersja ciągu tekstowego do ciągu tekstowego (zawierającego „cenę”);
- `StrToIntDef(S,Def)` - konwertuje ciąg `S` do liczby całkowitej; jeśli konwersja się nie powiedzie, metoda zwróci wartość domyślną `Def`;
- `StrToDateDef(S,Def)` - konwertuje ciąg `S` do daty `TDateTime`; jeśli konwersja się nie powiedzie, zwróci wartość `Def`;
- `StrToDateTimeDef(S,Def)` - konwertuje ciąg `S` do daty i czasu `TDateTime`; jeśli konwersja się nie powiedzie, zwróci wartość `Def`;
- `DateToStr(D)` - konwertuje datę `D` do formatu `string`;
- `DateTimeToStr(D)` - konwertuję datę i czas `D` do formatu `string`;
- `TimeToStr(D)` - konwertuje czas do formatu `string`;

Funkcje z serii `CorrectX2X()` służą do korygowania liczb zmiennoprzecinkowych na potrzeby wartości walutowych (cen). Dokonują one konwersji z taką dokładnością z jaką wymaga konfiguracja programu. Uwzględniają także poprawne metody zaokrąglania kwot.

Wyjątki

Jeśli podczas przetwarzania kodu wystąpi błąd generowany jest automatycznie tzw. *wyjątek* i przetwarzanie danej procedury zostaje przerwane. Istnieje możliwość przechwycenia takich sytuacji i napisania własnego kodu realizującego

określone czynności. W PascalBSX możemy przechwytywać wyjątki na dwa sposoby.

try… finally end

Czasami kiedy wystąpił błąd w pewnym obszarze programu, nie możemy pozwolić na jego proste zakończenie. Często są takie sytuacje, że pomimo błędu powinniśmy wykonać pewne dodatkowe czynności, na przykład, powinniśmy zwolnić pamięć z obiektów, zamknąć uchwyty do plików itp. W tym celu stosujemy składnię `try … finally … end`. Oto przykład:

```
L:=TStringList.Create;
try
  …
finally
   L.Free;
end;
```

W przykładzie tworzymy obiekt klasy `TStringList` i zapamiętujemy wskaźnik do niego w zmiennej `L`. Tworzenie obiektu wiąże się z przydzieleniem mu określonej pamięci. Każdy obiekt, który tworzymy samodzielnie, powinniśmy również samodzielnie zwolnić. Robimy to wywołując metodę `Free`. Gdyby jednak wystąpił błąd podczas wykonywania kodu przed zwolnieniem pamięci, wygenerowany zostałby wyjątek i przetwarzanie danej procedury zostałoby przerwane. Wywołanie `Free` nigdy by nie nastąpiło! W przykładzie pokazano jak należy sobie z tym radzić. Kod wykonany w bloku `finally…end` zostanie wykonany zawsze. Jeśli w obszarze `try…finally` wystąpi wyjątek, zanim dana procedura zostanie przerwana, zawsze wykona się kod z bloku `finally…end`.

try … except … end;

Ten sposób pozwala na przechwytywanie wyjątków i samodzielną ich obsługę. Jeśli w kodzie objętym słowami `try … except` wystąpi błąd generujący wyjątek, zostanie on przechwycony przez kod znajdujący się pomiędzy `except`

... `end`. Możemy tam napisać własny kod, np. wyświetlający odpowiedni komunikat. Przechwycenie wyjątku tą metodą nie przerwie zatem wykonywania danej procedury. Spójrzmy na przykład:

```
try
   …
except
  APP.ShowMessageError('Wystąpił błąd: '+LastExceptionMessage);
end;
```

Różnice pomiędzy except i finnaly

- Kod pomiędzy `finally ... end` - wykona się zawsze! Zarówno jeśli wystąpi wyjątek (błąd), jak i gdy błędu nie będzie; Gdy wyjątek wystąpi, kod ten zostanie wykonany, a następnie wyjątek nadal zostanie *przekazany*, co spowoduje przerwanie wykonania danej procedury;
- Kod pomiędzy `except ... end` - wykona się tylko wówczas, gdy wystąpi wyjątek (błąd); Wyjątek zostanie przetworzony, zatem nie będzie *przekazywany* dalej, a procedura będzie nadal przetwarzana;

Wczytywanie skryptów Pascala

W obrębie definicji widoków (znacznik `<view>`) jak i formularzy (znacznik `<form>`) istnieje możliwość określenia, jaki skrypt Pascala ma zostać załadowany dla danego obiektu (widoku lub formularza). Możemy zatem przygotować odpowiedni skrypt dla widoku, który wyświetlamy - lub dla naszego formularza. Z poziomu tego skryptu będziemy mogli odwoływać się do dowolnej kontrolki, pobierać jej wartość, zmieniać ją, wychwytywać różne zdarzenia itp.

Aby zdefiniować skrypt, który ma zostać załadowany należy użyć znacznika `<script>`. Poprzez atrybut `src` podajemy nazwę pliku z naszym skryptem. Oto przykład:

```
<form name="formularz" … >
  …
   <script src="plik.pas" />
</form>
```

Kiedy formularz ten zostanie wyświetlony, automatycznie załadowany do niego zostanie skrypt `plik.pas`.

Oto kilka przydatnych funkcji Pascala, które pozwalają na odwoływanie się z jego poziomu, do elementów powiązanego formularza:

- `GetValue(nazwa, domyślne)` - pobranie wartości z kontrolki o podanej `nazwie`. Jeśli BSX nie odnajdzie danej kontrolki lub nie ma ona zdefiniowanej wartości, wówczas funkcja zwróci wartość `domyślną`;
- `SetValue(nazwa, wartość)` - nadanie kontrolce o podanej `nazwie` przekazanej `wartości`;
- `GetValueL(nazwa, domyślne)` - działanie analogiczne do `GetValue()`, tylko zwracana wartość jest typu `Integer`; Parametr `domyślne` również powinien być liczbą całkowitą;

- `SetValueL(nazwa, wartość)` - działanie analogiczne jak `SetValue()`, tylko `wartość` powinno być typu `Integer`;
- `ParentGetValue(nazwa, domyślne)` - działanie analogiczne do `GetValue()`, tylko pobiera wartość z formularza „rodzica”;
- `ParentSetValue(nazwa, wartość)` - działanie analogiczne do `SetValue()`, tylko dotyczy pól z formularza „rodzica”;

Biblioteki

Tworząc własne rozszerzenia często będą się one składały z wielu plików PAS. Z reguły bowiem, każdy formularz czy widok - posiada własny tego typu plik. Często również pewne czynności można wykonywać z poziomu różnych miejsc, np. z poziomu otwartego formularza, jak też z poziomu widoku poprzez menu kontekstowe (pod prawym przyciskiem myszy). Aby w takich sytuacjach nie powielać tego samego kodu w kilku miejscach, warto jest zrobić tzw. bibliotekę. Biblioteka, to dowolny plik pascala, w którym tworzymy różnego rodzaju funkcje i procedury. Następnie taki plik możemy „dołączyć" do innych plików Pascala, w ten sposób zapewniając dostęp do zgromadzonych w tym pliku funkcji.

Biblioteki powinny być zapisywane w plikach rozpoczynających się od słowa `Lib`, np. `LibPlugin.pas`, `LibFunkcje.pas` itp. Oto przykład:

LibFunctions.pas

```
procedure pokazKomunikat;
begin
  APP.ShowMessageInfo('Komunikat z biblioteki!');
end;
```

W pliku, w którym chcemy skorzystać z powyższej biblioteki powinniśmy na samym jego początku użyć dyrektywy `{$uses}`. Przykład:

```
{$uses LibFunctions}
begin
  pokazKomunikat;
end;
```

Podpięcie tego pliku do dowolnego formularza spowoduje automatyczne uruchomienie kodu znajdującego się pomiędzy słowami `begin` i `end`. A ten wywoła procedurę `pokazKomunikat` pochodzącą z biblioteki `LibFunctions`.

System BSX analizując wszystkie pliki Pascala, zapamiętuje wszystkie biblioteki w specjalnej liście. Dzięki temu z bibliotek tych możemy korzystać również z poziomu innych wtyczek (innych folderów). System BSX automatycznie zlokalizuje miejsce danej biblioteki (jej pliku) i ją dołączy.

Na przykład, w module `BinSoft.Invoices` (odpowiedzialny za obsługę dokumentów handlowych) można odnaleźć bibliotekę `LibInvoices`. W module `BinSof.Stock` (odpowiedzialny za dokumenty magazynowe) - znajdziemy `LibStock`, a w module `BinSoft.System` (moduł główny, systemowy) - znajdziemy bibliotekę `LibSystem`. Z każdej z tych bibliotek możemy korzystać również we własnych wtyczkach. Wystarczy bowiem, że wpiszemy np.:

```
{$uses LibSystem, LibInvoices, LibStock}
```

i już możemy korzystać z wszystkich tych trzech rozszerzeń.

Na uwagę zasługuje fakt, że mechanizm ten działa w ten sposób, że jeśli biblioteka nie zostanie odnaleziona, nie będzie żadnego komunikatu o błędzie. Może się bowiem zdarzyć, że np. korzystamy z biblioteki `LibStock` pochodzącej z modułu magazynowego, a użytkownik nie zainstalował tego modułu lub posiada program, który tego moduły nie posiada w standardzie! Gdyby wówczas moduł musiał istnieć, w takich sytuacjach pojawiałby się błąd o braku niezbędnych plików. Z drugiej jednak strony, jeśli nie dołączymy jakiejś biblioteki (bo na przykład system BSX jej nie odnajdzie) - nie możemy korzystać z procedur i funkcji z niej pochodzących. Użycie ich w kodzie spowoduje bowiem pojawienie się komunikatu o błędzie, że dana funkcja nie została rozpoznana (odnaleziona). Dlatego stosując biblioteki, co do których nie mamy pewności że będą „istnieć" - kod programu korzystający z procedur z tych bibliotek powinniśmy objąć specjalnymi dyrektywami: `{$IFMODULE}... {$ENDIF}`. Oto przykład:

```
{$uses LibFunctions}
begin
```

```
  {$IFMODULE LibFunctions}
  pokazKomunikat;
  {$ENDIF}
end;
```

Powyższy kod działa w ten sposób, że przed jego uruchomieniem załadowywana jest do niego biblioteka `LibFunctions` i poprawnie wywołuje się procedura `pokazKomunikat`. Gdyby jednak się okazało, że biblioteka ta nie istnieje, kod objęty dyrektywną `{$IFMODULE}...{$ENDIF}` zostanie usunięty z kodu źródłowego i procedura się nie wywoła, ale nie będzie także żadnego komunikatu o błędzie.

Biblioteki wbudowane

Programy z serii MP/ABC posiadając określone moduły, udostępniają też pewne biblioteki pochodzące z tych modułów. Poniżej zebrano te z nich, które mogą okazać się przydatne dla osób tworzących własne rozszerzenia.

Uwaga! Chcąc lepiej zrozumieć wykorzystanie tych bibliotek należy zapoznać się ze strukturą danych w programach MP/ABC. Opis ten znajduje się w dalszej części podręcznika.

Moduł BinSoft.System - Biblioteka LibSystem

- `systemGetActiveStock(lCurrency : String) : Integer` - zwraca ID domyślnego magazynu dla waluty `lCurrency`;
- `systemUpdateSumDocument(lID : Integer;lTable:String;psMethod:Integer)` - powoduje przeliczenie podsumowania dokumentu; `lID` - id dokumentu, `lTable` - kompatybilna tabela, `psMethod` - metoda obliczeń (0 - od netto, 1 - od brutto);
- `systemAddressCorrect(lID : Integer;lTable:String;lPrefix:String)` - uaktualnia dane

adresowe o brakujące informacje (województwo, powiat itp.); Należy podać kompatybilną tabelę (`lTable`) oraz prefiks pól (np. `p`, `s`, `k` itp.);

Moduł BinSoft.Invoices - Biblioteka LibInvoice

- `invoiceCreateItem(lIDDOC:Integer;lName : String;lQuantity:Integer;lUnit:String;lType,lCalculation:Integer;lPriceN,lPriceB,lPriceV:Double;lVAT:String;lCostB:Double):Integer;` - tworzy pozycję do dokumentu handlowego;
- `invoiceRecalcProfitByDocument(IDDOC : Integer) : Boolean;` - przelicza zysk dla dokumentu;
- `invoiceUpdateSumDocument(lID : Integer;psMetoda:Integer;lWithProducts:Boolean):Double;` - aktualizuje wartość danego dokumentu;
- `invoiceCreateDocument(lType,lSubType : Integer; lProdId : Integer; lCalc : String; Import:String;nStatus : Integer;openDoc:Boolean) : Integer;` - tworzy dokument handlowy;

Moduł BinSoft.Stock - Biblioteka LibStock

- `stockUpdateRelationInfo(idInvoice : Integer; idDoc : Integer);` - odświeżenie relacji pomiędzy dokumentami handlowymi i magazynowymi;
- `StockRebuildServices;` - odbudowa usług (ich ilości);
- `StockRefreshBuyPrices;` - aktualizacja cen zakupu na indeksie;
- `StocksRefreshProduct(ID : Integer;MAG:Integer);` - odświeżenie stanu magazynowego dla danego produktu;

- `StocksRefreshProductsOnDocument(PID : Integer);` - odświeżenie stanów magazynowych produktów z określonego dokumentu magazynowego;
- `StocksCreateProductsAndRefreshByDocument(lIdDoc : Integer;newCurr,nCurr:String;AddNewProducts:Boolean; lRefreshStockCounts:Boolean);` - dodanie produktów z dokumentu magazynowego do indeksu produktów i odświeżenie ich stanów magazynowych;function Stock
- `StocksCreateRelationBetweenDocuments(lCreate:Boolean;lIDDOC : Integer;showInfo:Boolean) : Boolean;` - utworzenie relacji pomiędzy dokumentami magazynowmymi;
- `ClearRelationBetweenDocuments(lIDDOC : Integer;showInfo:Boolean):Boolean;` - usunięcie relacji pomiędzy dokumentami magazynowymi;
- `StocksCreateDocumentsFromInvoice(IdDoc : Integer;lZwrot:Boolean):Integer;` - utworzenie dokumentów magazynowych do dokumentu handlowego;
- `stockChangeStatusForDocument(ID : Integer; lNewStatus : Integer) : Integer;` - zmiana statusu dokumentu magazynowego;

Zdarzenia

System BSX pozwala na przechwytywanie różnego rodzaju zdarzeń w nim zachodzących. Tymi zdarzeniami są na przykład: moment otwierania okna, jego zamykania, zapisywania, wczytywania, określania wartości komórki w tabelce itp.

Kiedy zachodzi określone zdarzenie, system BSX automatycznie odszukuje funkcji w powiązanym skrypcie, o z góry zdefiniowanej nazwie. Jeśli funkcja taka zostanie odnaleziona, zostanie wywołana. Jeśli jej nie będzie, po prostu zostanie zignorowana. Aby zatem przechwycić określone zdarzenie, należy zwyczajnie utworzyć funkcję o specyficznej nazwie.

Wszystkie funkcje związane ze zdarzeniami rozpoczynają się od zwrotu `fOn`. Z tego też względu nie powinniśmy tworzyć własnych funkcji (nieobsługujących zdarzeń) o takich przedrostkach, by niechcący nie utworzyć funkcji obsługującej zdarzenie (jeśli nie mieliśmy takiego zamysłu).

Zdarzenia związane z formularzami

Poniżej podane zostały najważniejsze zdarzenia związane z formularzami. Parametry do nich przekazywane mają następujące znaczenie:

- `Sender` - obiekt okna, z którego pochodzi dane zdarzenie;
- `T` - obiekt wiersza pobranego z bazy danych;
- `ID` - id rekordu;
- `Key`, `KeyChar` - kod ASCII wciśniętego klawisza;

Lista zdarzeń:

- `fOnCreateForm(Sender)` - tworzenie okna;

- `fOnClearForm(Sender)` - czyszczenie okna;
- `fOnShowForm(Sender)` - wyświetlanie okna;
- `fOnShowDelayForm(Sender)` - ok. 0.5 sekundy po wyświetleniu okna;
- `fOnCloseQuery(Sender)` - zapytanie, czy można zamknąć okno;
- `fOnClose(Sender)` - zamykanie okna;
- `fOnDestroyForm(Sender)` - zwalnianie pamięci okna;
- `fOnBeforeLoadForm(Sender)` - przed załadowaniem rekordu do okna;
- `fOnLoadForm(Sender,T)` - załadowanie rekordu do okna;
- `fOnBeforeSave(Sender)` - przed zapisanie rekordu;
- `fOnSave(Sender)` - zapisanie rekordu;
- `fOnCheckForm(Sender)` - sprawdzenie czy formularz wypełniono poprawnie;
- `fOnAfterSave(Sender,ID)` - po zapisaniu rekordu;
- `fOnEndSave(Sender,ID)` - sam koniec zapisywania rekordu;
- `fOnLockedSave(Sender)` - próba zapisania rekordu, który jest zablokowany;
- `fOnBtnOK(Sender)` - kliknięto przycisk *OK*;
- `fOnBtnCancel(Sender)` - kliknięto przycisk *Anuluj*;
- `fOnBtnPrint(Sender)` - kliknięto przycisk *Drukuj*;
- `fOnKeyDown(Sender,Key,KeyChar)` - wciśnięto klawisz na klawiaturze;

Domyślnie system BSX stosujemy tzw. buforowanie okien formularzy. Mechanizm ten polega na tym, że kiedy próbujemy wyświetlić okno danego formula-

rza, jest ono tworzone i wyświetlane użytkownikowi. Kiedy następnie użytkownik je zamknie, okno nie jest usuwane z pamięci, a jedynie ukrywane. Kiedy następnie użytkownik ponownie spróbuje je wyświetlić, pokaże się ono znacznie szybciej, gdyż będzie już wygenerowane w pamięci. W związku z powyższym, zdarzenie `fOnCreateForm()` będzie wywoływane tylko raz, w momencie utworzenia formularza. Podobnie `fOnDestroyForm()` - tylko raz, w momencie usuwania formularza z pamięci. Po zdarzeniu `fOnCreateForm()` wywoła się zawsze `fOnClearForm()` - oznaczające, że formularz jest czyszczony, tzn. doprowadzany do wartości domyślnych. I to zdarzenie będzie wywoływane wiele razy, zawsze przed wyświetleniem formularza.

Większość z wymienionych zdarzeń, jeśli zwrócą wartość `"false"` (nie wartość logiczną `false`, lecz ciąg tekstowy „`false"`) - spowodują przerwanie dalszego wykonywania instrukcji. Jeśli na przykład, zdarzenie `fOnCloseQuery()` zwróci `"false"` - okno nie będzie zamknięte. Podobnie, jeśli `fOnSave()` lub `fOnBeforeSave()` zwrócą `"false"` - dane w oknie nie będą zapisane.

Zdarzenia związane z widokami

Poniżej zebrane zostały zdarzenia związane z widokami. Znaczenie parametrów przekazywanych przez te zdarzenia są następujące:

- `Sender` - obiekt okna, z którego pochodzi zdarzenie;
- `ID` - id rekordu;
- `IDName` - nazwa kolumny z kluczem głównym danej tabeli;
- `Col` - numer kolumny;
- `Row` - numer wiersza;
- `TBL` - obiekt tabelki;

- `Value` - wartość komórki;
- `ColName` - nazwa kolumny;
- `SelectedCount` - liczba zaznaczonych wierszy;

Lista zdarzeń:

- `fOnShowForm(Sender)` - wyświetlanie widoku;
- `fOnClose(Sender)` - zamykanie widoku;
- `fOnCellClick(Sender,Col,Row,IDName,ID)` - kliknięto w komórkę tabli;
- `fOnDblClick(Sender,ID,IDName,TBL)` - dwukrotnie kliknięto w tabeli;
- `fOnGetCellData(Sender,Col,Row,Value)` - pobieranie zawartości komórki do wyświetlenia;
- `fOnGetCellValue(Sender,T,ColName,Col,Row,Value)` - pobieranie zawartości komórki z bazy danych;
- `fOnGetCellStyle(Sender,Col,RowValue)` - ustalanie stylów dla komórki;
- `fOnCellAnchorClick(Sender,Col,Row,ColName,ID)` - kliknięto w hiperłącze w danej komórce;
- `fOnCellBitmapClick(Sender,Cell,Col,Row,ColName,ID)` - kliknięto w ikonkę powiązaną z daną komórką;
- `fOnCellButtonClick(Sender,Cell,Col,Row,ColName,ID)` - kliknięto w przycisk powiązany z daną komórką;
- `fOnCellCheckboxClick(Sender,Cell,Col,Row,ColName,ID)` - kliknięto w pole *checkbox* powiązane z daną komórką;

- `fOnCellCommentClick(Sender,Cell,Col,Row,ColName,ID)` - kliknięto w pole komentarza powiązane z daną komórką;
- `fOnCellRadioButtonClick(Sender,Cell,Col,Row,ColName,ID)` - kliknięto w pole *radio* powiązane z daną komórką;
- `fOnCellProperties(Self,Cell,Col,Row,ColName,ID)` - ustalenie właściwości komórki;
- `fOnCellClass(Self,Col,Row,ColName,ID)` - definiowanie klasy komórki;
- `fOnAdd(Sender)` - kliknięto przycisk *Dodaj*;
- `fOnEdit(Sender,ID,TBL)` - kliknięto przycisk *Edytuj*;
- `fOnSelectRecord(Sender,ID)` - kliknięto przycisk *Wybierz*;
- `fOnPrintRecord(Sender,ID,SelectedCount)` - kliknięto przycisk *Drukuj listę*;
- `fOnDeleteRecord(Sender)` - kliknięto przycisk *Usuń*;
- `fOnDeleteItem(Sender,ID,IDName,TBL)` - usuwanie pojedynczego rekordu;

Zdarzenia związane z kontrolkami

W formularzu może znajdować się wiele różnych kontrolek, np. kontrolki edycyjne, wyboru, daty itp. Istnieje możliwość przechwytywania zdarzeń związanych z tymi kontrolkami. Aby tego dokonać należy utworzyć funkcję o określonej nazwie. Na przykład, aby przechwycić moment „zmiany" zawartości pola tekstowego można przechwycić zdarzenie *onChange*. W tym celu należy stworzyć funkcję:

```
function fOnChange_NAZWAPOLA(Sender)
```

Po nazwie właściwej danego zdarzenia i po symbolu _ należy podać nazwę kontrolki, której dane zdarzenie dotyczy.

Nie wszystkie zdarzenia są wspólne dla wszystkich kontrolek. Jednak większość z nich posiada następujące zdarzenia:

- `fOnChane_XXX(Sender)` - wystąpiły zmiany w danym polu;
- `fOnKeyDown_XXX(Sender, Key, KeyChar)` - wciśnięto klawisz na klawiaturze;
- `fOnKeyUp_XXX(Sender, Key, KeyChar)` - puszczono klawisz klawiatury;
- `fOnEnter_XXX(Sender)` - kontrolka przejęła aktywność (*focus*);
- `fOnExit_XXX(Sender)` - kontrolka traci aktywność (*focus*);

`XXX` podane w nazwach funkcji powinno być zamienione nazwą kontrolki, którą chcemy przechwycić.

W momencie czyszczenia zawartości formularza lub ładowania do niego danych wczytanych z bazy danych, niektóre zdarzenia mogą być wywoływane automatycznie (np. zdarzenie *OnChange*). Często w tych sytuacjach nie chcemy wykonywać określonych czynności przypisanych do funkcji obsługujących te zdarzenia. Z tego względu istnieją dwie specjalne funkcje, za pomocą których możemy określić czy nie mamy do czynienia z tymi sytuacjami.

- `FormIsLoading` - funkcja zwraca `true`, jeśli właśnie odbywa się ładowanie danych do formularza z bazy danych;
- `FormIsClearing` - funkcja zwraca `true`, jeśli właśnie odbywa się czyszczenie formularza;

API

W kolejnych podrozdziałach tego rozdziału wyjaśnione zostaną różnego rodzaju klasy i biblioteki dołączone do systemu BSX. Zapewniają one programiście dostęp do wszystkich elementów systemu. Pozwalają na uzyskiwanie dostępu do aktywnej bazy danych, tworzenie połączeń z innymi bazami danych, pobieranie z nich danych, wykonywanie dowolnych zapytań itd. Ale to nie wszystko. Różnego rodzaju dostarczone klasy pozwalają na szyfrowanie danych, obsługę protokołów HTTP, FTP, SFTP, POP3, SMTP, IMAP, REST, SOAP i wiele więcej.

TBinDatabase - Bazy danych

Interpreter języka PascalBSX wyposażony został w klasę o nazwie `BinDatabase` zapewniającą dostęp do szeregu metod związanych z obsługą baz danych. Klasa ta pozwala na nawiązywanie połączeń z dowolnym serwerem (wspieranym przez BSX), wykonywanie dowolnych zapytań SQL, pobieranie danych itp.

Spójrzmy na przykład:

```
var B;
begin
  B:=TBinDatabase.CreateSQLite('c:\nazwa.db','');
  B.ExecSQL('UPDATE tabela SET x=1 WHERE id=2');
  B.Free;
end;
```

Powyżej nawiązano połączenie z bazą typu SQLite zapisaną w pliku „`c:\nazwa.db`". Jeśli takiego pliku nie ma, baza zostanie utworzona. Wykonane w niej zostanie zapytanie SQL typu `UPDATE`.

Nawiązywanie połączeń

Pierwszą czynnością jaką należy wykonać aby skorzystać z jakiejś bazy danych, jest utworzenie nowego obiektu typu `TBinDatabase`. Poprzez odpowiedni konstruktor podajemy dane dostępowe do serwera. Oto możliwe konstruktory:

- `CreateSQLite(NazwaBazy, Haslo)` - nawiązanie połączenia z bazą SQLite. Parametr `hasło` pozwala na dostęp do baz szyfrowanych. W chwili obecnej szyfrowanie dostępne jest tylko w środowisku Windows;

- `CreateFirebird(NazwaBazy)` - nawiązanie połączenia z bazą Firebird; wykorzystywana jest wbudowana obsługa baz tego typu; Metoda dostępna tylko w środowisku Windows;
- `CreateRemote(Protocol, Host, Port, Login, Pass, Database, HostTunel, SSHHost, SSHPort, SSHUser, SSHPort)` - nawiązanie połączenia z dowolną obsługiwaną bazą danych; W parametrze `Protocol` podajemy nazwę protokołu. Dostępne są wartości: `mysql`, `mariadb`, `sqlite`, `firebird`, `interbase`, `mssql`, `postgresql`; Parametr `HostTunel` może zawierać adres URL do skryptu, który będzie wykorzystany do tunelowania. Poprzez parametry z przedrostkiem `SSH` możemy podać parametry do serwera SSH, który ma być wykorzystany do tunelowania.

Pobieranie danych z bazy

Dostępnych jest szereg metod pozwalających na pobieranie danych z bazy danych. Oto niektóre z nich:

- `GetRow(SQL)` - pobiera jeden wiersz z bazy danych,
- `GetRows(SQL)` - pobiera wszystkie wiersze z bazy danych,
- `GetValueS(SQL, Default)` - pobiera jedną wartość z określonej tabeli; wartość zwracana jest typu `string`;
- `GetValueL(SQL, Default)` - pobiera jedną wartość z określonej tabeli; wartość zwracana jest typu `integer`;
- `GetValueD(SQL, Default)` - pobiera jedną wartość z określonej tabeli; wartość zwracana jest typu `double`;

Jeśli metoda `GetRow()` się powiedzie (odnaleziony będzie szukany wiersz) - zwróci ona obiekt wiersza odpowiedzi. W przeciwnym wypadku, zwróci wartość `nil`.

Wiersz odpowiedzi zwracany metodami `GetRow()` i `GetRows()` jest obiektem zapewniającym następujące właściwości i metody:

- `RowCount` - liczba wierszy odpowiedzi,
- `ColCount` - liczba kolumn odpowiedzi,
- `Eof` - czy aktualna odpowiedź jest ostatnią,
- `Next` - przejście do następnego wiersza odpowiedzi,
- `Previous` - przejście do poprzedniego wiersza odpowiedzi,
- `MoveFirst` - przejście do pierwszego wiersza odpowiedzi,
- `MoveLast` - przejście do ostatniego wiersza odpowiedzi,
- `Columns[index]` - pobranie nazwy kolumny o indeksie `index`,
- `Fields[index]` - pobranie zawartości kolumny o indeksie `index`,
- `Types[index]` - zwraca typ kolumny o indeksie `index`,
- `FieldByName[Nazwa]` - pobranie zawartości kolumny o podanej nazwie; zwraca wynik typu `string`,
- `IntFieldByName[Name]` - pobranie zawartości kolumny o podanej nazwie; zwraca wynik typu `integer`,
- `FloatFieldByName[Name]` - pobranie zawartości kolumny o podanej nazwie; zwraca wynik typu `double`;
- `FieldValueS[Name]` - pobranie zawartości kolumny o podanej nazwie; jeśli kolumna nie istnieje, zwróci ciąg pusty;
- `FieldValueL[Name]` - pobranie zawartości kolumny o podanej nazwie i zwrócenie wyniku typu `integer`; jeśli kolumna nie zostanie odnaleziona zwróci wynik 0;

- `FieldValueD[Name]` - pobranie zawartości kolumny o podanej nazwie i zwrócenie wyniku typu `double`; jeśli kolumna nie zostanie odnaleziona zwróci wynik 0;
- `FieldValueB[Name]` - pobranie zawartości kolumny o podanej nazwie i zwrócenie wyniku typu `boolean`; jeśli kolumna nie zostanie odnaleziona zwróci wynik `false`;
- `FieldTypeByName[Name]` - zwraca typ kolumny o podanej nazwie;
- `FieldExists(Name)` - sprawdzenie, czy istnieje kolumna odpowiedzi o danej nazwie;

Spójrzmy na przykład:

```
var B, T;
begin
  B:=TBinDatabase.CreateSQLite('nazwa.db','');

  T:=B.GetRow('SELECT * FROM tabela WHERE id=1');
  if T<>nil then
  begin
    ShowMessage(T.FieldByName['imie']);
    T.Free;
  end;

  B.Free;
end;
```

Kolejny przykład pobiera wszystkie wiersze z odpowiedzi:

```
var B, T;
begin
  B:=TBinDatabase.CreateSQLite('nazwa.db','');

  T:=B.GetRows('SELECT * FROM tabela WHERE imie LIKE ''%ol%''');
```

```
  while not T.Eof do
  begin
    ShowMessage(T.FieldByName['imie']);
    T.Next;
  end;
  T.Free;

  B.Free;
end;
```

Należy pamiętać o zwalnianiu pamięci z obiektów bazy danych oraz wierszy odpowiedzi. Dokonuje się tego metodą `Free`, np. `D.Free;`

Przydatne metody klasy TBinDatabase

Klasa `TBinDatabase` oprócz metod służących do pobierania danych, posiada dostępne jeszcze inne przydatne funkcje. Oto niektóre z nich:

- `ExecSQL(SQL)` - wykonanie dowolnego zapytania SQL,
- `DeleteRow(Table,Where)` - usunięcie wiersza z tabeli,
- `DuplicateRow(Table,Where)` - duplikowanie wiersza tabeli; metoda zwraca `ID` nowo utworzonego wiersza,
- `RowExists(Table,Where)` - sprawdzenie, czy istniej wiersz tabeli spełniający określony warunek,
- `TableExists(Table)` - sprawdzenie, czy istnieje tabela o podanej nazwie,
- `FieldExists(FieldName,TableName)` - sprawdzenie czy istnieje kolumna w danej tabeli,
- `FieldType(FieldName,TableName)` - sprawdzenie typu kolumny w tabeli,

- `Start` - rozpoczęcie transakcji,
- `Commit` - zatwierdzenie transakcji,
- `Rollback` - anulowanie transakcji,
- `InTransaction` - sprawdzenie, czy jest się w trakcie transakcji,
- `Ping(resume)` - wykonanie „ping" do bazy danych; parametr `resume` określa, czy w przypadku braku aktywnego połączenia, ma ono być automatycznie wznawiane;
- `AddSlashes(S)` - zabezpieczenie ciągu `S` przed znakami specjalnymi,
- `AddDate(date)` - dodanie daty (przekazanej jako `TDateTime`) do ciągu zapytania SQL,
- `AddDateS(dateS)` - dodanie daty (przekazanego jako `string`) do ciągu zapytania SQL,
- `AddDateTime(date)` - dodanie daty i godziny (przekazanej jako `TDateTime`) do ciągu zapytania SQL,
- `AddDateTimeS(dateS)` - dodanie daty i godziny (przekazanej jako `string`) do ciągu zapytania SQL;

Dodawanie i modyfikacja rekordów

Rekordy można dodawać i modyfikować poprzez metodę `ExecSQL` i skonstruowanie odpowiedniego zapytania SQL. Nie jest to rozwiązanie wygodne. Dlatego system PascalBSX obsługuje odpowiednią klasę, która to zadanie znacznie usprawnia. Obiekt tej klasy tworzymy wywołując metodę `InsertUpdateRow()` na rzecz obiektu bazy danych. W parametrze podajemy nazwę tabeli, na której ma być wykonana modyfikacja.

Mając utworzony obiekt tego typu możemy na nim wykonać następujące metody:

- `AddS(nazwa, typ, wartość)` - ustalenie wartości dla określonego pola (wartość typu `String`),
- `AddL(nazwa, typ, wartość)` - ustalenie wartości dla określonego pola (wartość typu `Integer`),
- `AddD(nazwa, typ, wartość)` - ustalenie wartości dla określonego pola (wartość typu `Double`),
- `AddB(nazwa, typ, wartość)` - ustalenie wartości dla określonego pola (wartość typu `Boolean`),
- `Execute(ID)` - wykonanie zapytania (parametr `ID` jest typu `Integer`),
- `ExecuteS(ID)` - wykonanie zapytania (parametr `ID` jest typu `String`);

Jeśli parametr `ID` ma wartość 0 lub ‚0' (zależnie od metody) - wykona się komenda `INSERT` i rekord doda się do bazy danych, w przeciwnym wypadku wykona się zapytanie typu `UPDATE`. Oto przykład:

```
var B, E;
begin
  B:=TBinDatabase.CreateSQLite('nazwa.db','');

  E:=B.InsertUpdateRow('imiona');
  E.AddS('imie','varchar','Karol');
  E.AddS('wiek','int','32');
  E.Execute(0);
  E.Free;

  B.Free;
end;
```

Dostęp do aktywnej bazy danych

Mając nawiązane połączenie z bazą danych, w każdym skrypcie mamy dostęp do obiektu o nazwie `BinDB`, który daje nam dostęp do bieżącego połączenia z bazą danych. Możemy zatem na rzecz tego obiektu korzystać z wszystkich opisanych w tym rozdziale metod i rozwiązań. Nie trzeba więc tworzyć tego typu obiektu, ani deklarować jakiejś zmiennej. **Nie wolno go również zwalniać!**

TBinFMCrypto - Kryptografia

Klasa `TBinFMCrypto` zapewnia implementację wielu algorytmów kryptograficznych oraz funkcji skrótu.

- **`TBinFMCrypto.MD5(Text)`** - oblicza funkcję skrótu MD5;
- **`TBinFMCrypto.SHA1(Text)`** - oblicza funkcję skrótu SHA1;
- **`TBinFMCrypto.TextCrypt(Text, Pass, Alg)`** - funkcja szyfruje `Text` algorytmem `Alg` z uwzględnieniem hasła `Pass`;
- **`TBinFMCrypto.TextDecrypt(Text, Pass, Alg)`** - funkcja deszyfruje `Text` zakodowany algorytmem `Alg` i hasłem `Pass`;

Jako nazwę algorytmu we wspomnianych funkcjach można użyć ciągów: `blowfish`, `cast128`, `cast256`, `des`, `3des`, `gost`, `ice`, `thinice`, `ice2`, `idea`, `mars`, `misty1`, `rc2`, `rc4`, `rc5`, `rc6`, `rijndael`, `serpent`, `tea`, `twofish`;

TBinFMUtils - Użyteczne funkcje

Klasa `TBinFMUtils` oferuje szereg przydatnych metod, różnego typu. Są to metody statystyczne, zatem korzystamy z nich poprzedzając ich nazwy nazwą klasy. W tym rozdziale przedstawione zostały metody, z których korzystać będziemy najczęściej.

- `DateTimeToUnix(D)` - konwertuje datę typ `TDateTime` do formatu Unix;
- `UnitToDateTime(D)` - konwertuje datę typu Unix do formatu `TDateTime`;
- `DaysInMonth(M)` - zwraca liczbę dni w miesiącu o numerze `M`;
- `AgeCalc(D1,D2)` - zwraca wiek osoby urodzonej w dniu `D1` na dzień `D2` (`D2` i `D2` są typu `TDateTime`);
- `getDate(S, D)` - zwraca datę (`TDateTime`) określoną identyfikatorem `S` na podstawie daty `D` (`TDateTime`); dostępne są identyfikatory: `monthstart` (początek miesiąca), `monthend` (koniec miesiąca), `yearstart` (początek roku), `yearend` (koniec roku), `todaystart` (początek dzisiejszego dnia), `todayend` (koniec dzisiejszego dnia), `yesterdaystart` (początek wczorajszego dnia), `yesterdayend` (koniec wczorajszego dnia), `monthXstart` (początek miesiąca `X`), `monthXend` (koniec miesiąca X);
- `CopyToClipboard(S)` - skopiowanie ciągu `S` do systemowego schowka;
- `PasteFromClipboard` - zwrócenie ciągu znajdującego się w systemowych schowku;
- `Random(Start, end)` - wylosowanie liczby całkowitej z przedziału `Start...End`;

- `ReplaceString(S, fromS, toS)` - zastąpienie w ciągu `S` podciągu `fromS` na podciąg `toS`;
- `BoolToStr(B)` - konwersja wartości logicznej `B` do ciągu tekstowego (`true=1`, `false=0`);
- `SizeToStr(L)` - zamiana liczby bajtów `L` na wartość opisową z jednostkami KB, MB, GB itp.
- `IntToStrFill(L, Len, Fill)` - konwersja liczby całkowitej `L` do ciągu tekstowego, którego długość wyniesie `Len` znaków; ciąg wypełniany będzie symbolem `Fill`;
- `isPhone(S)` - sprawdzenie, czy podany ciąg jest prawidłowym numerem telefonu;
- `isEmail(S)` - sprawdzenie, czy podany ciąg jest poprawnym adresem e-mail;
- `isDigits(S)` - sprawdzenie, czy podany ciąg zawiera tylko cyfry;
- `isAlpha(S)` - sprawdzenie, czy podany ciąg zawiera tylko znaki alfanumeryczne (litery i cyfry);
- `isIdentifier(S)` - sprawdzenie, czy podany ciąg jest poprawny identyfikatorem (składa się z liter, cyfr, znaku podkreślenia, przy czym nie może rozpoczynać się cyfrą);
- `LoadLinesFromFile(FileName, List)` - wczytuje plik tekstowy `FileName` i zwraca go w postaci obiektu `TStringList`; jeśli parametr `List` (typu `TStringList`) jest podany (`<>nil`), wykorzystany zostanie ten obiekt;
- `FileSize(FileName)` - zwraca wielkość pliku `FileName`;
- `PathToHomeFolder` - zwraca ścieżkę do folderu domowego;

- `MSL(L, A, B, C)` - na podstawie wartości liczby całkowitej `L` (i gramatyki języka polskiego) zwraca ciąg `A`, `B` lub `C`; przykład: `MSL(X,"sekunda", "sekundy", "sekund");`

APP - Obiekt aplikacji

System BSX oferuje obiekt `APP`, do którego mamy swobodny dostęp z poziomu interpretera PascalBSX. Obiekt ten zapewnia szereg metod pozwalających na sterowanie działaniem aplikacji i wykonywaniem różnych przydatnych funkcji na jej rzecz. W tym rozdziale zebrane zostały najważniejsze z nich.

- `APP.ShowMessageInfo(S)` - wyświetlenie komunikatu `S`; w komunikacje można używać prostych znaczników HTML;
- `APP.ShowMessageError(S)` - wyświetlenie komunikatu o błędzie `S`; w komunikacie można używać prostych znaczników HTML;
- `APP.isMACOSX` - sprawdzenie, czy aplikacja działa w środowisku OS X;
- `APP.isWindows` - sprawdzenie, czy aplikacja działa w środowisku Windows;
- `APP.ShowWait(S)` - wyświetlenie okna *Proszę czekać* z komunikatem `S`;
- `APP.HideWait` - ukrycie okna *Proszę czekać*;
- `APP.ShowWaitProgress(Position,Max)` - wyświetlenie paska postępu w oknie *Proszę czekać*;
- `APP.ShowWaitCancel(Show)` - wyświetlenie lub ukrycie przycisku **Anuluj** w oknie *Proszę czekać*; Metodą `APP.WaitCancelStatus` należy sprawdzać, czy przycisk **Anuluj** został kliknięty;
- `APP.WaitCancelStatus` - zwraca status kliknięcia przycisku **Anuluj** w oknie *Proszę czekać*;
- `APP.ProcessMessages` - przetworzenie kolejki komunikatów; należy z tej metody korzystać w różnego rodzaju pętlach, aby program nie *zamrażał* się, lecz stale odbierał komunikaty od użytkownika;

- `APP.Wait(MS)` - wstrzymanie działania programu na `MS` milisekund;

Formularze

Formularze to wszelkiego rodzaju okna pozwalające na wprowadzanie różnych danych. Definiujemy je znacznikiem `<form>` w obrębie pary znaczników `<forms>...</forms>`. W rozdziale *Elementy wtyczki* przedstawione już zostały podstawowe informacje na temat tworzenia okien tego typu. W tym rozdziale wyjaśnione zostaną te zagadnienia bardziej szczegółowo. Omówione również zostaną liczne znaczniki pozwalające na tworzenie różnego rodzaju kontrolek.

Znacznik <form>

Znacznik `<form>` posiada szereg atrybutów wpływających na sposób wyświetlania danego formularza.

Atrybuty:

- **`name`** (**string**) - nazwa formularza;
- **`caption`** (**string**) - tytuł formularza;
- **`width`** (**int**) - domyślna szerokość okna formularza,
- **`height`** (**int**) - domyślna wysokość okna formularza,
- **`table`** (**string**) - powiązana tabela danych w bazie danych,
- **`image`** (**string**) - powiązana ikona dla formularza,
- **`btnokvisible`** (**bool**) - czy ma być widoczny przycisk *OK*,
- **`btnsavevisible`** (**bool**) - czy ma być widoczny przycisk *Zapisz*,
- **`btncancelvisible`** (**bool**) - czy ma być widoczny przycisk *Anuluj*,
- **`btnprintvisible`** (**bool**) - czy ma być widoczny przycisk *Drukuj*,
- **`buffers`** (**bool**) - czy okno ma być buforowane;
- **`default`** (**string**) - ustawienie wartości domyślnych;

- **`focusto` (string)** - nazwa kontrolki, która ma być aktywna jako pierwsza, po wyświetleniu formularza;

Każdy formularz powinien mieć nadaną nazwę (`name`). Wszystkie pozostałe atrybuty są opcjonalne.

Domyślnie, wszystkie formularze są buforowane. Oznacza to, że są one generowane w momencie próby ich pierwszego wyświetlenia. Następnie kiedy okno danego formularza jest zamykane, fizycznie zostaje ono jedynie ukryte, po to, by przy ponownym jego wyświetleniu nie trzeba było go ponownie generować. Poprzez atrybut `buffers` można wyłączyć omówiony mechanizm.

Jeśli dla formularza zdefiniuje się atrybut `table` - oznaczać to będzie, że powiązany on jest z określoną tabelą w bazie danych. Dzięki temu, system BSX będzie oferował mechanizmy do automatycznego zapisywania danych do tej tabeli, oraz odczytywania z niej danych. Zdefiniowanie tego pola powoduje zatem, że w formularzu pojawią się przycisk „*Zapisz*”, „*OK*” oraz „*Anuluj*”. Jeśli atrybut `table` nie jest zdefiniowany, przycisk „*Zapisz*” będzie niewidoczny.

Mając powiązany formularz z tabelą w bazie danych, system BSX będzie automatycznie próbował zapisać w niej dane, na podstawie kontrolek dostępnych w tym formularzu. Powiązanie danych w bazie danych z kontrolkami, odbywa się poprzez nazwy tych kontrolek. Jeśli nazwy są tożsame (kontrolka ma tę samą nazwę do kolumna w bazie danych) - dane zostaną zapisane.

Każdy formularz posiada tytuł (atrybut `caption`). Istnieje jednak możliwość definiowania tytułu alternatywnego, wykorzystywanego w momencie otwierania rekordu z bazy danych. Dokonuje się tego w tym samym atrybucie `caption`, lecz po znaku separatora `|`. Przykład: `caption="Dodaj nowy rekord| Edytuj rekord"`. Przy tak zdefiniowanym tytule formularza, kiedy otwierany jest nowy rekord - tytułem okna będzie *Dodaj nowy rekord*, natomiast podczas edycji rekordu, pojawi się tytuł *Edytuj rekord*. W tytule `caption` można używać

tzw. odsyłaczy do kontrolek. Zapisuje się je jako nazwę kontrolki objętą nawiasami klamrowymi. W ten sposób w miejscu takiego odsyłacza, wstawiona zostanie zawartość odpowiedniej kontrolki. Przykład: `caption="Dodaj nową osobę|{nazwisko}"`. Przy tak zdefiniowanym tytule, podczas otwierania nowego rekordu, tytułem formularza będzie *Dodaj nową osobę*. Kiedy natomiast rekord zostanie otworzony do edycji, w tytule pojawi się nazwisko osoby.

Warunkowe ładowanie elementów formularza

Istnieje wiele znaczników powodujących wyświetlenie kontrolek różnego rodzaju, np. pola tekstowe, pola wyboru, panele z zakładkami itp. Wszystkie te znaczniki opisane zostały w kolejnych podrozdziałach tego rozdziału. Każdy z tych znaczników obsługuje jednak pewien wspólny zestaw atrybutów, poprzez który możemy wpływać na to, czy będzie on interpretowany przez system BSX, czy też zostanie zignorowany. Dzięki tym atrybutom możemy tworzyć elementy formularza, które będą widoczne tylko w określonych wersjach programu, tylko na określonych systemach operacyjnych itp.

Oto lista atrybutów:

- `ifmodule="moduł"` - załadowanie danego znacznika, jeśli istnieje określony `moduł`;
- `iftableexists="tabela"` - załadowanie danego znacznika, jeśli istnieje określona `tabela` w bazie danych;
- `ifosx="true/false"` - załadowanie danego znacznika, jeśli systemem operacyjnym jest OS X (lub nie jest OS X - zależnie od wartości);
- `ifwindows="true/false"` - załadowanie danego znacznika, jeśli systemem operacyjnym jest Windows (lub nie jest Windows - zależnie od wartości);
- `permission="uprawnienie"` - załadowanie danego znacznika, jeśli dane `uprawnienie` istnieje;

Kontrolki

System BSX oferuje kilkadziesiąt znaczników pozwalających na wstawianie kontrolek różnego rodzaju. Kontrolki te możemy podzielić na kontrolki edycyjne, czyli takie które umożliwiają wprowadzenie danych przez użytkownika. Jeśli formularz jest powiązany z bazą danych, powodują one tworzenie relacji pomiędzy nimi a kolumnami bazy danych. Dostępne są również kontrolki nieedycyjne. Czyli takie, które służą jedynie stronie wizualnej i nie wiążą się z danymi w bazie. Przykładem takich kontrolek są panele, panele z zakładkami itp.

Różne kontrolki oferują różne atrybuty. Część z atrybutów jest jednak wspólna dla wielu z nich. Oto najważniejsze z nich:

- **`name` (string)** - nazwa kontrolki; w przypadku formularza powiązanego z tabelą w bazie danych, nazwa ta jest też nazwą kolumny w bazie danych, z którą pole ma być w relacji;
- **`cname` (string)** - nazwa kontrolki w samym formularzu; z reguły nie używamy tego atrybutu; podajemy go wówczas, jeśli chcemy stworzyć kontrolkę o nazwie takiej jak kolumna w bazie danych, ale nie chcemy jej wiązać z tą kolumną (wówczas podajemy atrybut `cname`, a atrybut `name` pozostawiamy pusty) lub jeśli chcemy by kontrolka była powiązana z kolumną w bazie danych, ale w formularzu by była widoczna pod inną nazwą;
- **`left` (int)** - położenie kontrolki od lewej; jeśli atrybut nie jest podany, wówczas wynosi od `10px`;
- **`top` (int)** - położenie kontrolki od góry; jeśli atrybut nie jest podany, wówczas wynosi `10px` i jest automatycznie powiększany po wyświetleniu danego elementu; dzięki temu wymienienie kolejnych kontrolek bez podania atrybutów `left` i `top` spowoduje, że będą się one pojawiały jedna pod drugą;

atrybutowi `top` można nadać wartość specjalną `-1`. Oznacza ona, że dana kontrolka ma się pojawić na wysokości poprzedniej kontrolki, obok niej;

- **`x`** (**int**) - przesunięcie o zadaną liczbę pikseli w prawo;
- **`y`** (**int**) - przesunięcie o zadaną liczbę pikseli w dół;
- **`width`** (**int**) - szerokość kontrolki;
- **`height`** (**int**) - wysokość kontrolki;
- **`caption`** (**string**) - tytuł kontrolki;
- **`text`** (**string**) - zawartość kontrolki;
- **`default`** (**string**) - wartość domyślna;
- **`angle`** (**int**) - kąt obrotu kontrolki;
- **`opacity`** (**double**) - przezroczystość kontrolki;
- **`enabled`** (**bool**) - czy kontrolka jest włączona;
- **`readonly`** (**bool**) - czy kontrolka jest tylko do odczytu;
- **`visible`** (**bool**) - czy kontrolka jest widoczna;
- **`maxlength`** (**int**) - maksymalna długość wprowadzanych danych;
- **`noempty`** (**bool**) - oznaczenie, że zawartość kontrolki nie może być pusta;
- **`required`** (**bool**) - oznaczenie, że kontrolka jest wymagana do wypełnienia;
- **`unique`** (**bool**) - oznaczenie, że zawartość kontrolki musi być unikalna;
- **`textalign`** (**string**) - sposób wyrównania zawartości kontrolki; możliwe wartości: `left`, `center`, `right`;
- **`align`** (**string**) - sposób wyrównania kontrolki; możliwe wartości: `left`, `top`, `right`, `bottom`, `client`, `contents`, `center`;

- **anchors** (**string**) - ustawienie kotwic dla kontrolki; można użyć dowolnej kombinacji słów: `left`, `top`, `right`, `bottom` - ustalając tym samym kotwice na określone boki;
- **`margins`** (**int**) - ustawienie marginesów;
- **`margintop`**, **`marginleft`**, **`marginright`**, **`marginbottom`** (**int**) - ustalenie określonego marginesu;
- **`effects`** (**string**) - ustawienie efektów specjalnych dla kontrolki;
- **`tag`** (**int**) - pole do własnego użytku;
- **`tagstring`** (**string**) - pole do własnego użytku;

Nie podając atrybutów `left` ani `top` sprawimy, że kontrolki będą się pojawiać jedna pod drugą. Za pomocą wartości `top="-1"` sprawimy, że dana kontrolka pojawi się obok kontrolki poprzedniej.

Atrybuty `align` oraz `anchors` pozwalają na określanie sposobu wyświetlania danej kontrolki. Głównie będziemy z nich korzystać przy definiowaniu kontrolek nie-edycyjnych, czyli takich które będą określać jedynie wygląd formularza. Ale mogą być oczywiście stosowane w przypadku wszystkich kontrolek.

Atrybut `align` określa sposób wyświetlania danego elementu. Możliwe wartości:

- **`left`** - spowoduje, że kontrola „przyklei" się do lewej krawędzi; oznacza to, że zmieniając wymiary okna (lub kontrolki nadrzędnej) - zmieniać się będzie wysokość danej kontrolki tak, by wypełniała całą wysokość rodzica; szerokość kontrolki pozostanie stała;
- **`right`** - działanie analogiczne do `left`, z tym że kontrolka „przyklei" się do prawej krawędzi;

- **`top`** - działanie analogiczne do `left`, z tym że kontrolka „przyklei" się do górnej krawędzi; będzie zatem wypełniała całą szerokość kontrolki nadrzędnej (lub okna), a stała pozostanie wysokość;
- **`bottom`** - działanie analogiczne do `top`, z tym że kontrolka „przyklei" się do dolnej krawędzi;
- **`client`** - powoduje, że kontrolka wypełni całą dostępną (wolną) przestrzeń;
- **`contents`** - powoduje, że kontrolka wypełni cała przestrzeń kontrolki nadrzędnej, nawet nachodząc na inne kontrolki;
- **`center`** - kontrolka będzie miała szerokość i wysokość zdefiniowaną w odpowiednich parametrach, ale będzie zawsze wyśrodkowana względem kontrolki nadrzędnej;

Jeśli zatem tworzymy okna formularza i chcemy je podzielić - na przykład - na dwie części: górną i dolną, przy czym górna ma mieć stałą wysokość, a dolna „resztę" okna - dla kontrolki opisującej górną cześć zdefiniujemy atrybut `align="top"` i podamy atrybut `height`, natomiast dla dolnej kontrolki przypiszemy `align="client"`.

Kotwicami można uzyskać ten sam efekt co atrybutem `align`. Można też uzyskać jeszcze inne efekty. Ustawienie kotwicy na daną krawędź oznacza, że podczas zmiany wymiarów danej kontrolki, położenie danej krawędzi względem krawędzi nadrzędnej (rodzica) będzie stała.

Domyślnie wszystkie kontrolki mają ustawione kotwice na `left` i `top`. Oznacza to, że mając kontrolkę o wierzchołku (lewy górny róg) ustawionym na `left=10px`, `right=10px`, i określonej szerokości `width` i wysokości `height` - zmiana okna (kontrolki nadrzędnej, rodzica) - nie wpływa na położenie i wymiary danej kontrolki. Gdybyśmy jednak ustawili `anchor="right, bottom"` - wówczas dana kontrolka nie zmieniałaby położenia względem prawego dolnego rogu okna (lub kontrolki nadrzędnej). Czyli zmieniając wymi-

ary okna, nasza kontrolka „pływałaby” tak, że prawy dolny jej róg by był stale w tym samym miejscu.

Kontrolki budowy interfejsu

Kontrolki tego typu, to kontrolki nie-edycyjne. Oznacza to, że nie mają one powiązań z bazą danych, lecz służą jedynie wizualnej budowie formularza. Kontrolki takie można w sobie dowolnie zagnieżdżać. W ich wnętrzu można także wstawiać dowolne inne kontrolki, w tym oczywiście kontrolki edycyjne.

<layout> - Pudełko (ukryte)

Podstawowa kontrola określająca niewidoczny „prostokąt" grupujący inne kontrolki;

<flowlayout> - Pudełko FLOW

TODO

<panel> - Pudełko (widoczne)

Kontrolka wyświetla widoczny prostokąt grupujący inne kontrolki; Pudełko podlega stylom, czyli jego wygląd jest zależny od używanej skórki;

<rectangle> - Prostokąt

Kontrolka wyświetla prostokąt, niepodlegający stylom. Możemy samodzielnie określać sposób jego wyświetlania. Dostępne atrybuty:

- **`pencolor` (string)** - kolor obramowania;
- **`penstyle` (string)** - sposób rysowania obramowania; możliwe wartości: `solid` (wypełniony), `bitmap` (w oparciu o grafikę), `gradient` (w oparciu o gradient), `none` (brak);
- **pendash (string)** - styl rysowania obramowania; możliwe wartości: `dash` (kreski), `dashdot` (kreska-kropka), `dashdotdot` (kreska-kropka-kropka), `dot` (kropka);
- **`fillcolor` (string)** - kolor wypełnienia;

- **`fillstyle` (string)** - sposób wypełniania; możliwe wartości: `solid` (jednolite), `bitmap` (w oparciu o grafikę), `gradient` (w oparciu o gradient), `none` (brak);

<tabs> - Zakładki

Znacznikiem `<tabs>` możemy tworzyć panele zakładek. W obrębie tego znacznika powinny występować jedynie znaczniki `<tab>` opisujące kolejne zakładki. Wewnątrz znacznika `<tab>` mogą już się znajdować dowolne inne znaczniki, w tym również `<tabs>` - czyli ponownie panel z zakładkami.

Znacznik `<tabs>` posiada atrybut:

- **`tabposition` (string)** - sposób wyświetlania panelu z zakładkami; możliwe wartości: `top` (zakładki na górze), `bottom` (zakładki na dole), `dots` (zakładki w postaci kropek), `none` (zakładki niewidoczne);

<expander> - Panel zwijany

Expander to panel grupujący inne kontrolki, który może posiadać dwa stany: zwinięty i rozwinięty.

Atrybuty:

- **`expand` (boolean)** - czy panel ma być zwinięty czy rozwinięty;

Kontrolki edycyjne

Kontrolki edycyjne to takie, które mogą mieć powiązania z elementami bazy danych lub też pozwalające po prostu na interakcję z użytkownikiem poprzez wprowadzanie przez nich danych różnego typu.

<field> - Pole ukryte

Pole niewidoczne na formularzu. Jeśli posiada nazwę zgodną z kolumną w bazie danych, pozwala na dostęp do tej kolumny. Poprzez to pole możemy zatem odczytywać i zapisywać dane w bazie.

<edit> - Podstawowe pole tekstowe

Podstawowe, jednowierszowe pole tekstowe. Pozwala na wprowadzanie przez użytkownika prostych danych. Posiada szereg atrybutów wpływających na sposób jego wyświetlania lub kontroli wprowadzanych danych.

- **`fontsize` (int)** - wielkość czcionki;
- **`fontcolor` (string)** - kolor czcionki;
- **`fontstyle` (string)** - styl czcionki;
- **`textalign` (string)** - sposób wyrównania tekstu;
- **`edittype` (string)** - typ kontrolki; możliwe wartości: `alphanumeric`, `float`, `hex`, `lowercase`, `uppercase`, `money`, `numeric`, `signed-float`, `signednumeric`, `string`;
- **`password` (bool)** - pole do przechowywania hasła;
- **`precision` (int)** - dokładność wyświetlanych liczb zmiennoprzecinkowych;
- **`emptytext` (string)** - tekst wyświetlany kiedy pole jest niewypełnione;

<memo> - Wielowierszowe pole tekstowe

Wielolinijkowe pole tekstowe.

<dateedit> - Pole daty

Kontrolka `<dateedit>` powoduje wyświetlenie pola do wprowadzania daty. Datę można wybierać z kalendarza, który się rozwija po kliknięciu w symbol strzałki. Jeśli kontrolce tej przekażemy atrybut `mtype="datecheck"`, wówczas kontrolka wyposażona zostanie również w pole wyboru *checkbox*, którym można datę aktywować lub dezaktywować.

Jeśli kontrolka posiada pole *checkbox* i jest ono odznaczone, wówczas pobierana z tego pola wartość przyjmie postać daty: 1970-01-01. Data ta reprezentowana jest przez stałą `_EMPTY_DATE_` dostępną w interpreterze PascalBSX. Nadając taką wartość tej kontrolce również sprawimy, że stanie się ona nieaktywna.

<timeedit> - Pole czasu

Kontrolka `<timeedit>` wyświetla pole pozwalające na wprowadzanie czasu.

<combobox> - Pole combo

Pole `<combobox>` to pole wyboru elementu z listy. Wewnątrz tego znacznika powinny występować znaczniki `<option>` opisujące kolejne możliwe do przyjęcia wartości. Pomiędzy znacznikami `<option>` i `</option>` podajemy treść wyświetlaną jako opcję wyboru. Jeśli dla znacznika `<option>` określimy atrybut `id`, wówczas jego wartość zostanie przypisana do kontrolki, gdy dana pozycja zostanie wybrana z listy.

<comboedit> - Pole combo z możliwością edycji

Pole analogiczne do kontroli `<combox>`, z tym że oprócz wyboru opcji z listy, użytkownik ma możliwość samodzielnego nadania tej wartości wpisując ją w to pole.

<checkbox> - Pole checkbox

Pole wyboru *checkbox*.

Kontrolki inne

W tym podrozdziale przedstawione zostały inne kontrolki możliwe do wykorzystania w systemie BSX, jednakże nie wpisujące się w żadną z poprzednich grup.

<button> - Przycisk

Przycisk. Najważniejsze atrybuty to `caption` (wyświetlany na przycisku tekst) oraz `onclick` (nazwa procedury do wywołania po kliknięciu).

<shape> - Figura

TODO

<hintpanel> - Pole podpowiedzi

TODO

<showview> - Osadzenie widoku

Znacznikiem <showview> możemy osadzić dowolny widok wewnątrz danego formularza. Identyfikator formularza podajemy w atrybucie `view`. Inne ciekawe atrybuty zebrano poniżej:

- **`view` (string)** - identyfikator widoku do wyświetlenia;
- **`name` (string)** - nazwa widoku (by można się było do niej odnosić z kodu);
- **`panetitlevisible` (string)** - określa czy ma być widoczny panel tytułowy widoku;
- **`paneltopvisible` (string)** - określa czy ma być widoczny górny panel widoku;
- **`panelsearchvisible` (string)** - określa widoczność pola wyszukiwania;
- **`where` (string)** - dodatkowy warunek, który ma być dodany do zapytania SQL przy wyświetlaniu danego widoku; W warunku tym często korzystamy

z sekwencji: `{_parentrecordid_}` zwracającej ID formularza, na którym dany widok się znajduje;

Rozszerzenia

Oprócz tworzenia własnych modułów, system BSX pozwala na wykorzystywanie stworzonych przez siebie mechanizmów w innych swoich modułach, a nawet rozszerzanie funkcjonalności modułów już istniejących, dostarczonych przez innych producentów! Możliwe jest zatem tworzenia tzw. wtyczek (ang. *plugins*) realizujących różne funkcje, a następnie korzystanie z nich w innych swoich formularzach. Można także swoją wtyczkę „uruchomić" w obrębie innego, istniejącego już modułu.

Tworzenie rozszerzenia

Wtyczki tworzymy znacznikiem `<plugin>` w obrębie znacznika `<plugins>`. Plugin powinien mieć swoją nazwę, którą definiujemy w atrybucie `name`. W obrębie znacznika `<plugin>` możemy definiować dowolny design wtyczki tak, jak byśmy tworzyli formularz. Możemy również użyć znacznika `<script>` aby stworzyć skrypt powiązany z naszą wtyczką.

Przykład (P11):

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka" >


<plugins>

   <plugin name="Rozszerzenie">

         <edit name="pseudonim" caption="Pseudonim"
               width="100" top="160" />

         <layout align="client" parent="panelBottom">

            <button parent="panelBottom" name="btnWtyczka"
                    text="Kliknij" align="bottom" margins="5"
                    height="30" onclick="Klik" />

         </layout>

         <script src="plugin.pas" />

  </plugin>

</plugins>

</plugin>
```

plugin.pas

```
procedure Klik;
begin
  ShowMessage('Klik! :) ');
end;
```

Aby skorzystać ze stworzonego rozszerzenia w innym naszym formularzu, należy także użyć znacznika `<plugin>` z pełną ścieżką nazwy naszego rozszerzenia podaną w atrybucie `import`. Formularz może zatem wyglądać na przykład tak:

```
<forms>
  <form name="KontaktyForm" table="mc_kontakty"
        caption="Kontakt" width="500" height="400">
   <edit name="imie" caption="Imię" width="100" />
   <edit name="nazwisko" caption="Nazwisko" width="200" top="-1" />
   <edit name="wiek" caption="Wiek" width="80" textalign="center"
         mtype="int" />
   <combobox name="stanowisko" caption="Stanowisko" width="150">
    <option id="0">Nieznane</option>
    <option id="1">Dyrektor</option>
    <option id="2">Prezes</option>
   </combobox>
   <plugin import="Plugin.MyCompany.Wtyczka.Rozszerzenie" />
  </form>
</forms>
```

W ten sposób po wyświetleniu formularza, znajdzie się w nim oprócz pól *Imię*, *Nazwisko*, *Wiek* i *Stanowisko* - również pole *Pseudonim*. W prawym dolnym rogu okna pojawi się natomiast przycisk *Kliknij*, którego kliknięcie spowoduje wyświetlenie komunikatu.

Podłączenie się do istniejącego modułu

Definiując własną wtyczkę oprócz jej nazwy (atrybut `name`) możemy także określić miejsca, gdzie ma być ona ładowana automatycznie. Dokonujemy tego poprzez atrybut `placebyclassname`. Podajemy w nim pełną ścieżkę do formularza, w którym ma być załadowane nasze rozszerzenie. Jeśli chcemy by nasze funkcje ładowały się w wielu formularzach, podajemy wszystkie ich nazwy rozdzielając je średnikiem.

Spójrzmy na przykład (**P12**):

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka" >

<plugins>

   <plugin name="Rozszerzenie"
           placebyclassname="Plugin.MyCompany.Wtyczka.KontaktyForm">

         <edit name="pseudonim" caption="Pseudonim" width="100"
               top="160" />



         <layout align="client" parent="panelBottom">

            <button parent="panelBottom" name="btnWtyczka"
                    text="Kliknij" align="bottom" margins="5"
                    height="30" onclick="Klik" />

         </layout>

         <script src="plugin.pas" />

  </plugin>

</plugins>

</plugin>
```

Tym razem nasze rozszerzenie ponownie załaduje się w formularzu stworzonej wcześniej wtyczki. Jednakże teraz nie definiujemy tego jawnie (znacznikiem `<plugin>` w obrębie formularza), lecz na poziomie samego rozszerzenia. Podając jako wartość atrybutu `placebyclassname` ciąg `BinSoft.Invoices.SaleInvoiceForm` - nasza wtyczka ładować się będzie w oknie wystawiania faktury.

W opisany powyżej sposób możemy dodać do dowolnego istniejącego formularza własne rozszerzenia.

Przechwytywanie zdarzeń

Oprócz rozbudowy elementów interfejsu formularza poprzez nasze rozszerzenie, z poziomu kodu skryptu danego rozszerzenia możemy także przechwytywać różnego rodzaju zdarzenia. Robimy to w ten sam sposób, jak byśmy tworzyli te metody dla danego formularza. Jeśli zatem utworzymy metodę o nazwie `fOnClearForm` - będzie ona wywoływana zawsze w momencie czyszczenia formularza.

Raporty

W programach MP istnieje narzędzie *Raporty i zestawienie* dostępne w menu *Narzędzia*. Za jego pośrednictwem można tworzyć różnego rodzaju raporty, wykresy itp. W tym rozdziale wyjaśnione zostało, w jaki sposób tworzyć tego typu rozszerzenia do programów MP.

Chcąc stworzyć własny raport należy w dowolnym pliku XML utworzyć sekcję `<statements>`. Wewnątrz tej sekcji, kolejne raporty opisujemy znacznikiem `<statement>`. Dostępne są atrybuty: `caption` - tytuł raporty oraz opcjonalny `category` - kategoria raportu. Każdy z programów MP/ABC, po wywołaniu okna *Raporty i zestawienia*, w lewym panelu wyświetla wszystkie moduły (ich tytuły), które posiadają jakieś raporty, w każdym takim module wyświetla wszystkie kategorie, a w nich wszystkie raporty.

W obrębie znacznika `<statement>` możemy tworzyć formularz. Nie jest to pełny formularz, lecz jedynie obszar z filtrami, które się ukazują w górnej części raportu. W znaczniku tym powinien być również podpięty Pascal. Oto przykład:

```
<plugin name="Plugin.MyCompany.Wtyczka" caption="Moja wtyczka">

 <statements>

  <statement caption="Zestawienie" category="Zestawienia">

     <field name="ptype" text="1" />

     <combobox name="cbYear" caption="Rok"
       width="90" default="{_dateyear_}">
       <option id="2017">2017</option>
       <option id="2016">2016</option>
     </combobox>

     <combobox name="cbMonth" caption="Miesiąc" width="150"
default="{_datemonth_}" top="-1">

       <option id="1">Styczeń</option>

       <option id="2">Luty</option>

       <option id="3">Marzec</option>

       <option id="4">Kwiecień</option>

       <option id="5">Maj</option>
```

```
       <option id="6">Czerwiec</option>

       <option id="7">Lipiec</option>

       <option id="8">Sierpień</option>

       <option id="9">Wrzesień</option>

       <option id="10">Październik</option>

       <option id="11">Listopad</option>

       <option id="12">Grudzień</option>

     </combobox>

     <combobox name="cbCurrency" caption="Waluta"
         width="80" default="PLN" top="-1">
         <option table="bs_currency" id="pname">{pname}</option>
     </combobox>

     <combobox name="cbBranch" caption="Oddział" width="170"
        default="0" top="-1">
       <option id="0">Dowolny</option>
       <option table="bs_branches" where="pdel=0 AND
idcompany={_idcompany_}" orderby="pname" id="id">{pname}</option>

     </combobox>

     <combobox name="cbUser" caption="Użytkownik" width="170" default="0"
top="-1">

       <option id="0">Dowolny</option>

       <option table="bs_users" where="pdel=0 AND (idcompany IS NULL OR
idcompany=0 OR idcompany={_idcompany_})" orderby="pfirstname, plastname"
id="id">{pfirstname} {plastname}</option>

     </combobox>

     <checkbox name="cbFilters" text="Pokazuj filtry" width="200"
default="1" />

     <checkbox name="cbAddress" text="Pokazuj dane firmy" width="200"
default="1" top="-1" />

     <script src="raport.pas" />

  </statement>

 </statements>

</plugin>
```

Powyższy przykład spowoduje, że w oknie *Raporty i zestawienia* pokaże się moduł *Moja wtyczka*, w nim sekcja *Zestawienia*, a w niej raport *Zestawienie*. Kliknięcie tego raportu wyświetli formularz z filtrami widoczny w górnej części okna. Będą tam pola *COMBO* z wyborem roku, miesiąca, waluty, oddziału, użytkownika. Będą też dwa pola *CHECKBOX*, gdzie określimy czy mają być wyświetlane określone

informacje w raporcie. W przykładzie stworzono również ukryte pole `ptype`. Możemy z niego skorzystać w Pascalu podpiętym pod raport. Robi się to najczęściej wówczas, kiedy tworzymy wiele raportów, ale wszystkie maja ten sam kod Pascala. Wówczas poprzez to pole rozpoznajemy, który raport generujemy. Nazwy wszystkich pól określamy samodzielnie i nie są w żaden sposób narzucone. Musimy w Pascalu posługiwać się nimi, by uzyskać do nich dostęp.

Wyświetlając raport, użytkowniki ukażą się filtry w górnej części okna, po prawej od nich ukaże się przycisk *Generuj raport*. Taki sam przycisk pokaże się w środkowej części okna. Wywołana również zostanie funkcja `fOnShowForm` z podpiętego Pascala (o ile zostanie znaleziona). Z poziomu tej procedury możemy określić wartości domyślne dla filtrów - jeśli jest taka potrzeba. Kiedy użytkownik kliknie dowolny z przycisków *Generuj raport*, automatycznie wywołana zostanie procedura o nazwie Start - jeśli zostanie znaleziona w Pascalu. To w tej procedurze powinien znaleźć się główny kod generujący raport.

Oto prosty przykład:

```
var Browser;
   LHTML : TStringList;


procedure Start;
begin
  APP.ShowWait('Generowanie raportu...');
  APP.ShowWaitCancel(TRUE);
  LHTML := TStringList.Create;
  try
    LHTML.Add('<html>');
    LHTML.Add('<body>');
    LHTML.Add('<h1>Przykładowy raport</h1>');
    LHTML.Add('</body>');
    LHTML.Add('</html>');
    TBinFMUtils.SaveTextToUTF8File(LHTML.Text,_DIR_TMP_+'reports.html');
```

```
    Browser.LoadFromFile(_DIR_TMP_+'reports.html');
  finally
    LHTML.Free;
    APP.HideWait;
  end;
end;
begin
  Browser:=Home.FindComponent('Browser');
end;
```

Klikając w przycisk *Generuj raport* w ramach raportu wyświetli się biała strona z napisem *Przykładowy raport*.

Z raportami wiążą się następujące aspekty:

- Raport generowany jest w HTML; W oknie z raportami znajduje się przeglądarka, do której najpierw trzeba uzyskać „uchwyt". Robimy to instrukcją: `Browser:=Home.FindComponent('Browser');`
- W oknie z raportem znajdują się zakładki: `pgHTML` (zawierająca przeglądarkę) i `pgBar` (na potrzeby wykresów graficznych); Domyślnie druga z tych zakładek jest ukryta. Chcąc ją odkryć lub chcąc ukryć zakładkę HTML - podobnie jak przy `Browser` będzie trzeba uzyskać dostęp do tych zakładek.
- Generowanie raportu może być operacją czasochłonną. Dlatego dobrze jest przed nią wywołać `APP.ShowWait` by wyświetlić komunikat *Proszę czekać*. Na koniec należy ukryć ten komunikat metodą `APP.HideWait`.
- W oknie raportu, na dole, wyświetlają się przyciski *Zapisz jako PDF*, *Zapisz jako HTML*, *Zapisz jako XML*, *Zapisz jako CSV*. Domyślnie wszystkie one się wyświetlają. Należy samodzielnie je ukrywać, jeśli nie chcemy by były dostępne. Przycisk *Zapisz jako PDF* generuje PDF-a z wyświetlanego raportu przeglądarce. Kliknięcie pozostałych przycisków powoduje zapisanie raportu w określonym pliku. Podczas generowania raporty muszą już być wygenerowane.

Do przycisków przekazujemy ich ścieżki. Kliknięcie w te przyciski powoduje wyświetlenie okna dialogowego, odpowiedniego dla danego przycisku, i jedynie przekopiowanie wygenerowanego wcześniej pliku we wskazane miejsce.

- Aby nie generować oddzielnie pliku XML i HTML zalecamy wykorzystanie mechanizmu szablonów i klasy `TBinXML2HTML`. Klasa ta „pobiera" plik `XML`, plik szablonu `TPL` i generuje plik `HTML`.

Spójrzmy na bardziej zaawansowany raport.

```
var Browser : THtmlEditor;
    LXML,LCSV,barXML : TStringList;
    pType, lUser,lCompany,lBranch,lYear,lMonth,pFilters,pAddress : Integer;
    lCurrency : String;
    showReport,showChart : Boolean;
    pgB,pgR,lyB;

//-------------------------------------------------------------

function SprzedazMiesiac:Boolean;
var T : TBinTable;
    D,D1,D2 : TDateTime;
    m : Integer;
    DD : Integer;
    GSum,GLoc,LPrc,LVal,GSTemp : Double;
    Q,cN : String;
begin
  Result:=FALSE;

  LXML.Add('<title>Przychód: '+APP.LocalData.getMonthName(lMonth)+'
'+I2S(lYear)+'</title>');

  barXML.Add('<chart name="chart" align="client" title="Przychód w roku
'+APP.LocalData.getMonthName(lMonth)+' '+I2S(lYear)+'" skin="auto"
delayed="true" thread="true">');
  barXML.Add('<series>');
  barXML.Add('<serie title="Przychód" type="bar">');
```

```
  LXML.Add(' <legend>');

  LXML.Add('  <item name="date"
caption="Okres">'+APP.LocalData.getMonthName(lMonth)+' '+I2S(lYear)+'</
item>');

  LXML.Add('  <item name="currency" caption="Waluta">'+lCurrency+'</
item>');

  if lUser>0 then LXML.Add('  <item name="employee"
caption="Pracownik">'+GetValueV('cbUser','')+'</item>');

  if lBranch>0 then LXML.Add('  <item name="branch"
caption="Oddział">'+GetValueV('cbBranch','')+'</item>');

  LXML.Add(' </legend>');



  Q:='(nstatus=2 OR nstatus=3) AND nreturned=0 AND nnoinstat!=1 AND
ncurrency='''+lCurrency+'''';

  Q:=Q+' AND ntype=0 AND nsubtype>=1 AND nsubtype<=8';

  Q:=Q+' AND idcompany='+I2S(lCompany);

  if lBranch>0 then Q:=Q+' AND idbranch='+I2S(lBranch);

  if lUser>0 then Q:=Q+' AND idowner='+I2S(lUser);



  LXML.Add(' <header>');

  LXML.Add('  <item name="day">Dzień</item>');

  LXML.Add('  <item name="prc">Udział</item>');

  LXML.Add('  <item name="value">Wartość</item>');

  LXML.Add(' </header>');

  LXML.Add(' <data>');


  LCSV.Add('Dzień;Udział;Wartość');



  D:=EncodeDate(lYear,lMonth,1);



  cN:='nstotal_n';



  D1:=TBinFMUtils.getDate('monthstart',D);

  D2:=TBinFMUtils.getDate('monthend',D);

  DD:=TBinFMUtils.DaysInMonth(D);
```

```
  GSum:=BinDB.GetValueD('SELECT sum('+cN+') as suma FROM bs_invoices WHERE
'+Q+' AND ndate_issue>='+BinDB.AddDate(D1)+' AND
ndate_issue<='+BinDB.AddDate(D2),0);

  GLoc:=0;



  for m:=1 to DD do

  begin

      D1:=IncDay(D,m-1);



      APP.ProcessMessages;

      if APP.WaitCancelStatus then Exit;



      LVal:=BinDB.GetValueD('SELECT sum('+cN+') as suma FROM bs_invoices
WHERE '+Q+' AND ndate_issue>='+BinDB.AddDate(D1)+' AND
ndate_issue<='+BinDB.AddDate(D1),0);



      if GSum>0 then LPrc:=(LVal/GSum)*100 else LPrc:=0;



      LXML.Add(' <row lp="'+IntToStr(M)+'">');

      LXML.Add('   <item name="day" align="ac">'+DateToStr(D1)+'</item>');

      LXML.Add('   <item name="prc" align="ac">'+CVisS2S(CorrectD2S(LPrc))
+' %</item>');

      LXML.Add('   <item name="value"
align="price">'+CVisS2S(CorrectD2S(LVal))+'</item>');

      LXML.Add(' </row>');



      LCSV.Add(DateToStr(D1)+';'+CVisS2S(CorrectD2S(LPrc))
+';'+CVisS2S(CorrectD2S(LVal)));

      barXML.Add('<value label="'+DateToStr(D1)+'" y="'+CorrectD2S(LVal)+'"
/>');



      GLoc:=GLoc+LVal;

  end;

  LXML.Add(' </data>');

  LXML.Add(' <footer>');

  LXML.Add('  <item name="day" caption="Łącznie" align="ar"
colspan="1">Łącznie:</item>');

  LXML.Add('  <item name="prc" caption="Prc" align="price" colspan="1">100
%</item>');
```

```
  LXML.Add('  <item name="value" caption="Suma" align="price"
colspan="1">'+CVisS2S(CorrectD2S(GSum))+'</item>');
  LXML.Add(' </footer>');


  barXML.Add('</serie>');
  barXML.Add('</series>');
  barXML.Add('</chart>');


  showChart:=TRUE;


  Result:=TRUE;
end;


procedure Start;
var R : TBinFillTPL;
    T : TBinTable;
    W : Boolean;
begin
  LXML:=TStringList.Create;
  LCSV:=TStringList.Create;
  barXML:=TStringList.Create;


  APP.ShowWait('Generowanie raportu...');
  APP.ShowWaitCancel(TRUE);
  try
      //pobranie filtrów
      pType:=GetValueL('ptype',0);
      lUser:=GetValueL('cbUser',0);
      lBranch:=GetValueL('cbBranch',0);
      lCompany:=_idcompany_;
      lYear:=GetValueL('cbYear',2017);
      lMonth:=GetValueL('cbMonth',1);
      lCurrency:=GetValue('cbCurrency','PLN');
      pFilters:=GetValue('cbFilters',1);
      pAddress:=GetValue('cbAddress',0);
```

```
      //ukrywamy przyciski eksportu
      BtnVisible('btnSaveHTML',false);
      BtnVisible('btnSaveXML',false);
      BtnVisible('btnSaveCSV',false);

      //domyślnie pokazujemy raport HTML, ukrywamy wykresy
      showReport:=TRUE;
      showChart:=FALSE;

      //budujemy XML-a
      LXML.Add('<report>');

      //przekazujemy do XML-a dane firmy
      T:=BinDB.GetRow('SELECT * FROM bs_company WHERE id='+I2S(lCompany));
      if T<>nil then
      begin
        LXML.Add(T.ConvertToXML('',FALSE,'company'));
        T.Free;
      end;

      //w zależności, który wybraliśmy raport - wywołujemy odpowiednią
funkcję
      W:=FALSE;
      try
          case pType of
          1 : W:=SprzedazMiesiac;
          end;
      except
        APP.HideWait;
        APP.ShowMessageError('Wystąpiły problemy podczas przetwarzania
danych. %1%#'+_NL+'Msg:'+APP.TTLP(LastExceptionMessage));
      end;

      LXML.Add('</report>');
```

```
    //jak nie udało się wygenerować raportu - kończymy
    if not W then Exit;

    //zapisujemy wygenerowane XML i CSV do plików w folderze tymczasowym
    TBinFMUtils.SaveTextToUTF8File(LXML.Text,_DIR_TMP_+'reports.xml');
    TBinFMUtils.SaveTextToUTF8File(LCSV.Text,_DIR_TMP_+'reports.csv');

    //generujemy wersje HTML. Używamy klasy TBinXML2HTML
    //przetwarzamy szablon reports.tpl, z danymi z reports.xml
    //i zapisujemy do reports.html
    R:=TBinXML2HTML.Create(BinDB,nil,nil);
    R.AddVariable('_selfdir_',_DIRSELF);
    R.AddVariable('_showaddress_',I2S(pAddress));
    R.AddVariable('_showfilters_',I2S(pFilters));
    try
       R.LoadTemplate(_DIRSELF+'reports.tpl');
       R.LoadData(_DIR_TMP_+'reports.xml');
       R.SaveResult(_DIR_TMP_+'reports.html');
    finally
       R.Free;
    end;

    //ustawiamy ścieżki dla określonych przycisków i je wyświetlamy
    BtnHint('btnSaveHTML',_DIR_TMP_+'reports.html');
    BtnVisible('btnSaveHTML',true);

    btnHint('btnSaveXML',_DIR_TMP_+'reports.xml');
    BtnVisible('btnSaveXML',true);

    btnHint('btnSaveCSV',_DIR_TMP_+'reports.csv');
    BtnVisible('btnSaveCSV',true);

    //jak mamy pokazać raport w HTML, to go ładujemy
    if showReport then
    begin
```

```
         Browser.LoadFromFile(_DIR_TMP_+'reports.html');
      end;

      //jak mamy pokazać raport (wykres), to go również ładujemy
      if showChart then
      begin
          FormGetHandler().DeleteAllFieldsOnObject(lyB);
          FormGetHandler().ParseXMLOnObject(barXML.Text,lyB)
      end;

      //określamy widoczność zakładek
      pgR.Visible:=showReport;
      pgB.Visible:=showChart;
  finally
    LXML.Free;
    LCSv.Free;
    barXML.Free;
    APP.HideWait;
  end;
end;



begin
  Browser:=Home.FindComponent('Browser');
  pgB:=Home.FindComponent('pgBar');
  pgR:=Home.FindComponent('pgHTML');
  lyB:=Home.FindComponent('LayoutBar');
end;
```

Wyjaśniając poszczególne fragmenty:

1. W sekcji `begin...end` na końcu pliku uzyskujemy dostęp do przeglądarki (`Browser`), zakładek *Zestawienie* i *Wykres* (`pgBar`, `pgHTML`), *layoutu* w zakładce *Wykres* (`LayoutBar`).

2. Generowanie raportu polega na generowaniu zawartości dla plików `XML`, `CSV` oraz kodu do wykresu. Zapamiętujemy to w obiektach `LXML`, `LCSV`, `barXML`.

3. Pascal przygotowany jest tak, by można go było współdzielić dla wielu raportów. Typ raportu pobieramy z pola `ptype` i w zależności od niego wywołujemy odpowiednią funkcję. W przykładzie dla typu `ptype=1` wywołujemy funkcję `SprzedazMiesiac`. To ona musi wygenerować właściwy raport. Ponieważ jedne raporty będą posiadały wykresy, inne nie, stworzyliśmy zmienne `showReport` i `showChart`. Domyślnie pierwsza ma wartość `true`, a druga `false`. W funkcji `SprzedazMiesiac` zmieniamy wartość `showChart` na `true` - gdyż ten raport posiada wykres. W innych funkcjach, gdy byśmy je przygotowywali w przyszłości, będziemy mogli odpowiednio wpływać na te zmienne.

4. Przed generowaniem raportu ukrywamy przyciski do zapisywania ich w różnych formatach. Używamy w tym celu funkcji `BtnVisible`. Należy tu użyć nazw jakie zostały przypisane tym przyciskom. Te są już wbudowane w programy MP i brzmią dokładnie tak, jak podano w przykładzie.

5. Po wygenerowaniu raportu funkcją `SprzedazMiesiac` - zapisujemy pliki `XML`, `CSV` do folderu tymczasowego, generujemy wersję HTML z pomocą klasy `TBinXML2HTML` i również zapisujemy do folderu tymczasowego. Następnie pliki te podpinamy pod przyciski do zapisywania i je odkrywamy.

6. Jeśli `showReport=true` - czyli mamy pokazać raport w HTML - ładujemy go do przeglądarki.

7. Jeśli `showChart=true` - czyli mamy pokazać raport w postaci wykresu, jego również ładujemy. W tym celu korzystamy z kilku mechanizmów.

   7.1. Funkcja `FormGetHandler` zwraca „uchwyt” do bieżącego formularza;

7.2. Metoda formularza `ParseXMLOnObject` pozwala na parsowanie plików XML. Podajemy jej w parametrach kod XML oraz miejsce, w którym ma wygenerować formularz z przekazanego XML-a. W przykładzie, w zmiennej `barXML.Text` mamy zapamiętany XML ze znacznikiem `<chart>` rysującym wykres. Umieszczamy go w layoucie `lyB`.

7.3. Metoda formularza `DeleteAllFieldsOnObject` usuwa wszystkie kontrolki znajdujące się na obiekcie, który podamy w parametrze. W przykładzie usuwamy wszystko layoutu o nazwie `lyB`.

8. Określamy widoczność zakładek *Zestawienie* i *Wykres*.

9. W funkcji `SprzedazMiesiac` generujemy właściwy raport. Generujemy go w `LXML` - w formacie XML, `LCSV` - w formacie CSV i `barXML` - jako XML do wykresu (znacznik `<chart>`). Generowanie raportu może być operacją czasochłonną, dlatego w pętli wywołujemy metodę `APP.ProcessMessages;` - która przetwarza tzw. pętlę komunikatów programu i sprawia, że program nie przestanie odpowiadać. Metoda `APP.WaitCancelStatus` zwraca `true`, kiedy użytkownik kliknie *Anuluj* w widocznym oknie *Proszę czekać*. Gdy tak się stanie, przerywamy generowanie raportu.

Chcąc dokładnie zrozumieć zasadę działania powyższego przykładu należy jeszcze zobaczyć, jak wygląda wygenerowany XML dla wykresów oraz jak działa klasa `TBinXML2HTML`.

W przykładzie generowany jest kod XML wstawiający wykres w zakładce *Wykres*. Kod ten, to zwykły kod formularza - jak w innych miejscach programu MP. W tym jednak przypadku składa się on tylko ze znacznika `<chart>`. W naszym przykładzie może mieć on postać:

```
<chart name="chart" align="client" title="Przychód w roku Listopad 2017"
skin="auto" delayed="true" thread="true">

<series>
```

```
<serie title="Przychód" type="bar">
<value label="2017-11-01" y="0.00" />
<value label="2017-11-02" y="0.00" />
<value label="2017-11-03" y="0.00" />
<value label="2017-11-04" y="0.00" />
<value label="2017-11-05" y="0.00" />
<value label="2017-11-06" y="0.00" />
<value label="2017-11-07" y="0.00" />
<value label="2017-11-08" y="0.00" />
<value label="2017-11-09" y="0.00" />
<value label="2017-11-10" y="0.00" />
<value label="2017-11-11" y="0.00" />
<value label="2017-11-12" y="0.00" />
<value label="2017-11-13" y="34.00" />
<value label="2017-11-14" y="0.00" />
<value label="2017-11-15" y="0.00" />
<value label="2017-11-16" y="0.00" />
<value label="2017-11-17" y="0.00" />
<value label="2017-11-18" y="0.00" />
<value label="2017-11-19" y="0.00" />
<value label="2017-11-20" y="0.00" />
<value label="2017-11-21" y="0.00" />
<value label="2017-11-22" y="0.00" />
<value label="2017-11-23" y="0.00" />
<value label="2017-11-24" y="0.00" />
<value label="2017-11-25" y="0.00" />
<value label="2017-11-26" y="0.00" />
<value label="2017-11-27" y="0.00" />
<value label="2017-11-28" y="0.00" />
<value label="2017-11-29" y="0.00" />
<value label="2017-11-30" y="0.00" />
</serie>
</series>
</chart>
```

Znacznik <chart> wstawia wykres. W obrębie tego wykresu może znajdować się wiele serii (opisanych znacznikiem <series>). W ramach każdej serii umieszczamy „punkty" na wykresie. Używamy w tym celu znacznika <value>. Przekazujemy mu dwie wartości: label (etykietę) i y (wartość).

Klasa TBinXML2HTML

W przykładzie wykorzystano klasę TBinXML2HTML, która służy do generowania kodu HTML na podstawie szablonu (TPL) oraz danych zapisanych w pliku XML. Użycie tej klasy wygląda następująco:

```
R:=TBinXML2HTML.Create(BinDB,nil,nil);
R.AddVariable('_selfdir_',_DIRSELF);
R.AddVariable('_showaddress_',I2S(pAddress));
R.AddVariable('_showfilters_',I2S(pFilters));
try
   R.LoadTemplate(_DIRSELF+'reports.tpl');
   R.LoadData(_DIR_TMP_+'reports.xml');
   R.SaveResult(_DIR_TMP_+'reports.html');
finally
   R.Free;
end;
```

W konstruktorze tej klasy, pierwszy z parametrów to „uchwyt" do bazy danych. Dzięki temu w szablonie będziemy mogli pobierać dane bezpośrednio z bazy danych. W naszym przykładzie tak nie będzie, ale istnieje taka możliwość. Metodą AddVariable możemy przekazać do generatora dodatkowe zmienne. W naszym przykładzie przekazujemy ścieżkę do folderu, z którego uruchomiony jest raport oraz status *checkboxów*. Metodą LoadTemplate() ładujemy szablon, metodą LoadData() - ładujemy dane w XML, a metodą SaveResult() zapisujemy wygenerowany dokument.

Spójrzmy jak wyglądają dane w formacie XML wygenerowane przez nasz raport i przekazane do tej klasy.

```
<report>
<company>
 <id>1</id>
 <add_id_user>1</add_id_user>
 <add_time>2015-10-09 11:21:15</add_time>
 <modyf_id_user>1</modyf_id_user>
 <modyf_time>2017-08-03 09:44:24</modyf_time>
 <pname>BinSoft</pname>
 <pstreet>Podgórna</pstreet>
 …
</company>

<title>Przychód: Listopad 2017</title>
 <legend>
  <item name="date" caption="Okres">Listopad 2017</item>
  <item name="currency" caption="Waluta">PLN</item>
 </legend>
 <header>
  <item name="day">Dzień</item>
  <item name="prc">Udział</item>
  <item name="value">Wartość</item>
 </header>
 <data>
 <row lp="1">
   <item name="day" align="ac">2017-11-01</item>
   <item name="prc" align="ac">0,00 %</item>
   <item name="value" align="price">0,00</item>
 </row>
 <row lp="2">
   <item name="day" align="ac">2017-11-02</item>
   <item name="prc" align="ac">0,00 %</item>
   <item name="value" align="price">0,00</item>
```

```
 </row>
…
 </data>
 <footer>
  <item name="day" caption="Łącznie" align="ar" colspan="1">Łącznie:</item>
  <item name="prc" caption="Prc" align="price" colspan="1">100 %</item>
  <item name="value" caption="Suma" align="price" colspan="1">34,00</item>
 </footer>
</report>
```

(Powyżej pokazano jedynie fragment tego pliku)

Spójrzmy też na szablon `reports.tpl` użyty w szablonie.

```
<html>
<head>
  <title>{_title_}</title>
   <style type="text/css">
    *, body, table, th, td { font-size:10px;font-family:arial; }
    body { margin:10px; }
    h1 { font-size:25px; }
    table.dtable { width:100%;border-top:1px solid black;border-left:1px
solid black;border-spacing:0px;border-collapse: collapse;margin:0px 0px 5px
0px; }
    table.dtable th, table.dtable td { border-right:1px solid black;border-
bottom:1px solid black;padding:2px; }
    table.dtable th { background:#c0c0c0; }
    table.dtable tr.tr0 { background:#ffffff; }
    table.dtable tr.tr1 { background:#f0f0f0; }
    table.dtable .price { text-align:right; }
    table.dfilter { width:300px !important; }
    .ar { text-align:right !important; }
    .al { text-align:left !important; }
    .ac { text-align:center !important; }

    .fl { width:120px; }
  </style>
```

```
</head>
<body>
  {@if _showaddress_=1}
   <table class="dtable dfilter">
   <tr><th>{company.pname}</th></tr>
   <tr><td>
   {company.ppostcode} {company.ppost}<br />
   {company.pstreet} {company.pstreet_n1}<br />
   NIP: {company.pnip}<br />
   </td></tr>
   </table>
  {@/if}

  <h1>{title}</h1>

  {@if _showfilters_=1}
  <table class="dtable dfilter">
  <tr><th colspan="2">Filtry</th></tr>
  {@repeat legend.item}
  <tr><th class="ar fl">{$caption}</th><td class="ac">{$:value:}</td></tr>
  {@/repeat}
  </table>
  {@/if}

  <p> </p>

  <table class="dtable">
  <tr>
  {@repeat header.item}
  <th>{$:value:}</th>
  {@/repeat}
  </tr>
  {@repeat data.row}
  <tr>
    {@repeat item}
```

```
    <td class="{$align}">{$:value:}  </td>
    {@/repeat}
  </tr>
  {@/repeat}
  <tr>
  {@repeat footer.item}
  <th colspan="{$colspan}" class="{$align}">{$:value:}  </th>
  {@/repeat}
  </tr>
  </table>
  <p> </p><p> </p>
</body>
</html>
```

Jak widzimy, jest to zwykły kod HTML, w którym można odwoływać się do danych w podpiętym pliku XML używając specjalnej składni.

- Chcąc odwołać się do elementów z podpiętego XML-a należy podać do nich „ścieżkę" w nawiasach klamrowych. Nie podajemy znacznika głównego (root), lecz od kolejnego zagłębienia. W naszym przykładzie znacznikiem *root* jest znacznik `<report>`. Przykłady:
  - `{title}` - wstawi zawartość znacznika `<title>` (umieszczonego wewnątrz `<report>` - tego jednak nie podajemy, gdyż jak wspominano - pomijamy *root*-a);
  - `{company.pname}` - wstawi zawartość znacznika `<pname>` umieszczonego wewnątrz znacznika `<company>`;
  - `{legend.item.$caption}` - wstawi zawartość atrybutu `caption` ze znacznika `<item>` będącego w znaczniku `<legend>`. Ponieważ w znaczniku `<legend>` jest więcej znaczników `<item>` - użyty zostanie pierwszy ze znaczników;

~ `legend.item[1].$caption}` - wstawi zawartość atrybutu `caption` z drugiego znacznika `<item>` (numerujemy od 0) będącego w znaczniku `<legend>`;

- Oprócz odwoływania się do struktury XML możemy również stosować specjalne komendy. Mają one postać `{@...}`.
- `{@if ...} ... {@/if}` - składnia instrukcji IF. Do generowanego szablonu wstawi się umieszczony w obrębie tych znaczników kod tylko wówczas, kiedy warunek będzie spełniony. W warunku można użyć zmiennych (jak przykładzie `{@if _showaddress_=1}`) lub odwoływać się do struktury XML.
- `{@repeat ...} ... {@/repeat}` - wykona „pętlę". W parametrze podajemy ścieżkę do znacznika. Pętla będzie przechodziła po wszystkich znacznikach danego typu na tym samym zagłębieniu. Dla każdej pętli wstawiana będzie zawartość umieszczona pomiędzy tymi sekwencjami. Na przykład: `{@repeat data.row}...{@/repeat}` - wykona pętlę po wszystkich wierszach (znacznikach `<row>`) wewnątrz znacznika `<data>`. Wewnątrz pętli można odwoływać się do kolejnych znaczników. Podaje się wówczas ścieżkę względem `data.row`. Chcąc odwołać się do zawartości znacznika z pętli używamy `{$:value:}`. Chcąc odwołać się do atrybutów znacznika z pętli używamy sekwencji `{@atrybut}`.
- Zarówno warunki `{@if}`, jak i pętle `{@repeat}` można dowolnie zagłębiać.

Jak to zrobić?

W tym rozdziale przedstawione zostaną różne szablony kodu oraz wyjaśnienia - sposobu rozwiązania często występujących sytuacji. W przypadku większości precyzyjnych zapytań kierowanych do naszego działu wsparcia, odpowiedzi będą formułowane właśnie w postaci kolejnych „pytań i odpowiedzi" w tym rozdziale.

Jak zmienić ikonę formularza lub widoku?

Znaczniki `<form>`, `<view>` oraz `<item>` (w gałęzi `<category>`) posiadają atrybut `image` pozwalający na ustalenie ikony - odpowiednio dla formularza, widoku oraz elementu kategorii menu. Jako wartość tego atrybutu należy podać nazwę ikony „wbudowanej" w program lub nazwę pliku graficznego, który ma zostać załadowany. Nazwę pliku podajemy bez rozszerzenia, a sam plik powinien być w formacie PNG. Przykład:

```
<form image="ikona" … />
```

Powyższy fragment znacznika spowoduje, że w okienku formularza, w miejscu ikony, załaduje się plik graficzny `ikona.png`. Będzie on poszukiwany w katalogu, w którym znajduje się plik XML z powyższym znacznikiem.

Ciemna i jasna skórka

Wszystkie skórki dostępne w programie można podzielić na „jasne" i „ciemne". Dobrze jest również przygotowywać swoje ikony w obu takich odmianach. Dlatego podając nazwę ikony w atrybucie `image`, oprogramowanie BSX szuka w katalogu z danym plikiem XML w pierwszej kolejności plik o tej nazwie z przyrostkiem „dark" lub „light" - w zależności od aktualnej skórki. I dopiero jak nie zostanie odnaleziony, szuka pliku wprost o nazwie podanej przez użytkownika.

Jak dodać pole checkbox w widoku?

Widoki BSX pozwalają na dodawanie pól *checkbox* (wyboru) w dowolnej kolumnie. Aby tego dokonać, do widoku musi być podpięty skrypt, a w nim stworzone funkcje obsługujące zdarzenia: `fOnCellClass()` oraz `fOnCellProperties()`.

Pierwszym ze zdarzeń określamy tzw. klasę komórki. Dostępne są klasy:

- `checkbox` - pole typu *checkbox*,
- `radio` - pole typu *radio*,
- `button` - pole z przyciskiem,
- `comment` - pole z komentarzem,
- `bitmap` - pole z ikonką,
- `progress` - pasek postępu;

Ustalenie odpowiedniej klasy (poprzez zwrócenie jej nazwy w danej funkcji) - spowoduje pojawienie się odpowiedniego elementu w danej komórce. Zatem oprócz możliwości wyświetlenia pola *checkbox*, możemy także wyświetlić pole *radio*, przycisk, bitmapę, czy też pasek postępu. Aby modyfikować odpowiedni z tych elementów, używamy zdarzenia `fOnCellProperties()`. W zdarzeniu tym otrzymuje „uchwyt" do komórki (`Cell`), poprzez który możemy ją modyfikować. Oto przykład:

```
function fOnCellClass(Sender, ACol,ARow, IDName):String;
begin
  if IDName='pname' then Result:='checkbox';
end;


function fOnCellProperties(Sender, Cell, ACol, ARow, IDName, ID):String;
begin
  if IDName='pname' then
```

```
  begin
        Cell.Checkbox.IsChecked:=true;
  end;
end;
```

W zależności od klasy komórki, zmienna cell będzie zawierała odpowiednią właściwość powiązaną z wyświetlanym elementem, tj.:

- `Checkbox` - dla pól typu *checkbox*,
- `ProgressBar` - dla pól z paskami postępu;

Należy zwrócić uwagę, że obiekty wyświetlane w komórkach tabeli widoku, nie są „wstawiane" w danej kolumnie do wszystkich wierszy, a jedynie do tych widocznych. Kiedy zatem zawartość tabeli jest przewijana, zdarzenie `fOnCellProperties()` jest wywoływane dla każdej widocznej komórki, aby można było zaktualizować jej powiązany obiekt.

Jak dodać pasek postępu do widoku?

Patrz „*Jak dodać pole checkbox do widoku*?"

Jak określić domyślną kolumnę, po której mają być sortowane dane?

Należy w atrybucie `orderby` znacznika `<query>` podać nazwę kolumny. Można w nim również użyć dowolnego innego zapisu SQL, np.: `orderby="nazwisko, id DESC"`.

Czy można zablokować kolumny przed zmianą ich szerokości?

Tak. Należy dla znacznika `<view>` podać

atrybut `canresizecolumns="false"`.

Struktura danych

Chcąc samodzielnie rozbudowywać programy MP/ABC bardzo często będzie potrzeba samodzielnego modyfikowania tabel tych programów. W tym rozdziale zostały one opisane. Ponieważ jednak tabel tych jest tak dużo, a każda z nich składa się z dziesiątek pól, w rozdziale tym opisane zostaną tylko niektóre z tabel i tylko w ograniczonym zakresie.

Uwaga! Mając wyświetlony monitor Developera można w konsoli wpisać komendę `table("nazwaTabeli")` - wyświetlony zostanie opis danej tabeli.

Ogólne nazewnictwo

Istnieje kilka zasad wspólnych stosowanych w strukturze danych programów MP/ABC. Oto najważniejsze z nich:

- wszystkie standardowe tabele wchodzące w skład programów MP/ABC mają przedrostek `bs_`;
- każda tabela posiada klucz główny, którym jest kolumn `ID` (`INT`, `AUTO_INCREMENT`);
- większość tabel posiada standardowe kolumby: `add_id_user` (ID użytkownika, który dodał dany rekord), `add_time` (data dodania rekordu), `modyf_id_user` (ID użytkownika, który modyfikował dany rekord), `modyf_time` (data ostatniej modyfikacji);
- niektóre tabele posiadają opis pewnych dokumentów, np. `bs_invoices` (dokumenty handlowe), `bs_orders` (zamówienia), `bs_offers` (oferty), `bs_docsst` (dokumenty magazynowe) itp. Na tych dokumentach znajdują się „pozycje". Te zapisywane są w tabelach o analogicznych nazwach, lecz z przyrostkiem `_pr`, np.: `bs_invoices_pr` (pozycje na dokumentach handlowych), `bs_orders_pr` (pozycje na zamówieniach), `bs_offers_pr`

(pozycje na ofertach), `bs_docsst_pr` (pozycje na dokumentach magazynowych) itp. Kolumną wiążącą pozycje z dokumentem zazwyczaj jest `id-doc`;

- niektóre z ważnych tabel *systemowych*:
  - ~ `bs_company` - lista wszystkich firm;
  - ~ `bs_branches` - lista oddziałów w firmach;
  - ~ `bs_users` - lista użytkowników (i pracowników);

Ponieważ w jednej bazie danych można mieć wiele firm, a w tych wiele oddziałów i użytkowników, w wielu tabelach można odnaleźć kolumny: `idcompany` (ID firmy, której dotyczy dany wpis), `idbranch` (ID oddziału), `iduser/id-owner` (ID użytkownika); W tabelach tych można odnaleźć kolumnę `pdel`, która mówi czy dany rekord (firma, oddział, użytkownik) nie został usunięty (`pdel=1`). Usuwając bowiem dane z tych tabel, dla zachowania spójności danych, nie są one fizycznie usuwane a jedynie oznaczane jako usunięte;

- większość kolumn w tabelach posiada pewien *prefiks*, którym jest pojedyncza litera `p`, `n` itp. Stosowane jest to po to, by nie tworzyć kolumn, które mogłyby być słowami kluczowymi w danym systemie bazodanowym; dlatego nie ma kolumn `name`, `date`, `time`, tylko `pname`, `pdate`, `ndate_issue` itp.
- w tabeli `bs_contractors` - znajduje się opis kontrahentów; Kontrahenci są związani z większością danych w programie; Kontrahenta wybieramy w fakturach, zamówieniach, ofertach, dokumentach magazynowych, zadaniach itp. Dane adresowe kontrahentów umieszczane są w kolumnach: `pname` (nazwa), `pstreet` (ulica), `pstreet_n1` (numer budynku), `ppostcode` (kod pocztowy), `ppost` (poczta), `pcity` (miejscowość), `pprovince` (województwo), `pdistrict` (powiat), `pphone1` (telefon), `pemal` (e-email), `pnip` (numer NIP), `pregon` (numer REGON) itp. Z kontrahentem związane są również inne dane, np. dane adresu korespondencji; Użyto w tym celu tych

samych nazw kolumn, tylko zamiast przedrostka `p`, użyto `k`; W systemie BSX stosuje się zasadę *nadmiarowości*, tzn. tworząc jakiś dokument, na którym znajdą się dane kontrahenta - powiązanie z kontrahentem następuje poprzez kolumnę `pidcontractor`, ale oprócz tego kopiuje się do danego rekordu aktualną zawartość wszystkich danych adresowych kontrahenta; Na przykład, w tabeli `bs_invoices` (dokumenty handlowe) odnajdziemy kolumnę `pidcontractor`, ale też `pname`, `pstreet`, `pcity`, `kname`, `kstreet` itd. Dzięki temu można wystawić dokument dla kontrahenta, którego nie wprowadzimy do bazy kontrahentów (`pidcontractor=NULL`), jak również można wystawić dane dla istniejącego kontrahenta ale z innymi danymi adresowymi; Również zmiana danych adresowych kontrahenta nie wpływa na zmianę tych danych we wszystkich istniejących dokumentach; Często w dokumentach handlowych (`bs_invoices`), zamówieniach (`bs_orders`), dokumentach magazynowych (`bs_docsst`) i innych - oprócz danych „nabywcy" pojawiają się dane sprzedawcy; Te pochodzą z `bs_company`, są jednak kopiowane do kolumn `sname`, `sstreet`, `snip` itp. Są zatem również powielane do wszystkich dokumentów; To także sprawia, że zmiana danych adresowych firmy nie wpływa na wszystkie wystawione dokumenty;

- inne ważne tabele *systemowe* to m.in.:
  - `bs_sessions` - sesje użytkowników; kiedy użytkownik loguje się do bazy tworzony jest mu wpis w tej tabeli; gdy się wylogowuje - wpis zostanie usuwany;
  - `bs_settings` - różne ustawienia w programie;
  - `bs_symbols` - numeracje dokumentów;
- w programach sprzedażowo/magazynowych serii MP/ABC można mieć włączoną gospodarkę magazynową lub tzw. prosty indeks produktów; W przy-

padku prostego indeksu produktów - informacje o tych produktach przechowywane są w tabeli `bs_simplestock`; w przypadku pełnej gospodarki magazynowej - informacje o produktach pochodzą z tabeli `bs_stockindex`; Dodając pozycje do dokumentów (np. handlowych, zamówień, czy magazynowych), a więc tworząc wpisy do tabel `bs_invoices_pr`, `bs_orders_pr`, czy `bs_docsst_pr` - w tabelach tych znajdziemy zatem kolumny: `idsimpleproduct` oraz `idproduct`. Pierwsza z nich jest relacją do `bs_simplestock`, druga do `bs_stockindex`;

- większość dokumentów w programie posiada swoje statusy; Zazwyczaj użyta jest do tego kolumna `nstatus`. Statusy często wpływają na to, jak dany dokument jest traktowany; Jeśli dokument nie jest *Zatwierdzony* (najczęściej `nstatus=2`), nie jest uwzględniany przez program; Zatem zazwyczaj dokument handlowy tego typu nie jest uwzględniany w zestawieniach i raportach, w numeracjach dokumentów itp., a niezatwierdzony dokument magazynowy nie wpływa na gospodarkę magazynową;
- przy włączonej gospodarce magazynowej - produkty pobierane są z tabeli `bs_stockindex`; Jednakże z gospodarką magazynową wiąże się więcej tabel; Najważniejsze z nich to:
  - `bs_stocks` - lista wszystkich magazynów;
  - `bs_pricing` - lista wszystkich cenników;
  - `bs_stockrel` - stany magazynowe;
  - `bs_stockprice` - ceny produktów;
  - `bs_docsst` - dokumenty magazynowe;
  - `bs_docsst_pr` - pozycje na dokumentach magazynowych;
  - `bs_docspzwz` - powiązanie pomiędzy dokumentami PZ i WZ;
-

Zakończenie

Niniejszy podręcznik nie wyczerpuje wszystkich funkcjonalności jakie oferuje system BSX, a jedynie zakreśla podstawowe jego możliwości. Stale będzie jednak rozwijany i udoskonalany. W przypadku braku opisu określonych funkcji prosimy o kontakt mailowy.

e-mail: info@binsoft.pl