Politechnika Lubelska

PODRĘCZNIKI

Jakub Szabelski, Łukasz Sobaszek

Podstawy programowania w zastosowaniach inżynierskich

Visual Basic

Lublin 2023

Podstawy programowania w zastosowaniach inżynierskich

Visual Basic

Podręczniki – Politechnika Lubelska

LUBLIN UNIVERSITY
OF TECHNOLOGY
MECHANICAL
ENGINEERING FACULTY

Jakub Szabelski, Łukasz Sobaszek

Podstawy programowania w zastosowaniach inżynierskich

Visual Basic

POLITECHNIKA
LUBELSKA
WYDAWNICTWO

Lublin 2023

Recenzenci:
prof. dr hab. inż. Gabriel Kost, Politechnika śląska
dr hab. inż. Paweł Sitek, Politechnika Świętokrzyska

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-7947-567-4

Wydawca: Wydawnictwo Politechniki Lubelskiej
www.wpl.pollub.pl
ul. Nadbystrzycka 36C, 20-618 Lublin
tel. (81) 538-46-59
Druk: Agencja Reklamowa TOP Agnieszka Łuczak
www.agencjatop.pl

Nakład: 50 egz.

Spis treści

Streszczenie **9**

Abstract **10**

1. Wstęp **11**

2. Instalacja Visual Studio **13**

3. Tworzenie nowego projektu **15**

4. Resetowanie układu okien **17**

5. Panelowa struktura okna Visual Studio **19**

6. Zapisywanie aplikacji **25**

Ćwiczenie 1. Podstawy budowy aplikacji 26

Ćwiczenie 2. Podstawy tworzenia kodu 27

7. Szybkie uruchamianie aplikacji w środowisku Windows **29**

Ćwiczenie 3. *Spider's Legs* – podstawy programowania 29

8. Definiowanie innych zdarzeń **33**

9. Właściwości kontrolek **35**

Ćwiczenie 4. *PictureBox* 37

Ćwiczenie 5. Modyfikacja aplikacji *Spider's Legs* 39

Ćwiczenie 6. *Menu Format* 39

Ćwiczenie 7. Przyciski – kolory 41

10. Notacja węgierska **43**

Ćwiczenie 8. Przyciski – widoczność 43

Ćwiczenie 9. Etykiety – tekst 45

11. System podpowiedzi kontekstowych w Visual Studio **47**

12. Pole tekstowe – *TextBox* 49

Ćwiczenie 10. Pole tekstowe – wyrównanie tekstu 50

13. Okno informacyjne – ***MessageBox*** **51**

14. Kontrolka wyboru daty i czasu – ***DateTimePicker*** **53**

Ćwiczenie 11. *DateTimePicker* i *MessageBox* 53

15. Tablica znaków ASCII **55**

16. Kolejność tabulacji – *TabIndex i TabOrder* **57**
Ćwiczenie 12. Kolejność elementów w formularzu 57
Ćwiczenie 13. Składnia kodu podczas modyfikacji właściwości kontrolek .. 59

17. Suwak poziomy – *HScrollBar* **61**
Ćwiczenie 14. Suwaki, zatrzymanie programu 61
Zadanie 1. Odczytywanie wartości 61
Zadanie 2. Przesuwanie suwaka 62

18. Zmienne **63**
Ćwiczenie 15. Program wykonujący podstawowe operacje arytmetyczne ... 63

19. Pole grupy – *GroupBox* **71**
Ćwiczenie 16. rozwiązywanie równania kwadratowego 71

20. Funkcje matematyczne w Visual Studio **75**

21. Zaokrąglanie – *Round* **77**
Ćwiczenie 17. Obliczanie kwoty do zapłaty 77

22. Instrukcje warunkowe: *If ... Then* **79**
Ćwiczenie 18. Opis wieku 80

23. Instrukcje warunkowe: *Select Case* **83**
Ćwiczenie 19. Rozwiązywanie równania kwadratowego – wersja 2 83
Ćwiczenie 20. Obliczenia podatku 84
Ćwiczenie 21. Obliczenia podatku – wersja 2 85
Ćwiczenie 22. Paski narzędzi oraz kontrolki grupy 86
Zadanie 1. MenuStrip 86
Zadanie 2. ToolStrip 88
Zadanie 3. StatusStrip 88
Zadanie 4. NotifyIcon, ContextMenuStrip 89
Zadanie 5. ToolTip 90

24. Pętle **91**
Ćwiczenie 23. *For ... Next* 92
Ćwiczenie 24. *Do ... While/Until* 93
Ćwiczenie 25. Pętle – praktyczne zastosowanie 93

25. ZADANIA INŻYNIERSKIE DO SAMODZIELNEJ REALIZACJI ... 95
Zagadnienie 1. Obliczanie wytrzymałości wspornika ... 95
Zagadnienie 2. Czas pracy łożysk przekładni śrubowej ... 102
Zagadnienie 3. Czas pracy łożysk przekładni ślimakowej ... 105
Zagadnienie 4. Ugięcie belki wolnopodpartej obciążonej obciążeniem ciągłym 107
Zagadnienie 5. Wyznaczanie parametrów ceownika ... 108
Zakończenie ... 111
Bibliografia ... 113
Literatura uzupełniająca ... 113

Podstawy programowania w zastosowaniach inżynierskich. Visual Basic

Streszczenie

Wykorzystanie komputera stanowi współcześnie nieodzowny element pracy inżyniera. Dzięki technice komputerowej wiele obliczeń i analiz można wykonywać szybciej i sprawniej. Rozwiązywanie problemów inżynierskich wymaga jednak niejednokrotnie opracowywania narzędzi informatycznych poświęconych konkretnym zagadnieniom. Pożądaną wówczas umiejętnością jest samodzielne opracowywanie aplikacji komputerowych. Znajomość podstaw programowania komputerów pozwala bowiem na opracowywanie unikalnego i oryginalnego oprogramowania. Obecnie istnieje wiele języków programowania, które pozwalają w pełni wykorzystać możliwości komputera.

Przykładem dostępnego i intuicyjnego języka programowania jest Visual Basic, który znajduje szerokie zastosowanie w procesie opracowywania nowych aplikacji użytkowych oraz modułów rozszerzeń znanych programów. Niniejszy podręcznik powstał w odpowiedzi na zaobserwowany brak spójnego opracowania dotyczącego wykorzystania języka Visual Basic w celu analizy problemów inżynierskich z zakresu inżynierii mechanicznej. Praca została podzielona na 25 ćwiczeń uzupełnianiach kluczowymi zagadnieniami z zakresu programowania komputerów, które mają na celu wprowadzenie do praktycznego wykorzystania języka Visual Basic. Kluczowy element pracy stanowi przedstawienie zastosowania języka Visual Basic do rozwiązywania konkretnych zagadnień inżynierskich wraz z analizą przykładowych aplikacji użytkowych.

Słowa kluczowe: podstawy programowania, Visual Basic, Visual Studio, zastosowania inżynierskie

Programming fundamentals for engineering applications. Visual Basic

Abstract

The use of the computer is nowadays an indispensable part of an engineer's work. Thanks to computer technology, many calculations and analyses can be performed faster and more efficiently. However, solving engineering problems often requires the development of computer tools dedicated to specific issues. As a consequence independent development of computer applications is a key skill for modern engineer. This is because knowing the basics of computer programming allows to develop unique and original software. Currently, there are many programming languages that allow you to take full advantage of the computer's capabilities.

An example of an accessible and intuitive programming language is Visual Basic, which is widely used in the process of developing new utility applications and extension modules for well-known programs. This coursebook was written in response to the observed lack of a coherent study of the use of Visual Basic to analyze mechanical engineering problems. The work is divided into 25 exercises supplemented by key issues in computer programming to provide an introduction to the practical use of Visual Basic. A key element of the work is the presentation of the application of Visual Basic language to solve specific engineering problems, along with the analysis of sample utility applications.

Keywords: basics of programming, Visual Basic, Visual Studio, engineering applications

1. Wstęp

Visual Basic (VB) jest językiem programowania zbudowanym w oparciu o Basic, ale znacząco rozbudowanym przez firmę Microsoft. Jego bezpośrednim poprzednikiem był rozwijany od lat 80. QuickBasic. Historia VB sięga lat 90., kiedy to wydano jego pierwszą wersję. Dziś VB, obok C, C++, C#, Java, Pascal, Delphi i innych, należy do grupy języków programowania III generacji (3GL). Wygląd aplikacji projektuje się tu poprzez przeciąganie wybranych komponentów z panelu na formularz, a następnie modyfikowanie ich właściwości (koloru, rozmiaru, tekstu i wielu, wielu innych). Filozofia budowy programów w tym języku oparta jest o tzw. sterowanie zdarzeniami – działanie programu zależy od zdarzeń, takich jak: kliknięcie myszą, pisanie tekstu, zdarzeń pochodzących od innych programów/procedur, czujników zewnętrznych, itp. Język Visual Basic cechuje prostota, a jego nauka nie jest skomplikowana. Jest on, poza klasycznym zastosowaniem, tj. tworzeniem typowych, okienkowych aplikacji, wykorzystywany również do tworzenia makr w programach zewnętrznych – np. Visual Basic for Applications (VBA) jest częścią pakietu Microsoft Office i pozwala m.in. na automatyzację pewnych, często powtarzanych czynności.

Niniejsze opracowanie poświęcone jest nauce programowania w VB, w środowisku Visual Studio 2022. Istnieją oczywiście starsze wersje środowiska, które z powodzeniem można wykorzystać do budowy programów, a zakres czynności programistycznych nie będzie się znacząco różnił o tych, prezentowanych w niniejszej publikacji. Z uwagi na ten fakt, mogą występować niewielkie rozbieżności w stosunku do poprzednich wersji środowiska. Nie oznacza to jednak, że podręcznik ten będzie bezużyteczny dla użytkowników starszych wersji Visual Studio. Prezentowane treści dotyczą bowiem konkretnego języka, jakim jest Visual Basic, który w swoich podstawach jest identyczny, niezależnie od wersji, na której pracujemy.

Jaka jest zatem różnica pomiędzy środowiskami? Microsoft Visual Basic (2012 i starsze) to narzędzie umożliwiające budowę programów z wykorzystaniem tego języka programowania – ograniczona wersja zintegrowanego środowiska programistycznego (IDE) Microsoft Visual Studio. Microsoft Visual Studio w wersji 2022 to najnowsza wersja platformy programistycznej, przy pomocy której można tworzyć programy w różnych językach: C/C++, Visual Basic, C#, a także F#. Możliwe jest także doinstalowanie obsługi języków: M, Python i Ruby. Najnowsza wersja Visual Studio umożliwia także projektowanie aplikacji dla takich platform jak Android czy iOS. Różnorodne wersje platformy Visual Studio dostępne są na stronie firmy Microsoft[1].

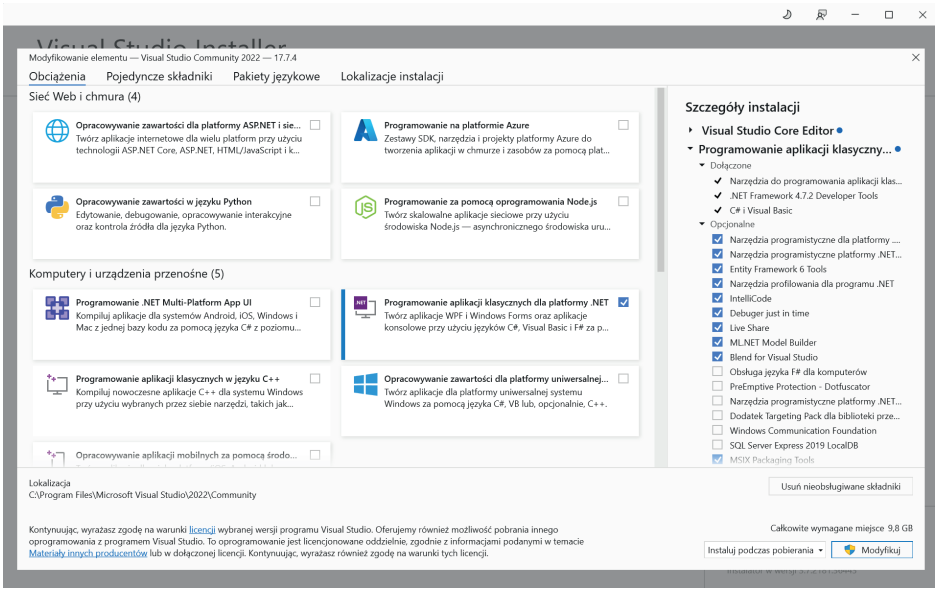
Poniższa książka przeznaczona jest w pierwszej kolejności jako materiał do zajęć dla studentów I stopnia kierunku robotyzacja procesów wytwórczych oraz mechanika i budowa maszyn Politechniki Lubelskiej, choć może ona także znaleźć zastosowanie w kształceniu studentów innych technicznych kierunków z zakresu inżynierii mechanicznej. Wiedza z zakresu opisanego w przygotowanej pracy ma stanowić narzędzie

[1] https://visualstudio.microsoft.com/pl (dostęp 10.11.2023).

wsparcia inżyniera (absolwenta) w jego typowej pracy zawodowej, choć może być mu przydatna jeszcze na etapie studiów. Zawarte treści są podstawowe, więc odbiorcą może być zarówno student, jak i pracujący w zawodzie inżynier, chcący rozszerzyć swoją wiedzę. Z uwagi jednak na zawarte tu przykłady do zajęć, które często odwołują się do treści z zakresu fizyki, mechaniki, wytrzymałości materiałów, podręcznik może okazać się wyzwaniem dla studentów i absolwentów nieposiadających podstawowej wiedzy z inżynierii mechanicznej.

2. Instalacja Visual Studio

W celu rozpoczęcia pracy w środowisku Visual Studio (VS), należy pobrać instalator programu ze strony internetowej Microsoftu[2].Sam instalator nie zajmuje dużo miejsca, gdyż potrzebne elementy pobiera dopiero podczas instalacji. W zależności od tego, ile komponentów zechcemy zainstalować, może być to od 3 GB do nawet ponad 40GB. Należy pamiętać, by dołączyć do naszej instalacji moduł „Programowania aplikacji klasycznych dla platformy .NET", gdyż w przeciwnym wypadku nie będziemy mieć możliwości tworzenia w środowisku typowych, okienkowych aplikacji w języku Visual Basic (rys. 1). Moduł ten można doinstalować również później, modyfikując bieżącą instalację VS (WINDOWS11: *panel sterowania* > *programy* > *programy i funkcji* > *odinstaluj lub zmień program* > *Microsoft Visual Studio* > *Zmień*) lub po prostu uruchamiając ponownie instalator i wybierając opcję: *Modyfikuj*.

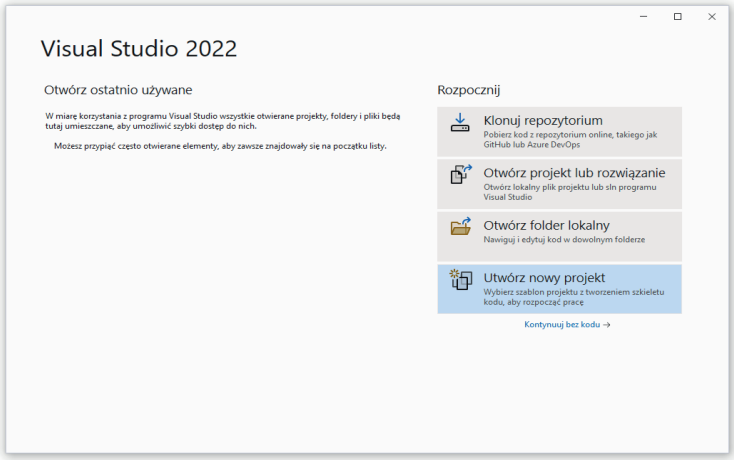


Rysunek 1. Instalator Microsoft Visual Studio 2022

[2] https://visualstudio.microsoft.com/pl (dostęp 10.11.2023).

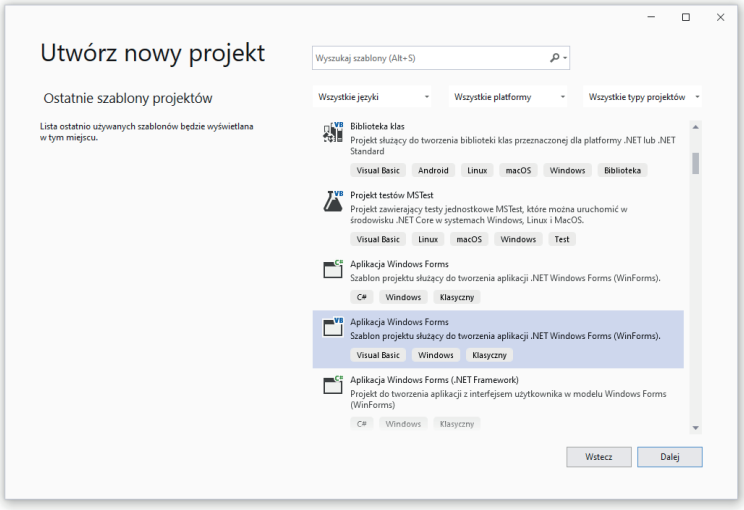
3. Tworzenie nowego projektu

Bezpośrednio po uruchomieniu środowiska Visual Studio użytkownik zostaje zapytany o podjęcie działań. W celu rozpoczęcia pracy należy zatem wybrać polecenie *Utwórz nowy projekt* (rys. 2).



Rysunek 2. Puste okno programu Microsoft Visual Studio 2022

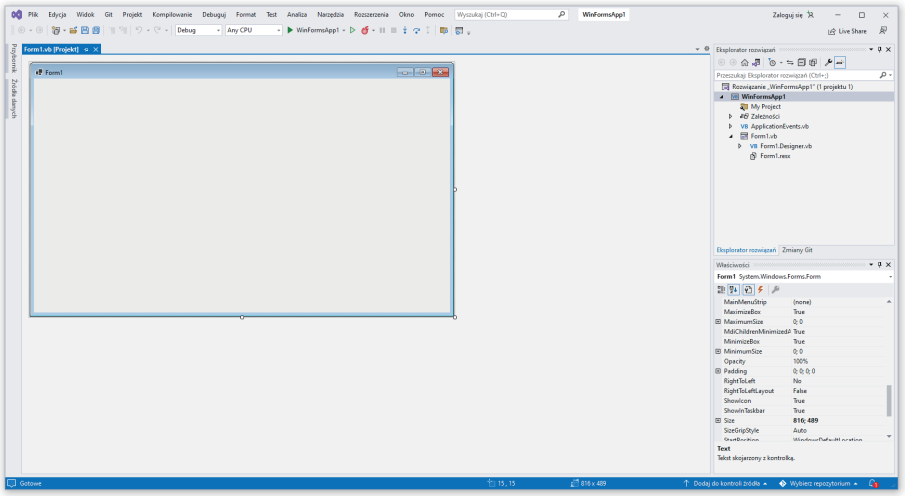
W kolejnym oknie należy odnaleźć wśród dostępnych opcji możliwość tworzenia aplikacji systemu Windows z wykorzystaniem języka Visual Basic *(Aplikacja Windows Forms – Visual Basic*) (rys. 3).



Rysunek 3. Wybór typu projektu z szablonu

Kolejny krok stanowi konfigurowanie projektu. Głównym parametrem będzie nadanie mu odpowiedniej nazwy. Domyślnie będzie to *WindowsApp1*. Przechodząc do następnego okna, trzeba zdefiniować platformę aplikacji (dla aplikacji systemu Windows z wykorzystaniem języka VB będzie to platforma .NET z dowolnej, stabilnej i wspieranej wersji).

W rezultacie utworzony zostanie nowy projekt, który otworzy się w głównym oknie środowiska VS (rys. 4). Okno to charakteryzuje się panelową budową – w jego skład wchodzą panele (rozwijalne lub stale widoczne), w których znajdują się opcje, komponenty i ustawienia.

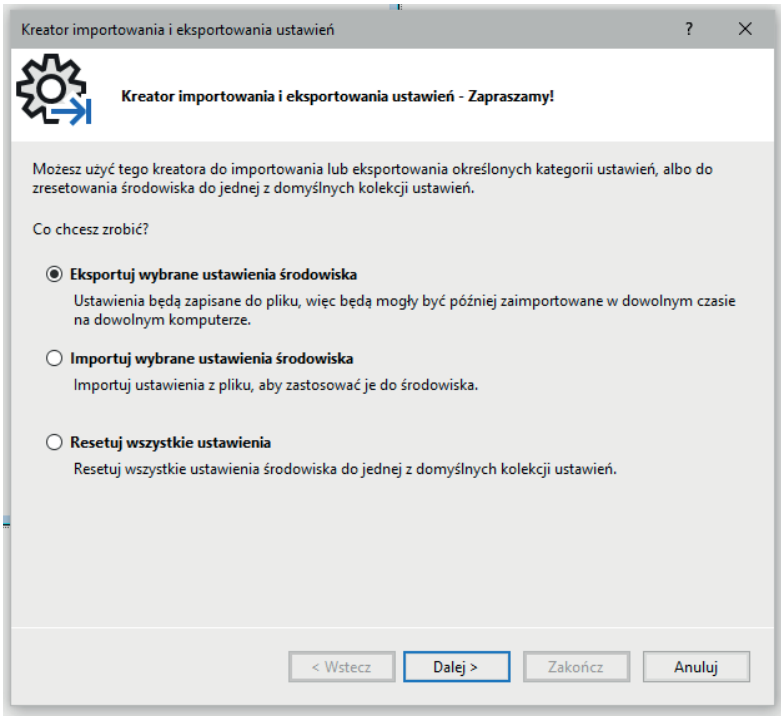


Rysunek 4. Tworzenie nowego projektu w Microsoft Visual Studio

Należy pamiętać, że Visual Studio zostanie uruchomiony w ostatnio używanym trybie ustawień (w tym rozmieszczenia paneli). Może być to domyślny wygląd dla programowania w Visual Basic, Visual C++, C# lub innego, dowolnego dostępnego języka programowania. Rozkład okien, paneli i ustawienia wszystkich dostępnych opcji może być też zdefiniowany przez poprzedniego użytkownika.

4. Resetowanie układu okien

Istnieje możliwość zmiany (przywrócenia domyślnego, wczytania zapisanego) środowiska pracy. W tym celu należy wybrać z menu: *Okno* > *Resetuj układ okna*. Jeśli jednak domyślne ustawienia zostały nadpisane i powyższa operacja nie przyniesie oczekiwanego rezultatu, należy wybrać: *Narzędzia* > *Import i eksport ustawień*… i zresetować wszystkie ustawienia (rys. 5).

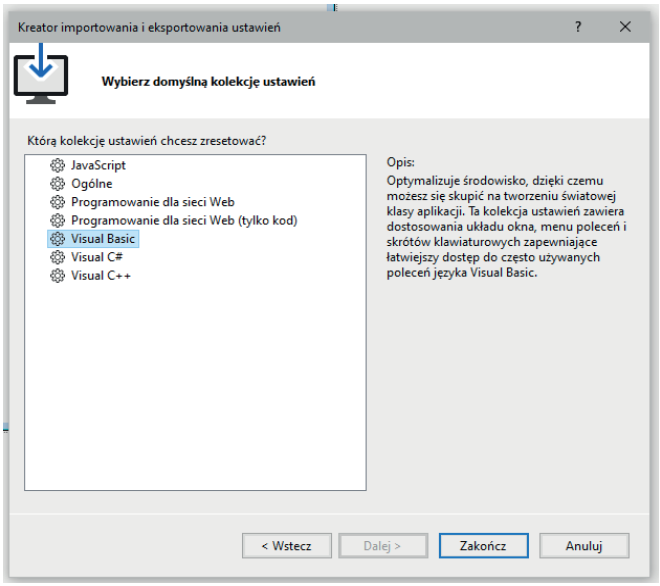


Rysunek 5. Kreator importowania i eksportowania ustawień

Okno importowania i eksportowania ustawień pozwala na wybranie jednej z trzech opcji. Po wybraniu ostatniej opcji – *Resetuj wszystkie ustawienia*, program pyta jeszcze użytkownika, czy bieżące ustawienia mają być zapisane, czy ma tylko przywrócić domyślne ustawienia, zastępując nimi obecne.

W następnym oknie należy już tylko wskazać, które środowisko chcemy wczytać. Dla programowania z wykorzystaniem języka Visual Basic będzie to oczywiście Visual Basic (rys. 6).

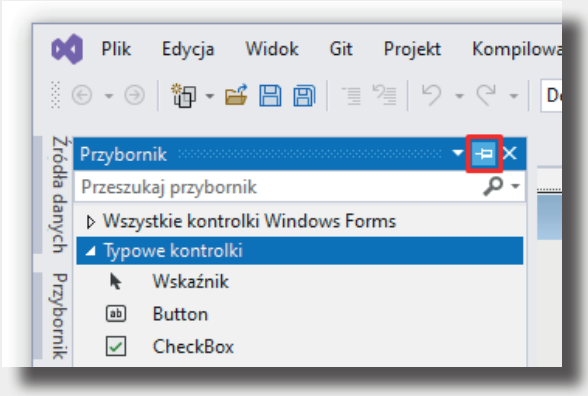
W przypadku pracy w poprzednich wersjach programu, tj. w Microsoft Visual Basic (nie Visual Studio), powyższy problem nie wystąpi, gdyż tam dostępne jest tylko środowisko pracy języka Visual Basic.



Rysunek 6. Kreator importowania i eksportowania ustawień (wybór środowiska)

Warto wiedzieć!

Jeżeli po wykonaniu procedury resetowania ustawień, niektóre z paneli zostaną ukryte, należy kliknąć na odpowiednią zakładkę lewym przyciskiem myszy i dezaktywować funkcję *Autoukrywanie.* Jest to możliwe poprzez przyciśnięcie symbolu pinezki w prawym górnym rogu okna/panelu (rys. 7).



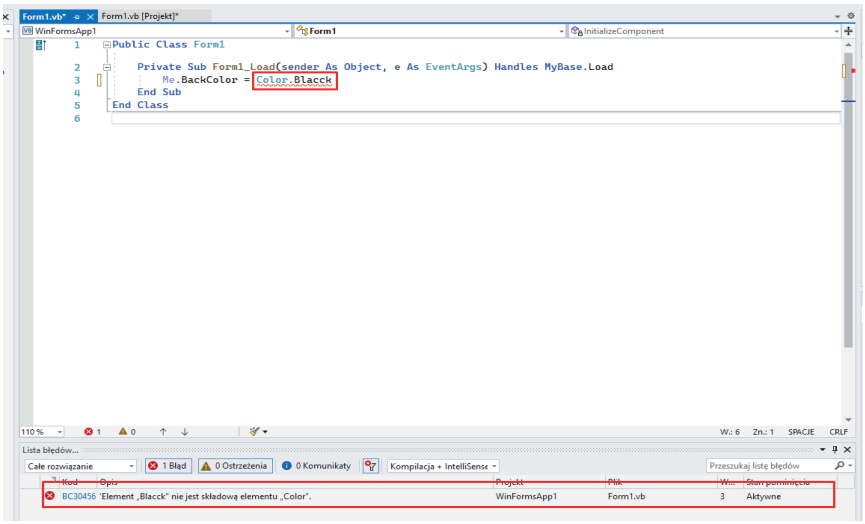
Rysunek 7. Kontrola nad autoukrywaniem panelu

5. Panelowa struktura okna VS

Środowisko Visual Studio składa się z wielu paneli, z których każdy pełni odpowiednią funkcję w procesie tworzenia programów.

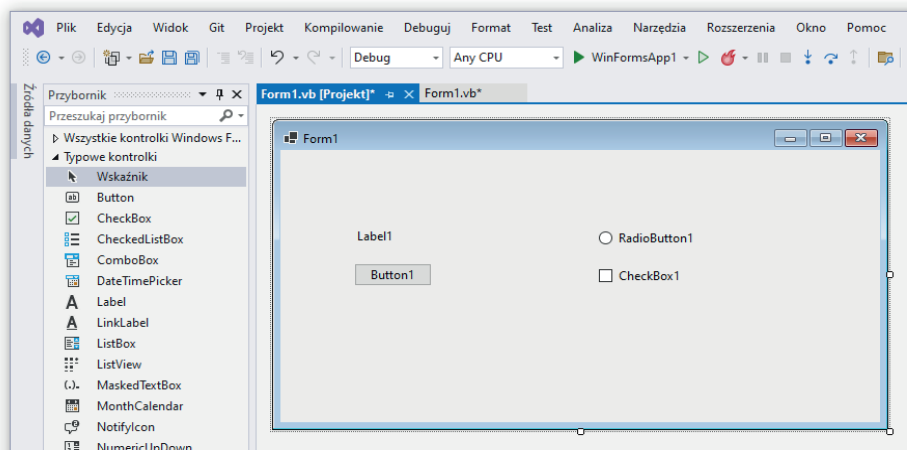
W lewej górnej części przestrzeni roboczej programu znajduje się zakładka: *Przybornik*, która rozwinie się z chwilą najechania na nią kursorem myszy. Wysunięty panel *Przybornika* to panel z kontrolkami (formantami). Za pomocą pinezki w pasku tytułowym, można go przypiąć do obszaru roboczego tak, żeby nie chował się do postaci zakładki. Panel wypełniony jest połączonymi w kilka grup kontrolkami, z których budowany będzie wygląd naszej aplikacji. Pierwsza grupa: *Wszystkie kontrolki Windows Forms* zawiera wszystkie dostępne kontrolki, ale ponieważ jest ich dużo, wygodnie jest tę grupę zwinąć i korzystać z drugiej: *Typowe Kontrolki,* czyli grupy najczęściej używanych kontrolek. Gdyby była potrzeba dodania innych kontrolek – należy ich szukać w kolejnych grupach. Będą to m.in.: *Kontenery*, *Menu i paski narzędzi*, *Dane*, *Składniki*, *Drukowanie*, *Okna dialogowe*. Ewentualnie, gdyby w oknie brakowało zakładki: *Przybornik,* można ją włączyć, korzystając z polecenia w pasku menu*: Widok > Przybornik.*

Kolejnym istotnym elementem środowiska jest panel *Lista błędów.* Można go wyświetlić w następujący sposób: *Widok > Lista błędów.* Ten poziomy panel umieszczony jest w dolnej części przestrzeni roboczej. Służy on do wyświetlania błędów w budowanym kodzie. To znaczy, że jeśli program będzie posiadał błędy składni (bądź błędy innego typu), to zostaną one wyświetlone w tym panelu wraz z podaniem pliku wewnątrz naszego projektu, w którym błąd znaleziono, a także ze wskazaniem linii i kolumny, w której błąd występuje (rys. 8). Dodatkowo podawany jest krótki opis błędu. W kodzie programu błąd ten będzie zaznaczony czerwoną falowaną linią.



Rysunek 8. Przykład zgłaszania błędów w kodzie programu

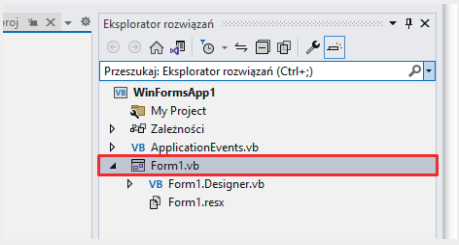
Po wczytaniu nowego projektu w centralnej części okna Visual Studio znajdzie się widok formularza programu, który będzie budowany (rys. 9). Na ten formularz (okno) dodawane będą kontrolki z omówionego wcześniej panelu: *Przybornik*.



Rysunek 9. Projektowanie wyglądu formularza

Warto wiedzieć!

Jeżeli użytkownik przez przypadek zamknie zakładkę z projektowanym formularzem, może ją ponownie otworzyć poprzez dwukrotne kliknięcie na elemencie *Form1.vb* w oknie *Eksplorator rozwiązań*, które znajduje się po prawej stronie okna głównego VS (rys. 10).



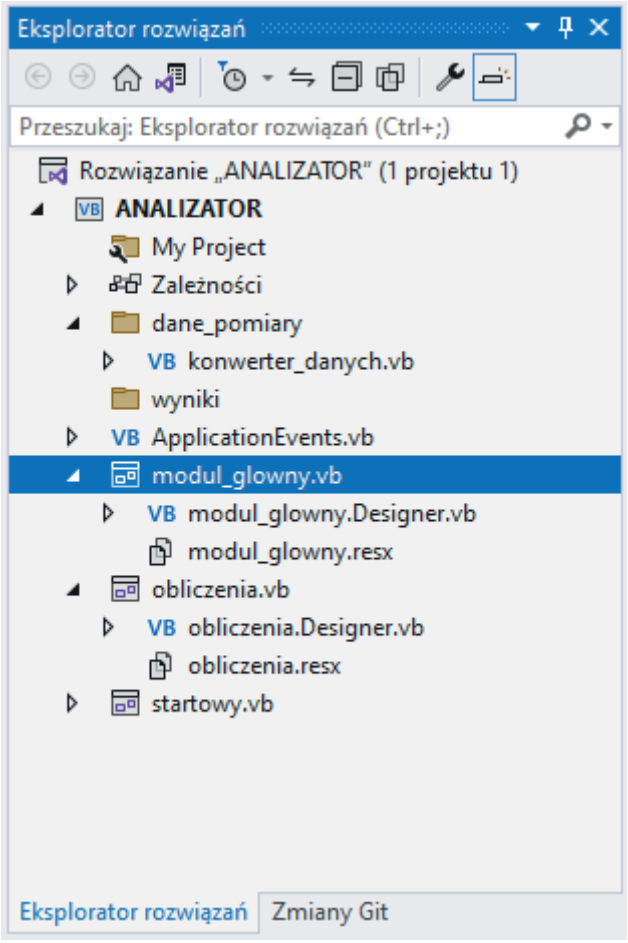
Rysunek 10. Kontrola nad autoukrywaniem panelu

Istnieją trzy sposoby na importowanie kontrolek:

- klasyczne przeciągniecie ich z przybornika na formularz lewym klawiszem myszy (tzw. metoda „*drag&drop*”),
- jednokrotne kliknięcie na wybraną kontrolkę lewym klawiszem myszy, a potem „narysowanie” jej w formularzu za pomocą lewego klawisza myszy,
- podwójne kliknięcie na wybraną kontrolkę w przyborniku, co spowoduje automatyczne dodanie jej w miejscu umieszczenia poprzedniej kontrolki.

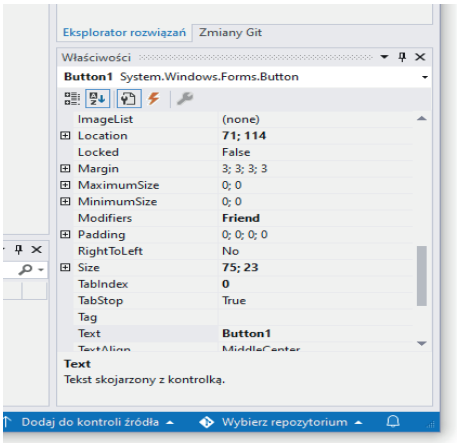
Pierwszy oraz trzeci sposób daje nam kontrolki domyślnych rozmiarów, w drugim rozmiar definiowany jest podczas rysowania kontrolki. Nie stanowi to jednak problemu, gdyż rozmiary kontrolek można zmieniać także później, przy wykorzystaniu panelu: *Właściwości*, który będzie omówiony w dalszej części rozdziału.

W prawym górnym oknie Visual Studio znajduje się panel *Eksplorator rozwiązań*. W miarę budowy projektu będzie się on wypełniał plikami wchodzącymi w skład projektu, np. plikami VB, zawierającymi opis wyglądu formularzy (okien) i kodu do nich przypisanego, a także dodanymi zasobami (np. plikami graficznymi, ikonami, plikami dźwiękowymi). Dwukrotne kliknięcie pliku *.vb powoduje jego otworzenie w widoku trybu tworzenia wyglądu aplikacji (rys. 11).



Rysunek 11. Przykładowy wypełniony panel Eksploratora rozwiązań

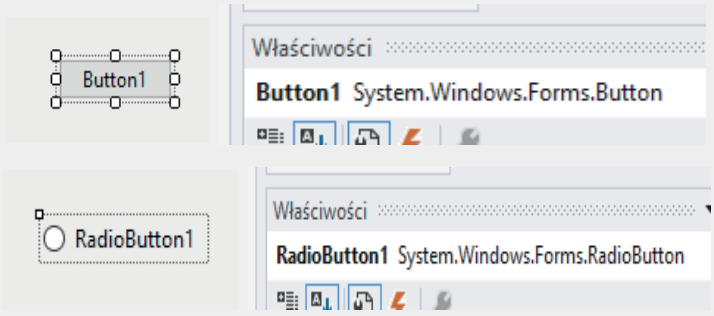
Poniżej panelu *Eksplorator rozwiązań* znajduje się okno: *Właściwości elementów* (kontrolek) tworzących program. W panelu tym znajduje się szereg opcji konfiguracji kontrolek, dodawanych do tworzonego projektu (rys. 12). Nie sposób w tym miejscu omówić wszystkich właściwości, dlatego też będą one sukcesywnie opisywane przy okazji realizowania kolejnych ćwiczeń i zadań praktycznych.



Rysunek 12. Panel właściwości

Warto wiedzieć!

Jeżeli użytkownik chce zmienić konkretną właściwość danej kontrolki, musi być ona aktywna, tzn. zaznaczona poprzez jednokrotne kliknięcie. Wówczas pojawia się wokół danej kontrolki przerywana linia z białymi znacznikami, a w nagłówku okna: *Właściwości* pojawia się jej nazwa (rys. 13).



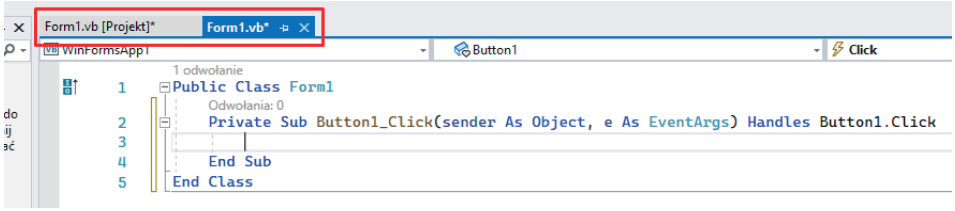
Rysunek 13. Wygląd aktywnej kontrolki

Tak jak pokazano wcześniej, rozpoczęcie pracy w środowisku Visual Studio pozwala na tworzenie aplikacji w widoku projektowania graficznego – w trybie *Projekt*. Oczywiście kluczowym elementem programowania jest tworzenie kodu w języku

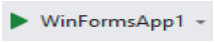
Visual Basic, dlatego też istotnym zagadnieniem jest przełączanie widoku pomiędzy kodem a projektem graficznym.

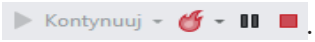
Widok kodu można wywołać, klikając na plik *From1.vb* (znajdujący się w panelu: *Eksplorator rozwiązań*) prawym klawiszem myszy i wybierając: *Pokaż kod* lub klawiszem F7. W tym momencie można po raz pierwszy zaobserwować tę dwudzielną budowę aplikacji.

Poza wyglądem projektanta użytkownik tworzy również kod programu. Powyżej centralnej części VS widoczne wówczas będą dwie zakładki: *Form1.vb [Projekt]* i *Form1.vb* (rys. 14). Służą one do przechodzenia między trybem tworzenia kodu (programowania rzeczywistego działania programu) a trybem projektowania wyglądu formularzy. Ponadto można przechodzić pomiędzy trybami za pomocą poleceń: *Widok > Kod* / *Widok > Projektant.*



Rysunek 14. Zakładki kodu i trybu projektowania wyglądu formularza

Gotowy program będzie uruchamiany (debugowany) za pomocą zielonego trójkąta na pasku narzędziowym 

(lub za pomocą klawisza F5). Po uruchomieniu uaktywni się czerwony kwadratowy przycisk zatrzymania programu 

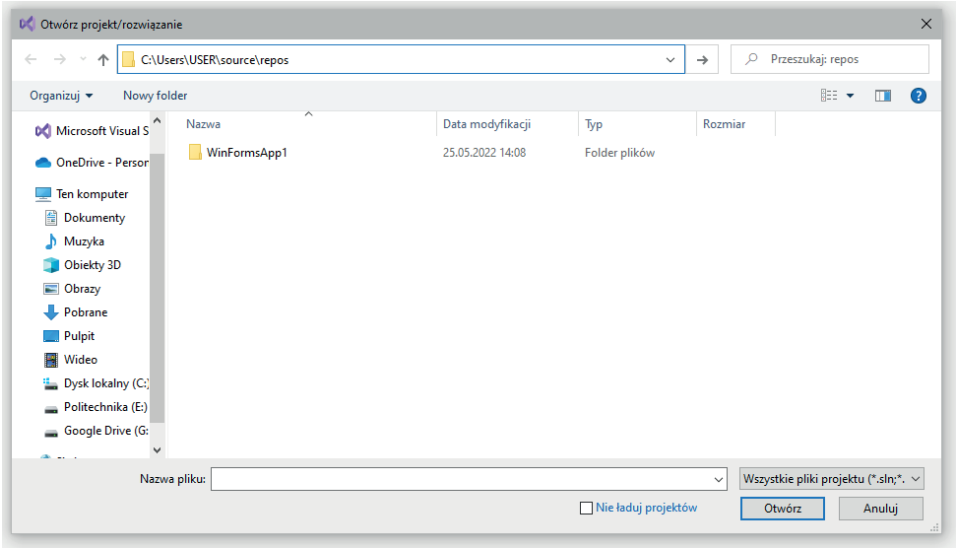
. Zamykania programu uruchomionego w środowisku Visual Studio należy zawsze dokonywać przy wykorzystaniu kwadratowego przycisku *STOP*. Wiąże się to ze szczególnymi przypadkami, w których zamknięcie formularza krzyżykiem w prawym górnym rogu okna nie zamknie w rzeczywistości całego programu (przykładem są tu programy wieloformularzowe z niewidocznym formularzem początkowym, o których więcej w dalszej części kursu).

Uwaga, gdy program jest uruchomiony, nie można go modyfikować (ani wyglądu, ani kodu). Jednocześnie, po uruchomieniu programu panel z kontrolkami będzie się ukrywać. Często na początku pracy z Visual Studio pojawiają się sytuacje, kiedy z jakiegoś powodu nie można edytować kodu lub wyglądu formularza. Trzeba się wówczas upewnić, czy program nie jest przypadkiem uruchomiony. W celu dalszej pracy nad projektem należy go zatrzymać.

6. Zapisywanie aplikacji

Utworzony projekt należy zapisać, aby w przyszłości móc kontynuować nad nim pracę. W tym celu należy w menu głównym wybrać polecenie: *Plik* > *Zapisz wszystko.* Bardzo ważne jest, aby skorzystać właśnie z tej opcji! Tworzony program to projekt, który przydatny jest tylko jako kompletne rozwiązanie. Zapisać więc należy jego wszystkie komponenty. Użycie wyłącznie polecenia: *Zapisz Form1.vb* będzie skutkować zapisaniem jedynie pojedynczego formularza, który samodzielnie będzie nieprzydatny.

Zapisany projekt będzie dostępny w lokalizacji, która była podana podczas konfigurowania parametrów aplikacji w chwili rozpoczęcia pracy. W łatwy sposób możemy ją odnaleźć i otworzyć ponownie, wybierając opcję: *Plik* > *Otwórz projekt…* Wówczas w oknie dialogowym zostanie otworzony folder zawierający nasz projekt (rys. 15).



Rysunek 15. Okno otwierania projektu

Po wejściu do folderu o nazwie projektu widoczny jest podfolder o tej samej nazwie oraz plik o nazwie projektu z rozszerzeniem **.sln*. Jest to właśnie plik projektu, który należy otworzyć, aby móc z powrotem go rozbudowywać i modyfikować.

W podobny sposób wygląda procedura otwierania projektu po zamknięciu środowiska VS. Wówczas w oknie powitalnym wybieramy opcję: *Otwórz projekt lub rozwiązanie*, po czym w oknie dialogowym wyświetli się miejsce, w którym domyślnie będą przechowywane projekty użytkownika.

Ćwiczenie 1. Podstawy budowy aplikacji

Dodaj do formularza i sprawdź działanie (przy uruchomionym programie) następujących, typowych kontrolek:

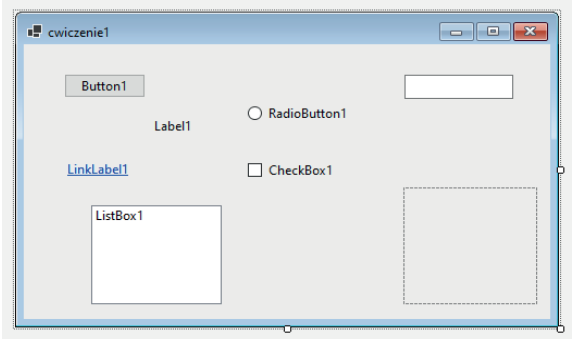
- *Button* – przycisk,
- *Label* – etykieta,
- *LinkLabel* – etykieta z linkiem,
- *ListBox* – pole listy,
- *PictureBox* – obraz,
- *TextBox* – pole tekstowe,
- *RadioButton* – przycisk opcji,
- *CheckBox* – przycisk wyboru,
- *MonthCallendar* – kalendarz miesięczny,
- *DateTimePicker* – selektor daty.

Pozmieniaj dodanym kontrolkom (włącznie z formularzem) poniższe, wybrane właściwości. Za pomocą przycisku na górze panelu: *Właściwości* możesz ustawić rozkład alfabetyczny właściwości, zamiast grupowania według kategorii. Zwróć uwagę na to, że dla niektórych kontrolek część właściwości może nie istnieć. Pamiętaj! Aby zmienić właściwość danej kontrolki, musisz uaktywnić ją poprzez jednokrotne kliknięcie. Aktualnie wybrana kontrolka będzie obrysowana, a na górze panelu *Właściwości* pojawi się jej nazwa. Dodatkowo pamiętaj, że część zmian może być zaobserwowana dopiero po uruchomieniu programu.

Właściwości do sprawdzenia:

- (*Name*) – nazwa,
- *BackColor* – kolor tła,
- *Cursor* – kształt kursora,
- *Font* – czcionka,
- *Forecolor* – kolor tekstu,
- *Location* – położenie,
- *Opacity* – nieprzezroczystość,
- *Size* – rozmiar,
- *Text* – tekst,
- *Visible* – widoczność.

Zapisz opracowany projekt pod nazwą *cwiczenie1*.



Rysunek 16. Przykładowy program z wykorzystaniem typowych kontrolek

Ćwiczenie 2. Podstawy tworzenia kodu

Zamknij poprzedni projekt (*Plik > Zamknij rozwiązanie*). Stwórz nowy i dodaj do formularza jeden przycisk. Program w takiej formie, poza tym, że ma wygląd klasycznego okna z dodanym przyciskiem, nie ma żadnej funkcji – nic się w nim nie dzieje. Dodamy więc kod, który będzie wykonany po kliknięciu w przycisk. W tym celu, będąc w widoku *Projekt* (widoku tworzenia wyglądu formularza), klikamy dwukrotnie w kontrolkę *Button* (przycisk). Visual Studio przeskoczy na zakładkę z kodem (ew. pierwszy raz wyświetli ją, jeśli do tej pory nie była widoczna). Zasadniczo, kod formularza zawsze będzie składał się z tych dwóch linii:

```
Public Class Form1

End Class
```

Pomiędzy tymi liniami będą znajdowały się (w miarę rozbudowy programu) dodatkowe linie opisujące, np. zdarzenia wywołujące określone działanie lub procedury.

W naszym przypadku kod będzie wyglądał następująco:

```
Public Class Form1
  Private Sub Button1_Click(sender As Object, e As EventArgs)
  Handles Button1.Click
  End Sub
End Class
```

Do wcześniej pokazanego fragmentu kodu dodane zostały dwie linie. Określają one początek i koniec procedury wywołanej zdarzeniem kliknięcia w pierwszy przycisk. Wszystko, co wpiszemy pomiędzy te dwie linie, zostanie wykonane, gdy w uruchomionym programie klikniemy ten przycisk, tj. przycisk o nazwie: *Button1*.

Zwróć uwagę, jak Visual Studio automatycznie dodaje wcięcia w kodzie. Zwiększa to jego czytelność, pozwalając na szybkie odnalezienie specyficznych fragmentów kodu. Wpisz do procedury następujący kod:

```
Button1.BackColor = Color.Red
```

Jeśli po zakończeniu wpisywania, żadna część kodu nie będzie podkreślona na niebiesko, a w dolnym panelu: *Lista błędów* nie będzie żadnych błędów, uruchom program i sprawdź, jakie działanie po kliknięciu w przycisk będzie miał wprowadzony kod. Zmodyfikowana zostanie wartość właściwości koloru tła – ustawiony zostanie kolor czerwony.

Dokładnie w ten sam sposób możesz modyfikować wszystkie właściwości każdego z elementów znajdujących się w formularzu (z formularzem włącznie). Nazwy poszczególnych właściwości można znaleźć w panelu: *Właściwości.*

Warto wiedzieć!

Ogólną zasadę modyfikowania właściwości kontrolek można opisać poniższym wzorem, który *elementowi 1* zmieni *właściwość 2* na *wartość 3*:

```
Element1.Właściwość2 = Wartość3
```

Warto wiedzieć, że jeżeli chcemy zmienić jakąkolwiek właściwość formularza, w którym aktualnie pracujemy, jako nazwy elementu w kodzie używamy wyrażenia: *Me*:

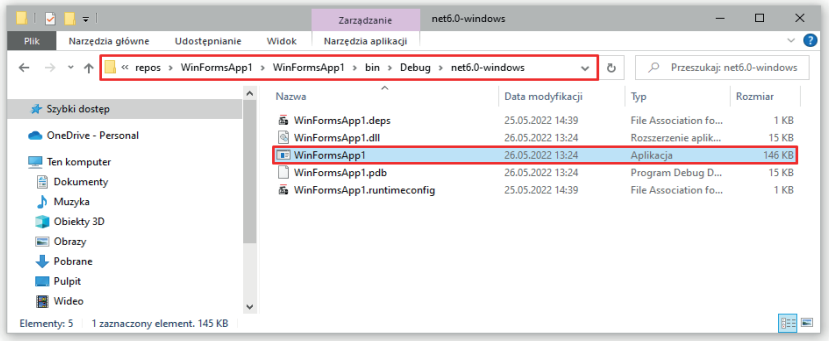
```
Me.Właściwość = Wartość
```

Przykładowo, kod zmieniający kolor tła całego formularza na kolor niebieski będzie mieć postać:

```
Me. BackColor = Color.Blue
```

7. Szybkie uruchamianie aplikacji w środowisku Windows

Odpowiednio wykonany i zapisany na dysku program można także uruchomić poza środowiskiem VS, ponieważ w takim właśnie celu tworzy się użytkowe aplikacje z wykorzystaniem języka VB. Aby to zrobić, należy na dysku komputera odnaleźć folder z projektem (będzie on nosił nazwę naszego projektu). Przechodząc do kolejnych podfolderów, trzeba przejść w głąb katalogu o nazwie *bin* aż do folderu *Debug*, w którym znajdują się właśnie zdebugowane pliki naszej aplikacji. Wybierając plik z rozszerzeniem *.exe (rys. 17), uruchomimy naszą aplikację w systemie Windows i będzie ona działać niezależnie od środowiska VS.



Rysunek 17. Uruchamianie opracowanej aplikacji poza środowiskiem VS

Warto wiedzieć, iż ścieżka dostępu może różnić się w zależność od wybranej wersji VS czy też zainstalowanego na komputerze środowiska uruchomieniowego .NET, jednak odnalezienie i uruchomienie programu będzie odbywać się w sposób analogiczny – pliki pozwalające na uruchomienie aplikacji zawsze przechowywane są w folderze projektu.

Ćwiczenie 3. *Spider's Legs* – podstawy programowania

Z lokalizacji zawierającej materiały pomocnicze pobierz program *Spider's_Legs* (adres do pobrania pliku dostępny jest u prowadzącego zajęcia lub u autorów podręcznika). Projekt należy zapisać na pulpicie, a następnie go rozpakować. W tym celu należy, klikając na niego prawym klawiszem myszy, wybrać z menu podręcznego pozycję: *Wyodrębnij wszystkie…*

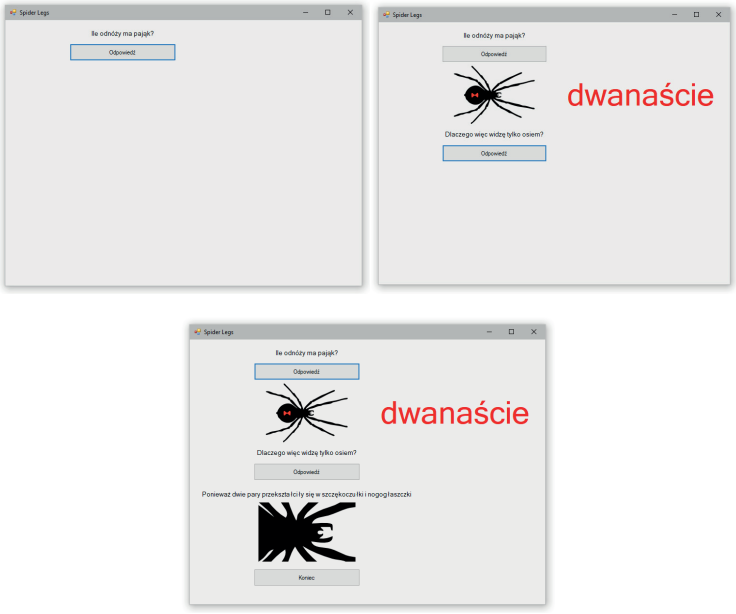
Plik otwierający projekt to ten, którego ikona ma kolorowy znak nieskończoności (odwróconą ósemkę), jego rozszerzenie to *.sln – jak *solution* (*rozwiązanie* – tak nazywa się projekty w VS). Kliknij w niego dwa razy, aby otworzyć rozpakowany projekt.

Jeżeli projekt został opracowany i zapisany w starszej wersji Visual Basic, konieczna może być jego konwersja do wersji posiadanej przez użytkownika. Procedura

konwertowania programu jest intuicyjna, a konwerter prowadzi użytkownika krok po kroku. W najnowszej wersji VB konwersja jest niemal automatyczna.

Gotowy projekt zostanie otworzony w Visual Studio, przy czym może się okazać, że nie zostanie od razu uruchomiony w trybie projektowania (będzie niewidoczny). Należy wówczas w panelu *Eksplorator rozwiązań* dwukrotnie kliknąć na *Form1.vb*.

W pierwszej kolejności trzeba sprawdzić, jak program działa. Aby to zrobić, uruchom go i klikaj na kolejne przyciski. Zasada działania programu jest następująca: zostaje zadane pytanie: „Ile odnóży ma pająk?" Po kliknięciu w pierwszy przycisk (*Button1*) z tekstem „Odpowiedź" zostaje wyświetlona etykieta (*Label2*) z odpowiedzią: „dwanaście" oraz pole obrazka (*PictureBox*) z obrazem pająka. Jednocześnie pojawia się kolejna etykieta z pytaniem: „Dlaczego więc widzę tylko osiem?" oraz drugi przycisk z tekstem: „Odpowiedź". Kliknięcie w drugi przycisk pokazuje drugą etykietę odpowiedzi: „Ponieważ dwie pary przekształciły się w szczękoczułki i nogogłaszczki" oraz drugie pole obrazka, ze zbliżeniem na przednią cześć głowotułowia pajęczaka (rys. 18).



Rysunek 18. Widok okna programu *Spider's Legs* na różnych etapach działania

Następnie przeanalizuj budowę wyglądu aplikacji. Program składa się z jednego formularza, na którym znajdują się 3 przyciski, 4 etykiety i 2 pola obrazka. Jakie znaczenie mają poszczególne kontrolki w programie? Przyciski służą do wyzwalania działania kodu, etykiety pełnią funkcję wyświetlania tekstu (bez możliwości ingerowania w tekst przez użytkownika), a użyteczność pól obrazkowych jest jasna.

Przejdź teraz do analizy kodu aplikacji. Wyświetl zakładkę z kodem. Zobacz, jakie procedury wchodzą w skład formularza. Są tam procedury wyzwalane przez klikanie kolejno w *Button1*, *Button2* i *Button3*. Przypomnij sobie, jakie było działanie pierwszego przycisku – było to odsłonięcie czerwonej etykiety z odpowiedzią, obrazka pająka, drugiego pytania i drugiego przycisku. Popatrz teraz na kod wywołany kliknięciem w pierwszy przycisk:

```
Private Sub Button1_Click(ByVal sender As System. Object, ByVal e
As System.EventArgs) Handles Button1.Click
   PictureBox1.Visible = True
   Label2.Visible = True
   Label3.Visible = True
   Button2.Visible = True
End Sub
```

Wewnątrz procedury znajdują się cztery linie. Pierwsza z nich ustawia elementowi *PictureBox1* właściwość *Visible* (widoczność) na *True* (prawda), tzn. obrazek odkrywa się/pojawia się. Druga linia – elementowi *Label1* (etykieta z czerwoną odpowiedzią) również ustawia właściwość *Visible* na T*rue* – odpowiedź pojawia się. Podobnie działają dwie kolejne linie. Widać tu więc wyraźnie wcześniej przedstawiony schemat, według którego należy postępować, gdy chcemy zmienić którąś z właściwości:

```
Element.Właściwość = Wartość
```

Dalej kod zawiera procedurę wywołaną kliknięciem w drugi przycisk. Znów pod jej działaniem odsłaniają się pewne kontrolki. Ostatnia procedura, wywołana kliknięciem w przycisk: *Zamknij* wykonuje funkcję zakończenia programu:

```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button3.Click
   End
End Sub
```

Po przeanalizowaniu kompletnego projektu widać wyraźnie, że program musi składać się z części graficznej (*Projekt*) i dopisanego do niej kodu, definiującego zachowanie programu przy różnych zdarzeniach.

Warto wiedzieć!

Oprócz polecenia *End,* w celu zakończenia działania aplikacji, można użyć także poniższego kodu:

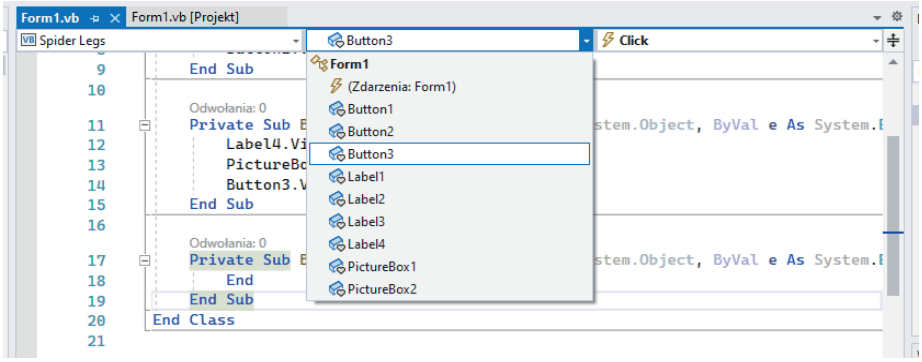
```
Me.Close()
```

Należy jednak pamiętać, iż polecenie to w aplikacjach składających się z kilku formularzy będzie powodowało zamknięcie jedynie aktywnego formularza. Polecenie *End* zatrzyma zaś całą aplikację.

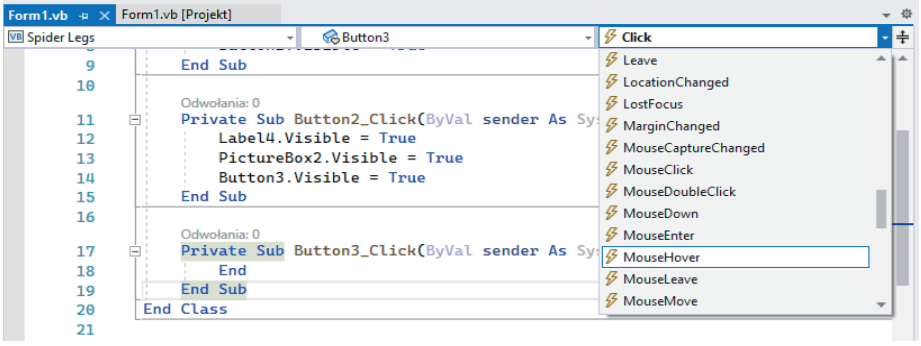
8. Definiowanie innych zdarzeń

W przypadku przedstawionego wcześniej programu, kod był wywoływany zdarzeniami kliknięcia w elementy (przyciski). Nie jest to regułą, gdyż nie tylko kliknięcie może wywoływać zdarzenia. Aby wybrać dowolne zdarzenie określające wykonanie procedury, należy przejść do trybu kodu i, stojąc w dowolnym miejscu kodu, wybrać z pierwszej rozwijalnej listy, znajdującej się powyżej okna z kodem, kontrolkę, której ma dotyczyć zdarzenie (rys. 19). Przykładowo, wybierając przycisk3 (*Button3*), a następnie rozwijając drugą listę opcji (na prawo od pierwszej), możemy określić inne zdarzenie niż kliknięcie myszą (rys. 20). Lista ta zawiera spis zdarzeń dotyczących wybranego elementu i będą to, np.:

- *MouseHover* – kod zadziała, gdy kursor myszy znajdzie się wewnątrz granic kontrolki,
- *DoubleClick* – kod zadziała, gdy użytkownik dokona podwójnego kliknięcia na kontrolce,
- *SizeChanged* – kod zadziała, gdy kontrolka zmieni rozmiar,
- *FormClosing* – kod zadziała, gdy formularz zacznie być zamykany
- i wiele innych.

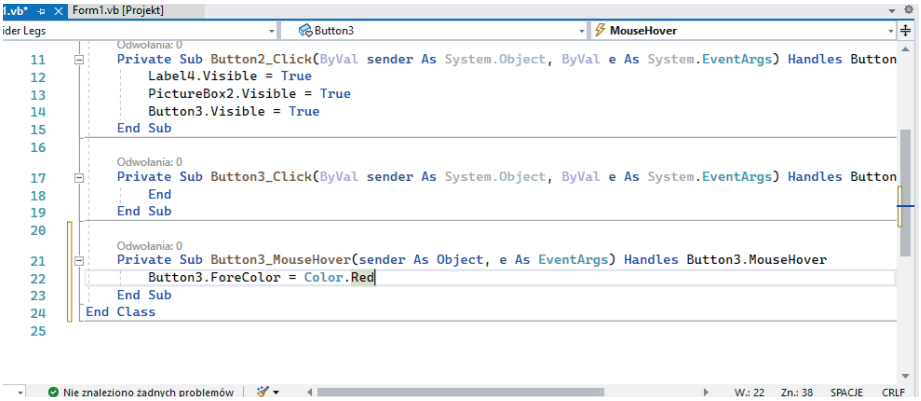


Rysunek 19. Wybór kontrolki do wywołania kodu



Rysunek 20. Wybór zdarzenia wybranej kontrolki do wywołania kodu

Kliknięcie w nazwę wybranego zdarzenia doda do formularza ramy kodu, który będzie wywołany przez to wydarzenie. W te ramy będzie można wprowadzać właściwy kod. Przykładowo, kod, który będzie wywołany przez najechanie kursorem myszy na trzeci przycisk, należy dodać poprzez wybranie z listy znajdującej się po lewej stronie elementu *Button3,* a z listy znajdującej się po prawo polecenia *MouseHover*. Wówczas można dodawać kod, który zmieni kolor tekstu na tym przycisku na czerwony (rys. 21).

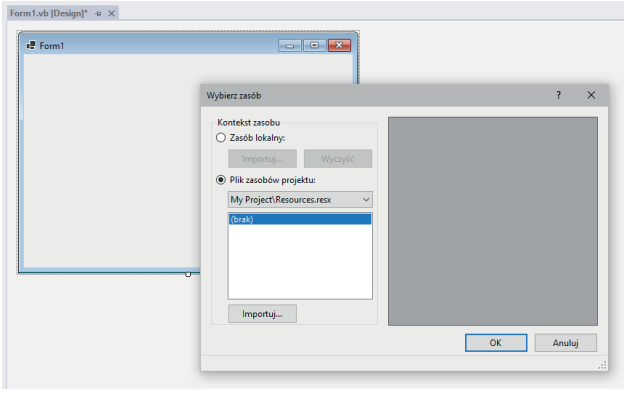


Rysunek 21. Ramy procedury wywołanej najechaniem na *Button3* i polecenie wywoływane przez to zdarzenie

9. Właściwości kontrolek

Każda z kontrolek posiada wiele różnorodnych cech, które opisują jej właściwości. Do najważniejszych z nich zalicza się:

- *Name*
  Właściwość ta decyduje o nazwie danej kontrolki w sposób domyślny, tuż po dodaniu jej z panelu *Przybornik* do formularz. Kontrolki nazywane są tak jak ich nazwy, dodatkowo na końcu zostaje dodana kolejna liczba. Stąd przyciski w bieżącym projekcie nazywają się: *Button1*, *Button2*, *Button3*, etykiety: *Label1*, *Label2*, *Label3*, *Label4,* a pola obrazków: *PictureBox1*, *PictureBox2*. Takie podejście do nazewnictwa kontrolek spełnia swoją rolę w przypadku prostych projektów, natomiast gdy w formularzu będzie musiało znaleźć się np. 15 przycisków czy 30 pól tekstowych, problemem może być odnajdywanie się w nich tylko po kolejnych numerach. Pomocna może być w tej sytuacji tzw. notacja węgierska, która będzie szczegółowo opisana w kolejnym rozdziale.
- *BackColor*
  Ta właściwość decyduje o kolorze tła danej kontrolki. W panelu *Właściwośc*i może on być wybrany spośród trzech grup w zakładkach: *Niestandardowy, Sieć WEB, System*. Pierwsza grupa określa klasyczną paletę 64 barw, druga to tzw. bezpieczna paleta kolorów do zastosowania w Internecie[3] i ostatnia to paleta kolorów, przy wykorzystaniu której widzimy domyślne elementy aplikacji wyświetlane przez system Windows[4].
- *BackgroundImage / Image / Error Image / Initial Image*
  To pozwala na umieszczenie obrazu w tle kontrolki. Akceptowane formaty plików graficznych to: *.gif, *.jpg, *.jpeg, *.bmp, *.wmf, *.png. Wczytanie pliku graficznego odbywa się z wykorzystaniem okna wyboru zasobu (rys. 22).



Rysunek 22. Okno wyboru zasobu

[3] http://pl.wikipedia.org/wiki/Kolory_w_Internecie#Bezpieczna_paleta_216_kolor.C3.B3w (dostęp 10.11.2023).

[4] http://msdn.microsoft.com/en-us/library/0zs041et.aspx dostęp 13.11.2023).

- *Cursor*
  Właściwość ta ustawia wygląd, jaki przyjmie kursor, gdy w uruchomionym programie najedziemy myszką na daną kontrolkę. Rodzaje kursorów zależą od obecnie wybranego schematu w: *Panel Sterowania > Mysz > Wskaźniki.*
- *Dock*
  Ustala sposób zadokowania kontrolki, tzn. przyklejenia jej wskazaną krawędzią (lub krawędziami) do krawędzi formularza. Zadokowanie aktywuje jednoczesną zmianę rozmiaru kontrolki (dopasowanie) przy zmianie rozmiaru formularza.
- *ForeColor*
  Ta właściwość decyduje o pierwszym kolorze kontrolki – najczęściej będzie dotyczyła koloru tekstu. Kolor definiowany jest w identyczny sposób jak kolor tła (*BackColor*).
- *Location*
  Określa parametry położenia kontrolki względem lewego górnego narożnika początku obszaru roboczego formularza (tzn. poniżej paska tytułowego okna). Właściwość ta jest pierwszą z omawianych, którą można rozbić na dwie składowe, rozwijając ją trójkątnym przyciskiem, znajdującym się po lewej stronie jej nazwy. Składowe X i Y określają odległość danej kontrolki od, kolejno, lewej pionowej i górnej poziomej krawędzi użytecznego obszaru formularza.
- *Margin*
  Właściwość ustalająca odstęp między kontrolkami. Gdy przesuniemy dowolną kontrolkę obok innej, w pewnej pozycji wyświetlać się będzie margines, tj. zdefiniowana dla kontrolki odległość odstępu między innymi kontrolkami. Nie jest ona bezwzględnie wymagana. Należy ją traktować jako wielkość pomocniczą przy ustalaniu układu kontrolek w formularzu. Podobnie jak *Location* można ją rozwinąć tak, aby zdefiniować naraz wszystkie krawędzie lub każdą krawędź oddzielnie.
- *MaximumSize/Minimum Size*
  Definiuje maksymalną i minimalną wielkość kontrolki w pikselach.
- *Size*
  Definiuje rozmiar kontrolki.
- *Text*
  Określa tekst znajdujący się na kontrolce.
- *TextAlign*
  Definiuje wyrównanie tekstu w kontrolce. Wybór polega na wskazaniu krawędzi/narożnika kontrolki, do której będzie przylegać tekst.
- *Visible*
  Właściwość ta ustala, czy kontrolka będzie w uruchomionym programie widoczna (*True*) czy niewidoczna (*False*).

Nie są to oczywiście wszystkie dostępne właściwości do modyfikacji, a tylko wybrane – najważniejsze/najczęściej modyfikowane. Więcej kontrolek będziemy poznawać przy okazji kolejnych tematów.

Należy pamiętać, że nie wszystkie właściwości będą aktywne/będą istniały dla różnych kontrolek. Oznacza to, że istnieją właściwości uniwersalne (np. nazwa), które istnieją dla wszystkich kontrolek, ale istnieją też indywidualne, tj. takie, które występują tylko dla konkretnych kontrolek.

Równie istotne jest to, że zmiana niektórych właściwości nie będzie od razu zauważalna w widoku projektowania. Widać to dokładnie w przypadku programu *Spider's Legs*. Cześć elementów jest widoczna w trybie projektowania, a po uruchomieniu programu znika. Dlaczego? Ponieważ ich właściwość widoczności ustawiona jest na *False.* W trybie *Designer* są one widoczne, ponieważ osoba tworząca program musi widzieć, co znajduje się w formularzu na etapie jego projektowania. Po uruchomieniu programu natomiast elementy takie jak: dwa pola obrazków, etykieta z czerwoną odpowiedzią, drugim pytaniem i drugą odpowiedzią oraz przycisk do drugiej odpowiedzi i przycisk zamykający program są niewidoczne. Dopiero działanie przycisków będzie zmieniało właściwość widoczności na *True*, czyli odsłaniało te elementy.

Podsumowując, właściwości ustawione w panelu: *Properties* będą aktywne po uruchomieniu programu, natomiast zmiany tych właściwości dokonujemy za pomocą kodu wywołanego zdarzeniami, np. kliknięciem w przycisk.

Ćwiczenie 4. *PictureBox*

Celem ćwiczenia jest budowa programu wyświetlającego wybrane grafiki. Utwórz nowy projekt. Dodaj do formularza kontrolkę *PictureBox* – służy ona do umieszczania w formularzu zawartości plików graficznych – obrazków.

Do wykonania ćwiczenia będą nam potrzebne przykładowe obrazki. W systemie Windows można znaleźć je w katalogu c:\windows\web\wallpaper. Może to być też przykładowa tapeta ściągnięta z Internetu (np. pexels.com, unsplash.com, pixabay.com czy maxpixel.net). Wstaw do kontrolki obrazek, modyfikując właściwość *Image*. Visual Basic zapyta o sposób umieszczenia pliku graficznego w kontrolce. Do wyboru są dwa tryby*: Zasób lokalny* (*Local Resource)* i *Plik zasobu projektu (Project Resource File).*

Pierwszy z nich, zasób lokalny*,* pozwala na umieszczenie pliku graficznego w zdefiniowanej przez programistę kontrolce na zasadzie pliku lokalnego, zapisanego na dysku. Nie jest wówczas umieszczany w projekcie fizyczny plik graficzny, a tylko odwołanie do niego. Jest to rozwiązanie, które musimy wybierać świadomie. Usunięcie wskazanego pliku graficznego z określonej lokalizacji będzie skutkowało brakiem możliwości wyświetlenia go w programie. Z drugiej strony jednak element może się np. zmieniać (póki nie zmienia się jego nazwa), a i tak będzie wyświetlany.

Drugim sposobem na umieszczenie pliku graficznego w kontrolce *PictureBox* jest wykorzystanie *Pliku zasobu projektu*. Jest to tzw. plik zasobów, obszar pamięci aplikacji, w którym fizycznie umieszczane są zasoby zewnętrzne (obrazki, dźwięki, ikony, itp.). Dodane w ten sposób zasoby – są dostępne z wewnątrz aplikacji. Nie ma żadnych zewnętrznych plików, wszystko jest wewnątrz skompilowanego programu (pliku .exe).

Dodane zasoby pojawiają się jako kolejne pozycje w oknie *Eksplorator rozwiązań* (*Solution Explorer)*. Plik zasobów można edytować (co zrobimy później).

Do naszego *PictureBoxa obrazek* wstawimy sposobem drugim.

I tu pojawia się pierwszy problem. Obrazek może mieć inny rozmiar niż jego pole. Do dopasowania tych wielkości wykorzystamy właściwość *SizeMode* (tryb rozmiaru) kontrolki *PictureBox*. Kolejne ustawienia tej właściwości pozwalają na dopasowanie wstawionego obrazka do pola:

- *Normal* – lewy narożnik pola obrazka pokrywa się z jego lewym narożnikiem, obrazek jest w skali 1 : 1, wszystko, co wypada poza prawy dolny narożnik pola obrazka zostaje ucięte,
- *StrechImage* – (rozciągnij obraz) – obraz zostaje dopasowany do całego obszaru pola obrazka, niezależnie od proporcji boków obrazu,
- *AutoSize* – (rozmiar automatyczny) – pole obrazka zostaje zwiększone do rozmiarów obrazka,
- *CenterImage* – (wyśrodkuj obraz) – pole obrazka zostaje wyśrodkowane względem obrazka, wszystko, co wypada poza pole obrazka zostaje ucięte,
- *Zoom* – podobnie jak *StrechImage* – obrazek zostaje dopasowany do obszaru pola obrazka z tą różnicą, że zachowane zostają jego proporcje.

Utwórz nowy projekt i dodaj do formularza nową kontrolkę *PictureBox*. Na jej przykładzie nauczymy się korzystać z jeszcze innego sposobu wstawiania obrazka, tj. wykorzystamy tu obrazki znajdujące się w Internecie. Aby wyświetlić taki obrazek w kontrolce, należy jego adres umieścić we właściwości *ImageLocation*. Działanie pola obrazka w takim wypadku będzie następujące: po uruchomieniu programu w polu obrazka wyświetli się obrazek zdefiniowany we właściwości *InitialImage* (obrazek początkowy), w tym czasie z lokalizacji sieciowej wczytany będzie obrazek określony w *ImageLocation* i jeśli zostanie pobrany do końca, zostanie wyświetlony w kontrolce *PictureBox.* W przeciwnym wypadku do kontrolki wstawiony zostanie obrazek błędu (zdefiniowany we właściwości *ErrorImage).*

Warto wiedzieć!

W kontrolce *PictureBox* można także implementować obrazy i grafiki dostępne w Internecie. Wówczas wykorzystujemy właściwość *ImageLocation*, wprowadzając w niej adres internetowy obrazu. Możliwość tę możemy także zrealizować za pomocą kodu, który umieścimy np. pod konkretnym przyciskiem. Kod będzie miał wówczas postać:

```
PictureBox1.ImageLocation = „https://strona.pl/grafika.jpg”
```

Ćwiczenie 5. Modyfikacja aplikacji *Spider's Legs*

Celem ćwiczenia jest modyfikacja programu *Spider's Legs* z zadania 3. Będzie ona dwuetapowa, tj. wizualna i logiczna:

1. Dodaj funkcję, która będzie wywołana podwójnym kliknięciem (*DoubleClick*) na etykietę ze słowem „*dwanaście*". Po wykonaniu takiej czynności tekst etykiety ma się zmienić w liczbę „12".
2. Zmień zasadę działania programu na odwrotną. Po starcie programu wszystkie elementy mają być widoczne. Przycisk pierwszej odpowiedzi będzie ukrywać obrazek i etykietę z czerwoną odpowiedzią. Przycisk drugiej odpowiedzi powinien zaś ukrywać drugi obrazek i drugą odpowiedź.
3. Pozmieniaj wygląd aplikacji – położenie elementów (przesuwaj je lub zmodyfikuj przy pomocy właściwości) i kolorystykę kontrolek oraz spolszcz program (pozmieniaj tekst wszystkich elementów angielskich na język polski).

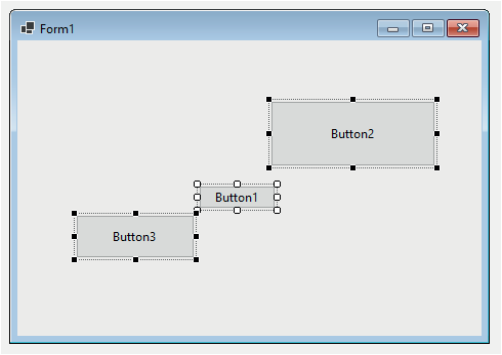
Ćwiczenie 6. *Menu Format*

W tym ćwiczeniu poznasz sposoby modyfikowania położenia elementów w formularzu. Dodaj do formularza trzy przyciski (kontrolka *Button*) o różnych rozmiarach, a następnie rozmieść je tam w dowolny sposób. Klasycznym sposobem przeniesienia kontrolki w inny obszar jest jej przeciągnięcie w miejsce docelowe z przytrzymanym, wciśniętym lewym klawiszem myszy. Visual Studio umożliwia na tym etapie skorzystanie z narzędzia podobnego do magnetycznych prowadnic, tj. pojawiających się w ściśle określonych sytuacjach linii, które przyciągają elementy w charakterystyczne położenia względem innych kontrolek. Przykładowo, przeciągając przycisk obok innego, można zauważyć, że w pewnych miejscach przykleja się on do przycisku nieruchomego (np. gdy ich górne krawędzie stykają się wzdłuż linii prostej). Istnieje także możliwość ustalenia marginesu odległości od każdego z elementów – jest to możliwe dzięki właściwości *Margin* danej kontrolki. Manualne rozmieszczanie elementów jest wygodne, gdy jest niewiele kontrolek do uporządkowania. W przypadku, gdy ich liczba jest znaczna, najlepiej posłużyć się zestawem narzędzi pozwalających na automatyczne układanie elementów. Narzędzia te umieszczone są w pasku *Menu,* w górnej części okna VISUAL STUDIO, w grupie poleceń: *Format*.

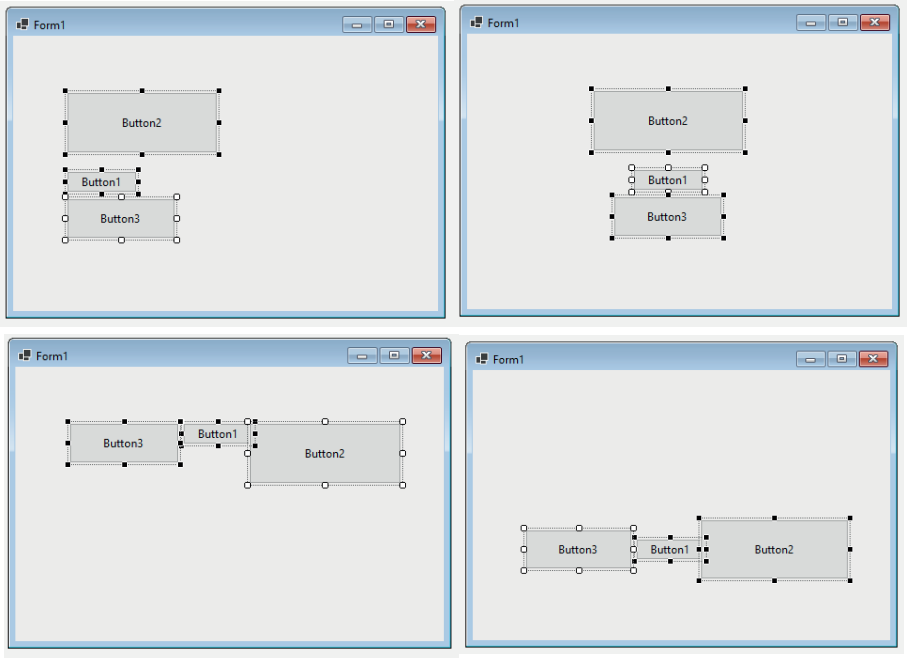
Aby móc z nich skorzystać, należy zaznaczyć wybrane elementy, klikając w nie kolejno z przyciśniętym na klawiaturze klawiszem *Control* (*Ctrl*), lub klasycznie obrysować je z przy pomocy lewego klawisza myszy. Warto zauważyć, że wszystkie zaznaczone elementy będą okrążone ciemnymi kropkami, a jeden wzorcowy, białymi (rys. 23). Dlaczego tak jest, dowiesz się za chwilę.

Pierwszym z poleceń menu *Format* jest grupa: *Wyrównaj*. Wykorzystując je, można rozkładać kilka elementów, wyrównując je ze sobą w poziomie: do lewej krawędzi, do osi środkowej i do prawej krawędzi. Natomiast w pionie możliwe jest wyrównanie do górnej krawędzi, do osi środkowej i do krawędzi dolnej. Wszystkie elementy

będą wyrównywane względem krawędzi (osi) elementu wzorcowego (nieruchomego). Elementem nieruchomym, do którego wyrównywane będą pozostałe kontrolki, będzie właśnie, wcześniej opisany, element wzorcowy, otoczony białymi punktami. Ponadto elementy można wyrównywać do siatki, jednak w celu wykorzystania tej opcji, należy zmienić ustawienia trybu projektowania (Pasek *MENU* > *Narzędzia* > *Opcje...* > *Projektant formularzy system Windows* > *Ustawienia układu: Pokaż siatkę i Przyciągaj do siatki* = *True* oraz Tryb układu = *SnaptoGrid*). Przykładowe zastosowanie wyżej opisanych metod przedstawia rysunek 24**.**



Rysunek 23. Zaznaczanie kontrolek w formularzu



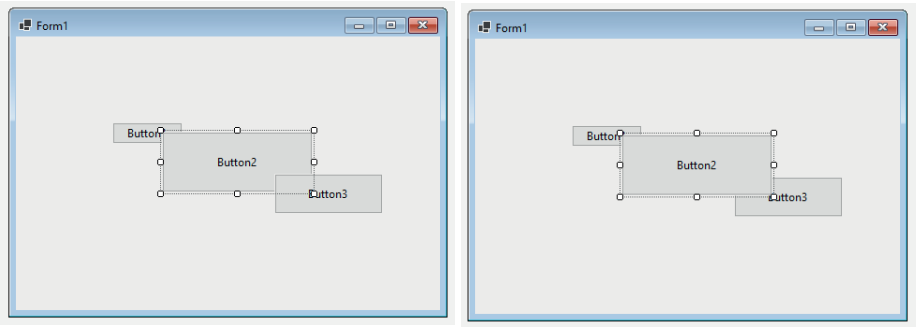
Rysunek 24. Działanie polecenia wyrównywania

Kolejnym poleceniem jednoczesnego formatowania wielu kontrolek jest: *Wyrównaj rozmiar*. Za jego pomocą można zmienić rozmiar zaznaczonych elementów tak, aby dopasować je do elementu wzorcowego. Istnieją cztery tryby dopasowania rozmiaru kontrolek: zmiana szerokości elementów, zmiana wysokości, zmiana obydwu wymiarów jednocześnie oraz dopasowanie rozmiaru do siatki**.**

Dwie następne grupy poleceń dotyczą poziomych i pionowych odstępów między elementami. Projektant aplikacji może wyrównać odstępy, zmniejszyć je, zwiększyć je lub ostatecznie usunąć.

Dalej znajdują się polecenia służące wyśrodkowaniu elementu lub zaznaczonej grupy elementów względem formularza. Jest to przeprowadzane na dwa sposoby: poziomo (*Horizontally*) albo pionowo (*Vertically*).

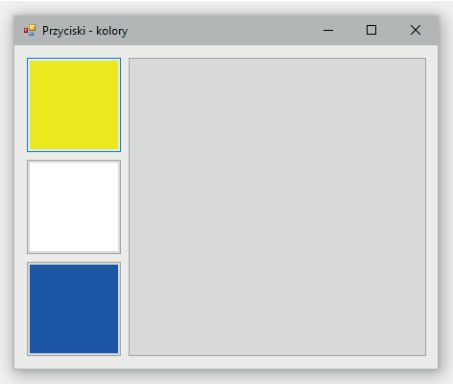
Przedostatnim są polecenia pozwalające na rozmieszczanie elementów „w stosie" – *Kolejność*. Przy budowie interfejsu aplikacji zdarza się, że niektóre kontrolki muszą znajdować się na wierzchu innych (*Przesuń na wierzch*) bądź pod nimi (*Przesuń na spód*). Działanie tych dwóch poleceń przedstawiono poniżej (rys. 25), gdzie dla początkowo rozmieszczonych w stosie przycisków zmieniane jest położenie przycisku *Button2*, tj. przesuwany jest on na wierzch lub na spód względem pozostałych elementów.



Rysunek 25. Polecenie ORDER (kolejność)

Ćwiczenie 7. Przyciski – kolory

W tym ćwiczeniu, wykorzystując omówione wcześniej metody rozmieszczania kontrolek w formularzu, zbuduj aplikację składającą się z czterech przycisków. Trzy pierwsze będą jednakowe, kwadratowe, umieszczone w równym szyku, jeden poniżej drugiego. Czwarty przycisk będzie umieszczony na prawo od kolumny trzech przycisków. Będzie duży i również kwadratowy (co do piksela). Jego górna krawędź powinna być w jednej linii z górną krawędzią pierwszego przycisku, a jego krawędź dolna w jednej linii z krawędzią dolną przycisku trzeciego. Na żadnym z przycisków nie powinno być tekstu (rys. 26).



Rysunek 26. Wygląd programu „Przyciski – kolory”

Pierwszy przycisk będzie miał tło koloru żółtego, drugi będzie biały, trzeci – niebieski. Przycisk kwadratowy na starcie będzie miał domyślny kolor (bez zmian). Działanie programu będzie następujące:

- kliknięcie w jeden z trzech przycisków zmieni kolor tła kwadratowego przycisku na odpowiednio kliknięty kolor,
- kliknięcie w przycisk kwadratowy zamyka program.

10. Notacja węgierska

Podczas programowania z wykorzystaniem języka Visual Basic (a także innych języków) używana jest często tzw. notacja węgierska. Jest to metoda nazewnictwa zmiennych i obiektów w programowaniu. W przypadku kontrolek polega ona na dodawaniu przed właściwą nazwą elementu odpowiedniego przedrostka określającego jej typ, np.:

- btn – *Button*, np.: *btn_zamknij, btnZamknij,*
- cbx – *CheckBox*, np.: *cbx_zielony, cbxZielony,*
- cmb – *ComboBox*,
- lbl – *Label*,
- lbx – *ListBox*,
- pbx – *PictureBox*,
- rbtn – *RadioButton*,
- tbx/txt – *TextBox*.

Przykładowy kod odnoszący się do etykiety wyświetlającej informację i pola tekstowego wyświetlającego wyniki:

```
Private Sub btnReset_Click(sender As Object, e As EventArgs)
Handles btnReset.Click
   lblInformacja.Text = „ Wprowadź dane na nowo...”
   txtWyniki.Clear()
End Sub
```

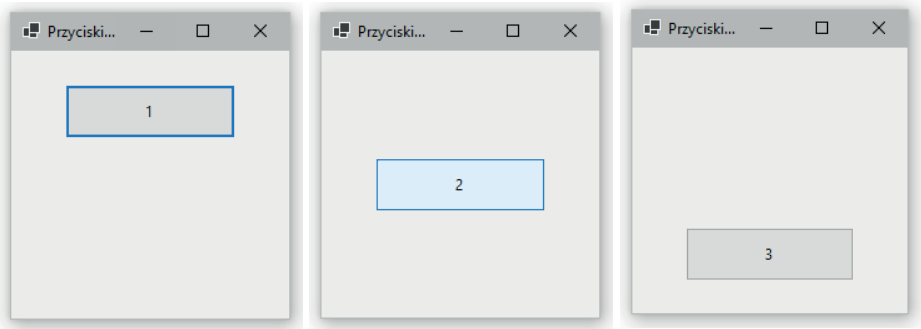
Notacja węgierska ma czynić kod bardziej przejrzystym, łatwiejszym do zrozumienia, ma usprawniać programowanie. Z pewnością jest ona pomocnym rozwiązaniem, gdy w tworzonym projekcie występuje dużo podobnych kontrolek – wówczas już po nazwie projektant wie, jaką funkcjonalność ma pełnić dany element. Notacja węgierska stosowana jest też w przypadku nazewnictwa zmiennych do określania ich typu. Ten przypadek wykorzystania będzie jednak omówiony w dalszej części. Krytycy twierdzą, że notacja węgierska jest zbędnym zanieczyszczaniem kodu, gdyż może prowadzić do niespójności (np. w przypadku zmiennych, gdy zmienimy typ zmiennej, musimy zmienić jej nazwę w całym kodzie).

Ćwiczenie 8. Przyciski – widoczność

Kolejne ćwiczenie będzie podobne do poprzedniego. Jego celem jest budowa aplikacji składającej się z trzech przycisków o równych rozmiarachw pionowym szyku, jeden pod drugim, na środku formularza. Tekst na przyciskach należy ustawić na kolejno: 1, 2, 3. Tytuł formularza należy zmienić na „Przyciski…” (rys. 27).

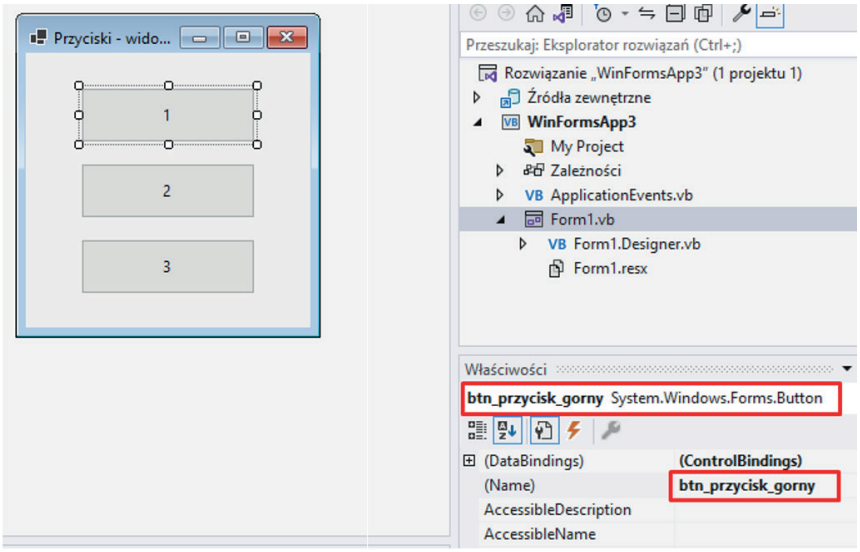
Działanie kodu należy zaprogramować następująco:

- po uruchomieniu programu widoczny jest pierwszy górny przycisk, pozostałe są niewidoczne,
- kliknięcie w pierwszy przycisk wywoła kod wykonujący dwie akcje – ukrycie pierwszego przycisku i wyświetlenie przycisku 2 (wykorzystaj właściwość widoczności),
- kliknięcie w drugi przycisk ukryje go i wyświetli przycisk 3,
- kliknięcie w trzeci przycisk zamknie program (*end*).



Rysunek 27. Działanie programu „Przyciski – widoczność”

Jeśli program jest gotowy i działa zgodnie z treścią polecenia, sprawdzimy jakie zastosowanie ma właściwość (*Name)* kontrolek. Zmień przyciskom nazwy, modyfikując ich właściwości (*Name)*. Wykorzystaj notację węgierską (rys. 28).



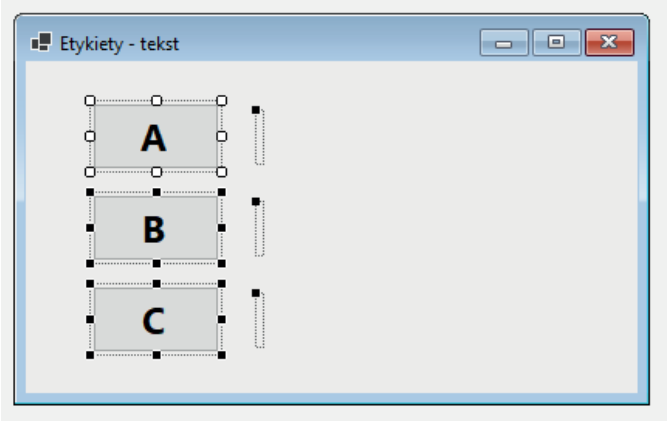
Rysunek 28. Zmiana nazwy kontrolki – zastosowanie notacji węgierskiej

Pozmieniaj nazwy przycisków na *btn_przycisk_gorny*, *btn_przycisk_srodkowy*, *btn_przycisk_dolny* (bez spacji w nazwie!). Przejdź do trybu pisania kodu i sprawdź, co się zmieniło. W przypadku aplikacji takich, jak obecnie budowana, nie ma wielkiej potrzeby nazywania kontrolek. Jednak, gdy ich liczba jest już znaczna, wygodnie jest skorzystać z takiej możliwości. Nie będzie później potrzeby zgadywania czy szukania w formularzu, który przycisk to *Button4* czy w której etykiecie wyświetlić wynik obliczeń: *Label5* czy *Label11*. Po prostu nazwiemy ją na początku z wykorzystaniem notacji węgierskiej, np.: *lbl_wynik_obliczen*. Budowa i późniejsza analiza kodu zrobi się wygodniejsza, a przede wszystkim efektywniejsza.

Ćwiczenie 9. Etykiety – tekst

Celem ćwiczenia jest budowa programu składającego się z czterech przycisków z powiększoną czcionką, kolejno z tekstem: A, B, C oraz czwartego przycisku: KONIEC. Na prawo od każdego przycisku z literami będzie znajdować się etykieta (kontrolka *Label*) bez tekstu (rys. 29). Do nazewnictwa kontrolek wykorzystaj notację węgierską (przykładowo: *lblEtykietaA*, *lblEtykietaB*, *lblEtykietaC*). Działanie programu będzie następujące:

- kliknięcie w przycisk A zmienia tekst na pierwszej etykiecie na „litera A”,
- kliknięcie w przycisk B zmienia tekst na dwóch pierwszych etykietach na „litera B”,
- kliknięcie w przycisk C zmienia tekst na pierwszej i trzeciej etykiecie na „litera C”,
- kliknięcie w przycisk „koniec” zamyka program.



Rysunek 29. Projekt aplikacji „Etykiety – tekst”

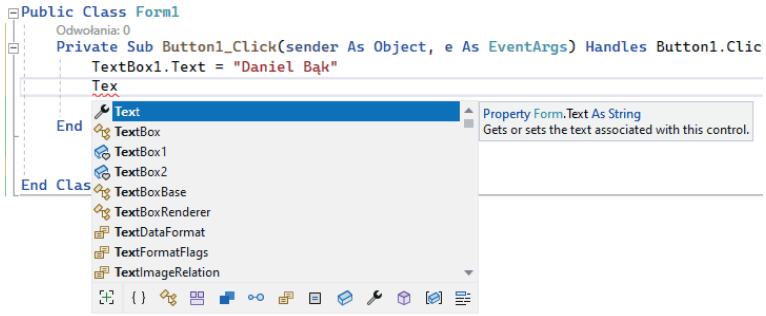
11. System podpowiedzi kontekstowych w Visual Studio

Przy pisaniu kodu aplikacji Visual Studio pozwala na wykorzystanie systemu podpowiedzi. Umożliwia on szybkie wprowadzanie partii typowego kodu poprzez wykorzystanie pojawiających się list zawierających typową składnię kodu (np. nazwy kontrolek, właściwości, obiektów, itp.). Przykładowo, jeżeli użytkownik zechce wprowadzić kod, który będzie zmieniał kolor tła wybranego pola tekstowego, musi użyć polecenia:

```
TextBox1.BackColor = Color.Red
```

Można także wygodnie wprowadzić ten kod, wykorzystując podpowiadające menu kontekstowe. Aby to zrobić szybko, nie wybierając podpowiadanych opcji kursorem myszy (tj. nie odrywając rąk od klawiatury), należy wykonać następujące czynności:

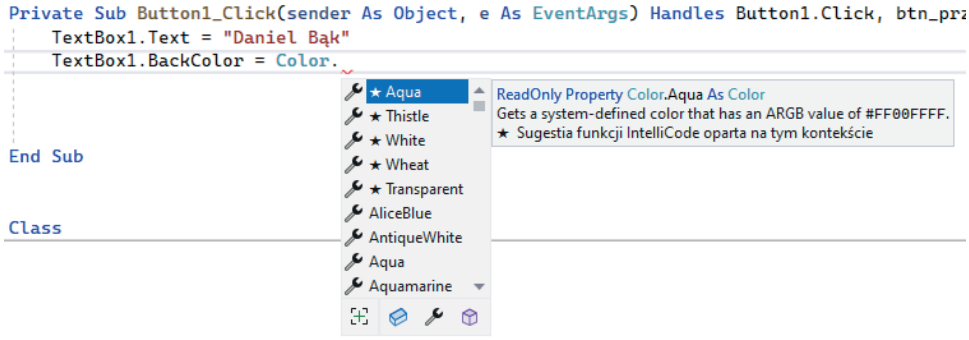
- rozpocząć pisanie nazwy kontrolki, której właściwość będziemy modyfikować (tj. *TextBox1*) – nie trzeba wpisywać całości, tylko kilka pierwszych liter nazwy – „tex" (rys. 30). UWAGA: w VS starszych niż 2017 wygodne jest zaznaczenie na dole pojawiającej się listy: zakładki *Common*.



Rysunek 30. Wpisywanie kodu z wykorzystaniem systemu podpowiedzi

- wybrać odpowiednią pozycję z listy przy pomocy lewego przycisku myszy (z tej opcji korzysta większość początkujących programistów). Jest to jednak niewygodne, ponieważ wymaga oderwania rąk od klawiatury i nakierowania kursora myszy na wybraną pozycję oraz jej kliknięcie. Istnieje wygodniejsze rozwiązanie. Należy przejść w górę lub dół listy podpowiedzi klawiszami strzałek na klawiaturze i stanąć na wybranym kawałku kodu, a następnie kliknąć znak, który powinien wystąpić w kodzie po wskazanym fragmencie (np. kropka, przecinek, znak równości). Ponieważ przykładowy kod po nazwie kontrolki będzie określał nazwę modyfikowanej właściwości, znakiem, który rozdziela te dwa elementy jest kropka. Częstym błędem jest klikanie klawisza *ENTER*. Co prawda, spowoduje to wybranie i wstawienie wybranego kodu, jednak przeniesie kursor do następnej linii.

- podobnie postąpić z nazwą właściwości (*BackColor*), z tą różnicą, że znakiem kończącym będzie tu znak równości.
- w identyczny sposób wybrać nazwę koloru (*Red*). Należy tu zauważyć, że właściwa składnia programu definiuje kolor jako *Color.Red*, ale przykład ten pokazuje, że programista, korzystając z tego typu podpowiedzi, nie musi znać dokładnych składni niektórych poleceń (rys. 31). Po wpisaniu „*Color.*", wybraniu odpowiedniej opcji okna kontekstowego z podpowiedzią oraz naciśnięciu klawisza *ENTER* Visual Basic sam zadba o właściwą składnię.



Rysunek 31. Wpisywanie kodu z wykorzystaniem systemu podpowiedzi

Jak można zauważyć, wykorzystanie systemu podpowiedzi ma kilka zasadniczych zalet:

1. Umożliwia szybsze wprowadzanie partii typowego kodu. Nie trzeba wpisywać pełnej, długiej nazwy kontrolki. Po nabraniu wprawy użytkownik tworzy kod zdecydowanie szybciej.
2. Uniemożliwia wprowadzanie błędnego kodu. System podpowiedzi pozwala wprowadzać tylko prawidłowy kod. Nawet w przypadku próby wprowadzenia błędnego polecenia np. „tekstboks", VS podpowie tylko prawidłowe: *TextBox*.
3. VS podpowiada składnię kodu, którego użytkownik może nie znać. Patrz wyżej: *Color.Red*.

12. Pole tekstowe – *TextBox*

Kontrolka *TextBox* (pole tekstowe) służy do komunikowania się użytkownika z programem. Dotychczas poznane i wykorzystywane kontrolki (np. etykiety) pozwalały programowi na wyświetlenie jakiejś informacji (np. wyników obliczeń, odpowiedzi na pytanie, itd.). Komunikacja ta była więc jednostronna – tylko w stronę użytkownika. Element typu „pole tekstowe" pozwala zarówno na wyświetlenie informacji przez program (jak powyżej opisany przykład z etykietą), jak i na wprowadzanie przez użytkownika informacji niezbędnych do pracy algorytmu programu, takich jak dane wejściowe do obliczeń, hasła, słowne informacje o obiektach, itp.

Poza typowymi właściwościami znanymi z innych kontrolek (*text*, *size*, *position*) pole tekstowe charakteryzuje kilka właściwości specyficznych tylko dla niego. Sprawdź, jakie działanie będzie mieć modyfikacja każdej z poniższych właściwości: *BorderStyle, Cursor, PasswordChar, Enabled, Multiline, WordWrap, ScrollBars, Dock*. Zwróć uwagę, że zmiana niektórych z ww. właściwości będzie widoczna już w widoku DESIGN w Visual Studio, inne zaś będą widoczne dopiero w uruchomionym programie. Wymienione powyżej właściwości odpowiadają za:

- *BorderStyle* – styl obramowania pola tekstowego,
- *Cursor* – rodzaj kursora, który będzie widoczny po najechaniu na kontrolkę (w przypadku kontrolki *TextBox*, różni się on od domyślnej, typowej strzałki),
- *PasswordChar* – maskowanie wprowadzanych do pola znaków przez wybrany symbol, np. *, wprowadzane znaki są wówczas niewidoczne dla użytkownika, ale program może je odczytywać,
- *Enabled* – włączanie/wyłączanie możliwości korzystania z pola przez użytkownika, program wciąż może z niego korzystać – kod wstawiający dane do pola zadziała, ale użytkownik nie będzie mógł nic poprawić ręcznie,
- *Multiline* – wieloliniowość, możliwość wprowadzania tekstu do pola o więcej niż jednym wierszu,
- *WordWrap* – zawijanie tekstu do następnego wiersza po wypełnieniu bieżącego wiersza (uwaga – może nie działać przy wyrównaniu tekstu innym niż do lewej strony),
- *ScrollBars* – paski z suwakami służące do przewijania widoku aktualnego tekstu, w sytuacji gdy nie był on zawijany do następnej linii, można zdefiniować widoczność pionowego, poziomego lub obydwu pasków,
- *Dock* – dokowanie kontrolki w formularzu – rozmieszczenie kontrolki tak, że przylega do określonej lub określonych krawędzi okna, a zmiana rozmiaru okna powoduje zmianę rozmiaru zadokowanej kontrolki poprzez jej dopasowanie do rozmiaru okna.

Ćwiczenie 10. Pole tekstowe – wyrównanie tekstu

Ćwiczenie polega na stworzeniu prostego programu, składającego się z przycisku i pola tekstowego. Kliknięcie przycisku będzie wywoływało kod zmieniający tekst w polu tekstowym na twoje imię i nazwisko. Należy pamiętać, że w kodzie, łańcuch tekstowy (imię i nazwisko) musi znajdować się wewnątrz cudzysłowu. Używając przedstawionego wcześniej systemu podpowiedzi kontekstowych i nie korzystając przy wpisywaniu kodu z myszy, dodaj odpowiednią linię.

Gotowy program należy następnie rozbudować, dodając do niego, trzy nieduże przyciski rozmieszczone w poziomie (jeden obok drugiego) poniżej istniejącego pola tekstowego. Na przyciskach ustaw kolejno tekst: na pierwszym „<<", na drugim „–" i na trzecim „>>". Kod pod tymi przyciskami będzie miał następujące działanie: kliknięcie w pierwszy przycisk wyrówna tekst do lewej strony pola tekstowego, w drugi – do środka, a w trzeci – do prawej. Wykorzystaj do tego odpowiednią właściwość pola tekstowego *TextAlign*. Do wprowadzenia jej prawidłowej składni użyj ponownie systemu podpowiedzi kontekstowej, który pozwoli wskazać kod bez znajomości jego dokładnego zapisu.

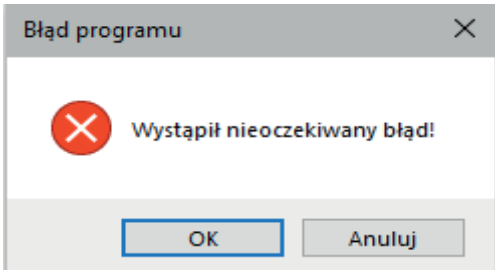
13. Okno informacyjne – *MessageBox*

Oprócz omówionych do tej pory kontrolek, środowisko Visual Studio umożliwia zastosowanie okna informacyjnego, które nazywa się *MessageBox*. Wywołanie tej funkcji pozwala na szybkie uruchomienie typowego okna z podstawowymi elementami, bez potrzeby oddzielnego planowania jego wyglądu. Składnia polecenia jest następująca:

```
MessageBox.Show(Komunikat, Tytuł, Przyciski, Styl okna)
```

Poniższe przykładowe okno (rys. 32), będzie wywoływane za pomocą następującego kodu:

```
MessageBox.Show(„Wystąpił nieoczekiwany błąd!”,
„Błąd programu”, MessageBoxButtons.OKCancel, MessageBoxIcon.Error)
```



Rysunek 32. Przykład okna typu MessageBox

Parametry takie jak tytuł okna, przyciski oraz styl okna są opcjonalne. Jeżeli nie zostaną zdefiniowane przez użytkownika, po prostu nie będą wyświetlane. Należy pamiętać, iż treść komunikatu oraz tytuł okna należy podawać w cudzysłowach. Możliwe jest także łączenie ciągu znaków z innymi elementami (znakami ASCII, parametrami kontrolek czy wartościami zmiennych). Wówczas do łączenia elementów używamy każdorazowo łącznika: & (SHIFT + 7) – przykładowo:

```
MessageBox.Show(„Szerokość przycisku Button1 to:
„ & Button1.Width, „Informacja”)
```

14. Kontrolka wyboru daty i czasu – *DateTimePicker*

Kontrolkę będziemy wykorzystywać wszędzie tam, gdzie konieczne jest wprowadzenie przez użytkownika daty lub czasu. Wprawdzie można byłoby do tego wykorzystać np. kontrolkę *TextBox*, ale w praktyce niosłoby to ze sobą pewne problemy. Użytkownik mógłby np. wprowadzić nazwę miesiąca w swojej dacie urodzenia w różny sposób (marzec, 03, III, 3, mar.). Prowadziłoby to w konsekwencji do znacznej trudności w przetwarzaniu takich danych, np. wyszukaniu wszystkich urodzonych w marcu. Kontrolka *DateTimePicker* z jednej strony umożliwia wyświetlanie daty i czasu w różnych formatach, z drugiej zaś zachowuje te dane w jednym, zawsze takim samym, uniwersalnym formacie. Zauważ, że w przeciwieństwie do dotychczas poznanych kontrolek, zawartość tej kontrolki (data i czas) przechowywane są w właściwości *Value* (nie *Text*). Dodaj do pustego formularza kontrolkę *DateTimePicker.* Uruchom program i zobacz jak ona działa. Prześledź, w jaki sposób zapisywana jest wybrana wartość we właściwości *Value*. Sprawdź właściwości: *MinDate, MaxDate*. Pozwalają one na ograniczanie z góry i z dołu zakresu możliwych do wprowadzenia dat. Jest to o tyle istotne, że to od programisty często zależy, czy użytkownik będzie pracował prawidłowo i nie generował błędnych danych (np. błędnych dat spoza zakresu).

W tej sytuacji należy zawsze przemyśleć jakie wartości daty i czasu może i powinien wybierać użytkownik i ograniczyć mu wybór tylko do tego zakresu. Zmniejszy to możliwość popełniania błędów. Sprawdź dozwolone opcje właściwości *Format* i to, w jaki sposób zmieniają one sposób wyświetlania daty/czasu zapisanego we właściwości *Value* kontrolki. Wybranie formatu *Custom* pozwala na zdefiniowanie własnego szablonu wyświetlania daty/czasu (we właściwości *Custom Format*) przy wykorzystaniu poniższych przypisań:

- *dd* – dzień,
- *dddd* – dzień tygodnia,
- *MM* – miesiąc krótki,
- *MMMM* – miesiąc długi,
- *yy* – rok krótki,
- *yyyy* – rok długi,
- *HH* – godzina w trybie 24 godz.,
- *hh* – godz. w trybie 12 godz.,
- *mm* – minuta,
- *ss* – sekunda.

W uruchomionym programie sprawdź działanie właściwości *SHOWUPDOWN***.**

Ćwiczenie 11. *DateTimePicker* i *MessageBox*

Celem ćwiczenia jest poznanie kontrolek wyboru daty i czasu oraz szybkiego wyświetlania okna informacyjnego.

W nowym formularzu dodaj kontrolkę: *DateTimePicker*. Poniżej dodaj przycisk – wykorzystamy go do połączenia kontrolki *DateTimePicker* z okienkiem typu *MessageBox*, które zostało omówione przed tym ćwiczeniem. Dodaj kod wywoływany kliknięciem w przycisk, wpisz:

```
MessageBox.Show("Twoja data urodzenia to: " & Chr(13)
                & DateTimePicker1.Value)
```

Sprawdź działanie tak zbudowanego programu, następnie zmodyfikuj kod przycisku do takiej postaci:

```
MessageBox.Show("Twoja data urodzenia to: " & Chr(13)
       & DateTimePicker1.Value, "Urodziny")
```

Ponownie sprawdź wynik wywołania okienka typu *MessageBox*, a potem jeszcze raz zmień kod przycisku do takiej postaci:

```
MessageBox.Show("Twoja data urodzenia to: " & Chr(13) & DateTimePicker1.
Value, "Urodziny", MessageBoxButtons.OKRetry, MessageBoxIcon.Information)
```

Zwróć uwagę na zawartość rozwijanego menu, które pojawia się po wpisaniu przecinka po „Urodziny" oraz później po wpisaniu kolejnego przecinka – po `MessageBoxButtons.OKRetry`. Wykorzystanie polecenia `Chr(13)` wyjaśniono w kolejnym rozdziale.

Wynikiem wywołania powyższych funkcji *MessageBox* jest wyświetlenie kolejnych okien, które będą różniły się wyglądem oraz parametrami. Pierwszym komunikatem jest tekst podany w kodzie w cudzysłowach, następnie stoi znak przejścia do następnego wiersza, a na końcu kawałek kodu, który pobiera wartość kontrolki wyboru daty. Te trzy elementy są ze sobą połączone za pomocą znaku „*&*". Przy wpisywaniu kodu zdarzyć się może, że znak „&" będzie zakwalifikowany jako błąd (podkreślony). Należy wówczas zadbać o to, by po jego lewej i prawej stronie znajdowały się spacje. Kolejne komunikaty są wzbogacone o przyciski *OK* i *Retry* oraz zmieniony jest charakter okna na okno informacyjne.

Zmodyfikuj teraz kod wywołania okna informacyjnego. Wejdź do kodu i postaw kropkę po *DateTimePicker1.Value*. Zaobserwuj, jakie właściwości (ikona klucza) będą do wyboru w rozwijalnym polu listy. Możliwe jest wyświetlenie dowolnej część daty: dnia, miesiąca, roku, godziny, dnia tygodnia. Zmień kod tak, żeby w jednej linii wyświetlał się tekst: „Urodziłeś się w" i dzień tygodnia urodzenia (*DayOfWeek*). Sprawdź działanie programu. Dodaj po nazwie właściwości (*DayOfWeek*) tekst *.ToString* (tzn. „do łańcucha tekstowego"). Sprawdź działanie programu. Zaobserwuj różnicę.

Sprawdź, jak w uruchomionym programie zachowa się kontrolka *DateTimePicker* gdy w widoku wyboru daty z kalendarza użyjemy kółka myszy, następnie użyj kółka myszy z wciśniętym na klawiaturze przyciskiem *CTRL*.

15. Tablica znaków ASCII

Wykorzystana w ćwiczeniu 11 formuła *Chr(13*) jest kodem znaku przejścia do następnej linii. Każdy znak na klawiaturze ma swój kod – przykładowo litera „a" ma przypisany kod 97. Ponadto, część znaków, których nie znajdziemy na klawiaturze, takich jak np. symbol brytyjskiego funta (£), stopnia (°) czy średnicy (ø), też ma przypisany kod. Podobnie jest w przypadku znaków niepisanych. Również takie znaki jak koniec linii (klawisz *End*), tabulator (klawisz *Tab*) czy przejście do następnej linii, tzw. powrót karetki (klawisz *Enter*), posiadają swoje reprezentacje w kodach ASCII. Standard ASCII (ang. *American Standard Code for Information Interchange* – Amerykański Standardowy Kod Wymiany Informacji) to kod liczbowy, który jest przyporządkowany każdemu znakowi. Tabela 1 przedstawia zestawienie kodów ASCII najczęściej stosowanych znaków. Tabela kodów ASCII składa się z dwóch grup: znaków o kodach od 0 do 32 (tzw. sterujących) i od 33 do 255 (tzw. drukowanych) i przedstawiona jest poniżej.

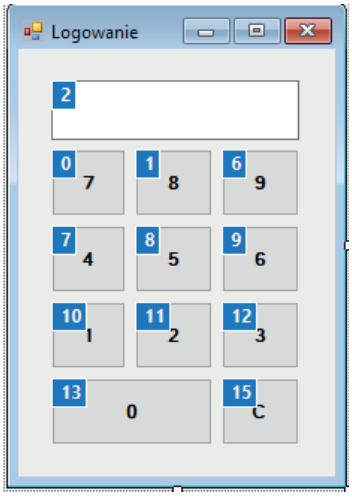
Tabela 1. Kody ASCII podstawowych znaków (opracowanie własne na podstawie [5])

Kod ASCII	**Znak**	**Kod ASCII**	**Znak**	**Kod ASCII**	**Znak**
97	a	65	A	32	spacja
98	b	66	B	33	!
99	c	67	C	34	„
100	d	68	D	35	#
101	e	69	E	36	$
102	f	70	F	37	%
103	g	71	G	38	&
104	h	72	H	39	,
105	i	73	I	40	(
106	j	74	J	41	)
107	k	75	K	42	*
108	l	76	L	43	+
109	m	77	M	44	,
110	n	78	N	45	-
111	o	79	O	46	.
112	p	80	P	47	/
113	q	81	Q	58	:
114	r	82	R	59	;
115	s	83	S	60	<
116	t	84	T	61	=
117	u	85	U	62	>

Kod ASCII	Znak	Kod ASCII	Znak	Kod ASCII	Znak
118	v	86	V	63	?
119	w	87	W	64	@
120	x	88	X	91	[
121	y	89	Y	92	\
122	z	90	Z	93	]
48	0	128	€	94	^
49	1	163	£	95	_
50	2	165	¥	123	{
51	3	36	$	124	|
52	4	169	©	125	}
53	5	153	™	126	~
54	6	176	°	147	“
55	7	152	˜	148	”
56	8	161	¡	149	•
57	9	191	¿	150	˜

16. Kolejność tabulacji – *TabIndex i TabOrder*

Podczas pracy z aplikacjami, które np. wymagają wprowadzenia dużej ilości danych w różne pola tekstowe (np. dane w kwestionariuszu osobowym) oraz innych podobnego typu, może okazać się, że praca w programie będzie wygodniejsza i bardziej efektywna, kiedy będziemy używać klawiatury. W uruchomionym programie, korzystając z klawisza *Tab*, możemy szybko, bez sięgania po mysz, przechodzić na następne pole tekstowe, przycisk czy inną kontrolkę i, np. po wypełnieniu jednego pola (imię), łatwo i szybko, bez odrywania rąk od klawiatury, przeskoczyć do następnego pola (np. drugie imię czy nazwisko). O kolejności przechodzenia między nimi będziemy decydować właściwością *TabIndex*, którą posiada każda kontrolka w formularzu. Jej numeracja jest zgodna z kolejnością dodawania kontrolek, od zera w górę.



Rysunek 33. Modyfikowanie kolejności przeskakiwania po elementach klawiszem TAB

W przypadku, gdy nie pilnowaliśmy odpowiedniej kolejności, budując nasz formularz albo po prostu chcemy zmienić ją na inną/lepszą – możemy to zrobić ręcznie lub korzystając z narzędzia *TabOrder* (*Widok >> Kolejność tabulacji*). Jeśli polecenie jest niewidoczne lub nieaktywne, wróć do formularza i kilka razy naciśnij klawisz *ESC*, aby odznaczyć wszystkie elementy formularza. Po uruchomieniu polecenia wyświetli się obecny, ponumerowany porządek elementów. Klikając w poszczególne elementy, możemy tę kolejność dowolnie zmieniać.

Ćwiczenie 12. Kolejność elementów w formularzu

Celem ćwiczenia jest budowa z przycisków (*Button)* klawiatury numerycznej, znanej z kalkulatorów. Do formularza dodaj 11 przycisków z cyframi od 0 do 9 oraz przyciskiem *C* pod nimi – zachowaj układ jak np. na klawiaturze numerycznej komputera,

tj. kolejność numeracji, rozmiary odpowiednich przycisków muszą być idealnie kwadratowe, klawisz „0” – szerszy (rys. 33). Ustaw kolejność elementów (*TabIndex*) w porządku od góry do dołu. Zwiększ rozmiar czcionki na przyciskach. Ponad przyciskami wstaw pole tekstowe. Zmodyfikuj kierunek wstawiania tekstu na tekst od prawej do lewej – *RightToLeft*. Przypisz każdemu przyciskowi kod wstawiający do pola tekstowego odpowiednią cyfrę (0–9) lub czyszczący pole tekstowe (*C*). Uruchom i przetestuj program.

Należy zastanowić się, jak zmodyfikować program tak, aby po kliknięciu w przycisk nie ginął dotychczasowy stan w polu tekstowym, a wybrana cyfra była dopisywana do pola. Przeanalizujmy działanie przycisku z cyfrą „1”. Zauważ, iż użycie standardowej formuły *TextBox1.Text = „1”* powoduje dodanie konkretnej, pojedynczej cyfry. Jeżeli niezbędne jest zwiększanie pola o kolejne cyfry, musisz zastosować tzw. znak zwiększania += , tj.:

```
TextBox1.Text += „1”
```

lub użyć formuły:

```
TextBox1.Text = TextBox1.Text & “1”
```

Warto wiedzieć!

Istnieje znacząca różnica pomiędzy użyciem polecenia:

`TextBox1.Text += “1”` oraz `TextBox1.Text +=1`

Pierwsze z poleceń (+=”1”) spowoduje dodanie znaku 1 do pola *TextBox1*, natomiast użycie drugiego z kodów (+=1) oznacza dodanie wartości 1 do obecnej wartości, znajdującej się w polu *TextBox1* (o ile znajduje się tam liczba).

Wówczas do pola zostanie dodana kolejna cyfra, bez utraty tego, co w nim wcześniej było. Przetestuj przedstawione powyżej kawałki kodu dla innych przycisków budowanej klawiatury.

Warto wiedzieć!

W celu wyczyszczenia zawartości pola *TextBox* można wykorzystać polecenia:

`TextBox1.Text = “”` lub `TextBox1.Clear()`

Jeżeli natomiast użytkownik chce usunąć jedynie ostatni znak z pola tekstowego, może wykorzystać funkcję *Remove*:

```
TextBox1.Text = TextBox1.Text.Remove(TextBox1.Text.Length – 1)
```

Ćwiczenie 13. Składnia kodu podczas modyfikacji właściwości kontrolek

Celem ćwiczenia jest zbudowanie aplikacji składającej się z dwóch pól tekstowych i dwóch przycisków. Rozmieść pola tekstowe obok siebie, a pod każdym polem umieść przycisk. Na lewym przycisku ustaw tekst „>>", na prawym „<<". Dodaj do programu kod, którego działanie będzie następujące: kliknięcie w przycisk „>>" powoduje wstawienie w prawe pole tekstowe tekstu, który stoi w polu lewym, kliknięcie w przycisk „<<" powoduje wstawienie w lewe pole tekstowe tekstu, który stoi w polu prawym. Pamiętaj o zasadzie: *Element.Właściwość = Wartość*. Który element ma się zmienić? Jaką ma przyjąć wartość? *TextBox1.Text = TextBox2.Text* i na odwrót. Uruchom program i sprawdź, czy działa prawidłowo.

Dodaj drugą linię kodu do obydwu przycisków, która rozszerzy działanie wywołane klikaniem w przyciski tak, żeby tekst po wstawieniu w przeciwne pole znikał z pola pierwotnego, tj. żeby pole pierwotne było czyszczone. Da to wrażenie przenoszenia tekstu z pola do pola.

Zmodyfikuj program, dodając blokadę zakresu zmiany rozmiaru formularza, tj. właściwości *MinimumSize* (300;300) i *MaximumSize* (800;600).

Sprawdź i zaobserwuj, co zmienia w formularzu zmiana każdej z następujących właściwości:

- *FormBorderStyle* – styl obramowania okna (m.in. style okna narzędziowego bez przycisków minimalizowania i maksymalizowania, style okna bez lub z możliwością zmiany rozmiaru, styl bez obramowania i inne),
- *SizeGripStyle* – trójkąt w prawym dolnym rogu formularza, symbolizujący możliwość zmiany rozmiaru okna,
- *Icon* – definiowanie ikony okna,
- *ShowIcon* – widoczność ikony okna,
- *MinimizeBox* – widoczność ikony minimalizowania okna,
- *MaximizeBox* – widoczność ikony maksymalizowania okna,
- *ControlBox* – widoczność zestawu ikon zmiany rozmiaru okna i zamykania okna,
- *ShowInTaskbar* – widoczność uruchomionej aplikacji na pasku zadań Windows,
- *Opacity* – pokrycie, nieprzejrzystość okna.

Uwaga – zmiana niektórych z właściwości może być widoczna dopiero w uruchomionym programie.

17. Suwak poziomy– *HScrollBar*

Kontrolka *HScrollBar* to klasyczny poziomy pasek z suwakiem. Suwak można przesuwać, łapiąc go kursorem myszy. Istnieją jeszcze dwa bardziej precyzyjne sposoby zmiany położenia suwaka:

- o małą wartość – przez kliknięcie kursorem myszy w strzałki na końcach paska lub przesunięciestrzałkami z klawiatury,
- o dużą wartość – przez kliknięcie kursorem myszy w puste pole na pasku dookoła suwaka lub użycie klawiszy *PageUp*, *PageDown* z klawiatury.

Najważniejszymi właściwościami dla kontrolki *HScrollBar* są:

- *Minimum* – minimalna wartość suwaka, wskazuje ją suwak wysunięty maksymalnie w lewo na pasku,
- *Maximum* – maksymalna wartość suwaka, wskazywana przez prawy koniec paska,
- *Value* – położenie suwaka określone jako liczba z zakresu (minimum, maksimum),
- *SmallChange* – wielkość, o jaką przesuwa się suwak przy wyżej opisanej „małej zmianie",
- *LargeChange* – wielkość, o jaką przesuwa się suwak przy wyżej opisanej „dużej zmianie", właściwość ta umożliwia jednocześnie zdefiniowanie szerokości suwaka.

Tę kontrolkę najczęściej wykorzystujemy do wybrania jakiejś wartości z zaplanowanego przez nas zakresu, np. jako danej wejściowej do dalszych obliczeń w naszym programie.

Ćwiczenie 14. Suwaki, zatrzymanie programu

Zadanie 1. Odczytywanie wartości

Celem ćwiczenia jest zapoznanie się z suwakami jako kontrolkami służącymi do wyboru wartości ze zdefiniowanego zakresu.

Dodaj do formularza kontrolkę *HScrollBar*. Jeśli nie jest widoczna w grupie *Typowe kontrolki* (*CommonControlls)*, poszukaj jej w: *Wszystkie kontrolki Windows Forms* (*AllWindowsForms)*. Zwiększ rozmiar kontrolki, tj. rozsuń ją na szerokość ok. 2/3 szerokości formularza. Uruchom program, aby sprawdzić, jak kontrolka działa.

Dodaj poniżej drugi suwak i ustaw suwakom następujące parametry:

- zakres: 0–1000, *SmallChange*: 10, *LargeChange*: *200*,
- zakres: 10000–50000, *SmallChange*: 100, *LargeChange*: 2000.

Na prawo od suwaków wstaw dwie etykiety. Pod spodem dodaj przycisk: „Wyczyść". Utwórz kod, który będzie wykonywany przy przesuwaniu każdego suwaka, tj. wstawienie do etykiety na prawo od suwaka wartości *Value* kontrolki.

Zadaniem przycisku będzie zerowanie obu pasków i etykiet (kod ustawiający każdemu z pasków jego wartość na minimum). Etykiety na starcie mają mieć wartość minimum suwaków, w tym celu, klikając dwukrotnie w puste miejsce w formularzu, dodaj do procedury wywołanej wyświetleniem tego formularza (tj. na starcie programu) kod, który wyczyści tekst na etykietach.

Uwaga:

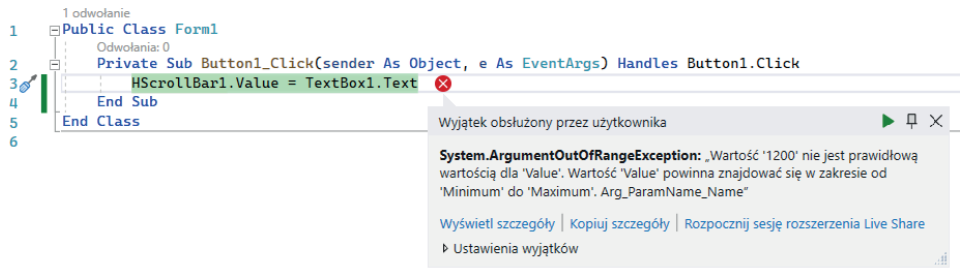
- właściwość *Minimum* wskazuje położenie początku paska, należy się nią posługiwać zamiast wpisywania konkretnej wartości przy zerowaniu suwaków,
- właściwość *Value* wskazuje położenie określone przez lewą krawędź suwaka, może mieć to wpływ na wyniki odczytywane na prawym końcu paska.

Uruchom program i sprawdź, czy działa poprawnie.

Zadanie 2. Przesuwanie suwaka

Celem ćwiczenia jest modyfikacja zadania z poprzedniego ćwiczenia tak, aby można było wartościami podanymi liczbowo przez użytkownikaprzesuwać suwak na pasku na określoną pozycję. Program zmodyfikujemy, dodając obok etykiet pola tekstowe i obok przyciski: „Ustaw". Zmieniony zostanie również kod. Teraz program będzie działał dodatkowo w drugą stronę. Kliknięcie w przycisk „Ustaw" przesunie suwak odpowiedniego paska do właściwej pozycji, wskazanej przez wartość w polu tekstowym.

Na tym przykładzie zaobserwujemy również tryb błędów w Visual Basic. W niektórych przypadkach program może się zawiesić. Rysunek 34 przedstawia widok trybu kodu po wywołaniu błędu przepełnienia (do suwaka próbowano wstawić wartość przekraczającą wartość *Maximum*). W takiej sytuacji Visual Basic ukrywa uruchomiony program i wraca do fragmentu kodu bezpośrednio odpowiadającego za błąd, zakreślając całą linię na zielono.



Rysunek 34. Tryb błędów

Jak widać powyżej, wyświetli się okno informacyjne z nazwą i opisem błędu, a także ze wskazówkami, jak można go usunąć i co zrobić, żeby błędy podobnego typu nie występowały w przyszłości. Należy też zwrócić uwagę na górny pasek z ikonami w Visual Studio. Obok ikony uruchamiania programu (zielony trójkąt), pojawi się aktywna ikona, służąca do jego zatrzymania (czerwony kwadrat). Dzieje się tak, gdyż zawieszony program jest wciąż uruchomiony. Należy posłużyć się czerwoną ikoną w celu zatrzymania zawieszonego programu i powrotu do Visual Studio.

18. Zmienne

Zmienna jest wydzielonym obszarem pamięci, który posiada swoją nazwę i może przechowywać określony rodzaj informacji. System operacyjny musi zarezerwować jakiś obszar pamięci dla każdej zmiennej i do jej obsługi. Dla zmiennych o małym zakresie rezerwuje małą ilość pamięci, dla dużych zakresów – więcej.

W programach takich jak ten, który będziemy budować, będą trzy zmienne, więc nie byłoby żadnego problemu, gdybyśmy nawet wybrali zmienne „na wyrost". Ale wyobraźmy sobie program operujący na dwustu zmiennych i wykonujący skomplikowane operacje matematyczne… Programista musi „powiedzieć" systemowi, jakie maksymalne zakresy zmiennych są wymagane do prawidłowej pracy programu, aby system mógł zarezerwować tyle pamięci, ile trzeba, ale musi jednocześnie wybrać najmniejszą zmienną, jaką tylko się da.

Ćwiczenie 15. Program wykonujący podstawowe operacje arytmetyczne

Celem ćwiczenia jest zapoznanie się z metodą budowy podstawowych aplikacji wykonujących obliczenia matematyczne oraz z pojęciem, rodzajami i wykorzystaniem zmiennych. Zbuduj prosty program typu: *dodaj/odejmij*. W tym celu umieść w formularzu dwa pola tekstowe, pod spodem przyciski z tekstem: „dodaj", „odejmij", „pomnóż", a pod nimi etykietę z tekstem „wynik = " i pole tekstowe.

Warto wiedzieć!

Visual Basic umożliwia przypisanie zmiennym wartości w bardzo łatwy sposób – poprzez podanie nazwy zmiennej i pola, w które ma być ona pobrana:

```
Liczba1 = TextBox1.Text
```

Niekiedy jednak można spotkać się z kodami, które posiadają także wzbogacenie o odpowiednie funkcje konwersji wartości do odpowiedniego typu danych. Przykładowo, dla zmiennej typu *Integer*:

```
Liczba1 = CInt(TextBox1.Text)
```

lub dla zmiennej typu *Double*:

```
Liczba1 = CDbl(TextBox1.Text)
```

CInt oznacza *Convert To integer* (przekonwertuj do zmiennej typu całkowitego), zaś ***CDbl*** oznacza *Convert do Double* (przekonwertuj do zmiennej typu zmiennoprzecinkowego).

Decyzja o formie przypisywania zmiennym wartości zależy tylko i wyłącznie od programisty.

Kod będziemy budować krok po kroku, wyjaśniając na bieżąco kolejne linie programu. Utwórz teraz procedurę wywołaną kliknięciem w przycisk: „dodaj". Cały kod funkcji dodawania będzie umieszczony w tej procedurze. Jako pierwszą linię wprowadź:

```
pierwsza = TextBox1.Text
```

Linia ta będzie wstawiała, do zmiennej o nazwie: „pierwsza", zawartość pola tekstowego 1, a dokładniej jego właściwości: *Text*.

Jednak po przejściu do kolejnej linii kodu Visual Basic podkreśli tę linię falistą kreską, sygnalizując błąd składni:

```
pierwsza = TextBox1.Text
```

Jednocześnie na dole, w oknie: *Error list* pojawi się informacja o charakterze błędu:

```
'pierwsza' is not declared
```

Dzieje się tak dlatego, że Visual Basic nie rozumie, co w kodzie oznacza tekst: „pierwsza". Jeśli chcemy ten tekst wykorzystywać jako nazwę zmiennej, musimy najpierw tę zmienną zadeklarować, tj. pokazać programowi, że wewnątrz kodu będzie znajdował się tekst: „pierwsza" i że będzie to zmienna przechowująca określony typ danych, z wcześniej określonego zakresu. Do deklarowania zmiennych służy polecenie: *Dim*. Należy jednak pamiętać o tym, że zmienna musi być zadeklarowana, zanim zostanie wywołana w kodzie. Dlatego też trzeba dodać poniższą linię, powyżej linii wstawiającej zawartość pola tekstowego do zmiennej:

```
Dim pierwsza As Short
```

Warto wiedzieć!

Każda zmienna posiada swoją własną, unikalną nazwę. Należy pamiętać, iż podczas nazywania zmiennych, należy unikać polskich znaków (ą, ę, ź, ć, itp.), a także znaków specjalnych (@, # , $,) oraz spacji. Spacja może zostać zastąpiona w nazwie zmiennej poprzez znak podkreślnika:

```
Dim moment_sily As Double
```

Podczas nazywania zmiennych można także posługiwać się notacją węgierską (ćw. 8–9). Przykładowo, dla zmiennej typu *Integer*:

```
Dim intZmienna1 As Integer
```

lub dla zmiennej typu *Double*:

```
Dim dblZmienna1 As Double
```

Wówczas programista poprzedza nazwę zmiennej odpowiednim prefixem, który jest skrótem typu danych zmiennej.

Po zadeklarowaniu zmiennej, podkreślenia w kodzie i błędy w panelu: *Error List* znikną. Typ danych określane przy deklaracji: *Dim <nazwa> As <typ danych>* dobiera się na podstawie tabeli 2, zestawiającej dostępne typy oraz zakresy przechowywanych wartości. Przedstawiono tam również składnię przykładowych poleceń deklaracji. Należy pamiętać, że nazwa zmiennej nie może zawierać spacji. Do separowania należy użyć np. znaku podkreślenia _ (pot. podkreślnik/„podłoga").

Tabela 2. Typy danych, zakresy przechowywanych danych, przykłady deklaracji i zastosowania

Typ	**Wielkość**	**Zasięg**	**Przykładowe zastosowanie**
Short	16-bitowy	Od -32 768 do 32 767	Dim Ptaki As Short Ptaki = 12500
UShort	16-bitowy	Od 0 do 65 535	Dim Dni As UShort Dni = 55000
Integer	32-bitowy	Od -2 147 483 648 do 2 147 483 647	Dim Insekty As Integer Insekty = 37500000
Ulnteger	32-bitowy	Od 0 do 4 294 967 295	Dim Chińczycy As Ulnteger Chińczycy = 3000000000
Long	64-bitowy	Od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	Dim Populacja As Long Populacja = 4800000004
ULong	64-bitowy	od 0 do 18 446 744 073 709 551 615	Dim Gwiazdy As ULong Gwiazdy = 1800000000000000000
Single	32-bitowy zmienno-przecinkowy	od 3,402823E38 do 3,4028235E38	Dim Cena As Single Cena = 899.99
Double	64-bitowy zmienno-przecinkowy	od -1,79769313486231E308 do 1,79769313486231E308	Dim Pi As Double Pi = 3.1415926535
Decimal	128-bitowy	Wartości do +/-79228 **x** 1024	Dim Dług As Decimal Dług = 7600300.50
Byte	8-bitowy	od 0 do 255 (bez wartości ujemnych)	Dim Pokój As Byte Pokój = 13
SByte	8-bitowy	od -128 do 127	Dim Tempretura As SByte Temperatura = -20
Char	16-bitowy	Jakikolwiek symbol Unicode w zakresie 0–65 535	Dim ZnakUnicode As Char ZnakUnicode = „A"
String	Zazwyczaj 16-bitowy dla każdego znaku	od 0 do około 2 miliardów 16-bitowych znaków Unicode	Dim Pies As String Pies = „jamnik"
Boolean	16-bitowy	True lub False (konwersja zmienia 0 na False a inne wartości na True)	Dim Prawda as Boolean Prawda = True
Dote	64-bitowy	od 1 stycznia 0001 do 31 grudnia 9999	Dim Urodziny as Date Urodziny = #3/1/1963*

W naszym programie dodamy teraz linię wstawiającą do zmiennej: „druga" zawartość drugiego pola tekstowego. Podobnie jak poprzednio, zmienną należy wcześniej zadeklarować.

Dobrą regułą deklarowania zmiennych jest umieszczanie deklaracji na początku kodu. Przenieś linie z deklaracjami do góry. Obecnie w dwóch liniach znajdują się deklaracje dwóch zmiennych. Jeśli deklarujemy zmienne tego samego typu, możemy je umieszczać w jednej linii, po przecinkach. Deklaracja będzie więc wyglądać następująco:

```
Dim pierwsza, druga As Short
```

Dalej wpiszemy linię kodu, wstawiającą do zmiennej wynik (trzeba będzie ją również zadeklarować), czyli sumę zmiennych „pierwsza" i „druga".

```
wynik = pierwsza + druga
```

Ostatnia linia kodu będzie wyświetlała wynik w określonym polu tekstowym:

```
TextBox3.Text = wynik
```

Sprawdźmy działanie tak zaprogramowanego przycisku. Spróbuj dodać 2 i 3. Spróbuj dodać 2,5 i 3,5. Dlaczego drugi wynik jest błędny? Zmień typy danych na takie, które przechowują części ułamkowe.

Rozpoczynając od znaku apostrofu (‘), dodaj w kodzie, na końcach poszczególnych wierszy, komentarze opisujące typowe bloki logiczne:

- deklaracja zmiennych,
- wstawienie danych wejściowych do zmiennych,
- obliczenia wyników,
- wyświetlenie wyników.

Zwróć uwagę, że tekst, wprowadzony wewnątrz kodu w postaci komentarza, nie jest w żaden sposób analizowany przez VB, jest formatowany w kolorze zielonym i wyraźnie odznacza się w kodzie funkcjonalnym.

Uzupełnij teraz program o funkcję odejmowania i mnożenia, zgodnie z powyższym schematem budowy kodu. Ostateczny kod będzie wyglądał następująco:

```
Public Class Form1
   Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
   As System.EventArgs) Handles Button1.Click
       Dim pierwsza, druga, wynik As Single
       pierwsza = TextBox1.Text
       druga = TextBox2.Text
       wynik = pierwsza + druga
       TextBox3.Text = wynik
    End Sub
```

```
    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles Button2.Click
        Dim pierwsza, druga, wynik As Single
        pierwsza = TextBox1.Text
        druga = TextBox2.Text
        wynik = pierwsza - druga
        TextBox3.Text = wynik
    End Sub

    Private Sub Button3_Click(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles Button3.Click
        Dim pierwsza, druga, wynik As Single
        pierwsza = TextBox1.Text
        druga = TextBox2.Text
        wynik = pierwsza * druga
        TextBox3.Text = wynik
    End Sub
End Class
```

Możemy w tym kodzie dokonać jeszcze jednej modyfikacji dotyczącej deklarowania zmiennych. W tej chwili, w każdej procedurze (dodawanie, odejmowanie) musimy niezależnie deklarować zmienne. Zmienna zadeklarowana w procedurze przycisku dodawania jest niewidoczna (niezadeklarowana) w procedurze przycisku odejmowania. Zmienne te deklarowane są obecnie w sposób tzw. lokalny. Aby uprościć program, wyciągniemy deklaracje poza procedury. Kod powinien wówczas wyglądać tak:

```
Public Class Form1
Dim pierwsza, druga, wynik As Single
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles Button1.Click
        pierwsza = TextBox1.Text
        druga = TextBox2.Text
        wynik = pierwsza + druga
        TextBox3.Text = wynik
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles Button2.Click
        pierwsza = TextBox1.Text
        druga = TextBox2.Text
        wynik = pierwsza + druga
        TextBox3.Text = wynik
    End Sub
```

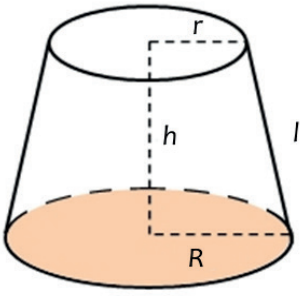
```
    Private Sub Button3_Click(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles Button3.Click
        pierwsza = TextBox1.Text
        druga = TextBox2.Text
        wynik = pierwsza * druga
        TextBox3.Text = wynik
    End Sub
End Class
```

Zmienne, z lokalnych, stały się globalne – są teraz zadeklarowane w całym kodzie tego formularza.

Jaki jest w ogóle sens deklarowania zmiennych? Przecież proste zadanie sumujące moglibyśmy zapisać za pomocą jednej linijki kodu:

```
TextBox3.Text = TextBox1.Text + TextBox2.Text
```

Rzeczywiście, VB (w przeciwieństwie do wielu innych języków) pozwoliłby na wykonanie takiego kodu i jeśli dane wejściowe byłyby liczbami, zsumowałby je i wyświetlił wynik. Jest to właśnie jedna z cech tego prostego języka. Pozwala on na więcej niż bardziej zaawansowane języki. Nie należy jednak wszystkiego bezrefleksyjnie upraszczać. Taki kod, wbrew pozorom, nie jest prawidłowy. Będzie to widać to np. w sytuacji konieczności przeprowadzenia, choćby odrobinę trudniejszych, obliczeń, np. objętości stożka ściętego (rys. 35). Kiedy liczba danych wejściowych rośnie do 4, posługiwanie się wyłącznie odniesieniami do pól tekstowych, do tego stopnia zaciemni widok kodu, że trudno go będzie przeczytać, zrozumieć, znaleźć ew. błędy i je poprawić.



$$V=\pi/3\cdot h(R^2+R\cdot r+r^2)$$

gdzie:
V – objętość stożka ściętego,
h – wysokość stożka ściętego,
R, r – promienie podstaw stożka.

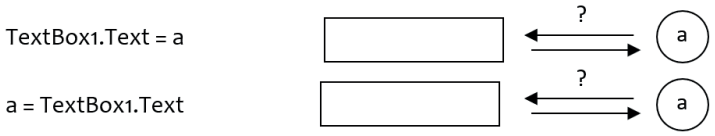
Rysunek 35. Obliczenia geometryczne stożka ściętego

W uruchomionym programie spróbuj kliknąć w przycisk: *Dodaj*, w momencie, gdy jedno z pól tekstowych będzie puste. Program zatrzyma się i wskaże linię wywołującą błąd. Mogłoby się wydawać, że puste pole oznacza zero. Tak jednak nie jest. Pojęcie: *NULL* (nic) to nie to samo, co zero.

Najlepszym przykładem ilustrującym owe pojęcie jest sytuacja, w której codziennie notujemy bieżącą temperaturę powietrza. Jednego dnia będzie +3°C, drugiego 0°C, trzeciego 7°C, a czwartego nie wpiszemy nic. *Nic* to nie zero. Zero to konkretna liczba. Program zatrzymał się, dlatego że deklarując zmienne, ustaliliśmy ich typ na któryś z typów przechowujących liczby. Teraz, klikając w przycisk, wywołujemy procedurę, która do zmiennej próbuje wstawić: *NULL* – nie liczbę. Program zachowa się podobnie, jeśli spróbujemy wywołać zaprogramowane procedury przy znajdujących się w polach tekstowych literach lub innych znakach.

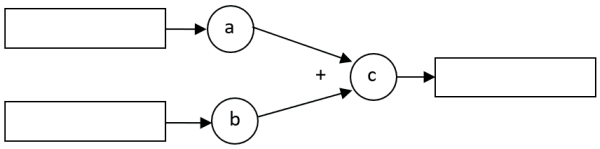
Podsumowanie:

1. Wskaż odpowiedni kierunek strzałki – skąd zawartość wstawi się dokąd (rys. 36):



Rysunek 36. Kolejność działania linii przypisania

2. Omów schematyczne działanie programu (rys. 37): elementy potrzebne do kolejnych kroków i ich rezultaty, kolejność poszczególnych kroków



Rysunek 37. Schemat programu sumującego

19. Pole grupy – *GroupBox*

Kontrolka: *GroupBox* i kontrolka: *Panel* są reprezentantami grupy kontrolek: Kontenery (*Containers*). Elementy te pełnią w programach funkcję oddzielania części kontrolek formularza od innych. Różnica między nimi polega na tym, że pole grupy może posiadać tekst nagłówkowy, a panel nie. Domyślny wygląd ramki kontrolki jest też różny. Pole grupy jest obrysowane przerywana linią, natomiast panel jest (w uruchomionym programie) niewidoczny. Brak obrysu panelu związany jest z jego rolą w formularzu. Ma on wydzielać kilka kontrolek w sposób logiczny, ale nie wizualny. Stosujemy go między innymi tam, gdzie mamy opcje wyboru (*RadioButton*, *CheckBoxy* i inne). Zastosowanie panelu pozwala między innymi wydzielić logicznie opcje różnych grup, np. gdy mamy 3 opcje wyboru koloru towaru, a obok 3 opcje wyboru formy płatności za towar. Dodatkowo, panel może posiadać suwaki do przewijania zawartości kontrolki, w przypadku, gdyby wykraczała ona poza rozmiar kontrolki (właściwość: *AutoSize*).

Ćwiczenie 16. rozwiązywanie równania kwadratowego

W ćwiczeniu tym należy zbudować aplikację, która będzie wyznaczała deltę i rozwiązania równania kwadratowego typowej postaci: $ax^2 + bx + c = 0$, zgodnie z następującymi wzorami:

$$\Delta = b^2 - 4ac, \qquad x_1 = \frac{-b+\sqrt{\Delta}}{2a}, \qquad x_2 = \frac{-b-\sqrt{\Delta}}{2a}.$$

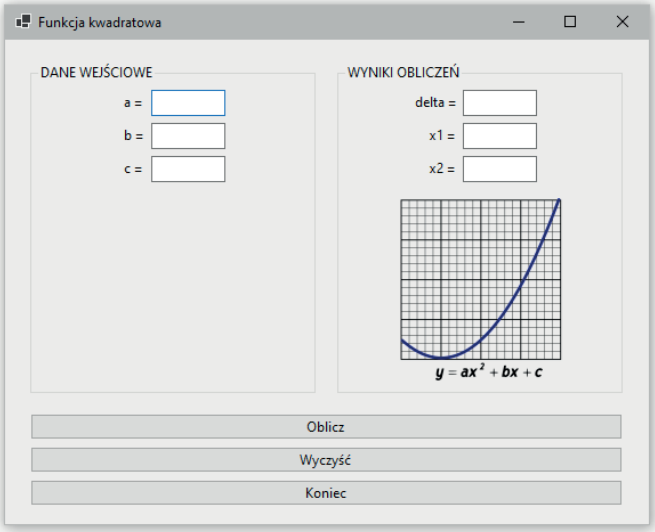
W tym ćwiczeniu nie będziemy jeszcze zwracać uwagi na znak delty ani inne warunki istnienia równania kwadratowego i jego rozwiązań.

Wygląd aplikacji ma opierać się o dwie kontrolki: *GroupBox*. Formularz ma być podzielony na pół. W polu grupy o nazwie: „*Dane Wejściowe*”, znajdującym się po lewej stronie formularza, będą znajdować się etykiety: „*a* = ”, „*b* = ” i „*c* = ”, obok nich pola tekstowe do wprowadzania danych wejściowych. W polu grupy o nazwie: „*Wyniki Obliczeń*” znajdą się etykiety opisujące wyniki i pola tekstowe, w których te wyniki będą wyświetlane. Dodatkowo, poniżej wyników, ale wewnątrz pola grupy znajdzie się obrazek paraboli, wyświetlany jako obraz z Internetu (w przypadku problemów z połączeniem – któraś z tapet systemu Windows). Poniżej obu pól grupy, na środku znajdą się trzy przyciski: *Oblicz*, *Wyczyść* i *Koniec* (rys. 38).

Działanie programu będzie opierać się o zdarzenia wywołane klikaniem w któryś z przycisków. I tak: kliknięcie w przycisk: „Koniec” – zamknie program, a przycisk: „Wyczyść” opróżni wszystkie pola tekstowe. Przycisk: „Oblicz” będzie natomiast przeprowadzał (podobnie jak wcześniejszy program sumujący liczby) następujące czynności:

- deklaracje zmiennych,
- wstawienie do zmiennych wartości, pobranych z odpowiadających im pól tekstowych,

- obliczenia wyników,
- wstawienie uzyskanych wyników do wynikowych pól tekstowych.



Rysunek 38. Wygląd aplikacji do wyznaczania pierwiastków funkcji kwadratowej (opracowanie własne na podstawie [6])

W równaniach na pierwiastki i równania kwadratowego występuje funkcja pierwiastka kwadratowego. Tymczasowo poradzimy sobie z tym problemem, znając zależność matematyczną:

$$\sqrt[n]{x} = x^{\frac{1}{n}}, \text{ więc } \sqrt[2]{x} = x^{\frac{1}{2}} = x^{0,5} \quad \text{czyli} \quad x^{\frac{1}{2}} \quad \text{lub} \quad x\text{^}0.5$$

Zestaw danych dla testowania poprawności programu przedstawiono w poniższej tabeli:

Tabela 3. Przykładowe dane i rozwiązanie równania kwadratowego

a	b	c	Δ	x_1	x_2
2	8	6	16	-1	-3

Zwróć uwagę na poprawność wpisywania wzorów do kodu. Sprawdź kolejność wykonywania działań. Częstym błędem jest wprowadzenie zbyt dużej lub małej ilości nawiasów, co skutkuje wprowadzeniem błędnej formuły i uzyskiwaniem nieprawidłowych wyników.

```
Imports System.Math
Public Class Form1
   Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
   As System.EventArgs) Handles Button1.Click
      Dim a, b, c, delta, x1, x2 As Single
      a = TextBox1.Text
      b = TextBox2.Text
      c = TextBox3.Text
      delta = b ^ 2 - 4 * a * c
      TextBox4.Text = delta
      x1 = (-b + delta ^ 0.5) / (2 * a)
      TextBox5.Text = x1
      x2 = (-b - delta ^ 0.5) / (2 * a)
      TextBox6.Text = x2
 End SubEnd Class
```

20. Funkcje matematyczne w Visual Studio

Nazwa funkcji matematycznej pierwiastka kwadratowego w VB to *Sqrt* (ang. *square root*). W gotowym programie zamień tylko w linii obliczającej :

```
delta ^ 0.5
```

na

```
Sqrt(delta)
```

Zwróć uwagę, że tak wprowadzona funkcja wygeneruje błąd składni kodu. VB nie rozumie, że wyrażenie *Sqrt* ma oznaczać nazwę funkcji matematycznej, choć tak ma być. Dopisz przed nazwą funkcji *Sqrt* następujący fragment kodu:

```
System.Math.
```

Po wstawieniu drugiej kropki, przejrzyj dokładnie zawartość przewijanej listy. Znajdziesz tam przestrzeń nazw typowych funkcji matematycznych, między innymi: wartość bezwzględna, funkcje trygonometryczne, logarytmy, liczby *e* i *pi*, minimum, maksimum, zaokrąglanie, itd. Wskazanie *System.Math.Sqrt()* pozwoli więc VB zrozumieć, że chodzi o funkcję pierwiastka kwadratowego. W sytuacji gdy nasze wzory w kodzie będą zawierały wiele nazw funkcji matematycznych, możemy zmusić VB do wczytania i rozumienia całej tej przestrzeni nazw już na początku programu. Służy do tego polecenie, które należy umieścić **na początku** kodu (powyżej *Public Class Form*):

```
Imports System.Math
```

Od tej pory, już od startu programu, VB będzie prawidłowo interpretował wszystkie funkcje z tej grupy i nie będzie potrzeby pisać więcej w kodzie odniesienia do *System.Math*.

21. Zaokrąglanie – *Round*

Do zaokrąglenia wartości wykorzystujemy funkcję *Round*. Składnia funkcji jest następująca:

```
Round(co?, o_ile?)
```

W nawiasie podajemy, rozdzielane przecinkami: nazwę zmiennej, która będzie zaokrąglana oraz ilość miejsc po przecinku, do której chcemy wybrany element zaokrąglić. Należy się również zastanowić, w którym miejscu kodu zastosujemy zaokrąglanie. Teoretycznie do wyboru mamy dwa sposoby zaokrąglania:

- na etapie liczenia:

```
wynik = Round(wynik, 2)
```

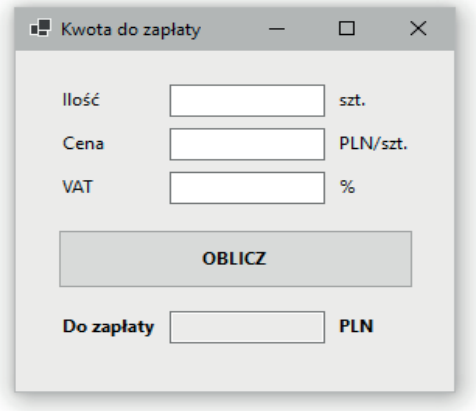
- na etapie wyświetlania wyników:

```
TextBox.Text = Round(wynik,2)
```

Lepszym rozwiązaniem będzie oczywiście to drugie, tj. najpierw należy obliczyć wynik, a dopiero, wyświetlając go, dokonać zaokrąglenia tego, co się wyświetli. Dlaczego? Dlatego, że wówczas w zmiennej: *wynik* cały czas będzie przechowywany dokładny wynik obliczeń, a tylko w polu tekstowym pojawi się zaokrąglony, więc nie stracimy tej zaokrąglonej końcówki. W przypadku wykorzystania otrzymanego wyniku w dalszych obliczeniach, mogłoby mieć to wpływ na rezultaty kolejnych wyliczeń.

Ćwiczenie 17. Obliczanie kwoty do zapłaty

Należy wykonać aplikację (rys. 39), która będzie obliczała końcową kwotę do zapłaty, przy następujących danych wejściowych: cena towaru netto w PLN, liczba sztuk towaru, kwota VAT (wpisana jako liczba, ale traktowana później w kodzie jako wartość procentowa).



Rysunek 39. Wygląd aplikacji do obliczania kwoty do zapłaty

Podczas sprawdzania gotowego programu może okazać się, że format wyświetlanego wyniku obliczeń nie będzie nas satysfakcjonował, gdyż obliczona wartość może mieć dużo miejsc po przecinku. W przypadku klienta, kupującego coś w sklepie, nie ma to sensu, gdyż zwykle podawana jest nam cena zaokrąglona do pełnych groszy. Zastosuj więc zaokrąglanie uzyskanego wyniku końcowego.

Kolejną dobrą praktyką przy budowie programów obliczeniowych jest blokowanie pól tekstowych wyświetlających wyniki obliczeń tak, aby użytkownik nie mógł ręcznie zmienić uzyskanego wyniku. Wykorzystać do tego należy właściwość *ReadOnly* (tylko do odczytu) pola tekstowego.

```
do_zaplaty = cena * ilosc * VAT/100 + cena * ilosc
     do_zaplaty = cena * ilosc (1 + VAT/100)
```

22. Instrukcje warunkowe: *If ... Then*

Programy, które budowaliśmy do tej pory były niezwykle proste. Programy, które będziemy budować w dalszych ćwiczeniach często będą działały różnie w zależności od jakiegoś warunku, np. gdy wyliczony podatek wyjdzie ujemny, kwota wynikowa będzie „do zwrotu" podatnikowi, gdy wpisane hasło będzie nieprawidłowe, program nie pozwoli przejść dalej. Służyć do tego będą tzw. struktury decyzyjne, tj. zestawy instrukcji rozdzielających przebieg kodu w taki sposób, że, w zależności od spełnienia postawionego warunku, część kodu będzie wykonywana lub nie.

Podstawową instrukcją pozwalającą w taki sposób sterować wykonywaniem kodu jest klasyczna instrukcja *If ... Then* (jeżeli ... to). Najprostszy przypadek jej składni przedstawiono poniżej:

```
If warunek Then linijka_kodu
```

np.

```
If godzina > 24 Then TextBox1.Text="Doba ma tylko 24h."
```

W przypadku, gdy kod wykonywany po spełnieniu warunku jest dłuższy, składnia nieco zmienia swój kształt, kod staje się wieloliniowy, dochodzi linia zakończenia instrukcji *End If*:

```
If warunek Then
   linijka_kodu1
   linijka_kodu2
End If
```

Instrukcję *If* można również poszerzyć o fragment kodu, który będzie wykonany, gdy warunek nie będzie spełniony *Else*:

```
If warunek Then
   linijka_kodu1
   linijka_kodu2
Else
   linijka_kodu3
   linijka_kodu4
End If
```

np.

```
If TextBox1.Text = "Kowalski" Then
   TextBox1.Text = "Witaj Jasiu!"
   Label1.Text = "szef dzwonił 5 razy"
Else
   TextBox1.Text = "Ktoś ty?"
End If
```

Warunki można ze sobą łączyć za pomocą operatorów logicznych. Mamy do wyboru kilka operatorów, m.in.: *and* (i), *or* (lub), *not* (nie), *xor* (ang. *exclusive or* – szczególne lub). Poniższy przykład przedstawia warunek sprawdzający nazwiska i dwa kody *pin* użytkowników:

```
If Not nazwisko = “” And PIN = 12345 Or PIN = 67890 Then
        TextBox1.Text = “Witaj drogi kliencie!”
```

Kod wyświetli w polu tekstowym odpowiedź, tylko gdy zmienna *nazwisko* nie będzie pusta i zmienna *pin* przyjmie jedną z dwóch wartości.

Pierwsze trzy warunki powinny być zrozumiałe. Problem bywa ze zrozumieniem ostatniego. Operator *xor* działa podobnie jak *or,* z tą jedną różnicą, że gdyby nawet obydwa, łączone przy jego pomocy, warunki zostały spełnione, łączny warunek będzie niespełniony. Lub, patrząc odwrotnie, warunki połączone operatorem *xor* będą spełnione tylko wtedy, gdy jeden (i tylko jeden) z łączonych warunków będzie spełniony.

Przykład fizyczny wyjaśniający *xor.* Mamy dwa włączniki światła w pomieszczeniu. Na początku obydwa są wyłączone (*False*). Wchodzimy do pomieszczenia i naciskamy pierwszy przycisk (*True*) – światło się zapala. Teraz, aby światło zgasło, mamy dwie opcje. Wyłączyć pierwszy przycisk (obydwa wyłączone) lub włączyć drugi (dwa włączone też dają finalnie wyłączone światło). I znowu odwrotnie – światło będzie się świecić tylko wtedy, gdy wyłącznie jeden przycisk (dowolny) będzie włączony.

Ćwiczenie 18. Opis wieku

Wykorzystując instrukcję warunkową *If… Then,* zbuduj program, który, w zależności od wieku podanego przez użytkownika w polu tekstowym, po przyciśnięciu przycisku wyświetli w etykiecie informację opisującą ten wiek. Przedziały wiekowe należy dobrać zgodnie z poniższym schematem (rys. 40):

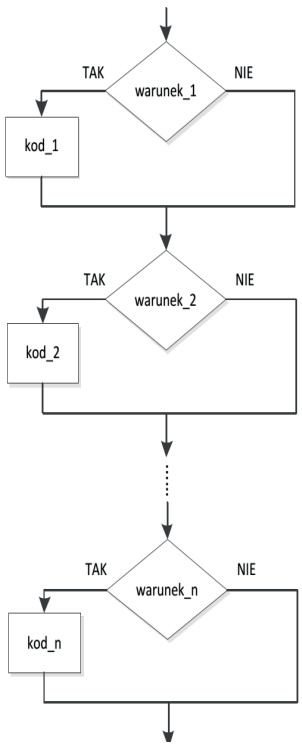
Rysunek 40. Schemat przedziałów wiekowych

Zbuduj program, sprawdź, czy działa poprawnie, jeśli nie zrobiłeś tego wcześniej, spróbuj zmodyfikować kod tak, aby dodatkowo wyświetlał się dowolny opis dla wszystkich lat nieuwzględnionych na powyższym rysunku.

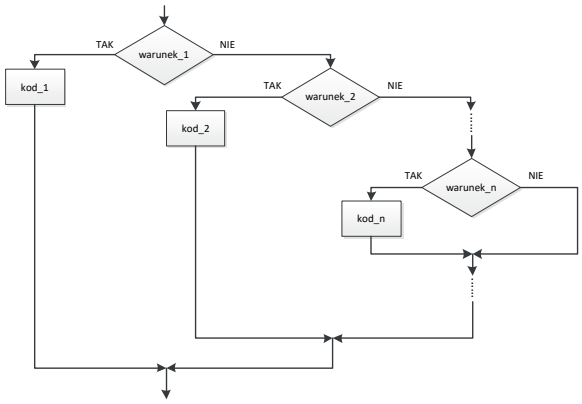
Poniższe wyjaśnienie przeczytaj dopiero po skończeniu zadania!

Od strony kodu program można zbudować na dwa, nieco odmienne, choć finalnie działające, sposoby. Pierwszym jest klasyczne umieszczanie niezależnych od siebie instrukcji warunkowych *If ... Then*, jedna pod drugą. Będą one sprawdzały kolejne warunki i, w przypadku ich spełnienia, wyświetlały zaprogramowany tekst. Takie szeregowe podejście można przedstawić za pomocą poniższego schematu (rys. 41).

Najczęściej jednak 2–3 osoby w grupie budują kod do tego zadania według poniższego, zagnieżdżonego schematu (rys. 42):



Rysunek 41. Schemat kodu ćwiczenia: wariant 1 – szeregowy



Rysunek 42. Schemat kodu ćwiczenia: wariant 2 – zagnieżdżony

Działanie takiego programu będzie bardziej efektywne, a kod sprawniejszy. Kolejność wykonywania kodu będzie następująca: sprawdzony zostaje pierwszy *warunek_1*. Jeśli zwróci on prawdę (warunek będzie spełniony), zostanie wykonany *kod_1*, następnie cała wielka instrukcja zostanie zakończona. Nie będą wykonywane dalsze sprawdzania warunków. Jeśliby jednak *warunek_1* nie był spełniony, w sekcji *Else* zostanie umieszczona (zagnieżdżona) kolejna struktura *If ... Then.* Różnica w stosunku do kodu „szeregowego" polega na tym, że tam, niezależnie od wyniku sprawdzenia

warunku_1, wykonywane były kolejne instrukcje *If*. Tu są one sprawdzane tylko, gdy *warunek*_1 nie zostanie spełniony.

Takie podejście jest bardziej efektywne od pierwszego, choć wymaga od programisty dyscypliny przy otwieraniu i zamykaniu kolejnych instrukcji *If* (każdy *If* musi kończyć się *End If*). Bardzo pomocna w tym wypadku okazuje się funkcja autorozmieszczania kodu w Visual Studio. Kolejne instrukcje warunkowe będą się znajdowały w pionowych kolumnach, każda nieco dalej na prawo od poprzedniej.

Przyjrzyj się poniższemu przykładowi i spróbuj przeanalizować jego działanie.

```
If TextBox1.Text <= 0 Then
    Label1.Text = "jest ciałem stałym"
Else
    If TextBox1.Text <= 100 Then
        Label1.Text = "jest cieczą"
    Else
        If TextBox1.Text < 1000 Then
            Label1.Text = "jest gazem"
        Else
            Label1.Text = "jest strasznie gorącym gazem"
        End If
    End If
End If
```

Zauważ, że w powyższym przykładzie, w jego ostatnim segmencie (ostatnia sekcja *Else*), nie ma już potrzeby sprawdzania, czy temperatura jest większa od 1000. Jeśli kod dotrze do tego momentu to znaczy, że żadna z dotychczasowych instrukcji *If* nie zakończyła całej struktury instrukcji warunkowej. To znaczy też, że podana temperatura ani nie jest mniejsza od zera, ani z zakresu 0–100, ani z zakresu 100–1000, więc musi być większa od 1000! Popraw swój program tak, aby działał według zasady zagnieżdżania instrukcji *If ... Then*.

23. Instrukcje warunkowe: *Select Case*

Inną strukturą decyzyjną jest instrukcja: *Select Case* (wybierz przypadek). Jej działanie jest właściwie identyczne z *If ... Then,* natomiast składnia będzie zdecydowanie czytelniejsza. W niektórych przypadkach, zwłaszcza tych bardziej skomplikowanych, wygodniej jest wykorzystać do budowy kodu właśnie tę instrukcję. Składnia polecenia jest następująca:

```
Select Case
   Case
   Case
   Case
   Case
   Case Else
End Select
```

Fragment kodu wyświetlającego opis miesiąca/miesięcy, prezentujący przykładową składnię polecenia, przedstawiono poniżej. Zwróć uwagę, w jaki sposób określane są konkretne wartości, w jaki przedział otwarty, a w jaki zamknięty. Podobnie jak w instrukcji *If ... Then*, tu też mamy sekcję *Else (Case Else)*.

```
Select Case miesiac
   Case 2, 6
      Label1.Text = "sesja"
   Case Is > 12
      Label1.Text = "błąd – rok ma 12 miesięcy!"
   Case 7 To 9
      Label1.Text = "wakacje"
   Case Else
      Label1.Text = „jakiś inny, nieistotny miesiąc"
End Select
```

Instrukcja *Select Case* podobnie jak *If ... Then* musi się kończyć linią *end,* w tym przypadku będzie to *End Select*. Różnicą w stosunku do poprzedniej struktury jest to, że na samym początku instrukcji nie definiujemy warunku, tylko nazwę zmiennej (lub właściwości kontrolki i innych elementów). Wewnątrz kodu, w sekcjach, określamy kolejne przypadki wybranej na początku zmiennej oraz działanie kodu w przypadku osiągnięcia danego przypadku. Zastanów się, jak można byłoby napisać powyższy kod, wykorzystując instrukcję *If ... Then*.

Ćwiczenie 19. Rozwiązywanie równania kwadratowego – wersja 2

W tym ćwiczeniu będziemy budować program, który wykonywaliśmy już wcześniej. Będzie on wyznaczać rozwiązania równania kwadratowego. Tym razem jednak będziemy uwzględniać założenia matematyczne równania kwadratowego oraz przy-

padki znaku delty, od której będzie zależeć liczba rozwiązań i sposób ich wyznaczania. Wykorzystaj poniższe zależności (tab. 4).

Tabela 4. Zależności matematyczne do rozwiązania równania kwadratowego

$\Delta = b^2 - 4ac$		
Znak delty	**Liczba rozwiązań**	**Wzory**
$\Delta > 0$	dwa rozwiązania	$x_1 = \frac{-b + \sqrt{\Delta}}{2a}$ $x_2 = \frac{-b - \sqrt{\Delta}}{2a}$
$\Delta = 0$	jedno rozwiązanie	$x = \frac{-b}{2a}$
$\Delta < 0$	brak rozwiązań	–

Program powinien wyglądać funkcjonalnie, część do wprowadzania danych wejściowych powinna być oddzielona od części wyświetlającej wyniki. W zależności od tego, ile rozwiązań będzie miało równanie w danym przypadku, tyle powinno być widocznych pól tekstowych. Wyświetlane wyniki to: obliczone wartości, tekst: „delta jest (większa/równa/mniejsza) niż zero, więc równanie nie ma/ma 1/2 rozwiązania". Kod wyzwalamy kliknięciem w przycisk: *„Oblicz"*. Dodatkowo należy umieścić w formularzu przyciski *„Reset"* i *„Zamknij"*.

Tabela 5. Warianty danych wejściowych i rozwiązania równania kwadratowego

a	***b***	***c***	***Δ***	x_1	x_2
2	8	6	16	-1	-3
1	2	1	0	-1	–
2	1	2	-15	–	–

Ćwiczenie 20. Obliczenia podatku

Wybierając z paska menu polecenie: *„Save All"*, zapisujemy projekt w *Moich Dokumentach* pod nazwą: „Imię i Nazwisko studenta – data". Celem ćwiczenia jest zbudowanie programu, który po wprowadzeniu, jako danych wejściowych, **miesięcznego dochodu brutto,** będzie wyznaczał i wyświetlał dochód roczny oraz, według progów podanych w poniższej tabeli: numer grupy podatkowej i kwotę podatku w skali rocznej**.** Interfejs powinien być oparty o pola grupy, rozdzielające dane wejściowe od wyświetlanych wyników, oraz przyciski: *Oblicz*, *Resetuj* i *Koniec* (tab. 6). Wyniki wyświetlane są w nieedytowalnych polach tekstowych. Wszystkie pola opisane zostały etykietami. Ćwiczenie należy wykonać dwukrotnie – oddzielnie z wykorzystaniem struktury *If… Then* oraz osobno *Select Case*.

Tabela 6. Progi podatkowe

Roczny dochód	Grupa podatkowa	Sposób obliczenia podatku
do 8000 zł	0	kwota wolna od podatku
8 000 – 120 000 zł	1	17% dochodu – 5 100 zł
od 120 000 zł	2	32% od nadwyżki ponad 120 000 + 15 300 zł

Ćwiczenie 21. Obliczenia podatku – wersja 2

Celem ćwiczenia jest zbudowanie podobnego programu jak w poprzednim ćwiczeniu, z tą różnicą, że na początku użytkownik będzie miał do wyboru rok podatkowy. Jako że w różnych latach obowiązywały różne zasady rozliczania, wzory na poszczególne grupy podatkowe będą różne. Ponadto, w różnych latach niejednakowe były wysokości kwoty wolnej od podatku (tab. 7). Zwróć uwagę na ostatnią tabelę z kwotami wolnymi od podatku. W programie musi się dodatkowo znaleźć obrazek banknotów polskich złotych.

Tabela 7. Progi podatkowe w różnych latach

Rok	Roczny dochód	Grupa podatkowa	Sposób obliczenia podatku
2009, 2010	do 85528 zł	1	18% dochodu ponad kwotę wolną od podatku
	od 85528 zł	2	14839,02 zł + 32% od dochodu ponad 85528 zł
2008	do 44490 zł	1	19% dochodu ponad kwotę wolną od podatku
	44490–85528 zł	2	7866,25 zł + 30% od dochodu ponad 44490 zł
	od 85528 zł	3	20177,65 zł + 40% od dochodu ponad 85528 zł
2007	do 43405 zł	1	19% dochodu ponad kwotę wolną od podatku
	43405–85528 zł	2	7674,41 zł + 30% od dochodu ponad 43405 zł
	od 85528 zł	3	20311,31 zł + 40% od dochodu ponad 85528 zł
2003–2006	do 37024 zł	1	19% dochodu ponad kwotę wolną od podatku
	37024–74048 zł	2	6504,48 zł + 30% od dochodu ponad 37024 zł
	od 74048 zł	3	17611,68 zł + 40% od dochodu ponad 74048 zł

Rok	Kwoty dochodu wolnego od podatku
2009–2010	3091,00 zł
2008	3089,00 zł
2007	3013,37 zł
2003–2006	2789,89 zł

Ćwiczenie 22. Paski narzędzi oraz kontrolki grupy

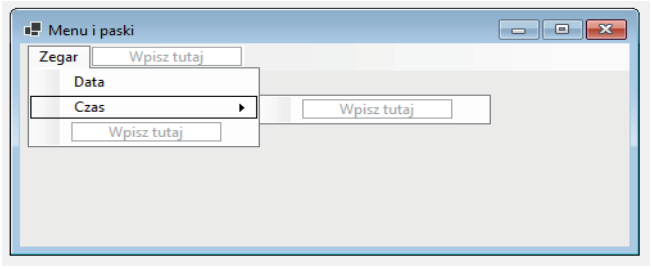
Menu

Celem zadania jest poznanie i wykorzystanie szeregu kontrolek, m.in. z grupy: *Menu i Paski narzędzi* (*Menus & Toolbars)*.

Zadanie 1. MenuStrip

Celem ćwiczenia jest zapoznanie z kontrolką: *MenuStrip*. Dodaj ją do formularza nowego programu. Powinieneś zauważyć, że kontrolka ta zostanie umieszczona na poziomym panelu poniżej formularza, na tzw. *tacce komponentów*. Tu będą pojawiać się kontrolki, które same w sobie nie mają wizualnej reprezentacji (nie są widoczne w formularzu). Kontrolka, którą się zajmujemy, stanowi szablon klasycznego paska *Menu*, znanego z aplikacji systemu Windows. Będziemy go rozbudowywać, dodając do niego kolejne pozycje. Po zaznaczeniu kontrolki lewym klawiszem myszy, w formularzu zostanie wyświetlony zarys tworzonego przez nas *Menu*. Kolejne pozycje możemy dodawać w prosty sposób, wpisując ich nazwy w pole z tekstem: Wpisz tutaj *(Type Here)* na pasku *Menu*.

Dla potrzeb tego ćwiczenia dodaj jedną pozycję menu o nazwie *Zegar*, która po rozwinięciu będzie posiadała dwie opcje: *Data* i *Czas* (rys. 43). Zauważ, że po dodaniu kolejnych pozycji możesz definiować podmenu lub dodawać nowe pozycje w menu głównym. Każda pozycja w podmenu może również posiadać kolejne poziomy.Ważne, by zauważyć, będąc w panelu *Właściwości*, że zarówno kontrolka: *MenuStrip*, jak i nowo dodane pozycje menu: *Zegar, Data, Czas* – są odrębnymi, niezależnymi kontrolkami. Można im oddzielnie zmieniać właściwości.



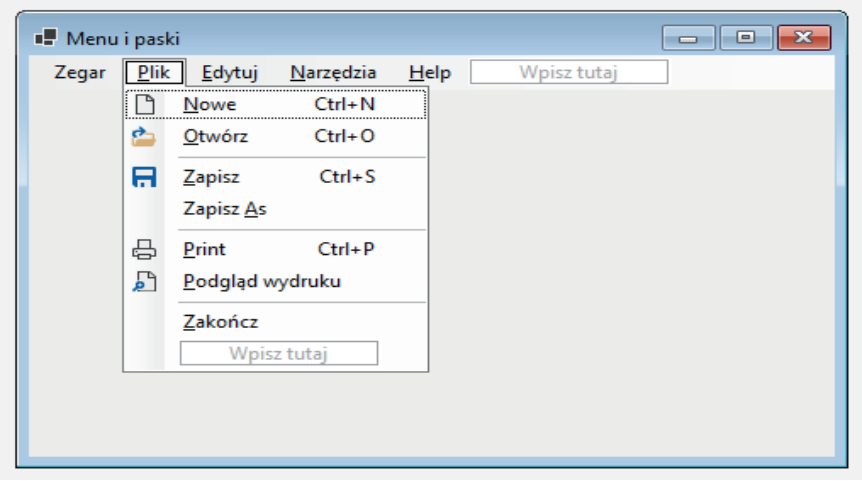
Rysunek 43. Budowa *MenuStrip*

W ćwiczeniu będzie nam jeszcze potrzebna etykieta (*Label)*. Umieść ją na środku formularza, zwiększ rodzaj i rozmiar czcionki. Aby zdefiniować kod wykonywany po wybraniu określonej pozycji menu, należy, podobnie jak w przypadku kontrolki: *Przycisk*, kliknąć dwukrotnie wybraną pozycję menu w formularzu. W kodzie zostanie stworzona procedura, wywołująca ten kod po zdarzeniu wybrania danej pozycji menu. Wpisz tam poniższy kod:

```
Label1.Text = DateString
```

Uruchom program i sprawdź działanie dodanego kodu. Program powinien wstawić do etykiety datę, ustawioną w chwili bieżącej w komputerze. Zbuduj w podobny sposób funkcję wyświetlania godziny po wybraniu z *Menu* pozycji: *Czas*. Łańcuchem przechowującym bieżącą godzinę jest *TimeString*.

Przydatną opcją przy budowie paska Menu jest możliwość automatycznego wstawienia najbardziej typowych pozycji. Aby to zrobić, kliknij prawym klawiszem myszy w puste pole na pasku Menu i z menu kontekstowego wybierz: *Wstaw elementy standardowe* (*Insert Standard Items*). Do menu powinny zostać dodne pozycje: *Plik, Edytuj, Narzędzia, Pomoc* (*File, Edit, Tools, Help*) wraz z rozszerzającymi je po rozwinięciu podpozycjami (rys. 44). Tak dodane menu nie posiada żadnego kodu, jest tylko pustą strukturą. Warto jednak na tym przykładzie prześledzić, jak może wyglądać pasek Menu.



Rysunek 44. Typowe elementy kontrolki *MenuStrip*

Pierwszą, rzucającą się w oczy, właściwością tak dodanego Menu są ikony przy niektórych pozycjach. Definiuje je właściwość: *Image*.

Zwróć uwagę, że część pozycji posiada skrót klawiszowy do szybkiego uruchamiania, np. typowo *ctrl+p* to drukowanie, *ctrl+s* – zapisywanie, itd. Skróty opisuje właściwość: *ShortcutKeys*. Właściwość *ShortcutKeyDisplayString* pozwala na zmia-

nę domyślnego wyświetlania klawisza skrótu. Możemy samodzielnie zdefiniować, jaki tekst będzie wyświetlany na prawo od wybranej pozycji menu. Dodatkowo, przy pomocy właściwości: *ShowShortcutKeys* można całkowicie ukryć wyświetlanie opisu skrótu w menu (skrót jednak cały czas jest aktywny).

W nazwach wszystkich pozycji menu można dostrzec, że jedna z liter (często pierwsza) jest podkreślona. Jest to litera szybkiego wybierania danej pozycji. W uruchomionej aplikacji Windows (jak np. w programie Visual Basic) naciśnięcie lewego klawisza *Alt* zaznacza pierwszą pozycję menu (najczęściej Plik – *File*). Za pomocą klawisza kursora w dół lub *ENTER* można rozwinąć obecnie zaznaczone menu i, wybierając z klawiatury klawisze skrótu szybkiego wybierania, od razu przejść do określonej pozycji.

Zadanie 2. ToolStrip

Celem ćwiczenia jest zapoznanie z kontrolką: *Pasek Narzędziowy z ikonami*, często spotykaną cechą aplikacji systemu Windows.

Dodaj kontrolkę *Toolstrip* do formularza. Podobnie jak poprzednio, zostanie ona umieszczona na tacce komponentów poniżej formularza. Sam pasek zaś będzie znajdować się w górnej części.

Po kliknięciu w pasek na górze lub jego kontrolkę, w tacce pojawi się rozwijalny przycisk wyboru kontrolki do umieszczenia na pasku. Do wyboru będą: przycisk (*Button*), etykieta (*Label*), przycisk podziału (*SplitButton*), rozwijalny przycisk (*DropDownButton*), separator, pole combo (*ComboBox*), pole tekstowe (*TextBox*) i pasek postępu (*ProgressBar*). Każda kontrolka dodana do paska staje się oddzielną kontrolką, której właściwości możemy modyfikować.

Podobnie jak w przypadku wyżej omawianej kontrolki *MenuStrip*, przydatną opcją jest możliwość automatycznego wstawienia typowych pozycji (prawy klawisz myszy: Wstaw elementy standardowe (*Insert Standard Items*). Do menu powinny zostać dodane typowe przyciski: *Nowy, Otwórz, Zapisz, Drukuj, Wytnij, Kopiuj, Wklej, Pomoc.*

Dodaj funkcjonalność do przycisku: N*ożyczki,* ich kliknięcie spowoduje wyczyszczenie etykiety znajdującej się na środku formularza.

Zadanie 3. StatusStrip

Celem ćwiczenia jest zapoznanie z kolejną, często wykorzystywaną w typowych okienkowych programach kontrolką – *Paskiem Statusu* (*StatusStrip*). Dodaj go do formularza. Znów, jak poprzednio, znajdzie się on na tacce komponentów, natomiast sam pasek zostanie umieszczony w dolnej części formularza. Po wybraniu paska pojawi się na nim rozwijalny przycisk wyboru kontrolki do umieszczenia. Do wyboru będą: etykieta statusu (*StatusLabel*), pasek postępu (*ProgressBar*), przycisk podziału (*SplitButton*) i rozwijalny przycisk (*DropDownButton*). Podobnie jak wcześniej, każdy dodany element staje się oddzielną kontrolką, której właściwości możemy indywidualnie modyfikować.

Do przycisku: *Nożyczki* dodaj funkcjonalność, która spowoduje wyczyszczenie etykiety znajdującej się na środku formularza. Dodaj etykietę statusu: *Pasek Statusu* i dołącz kod do kliknięcia w ikonę nożyczek (z poprzedniego zadania), któr czyści etykietę na środku formularza. Kod powinien wstawić tekst: „wyczyszczono etykietę" w nowo dodaną etykietę statusu .

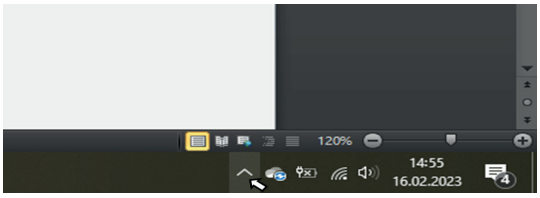
Zadanie 4. NotifyIcon, ContextMenuStrip

Celem ćwiczenia jest zapoznanie z kontrolkami ikony powiadomień i menu kontekstowym.

Dodaj do tacki komponentów kolejną kontrolkę – ikonę powiadomienia (*NotifyIcon*). Jest do element, który będzie wyświetlał się w tzw. *Pasku Zasobnika Systemu Windows* – polu znajdującym się w prawym dolnym rogu ekranu, obok zegara systemowego. Dopóki jednak nie zdefiniujemy, jaką ikonę będzie wyświetlać ta kontrolka, po uruchomieniu programu nic nie zobaczymy.

We właściwości: *Icon* ustaw dowolną ikonę, np. pobrany z sieci plik z rozszerzeniem *.ico, np. *flaticon.com* lub *icon-icons.com*. W Windowsie można szukać ikon m.in.: w folderach *C:/program files* lub *C:/windows,* np.: *c:/Program Files (x86)/ Microsoft Office/Office##.*

Sprawdź, czy teraz, po uruchomieniu programu, ikona pojawia się obok zegara systemowego. Upewnij się, że nie jest ukryta – przycisk: *Pokaż ukryte ikony* (rys. 45).



Rysunek 45. Pokaż ukryte ikony (*NotifyIcon*)

Tak zdefiniowana ikona nie ma jednak w tej chwili żadnej funkcjonalności. Nic się nie stanie, kiedy ją klikniemy. Dlatego, wykorzystując kontrolkę menu kontekstowego (*ContextMenuStrip*), stworzymy menu, które następnie podepniemy pod ikonę.

Dodaj dwie kontrolki menu kontekstowego. Jak większość kontrolek w tym ćwiczeniu, te też umieszczą się w tacy komponentów poniżej formularza. Kiedy żaden *ContextMenuStrip* nie będzie zaznaczony na tacce, nie zobaczymy naszego menu nigdzie w formularzu.

Wybierz pierwsze menu (*ContextMenuStrip1)*. Na górze formularza pojawi się szkielet menu kontekstowego. Uzupełnij go trzema pozycjami: A, B i C. Drugie menu, dla odmiany, wypełnij pozycjami: 1, 2, 3. Zauważ, że nasze menu kontekstowe, podobnie jak pasek Menu na górze formularza, możemy rozbudowywać w dół, ale również w prawo, dodając kolejne pozycje podmenu. Do pozycji A dodamy kod, który za-

mknie program (*End*). W tym celu zaznaczamy pierwsze menu kontekstowe i klikamy dwukrotnie w pozycję *A*.

Menu kontekstowe, jak sama nazwa wskazuje, nie będzie wyświetlane cały czas, będzie pojawiać się tylko w jakimś kontekście, w konkretnej sytuacji. Dlatego, jeśli teraz uruchomisz program, nigdzie nie znajdziesz, przed chwilą zbudowanego, menu kontekstowego. Pierwsze menu dodamy do, przed chwilą analizowanej, ikony powiadomienia (*NotifyIcon*). W panelu z właściwościami znajdź dla niej właściwość: *ContextMenuStrip* i ustaw tam pierwsze menu (*ContextMenuStrip1)*. Uruchom program, zlokalizuj swoją ikonę obok zegarka Windows i kliknij w nią prawym klawiszem myszy. Wybierz pozycję *A*, żeby zamknąć program.

W analogiczny sposób spróbuj dołączyć drugie menu kontekstowe (*ContextMenuStrip2)* do formularza (*Form1*). Uruchom program, kliknij prawym klawiszem myszy w puste miejsce w formularzu.

Zadanie 5. ToolTip

Celem ćwiczenia jest zapoznanie z etykietą podpowiedzi, wizualnym elementem w postaci chmurki lub dymka, który będzie wyświetlany po najechaniu kursorem myszy na dany element. Jego zadaniem jest najczęściej wyświetlenie opisu lub wyjaśnienia do danej kontrolki lub funkcjonalności, jaką ona uruchamia.

Dodaj dwie kontrolki *ToolTip*. Znów zostaną one umieszczone w tacce komponentów, ale nie pojawią się nigdzie w formularzu. Wynika to z tego, że kontrolki same w sobie są tylko szablonami tych etykiet, natomiast treść, jaką będą wyświetlać, będziemy definiować indywidualnie dla odpowiednich kontrolek. Pierwszej kontrolce nie zmieniaj żadnych właściwości, natomiast w *ToolTip2* zmodyfikuj cechy, takie jak: *AutoPopDelay* – czas widoczności, *BackColor*, *ForeColor*, *InitialDelay* – opóźnienie do pierwszego wyświetlenia, *lsBalloon* – kształt dymka, *ReshowDelay* – czas do kolejnego wyświetlenia, *ShowAlways* – wyświetlanie, gdy okno nie jest główne, *ToolTiplcon* – ikona, *ToolTipTitle* – tytuł, *UseAnimation* – animacje, *UseFading* – efekt zanikania.

Do właściwości etykiety stojącej na środku formularza wstaw: *ToolTip on ToolTip1* – swoje imię, a do *ToolTip on ToolTip2* – nazwisko. Sprawdź działanie programu.

24. Pętle

Pętle to instrukcje o charakterze cyklicznym, powtarzane dopóki nie zostanie spełniony zadany warunek logiczny. Pętla powinna być skończona, tzn. warunek logiczny, stojący u jej podstawy, musi być spełniony tak, żeby pętla mogła się zakończyć. Instrukcje tego typu stosujemy wszędzie tam, gdzie potrzebne jest wielokrotne wykonanie tej samej instrukcji lub instrukcji zmienionej w każdej iteracji/cyklu, np. obliczenia kolejnych wartości funkcji w celu przygotowania danych do rysowania wykresów.

Struktura *For ... Next ... Step*

Najbardziej podstawową pętlą, istniejącą w niemal identycznej formie w różnych językach programowania, jest pętla: *For ... Next*. Jej ogólna składnia w Visual Basic jest następująca:

```
Dim i as Integer
For i = x To y
   ,polecenie 1
   ,polecenie 2
   ,polecenie 3 z odniesieniem do bieżącej wartości i
   ,itd. …
 Next i
```

Tu*x*" i *y* to wartości początku i końca pętli.

Działanie takiej instrukcji będzie następujące: po zadeklarowaniu zmiennej „i", w pierwszym cyklu zostanie jej przyporządkowana wartość *x*, następnie zostanie wykonany do końca szereg instrukcji wewnątrz pętli (polecenia 1, 2, 3, itd.). Poleceniem *Next i* w ostatniej linii kodu, działanie programu zostanie cofnięte do polecenia *For*. Zmiennej „i" zostanie przyporządkowana kolejna liczba naturalna o 1 większa od bieżącej. Pętla będzie zataczać kolejne i kolejne koło, dopóki wartość *i* nie osiągnie założonego *y*. Wtedy pętla zostanie wykonana po raz ostatni, zakończy się, a dalej będzie wykonywany kod znajdujący się po niej.

Przyrost wartości *i* w pętli nie musi koniecznie mieć wartość 1. Dodając do polecenia *For* frazę *Step z*, można dowolnie zdefiniować wartość przyrostu, np. 2 lub 3, itd. W przypadku, gdybyśmy chcieli, by przyrost wynosił np. 0.5, to deklarując rodzaj zmiennej *i*, musimy pamiętać o odpowiednim typie danych, obsługującym części ułamkowe liczb, np. single. I tak polecenie:

```
For i = 5 To 9 Step 2
```

wygeneruje pętlę o wartościach *i*: 5, 7, 9.

Struktura *Do … Loop*

Kolejną strukturą pozwalającą tworzyć pętle jest: *Do … Loop*. Najważniejsza różnica w stosunku do poprzednio zaprezentowanej instrukcji: *For … Next* polega na tym, że w pętli *For … Next* precyzyjnie podajemy moment zakończenia pętli. Jest to ostatnia wartość w linii *For*.

W *Do … Loop* nie musimy dokładnie wskazywać, kiedy pętla będzie zakończona, ale stanie się to na pewno, kiedy tylko zostanie osiągnięty warunek zakończenia. Składnia tej instrukcji jest następująca:

```
Do (while | until) wyrażenie
   ,polecenie 1
   ,polecenie 2
   ,polecenie 3
Loop
```

I tak, możemy wykonywać naszą pętlę z wyrażeniem: *while* tzn., tak długo, jak długo wyrażenie będzie prawdziwe. A kiedy w kolejnej pętli, na skutek poleceń wewnątrz niej, wyrażenie przestanie być prawdziwe, pętla zakończy się.

Przy użyciu wyrażenia *until* będzie odwrotnie. Pętla będzie wykonywana tak długo, aż niespełnione wyrażenie warunkowe zostanie wreszcie spełnione.

Oto przykładowa instrukcja, która w polu tekstowym będzie wyświetlać kolejne wielokrotności liczby 7, aż osiągnie 77:

```
Public Class Form1
   Private Sub Button1_Click(sender As Object, e As EventArgs)
   Handles Button1.Click
      Dim zmienna As Integer
         zmienna = 0
      Do While zmienna <= 77
         TextBox1.Text &= zmienna & „, „
         zmienna = zmienna + 7
      Loop
   End Sub
End Class
```

Ćwiczenie 23. *For … Next*

Celem ćwiczenia jest budowa programu wykorzystującego pętlę: *For … Next*. Należy stworzyć program, w którym do pola tekstowego zostaną wstawione, zaokrąglone do dwóch miejsc po przecinku i oddzielone między sobą przecinkami, kolejne wielokrotności liczby π, z zakresu od 2π do 12π co 2π, tj.: 2π, 4π, 6π … 12π, tzn.: 6.28, 12.57, …, 37.7

Przykładowe rozwiązanie:

```
Imports System.Math
Public Class Form1
   Private Sub Button1_Click(sender As Object, e As EventArgs)
   Handles Button1.Click
      Dim a As Single
      For a = 2 * PI To 12 * PI Step 2 * PI
         TextBox1.Text &= Round(a, 2) & “, “
      Next a
         TextBox1.Text &= “ - KONIEC”
   End Sub
End Class
```

Ćwiczenie 24. *Do … While/Until*

Celem ćwiczenia jest budowa programu wykorzystującego strukturę: *do while.* Stwórz program wyświetlający w polu tekstowym kolejne numery i odpowiadające im kody ASCII w zakresie od 32 do 126 co 1 (funkcja CHR).

Przykładowe rozwiązanie:

```
Do While kod <= 126
   TextBox1.Text &= kod & “: “ & CHR(kod) & “, “
   kod += 1
Loop
```

Ćwiczenie 25. Pętle – praktyczne zastosowanie

Celem ćwiczenia jest utrwalenie wiadomości i umiejętności wykorzystania pętli.

Z wykorzystaniem dowolnej instrukcji pętli (ćw. 23–24), opracuj program, który będzie wyznaczać wartości dowolnej funkcji kwadratowej, w każdym punkcie dowolnego przedziału jej argumentów. W tym celu można opracować program przedstawiony poniżej (rys. 46).

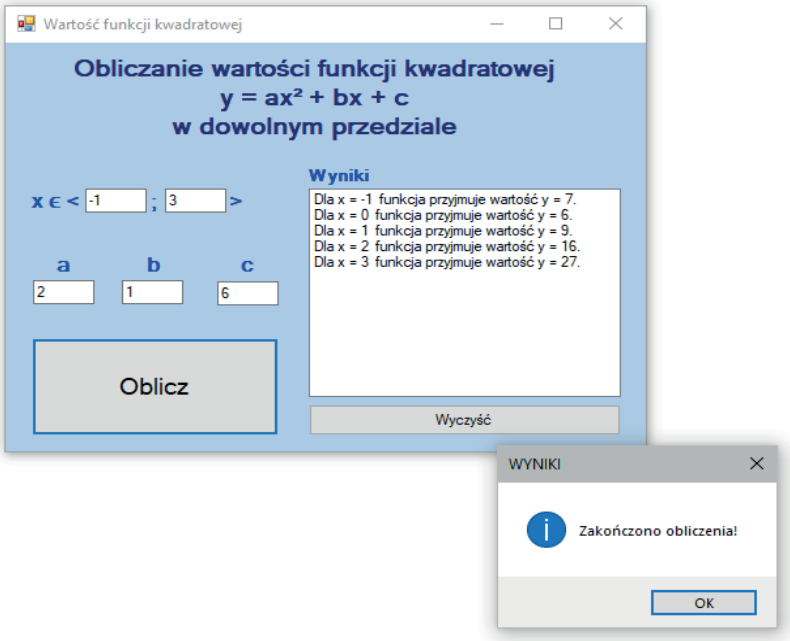
Przykładowe rozwiązanie:

```
Public Class Form1
   Dim a, b, c, y, x1, x2 As Double
   Private Sub btnOblicz_Click(ByVal sender As System.Object,
   ByVal e As System.EventArgs) Handles btnOblicz.Click
      a = CDbl(txtA.Text)
      b = CDbl(txtB.Text)
      c = CDbl(txtC.Text)
      x1 = CDbl(txtX1.Text)
      x2 = CDbl(txtX2.Text)
      For x = x1 To x2
         y = a * x ^ 2 + b * x + c
```

```
            lstWartosci.Items.Add(„Dla x = „ & x & „ funkcja przyjmuje war-
            tość y = „ & y & „.”)
        Next
            MessageBox.Show(„Zakończono        obliczenia!”,        „WYNIKI”,
            MessageBoxButtons.OK, MessageBoxIcon.Information)
    End Sub

    Private Sub btnWyczysc_Click(sender As Object, e As EventArgs) Handles
    btnWyczysc.Click
        lstWartosci.Items.Clear()

    End Sub
End Class
```



Rysunek 46. Przykładowy wygląd aplikacji

Wykorzystując typowe elementy środowiska VS, można wprowadzić parametry o wartościach, zgodnie z którymi będzie realizowana pętla. W przytoczonym przykładzie widać, iż elementami decydującymi o realizacji liczby wykonanych obliczeń są wartości zmiennej „x”, zmieniające się w zadanym zakresie (definiowanym przez użytkownika z wykorzystaniem pól tekstowych).

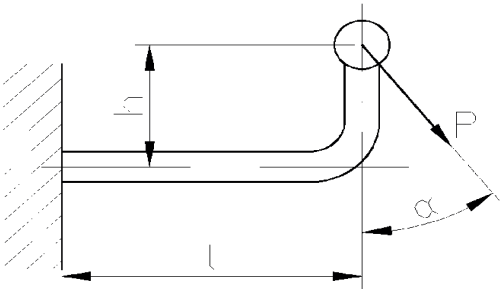
Dodatkowo warto zwrócić uwagę na działanie kontrolki: *ListBox*, która w środowisku VS musi być stosowana wraz z odpowiednią pętlą. Wówczas do kontrolki typu: *ListBox* w sprawny sposób można dodawać kolejne elementy lub je modyfikować.

25. ZADANIA INŻYNIERSKIE DO SAMODZIELNEJ REALIZACJI

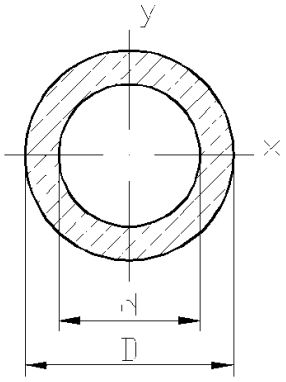
Zagadnienie 1. Obliczanie wytrzymałości wspornika

Jednym z podstawowych zagadnień z zakresu badań konstrukcji są obliczenia wytrzymałościowe. Na podstawie schematu obciążenia elementu oraz jego geometrii wyznacza się występujące w nim naprężenia i porównuje z dopuszczalnymi dla danego materiału. Zagadnienie to, dotyczące różnego rodzaju konstrukcji, zostało szczegółowo opisane w publikacjach [1, 3, 4].

Jednym z przykładów jest obliczanie wytrzymałości wspornika. Wykorzystując odpowiednie wzory oraz zależności matematyczne, należy stworzyć program umożliwiający wykonanie częściowych obliczeń wytrzymałościowych wspornika [1]. Rysunek 47 przedstawia schemat obciążenia wspornika. Rysunek 48 pokazuje przekrój poprzeczny rury, z której został on wykonany. Szczegółowe informacje dotyczące obliczeń znajdują się pod poniższymi rysunkami.



Rysunek 47. Analizowany wspornik (opracowanie własne na podstawie [1])



Rysunek 48. Przekrój poprzeczny wspornika (opracowanie własne na podstawie [1])

Do kluczowych parametrów niezbędnych do przeprowadzenia obliczeń należą: długość l i wysokość h wspornika, siła P oraz jej składowe ($P1$ oraz $P2$), kąt przyłożenia siły α, a także średnica zewnętrzna D oraz średnica wewnętrzna d rury, z jakiej wykonano wspornik.

Poniżej przedstawiono przykładowe obliczenia dla danych: $l = 150$ mm, $h = 100$ mm, $P = 1{,}8$ kN $= 1800$ N, $\alpha = 60^{\circ}$, $D = 30$ mm, $d = 20$ mm.

Na podstawie wymienionych powyżej parametrów można wyznaczyć kluczowe wartości szukane, obliczone poniżej.

Składowe siły P:

$$P_1 = P \cdot sin\alpha = 1800 \cdot sin60° = 1558{,}85 \text{ [N]}$$

$$P_2 = P \cdot cos\alpha = 1800 \cdot cos60° = 900 \text{ [N]}$$

Moment gnący:

$$M_g = (P_1 \cdot h) + (P_2 \cdot l) = (1558{,}85 \cdot 100) + (900 \cdot 150) = 290885 \text{ [Nmm]}$$

Pole przekroju rury:

$$S = \frac{\pi}{4} \cdot (D^2 - d^2) = \frac{\pi}{4} \cdot (30^2 - 20^2) = 392{,}7 \text{ [mm}^2\text{]}$$

Wskaźnik wytrzymałości:

$$W_x = \frac{\pi}{32} \cdot \frac{(D^4 - d^4)}{D} = \frac{\pi}{30} \cdot \frac{(30^4 - 20^4)}{30} = 2127{,}12 \text{ [mm}^3\text{]}$$

Naprężenia rozciągające:

$$\sigma_r = \frac{P_1}{S} = \frac{1558{,}85}{392{,}7} = 3{,}97 \text{ [MPa]}$$

Naprężenia tnące:

$$\tau_t = \frac{P_2}{S} = \frac{900}{392{,}7} = 2{,}29 \text{ [MPa]}$$

Naprężenia gnące:

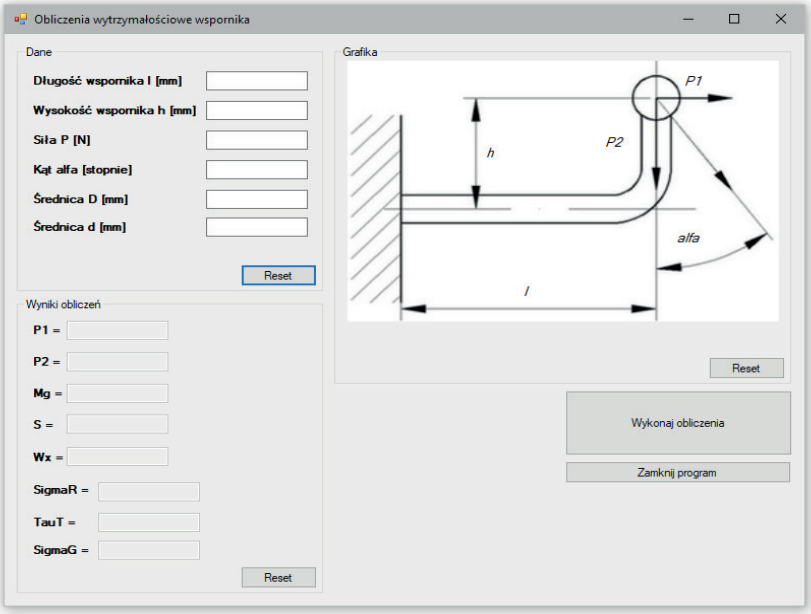
$$\sigma_g = \frac{M_g}{W_x} = \frac{290885}{2127{,}13} = 136{,}75 \text{ [MPa]}$$

Przedstawione obliczenia można zaimplementować w środowisku Visual Studio w celu budowy odpowiedniego programu komputerowego, wspierającego analizę wytrzymałości dla różnych wartości parametrów. Przykład takiego programu został przedstawiony poniżej (rys. 49).

Aplikację umożliwiającą realizację wybranych obliczeń wytrzymałościowych wspornika opracowano, bazując na następujących założeniach:

1. Wszystkie wyniki obliczeń powinny być zaokrąglone do 2 miejsc po przecinku.
2. Pola w grupie: „*Dane*" powinny być zabezpieczone w odpowiedni sposób – jeżeli użytkownik nie wprowadzi danych w którekolwiek pole, zostanie wyświetlony odpowiedni komunikat.
3. Poszczególne kontrolki realizują odpowiednie zdarzenia.
4. Obliczone wartości mają być wyświetlane wraz z jednostkami.
5. Wartości *P*, *P*1, *P*2, *α*, *l*, *h* po wykonaniu obliczeń powinny zostać wyświetlone na rysunku pomocniczym.

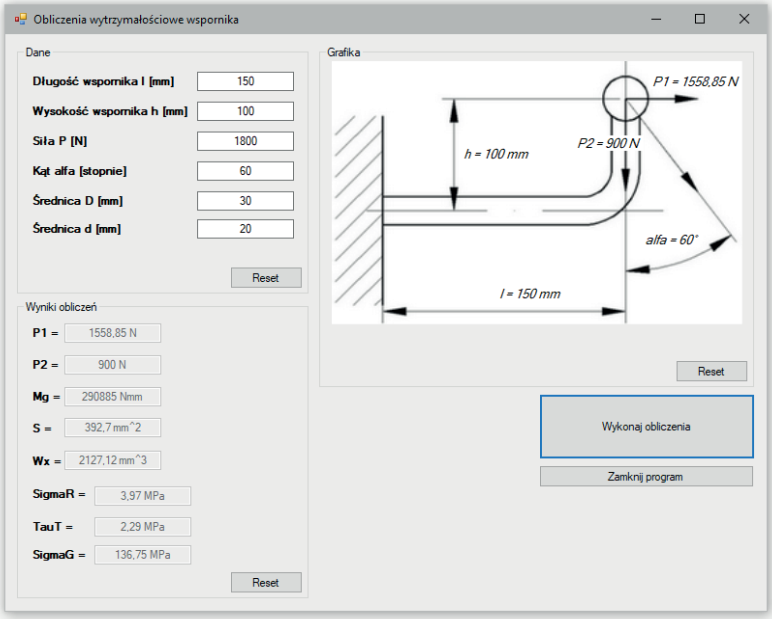
Przykładowe rozwiązanie:



Rysunek 49. Przykładowy wygląd okna opracowanej aplikacji

W opracowanej aplikacji, w celu pogrupowania poszczególnych elementów, wykorzystano kontrolki typu *GroupBox*, zaś w ich wnętrzu umieszczono kolejne kontrolki funkcyjne. Elementy typu *Label* pozwoliły na wyświetlenie informacji o zmiennych wprowadzanych lub wyświetlanych w kontrolkach typu *TextBox*. W przypadku pól tekstowych, w których zostają wyświetlane wyniki, wykorzystano blokowanie pola tak, aby nie była możliwa modyfikacja uzyskanych wyników (właściwość *Enabled* kontrolki *TextBox* ustawiono na wartości *False*). Dodatkowo wykorzystano elementy typu *Button*, które spełniały opisane na nich funkcje (np. służyły do resetowania poszczególnych pól – czyszczenia pól tekstowych lub przywracania etykietom pierwot-

nych wartości). W celu wyświetlenia obrazu pomocniczego zastosowano kontrolkę typu *PictureBox*, zaś na rysunku dodano elementy typu *Label*, które po wprowadzeniu danych zmieniały swoje wartości (rys. 50). Wszystkie kontrolki nazywano zgodnie z założeniami notacji węgierskiej.



Rysunek 50. Wyniki wykonanych obliczeń

Opracowanie warstwy graficznej aplikacji pozwoliło na rozpoczęcie prac nad kodem źródłowym, który został przedstawiony poniżej:

```
Imports System.Math

Public Class Form1
   Dim l, h, P, alfa, Dzew, Dwew, P1, P2, Mg, S, Wx, SigmaR, TauT, SigmaG
   As Double
   Private Sub btnWykonaj_Click(ByVal sender As System.Object, ByVal e As
   System.EventArgs) Handles btnOblicz.Click

      'Sprawdzanie warunku uzupełnionych pół
      If txtL.Text = „” Then
         MessageBox.Show(„Podaj wartość długości l wspornika!”,
         „Brak danych”, MessageBoxButtons.OK, MessageBoxIcon.Warning)
      Else If txtH.Text = „” Then
         MessageBox.Show(„Podaj wartość wysokości h wspornika!”,
         „Brak danych”, MessageBoxButtons.OK, MessageBoxIcon.Warning)
      Else If txtP.Text = „” Then
```

```
    MessageBox.Show(„Podaj wartość siły P działającej na wspornik!”,
    „Brak danych”, MessageBoxButtons.OK, MessageBoxIcon.Warning)
Else If txtAlfa.Text = „” Then
    MessageBox.Show(„Podaj wartość kąta przyłożenia siły!”,
    „Brak danych”, MessageBoxButtons.OK, MessageBoxIcon.Warning)
Else If txtDzew.Text = „” Then
    MessageBox.Show(„Podaj wartość średnicy zewnętrznej D rury!”,
    „Brak danych”, MessageBoxButtons.OK, MessageBoxIcon.Warning)
Else If txtDwew.Text = „” Then
    MessageBox.Show(„Podaj wartość średnicy wewnętrznej d rury!”,
    „Brak danych”, MessageBoxButtons.OK, MessageBoxIcon.Warning)
Else
    'Pobieranie zmiennych
    l = CDbl(txtL.Text)
    h = CDbl(txtH.Text)
    P = CDbl(txtP.Text)
    alfa = CDbl(txtAlfa.Text)
    Dzew = CDbl(txtDzew.Text)
    Dwew = CDbl(txtDwew.Text)\

    'Obliczenia siły składowych
    P1 = P * Sin(alfa * (PI / 180))
    P1 = Round(P1, 2)
    txtP1.Text = P1 & “ N”

    P2 = P * Cos(alfa * (PI / 180))
    P2 = Round(P2, 2)
    txtP2.Text = P2 & “ N”

    'Obliczenia momentu gnącego
    Mg = (P1 * h) + (P2 * l)
    Mg = Round(Mg, 2)
    txtMg.Text = Mg & “ Nmm”

    'Obliczenia pola przekroju i wskaźnika wytrzymałości
    S = (PI / 4) * ((Dzew ^ 2) - (Dwew ^ 2))
    S = Round(S, 2)
    txtS.Text = S & “ mm^2”

    Wx = (PI / 32) * (((Dzew ^ 4) - (Dwew ^ 4))) / Dzew
    Wx = Round(Wx, 2)
    txtWx.Text = Wx & „ mm^3”

    'Obliczenia naprężeń
    SigmaR = P1 / S
    SigmaR = Round(SigmaR, 2)
    txtSigmaR.Text = SigmaR & “ MPa”

    TauT = P2 / S
    TauT = Round(TauT, 2)
    txtTauT.Text = TauT & “ MPa”
```

```
            SigmaG = Mg / Wx
            SigmaG = Round(SigmaG, 2)
            txtSigmaG.Text = SigmaG & „ MPa”'

            ‚Wyświetlanie wartości na rysunku w grupie „Grafika”
            lblL.Text = “l = “ & l & “ mm”
            lblH.Text = “h = “ & h & “ mm”
            lblP1.Text = “P1 = “ & P1 & “ N”
            lblP2.Text = “P2 = “ & P2 & “ N”
            lblAlfa.Text = “alfa = “ & alfa & Chr(176)
        End If
    End Sub

    Private Sub btnDane_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnResetDane.Click
        txtL.Clear()
        txtH.Clear()
        txtP.Clear()
        txtAlfa.Clear()
        txtDzew.Clear()
        txtDwew.Clear()
    End Sub

    Private Sub btnObliczenia_Click(ByVal sender As System.Object, ByVal
    e As System.EventArgs) Handles btnResetObliczenia.Click
        txtP1.Clear()
        txtP2.Clear()
        txtMg.Clear()
        txtS.Clear()
        txtWx.Clear()
        txtSigmaR.Clear()
        txtTauT.Clear()
        txtSigmaG.Clear()
    End Sub

    Private Sub btnResetGrafika_Click(sender As Object, e As EventArgs)
    Handles btnResetGrafika.Click
        lblL.Text = “l”
        lblH.Text = “h”
        lblP1.Text = “P1”
        lblP2.Text = “P2”
        lblAlfa.Text = “alfa”
    End Sub

    Private Sub btnZamknij_Click(sender As Object, e As EventArgs) Handles
    btnZamknij.Click
        Me.Close()
    End Sub
End Class
```

Do najważniejszych elementów opracowanego kodu należy zaliczyć:

1. Zaimportowanie biblioteki funkcji matematycznych, których wykorzystanie jest konieczne do realizacji obliczeń:

```
Imports System.Math
```

2. Zadeklarowanie niezbędnych zmiennych:

```
Dim l, h, P, alfa, Dzew, Dwew, P1, P2, Mg, S, Wx, SigmaR, TauT, SigmaG
As Double
```

3. Zastosowanie instrukcji warunkowych do sprawdzania warunku umieszczenia danych w polach tekstowych. Przykładowo:

```
If txtL.Text = „” Then
   MessageBox.Show(„Podaj wartość długości l wspornika!”, „Brak danych”,
   MessageBoxButtons.OK, MessageBoxIcon.Warning)
ElseIf txtH.Text = „” Then
   MessageBox.Show(„Podaj wartość wysokości h wspornika!”, „Brak danych”,
   MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

4. Zastosowanie poleceń pobierania danych z pól tekstowych. Przykładowo:

```
P = CDbl(txtP.Text)
alfa = CDbl(txtAlfa.Text)
```

5. Realizacja obliczeń, zastosowanie odpowiednich funkcji matematycznych. Przykładowo, wyznaczanie wartości funkcji sinus (polecenie *Sin*) oraz zaokrąglania wyniku za pomocą funkcji *Round*:

```
P1 = P * Sin(alfa * (PI / 180))
P1 = Round(P1, 2)
txtP1.Text = P1 & “ N”
```

6. Zastosowanie elementów kodu, pozwalających na łączenie obliczonych wartości z ich jednostkami w polach tekstowych oraz etykietach. Przykładowo:

```
lblP1.Text = “P1 = “ & P1 & “ N”
lblP2.Text = “P2 = “ & P2 & “ N”
lblAlfa.Text = “alfa = “ & alfa & Chr(176)
```

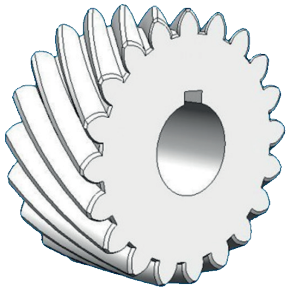
7. Wykorzystanie instrukcji czyszczenia pól. Przykładowo:

```
txtAlfa.Clear()
txtDzew.Clear()
txtDwew.Clear()
```

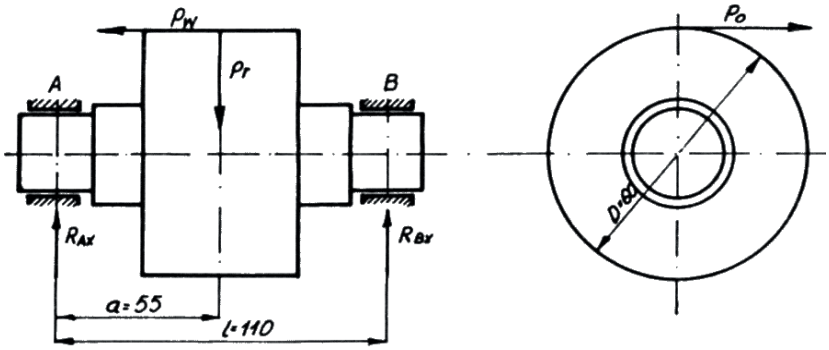
Zastosowanie wymienionych elementów pozwoliło na opracowanie w pełni funkcjonalnej aplikacji. Wykorzystując zaprezentowany przykład, można w analogiczny sposób opracować użytkowe programy komputerowe do analizy innych, kluczowych zagadnień inżynierskich, które zostały przedstawione w dalszej części publikacji.

Zagadnienie 2. Czas pracy łożysk przekładni śrubowej

Wałek, na którym osadzone jest koło walcowe o zębach śrubowych, obciążony siłami [1]: obwodową P_o = 1530N, promieniową P_r = 590N i wzdłużną P_w = 410N ułożyskowany jest w łożyskach tocznych. Podporę A stanowi łożysko kulkowe zwykłe 6206, C_1 = 14200N, n_{gr} = 13000 obr./min., zaś podporę B łożysko kulkowe skośne dwurzędowe 3206, C_2 = 25000N, C_3 = 20400N, n_{gr} = 8000 obr./min. Wałek wykonuje n = 1500 obr./min, a trwałość przekładni wynosi L_h = 10000. Oblicz czas pracy łożysk.



Rysunek 51. Koło walcowe o zębach śrubowych (opracowanie własnena podstawie [1])



dane wejściowe: **a, l, d**

Rysunek 52. Koło walcowe o zębach śrubowych – schemat obciążenia (opracowanie własne na podstawie [1])

Obciążenie podpór (poprzeczne):
Płaszczyzna X
Z warunku równowagi momentów względem punktu A:

$$\sum M_A = -P_r \cdot a + P_w \cdot \frac{D}{2} + R_{BX} \cdot l = 0,$$

więc:

$$R_{BX} = \frac{P_r \cdot a - P_w \cdot \frac{d}{2}}{l} = \frac{590 \cdot 55 - 410 \cdot 30}{110} = 183{,}18N$$

Z warunku równowagi momentów względem punktu B:

$$\sum M_B = R_{AX} \cdot l - P_r \cdot (l - a) - P_w \cdot \frac{D}{2} = 0,$$

więc:

$$R_{AX} = \frac{P_r \cdot (l - a) + P_w \cdot \frac{d}{2}}{l} = \frac{590 \cdot 55 + 410 \cdot 30}{110} = 406{,}82N$$

Płaszczyzna Y

$$R_{AY} = R_{BY} = \frac{P_o}{2} = \frac{1530}{2} = 765N$$

Wypadkowe reakcje podpór:

$$R_A = \sqrt{\left({R_{AX}}^2 + {R_{AY}}^2\right)} = \sqrt{406{,}82^2 + 765^2} = 866N$$

$$R_B = \sqrt{\left({R_{BX}}^2 + {R_{BY}}^2\right)} = \sqrt{183{,}18^2 + 765^2} = 787N$$

Łożysko A
Obciążenie zastępcze:

$$P = X \cdot V \cdot F_r + Y \cdot F_a; (F_r = R_a,\ F_a = P_w),$$

Dobierzemy współczynniki X i Y (dla naszego programu zakładamy ich wartość, w rzeczywistości odczytujemy z tabel):

$$X = 0{,}56 \text{ i } Y = 1{,}8, V = 1$$

$$P = 0{,}56 \cdot 1 \cdot 866 + 1{,}8 \cdot 410 = 1223N;$$

Czas pracy łożyska A:

$$L_h = \frac{10^6}{n \cdot 60}\left(\frac{C}{P}\right)^3 = \frac{10^6}{1500 \cdot 60}\left(\frac{14200}{1223}\right)^3 = 17300\ godzin$$

Łożysko B
Obciążenie zastępcze:

$$P = X \cdot V \cdot F_r + Y \cdot F_a;$$

($F_r = R_A \; lub \; R_B$ (większa wartość) – funkcja MAX(Ra, Rb), $F_a = P_w$),

Dobierzemy współczynniki X i Y (zakładamy w naszym programie:
$X = 1$ i $Y = 0{,}73, V = 1$)

$$P = 1 \cdot 1 \cdot 866 + 0{,}73 \cdot 410 = 1165 \; N$$

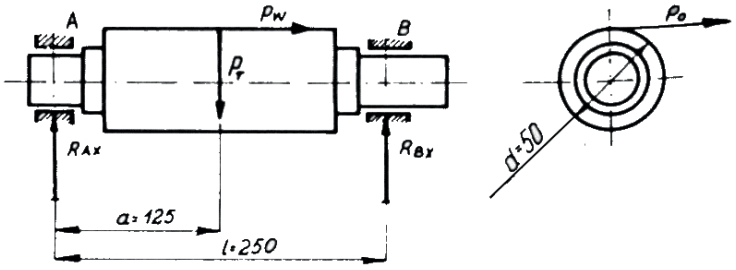
Czas pracy łożyska B:

$$L_h = \frac{10^6}{n \cdot 60}\left(\frac{C}{P}\right)^3 = \frac{10^6}{1500 \cdot 60}\left(\frac{25000}{1185}\right)^3 = 109589 \; godzin$$

Zagadnienie 3. Czas pracy łożysk przekładni ślimakowej

Łożyska ślimaka walcowego przekładni urządzenia dźwigowego stanowią [1]: B – dwa łożyska stożkowe 32306 o nośności C_1 = 55000N, C_{10} = 43500N; *A* – łożysko walcowe NU 1006 o nośności C_2 = 13600N, C_{20} = 9200N. Ślimak wykonuje n = 960 obr./min i obciążony jest siłami: obwodową P_o = 1600N, promieniową P_r = 1400N i wzdłużną P_w = 4000N. Żądany czas pracy łożysk wynosi L_{hmin} = 5000h. Oblicz trwałość łożysk.

Rysunek 53. Ślimak i ślimacznica (opracowanie własne na podstawie [1])



Dane wejściowe: **a, l, d**

Rysunek 54. Ślimak i ślimacznica – schemat obciążenia (opracowanie własne na podstawie [1])

Promieniowe obciążenie łożysk:
Płaszczyzna X
Z warunku równowagi momentów względem punktu A:

$$\sum M_A = R_{Ax} \cdot l - P_r \cdot (l - a) + P_w \cdot \frac{d}{2} = 0, \text{ więc:}$$

$$R_{AX} = \frac{-P_w \cdot \frac{d}{2} + P_r \cdot (l - a)}{l} = \frac{-4000 \cdot 25 + 1400 \cdot 125}{250} = 300N$$

Z warunku równowagi momentów względem punktu B:

$$\sum M_B = -R_{Bx} \cdot l + P_w \cdot \frac{D}{2} + P_r \cdot a = 0, \text{ więc:}$$

$$R_{BX} = \frac{P_r \cdot a + P_w \cdot \frac{d}{2}}{l} = \frac{1400 \cdot 125 + 4000 \cdot 25}{250} = 1100N$$

Płaszczyzna Y

$$R_{AY} = R_{BY} = \frac{P_o}{2} = \frac{1600}{2} = 800N$$

Wypadkowe reakcje podpór:

$$R_A = \sqrt{\left({R_{AX}}^2 + {R_{AY}}^2\right)} = \sqrt{300^2 + 800^2} = 854N$$

$$R_B = \sqrt{\left({R_{BX}}^2 + {R_{BY}}^2\right)} = \sqrt{1100^2 + 800^2} = 1360N$$

Łożysko A
Obciążenie zastępcze:

$P = V \cdot F_r$ ($F_r = R_A \; lub \; R_B$ – większa wartość) – funkcja MAX(Ra, Rb)

Zakładamy dla naszego programu: $V = 1$

$$P = 1 \cdot 1360 = 1360N$$

Trwałość łożyska A:

$$L_h = \frac{10^6}{n \cdot 60}\left(\frac{C_2}{P}\right)^{\frac{10}{3}} = \frac{10^6}{960 \cdot 60}\left(\frac{13600}{1360}\right)^{\frac{10}{3}} = 37400 \; godzin$$

Łożysko B (dwa łożyska stożkowe)
Obciążenie zastępcze:

$P = X \cdot V \cdot \frac{F_r}{2} + Y \cdot F_a$; ($F_r = R_A \; lub \; R_B -$ większa wartość, $F_a = P_w$),

Dobierzemy współczynniki X i Y (w naszym programie zakładamy je, ale w rzeczywistości – odczytujemy z tabel): $X = 0{,}4$ i $Y = 2, V = 1$

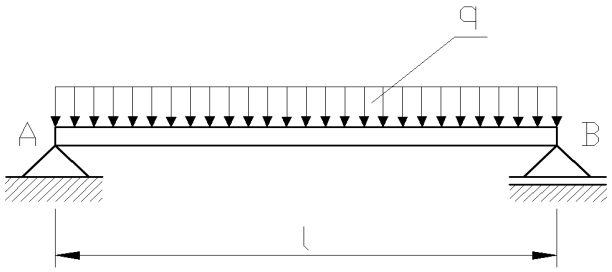
$$P = 0{,}4 \cdot 1 \cdot \frac{1360}{2} + 2 \cdot 4000 = 8272N$$

Trwałość łożysk B:

$$L_h = \frac{10^6}{n \cdot 60}\left(\frac{C_1}{P}\right)^{\frac{10}{3}} = \frac{10^6}{960 \cdot 60}\left(\frac{55000}{8272}\right)^{\frac{10}{3}} = 9500 \; godzin$$

Zagadnienie 4. Ugięcie belki wolnopodpartej obciążonej obciążeniem ciągłym

Dla belki wolnopodpartej o długości l, obciążonej obciążeniem ciągłym o wartości q na całej długości (rys. 55), należy wyznaczyć wartość reakcji w podporach R_A i R_B, wartość momentu gnącego Mg, obliczyć strzałkę ugięcia f oraz kąt ugięcia Θ. Belka ma przekrój prostokątny o wysokości h i szerokości b, zaś moduł Younga materiału, z jakiego wykonano belkę, wynosi E [2].



Rysunek 55. Schemat obciążenia belki (opracowanie własne na podstawie [2])

Poniżej przedstawiono przykładowe obliczenia dla danych l = 4 m = 4000 mm, q = 1,5 N/mm, h = 100 mm, b = 50 mm, E = 0,11 10^5 MPa (belka wykonana jest z drewna).

Na podstawie wymienionych powyżej parametrów można wyznaczyć kluczowe wartości szukane, które obliczone są poniżej.

Reakcje w podporach:

$$R_A = R_B = \frac{ql}{2} = \frac{1{,}5 \cdot 4000}{2} = 3000\ [\mathrm{N}]$$

Wartości momentu gnącego w połowie belki:

$$M_g = \frac{ql^2}{8} = \frac{1{,}5 \cdot 4000^2}{8} = 3000000\ [\mathrm{Nmm}] = 3000\ [\mathrm{Nm}]$$

Moment bezwładności belki:

$$M_g = \frac{bh^3}{12} = \frac{50 \cdot 100^3}{12} = 4166{,}67 \cdot 10^3\ [\mathrm{mm}^4]$$

Strzałka ugięcia belki:

$$f = \frac{5}{384} \cdot \frac{ql^4}{EI} = \frac{5}{384} \cdot \frac{1{,}5 \cdot 4000^4}{0{,}11 \cdot 10^5 \cdot 4166{,}67 \cdot 10^3} = 109{,}09\ [\mathrm{mm}]$$

Kąt ugięcia belki:

$$\theta = \frac{1}{24} \cdot \frac{ql^3}{EI} = \frac{1}{24} \cdot \frac{1{,}5 \cdot 4000^3}{0{,}11 \cdot 10^5 \cdot 4166{,}67 \cdot 10^3} = 0{,}0873 \text{ [mm]}$$

Na podstawie przedstawionych wzorów i obliczeń można wykonać aplikację pozwalającą analizować ugięcie belki o alternatywnych wartościach obciążenia, odmiennych wymiarach czy wykonanej z innych materiałów konstrukcyjnych.

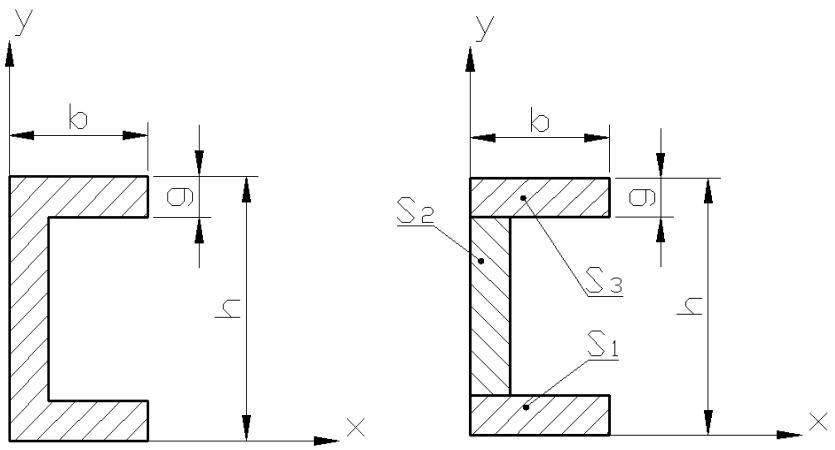
W poniższej tabeli 8 przedstawiono przykładowe wartości modułu Younga *E* dla różnych materiałów.

Tabela 8. Wartości modułu Younga E dla wybranych materiałów – na podstawie [3, 4].

Materiał	**Wartość modułu Younga *E* x 10^5 MPa**
Stal konstrukcyjna St3	2,1
Żeliwo ZL250	1,1
Aluminium	0,7
Miedź	1,1
Drewno (grab, sosna)	0,11
Beton	0,20

Zagadnienie 5. Wyznaczanie parametrów ceownika

Dla ceownika o wysokości *h*, szerokości *b* i grubości *g* (rys. 56) wyznacz pole powierzchni przekroju poprzecznego *S* ceownika oraz współrzędne środka ciężkości x_c i y_c [7].



Rysunek 56. Przekrój poprzeczny rozpatrywanego ceownika (opracowanie własne na podstawie [7])

Poniżej przedstawiono przykładowe obliczenia dla danych h = 100 mm = 10 cm, b = 50 mm = 5 cm, g = 6 mm = 0,6 cm. Wyznaczanie rozpatrywanych parametrów ceownika o konkretnych wymiarach wygląda następująco:

Pole przekroju ceownika:

$$S = S_1 + S_2 + S_3$$

$$S_1 = S_3 = b \cdot g = 5 \cdot 0{,}6 = 3\ [\mathrm{cm}^2]$$

$$S_2 = (h - 2g) \cdot g = (10 - 2 \cdot 0{,}6) \cdot 0{,}6 = 5{,}28\ [\mathrm{cm}^2]$$

$$S = 3 + 5{,}28 + 3 = 11{,}28\ [\mathrm{cm}^2]$$

Współrzędne środka ciężkości:

$$x_c = \frac{S_1 \cdot x_1 + S_2 \cdot x_2 + S_3 \cdot x_3}{S_1 + S_2 + S_3}$$

$$S_1 = 3\ [\mathrm{cm}^2]$$

$$S_2 = 5{,}28\ [\mathrm{cm}^2]$$

$$x_1 = x_3 = \frac{1}{2}b = \frac{1}{2} \cdot 5 = 2{,}5\ [\mathrm{cm}]$$

$$x_2 = \frac{1}{2}g = \frac{1}{2} \cdot 0{,}6 = 0{,}3\ [\mathrm{cm}]$$

$$x_c = \frac{3 \cdot 2{,}5 + 5{,}28 \cdot 0{,}3 + 3 \cdot 2{,}5}{3 + 5{,}28 + 3} = 1{,}47\ [\mathrm{cm}]$$

$$y_c = \frac{1}{2}h = \frac{1}{2} \cdot 10 = 5\ [\mathrm{cm}]$$

Zakończenie

Szanowny Czytelniku (najpewniej Studencie), jeśli to czytasz, to najwyraźniej dotarłeś do końca naszego podręcznika. Mamy nadzieję, że droga którą przeszedłeś i która doprowadziła Cię na ostatnie karty, nie była specjalnie skomplikowana oraz że możesz o sobie już powiedzieć, że teraz umiesz programować w języku Visual Basic. Staraliśmy się, żeby ta publikacja stanowiła kompletny i praktyczny podręcznik, który wprowadzi w świat programowania nawet osobę nieposiadającą żadnej wiedzy czy umiejętności z tego zakresu. Nasza książka nie tylko kładzie nacisk na teoretyczne podstawy języka Visual Basic, ale również oferuje liczne ćwiczenia praktyczne, co sprawia, że czytelnik ma okazję przetestować swoje umiejętności na konkretnych zadaniach. Jednocześnie skupiliśmy się na aspektach istotnych dla studentów kierunków związanych z inżynierią mechaniczną, umożliwiając im wykorzystanie programowania w rozwiązywaniu rzeczywistych problemów z ich dziedziny.

W naszej opinii książka ta stanowi solidne wsparcie dla studentów, przygotowując ich do efektywnego korzystania z języka Visual Basic zarówno w toku dalszego studiowania, jak i w przyszłych projektach inżynierskich.

Jesteśmy jednocześnie zainteresowani Waszym odbiorem naszej publikacji. Jeśli chcecie nam przekazać jakieś uwagi, szczególnie krytyczne – czekamy na nie z niecierpliwością. Chcemy, żeby ta książka była jak najlepsza, więc jeśli uważacie, że coś warto w niej zmienić lub dodać – prosimy o kontakt z jednym z autorów.

Bibliografia

[1] Czarnigowski J., Ferdynus M., Kuśmierz L., Ponieważ G., *Podstawy konstrukcji maszyn. Cz. I. Zbiór zadań*, Instytut Zastosowań Techniki, Lublin 2005.
[2] Iwulski Z., *Wyznaczanie sił tnących i momentów zginających w belkach. Zadania z rozwiązaniami*, Uczelniane Wydawnictwa Naukowo-Techniczne, Kraków 2001.
[3] Niezgodziński M. E., Niezgodziński T., *Wytrzymałość materiałów*, Wydawnictwo Naukowe PWN, Warszawa 2002.
[4] Niezgodziński M. E., Niezgodziński T., *Zadania z Wytrzymałości materiałów*, Wydawnictwa Naukowo-Techniczne, Warszawa 2016.
[5] Portal 101Computing.net: Tablica kodów ASCII, https://www.101computing.net/wp/wp-content/uploads/ASCII-Table.pdf [data dostępu: 13.02.2023 r.]
[6] Serwis FreeSVG, *Obraz paraboli*, https://freesvg.org/img/parabola.png [data dostępu: 13.02.2023 r.]
[7] Siuta W., *Mechanika techniczna*, Wydawnictwa Szkolne i Pedagogiczne, Warszawa 1985.

Literatura uzupełniająca

Halvorson M., *Microsoft Visual Basic 2010. Krok po kroku*, APN Promise, Warszawa 2010.
Liberty J., *Learning Visual Basic .NET*, O'Reilly Media, 2002.
Matulewski J., *Visual Basic .NET w praktyce. Błyskawiczne tworzenie aplikacji*, Helion, Gliwice 2012.
Treichel W., *Visual Basic dla studentów. Podstawy programowania w Visual Basic 2010,* Witkom (Salma Press), Warszawa 2011.
Wrotek W., *VBA dla Excela 2021 i 365 PL: 234 praktyczne przykłady*. Helion, Gliwice 2022.