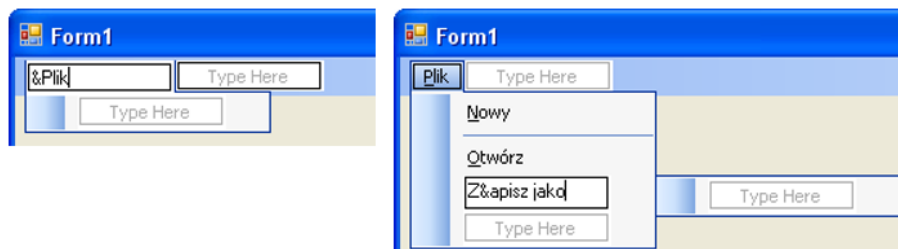


Edytor tekstu

Komponenty edytora

MenuStrip – pasek menu (menu główne)

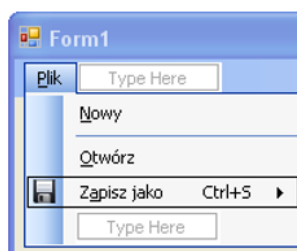
- Automatycznie lokuje się u góry okna (domyślnie właściwość *Dock = Top*)



- Wyposażone w wygodny edytor, umożliwiający tworzenie wielopoziomowego menu bezpośrednio na formularzu,
- Wpisanie opisu i potwierdzenie klawiszem <Enter> powoduje utworzenie komponentu *ToolStripMenuItem*, wpisanie znaku „-” (minus) dodaje separator,
- Dodawane w ten sposób elementy menu mają nietypowe nazwy: **opisElementuToolStripMenuItem** (np. **zapiszJakoToolStripMenuItem**), co ułatwia identyfikację funkcji obsługi zdarzeń – np. **zapiszJakoToolStripMenuItem_Click**

ToolStripMenuItem – pozycja menu

- Dostępny wyłącznie jako element menu (*MenuStrip* lub *ContextMenuStrip*) – nie ma w Toolboxie Visual Studio

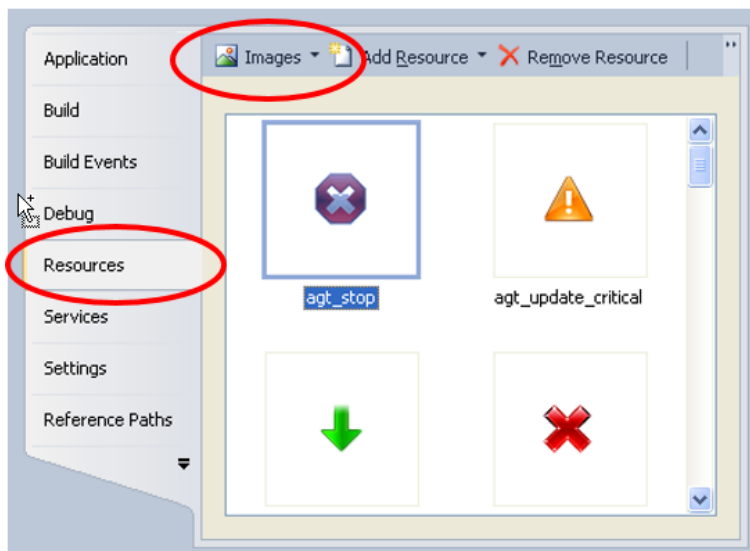


Właściwości:

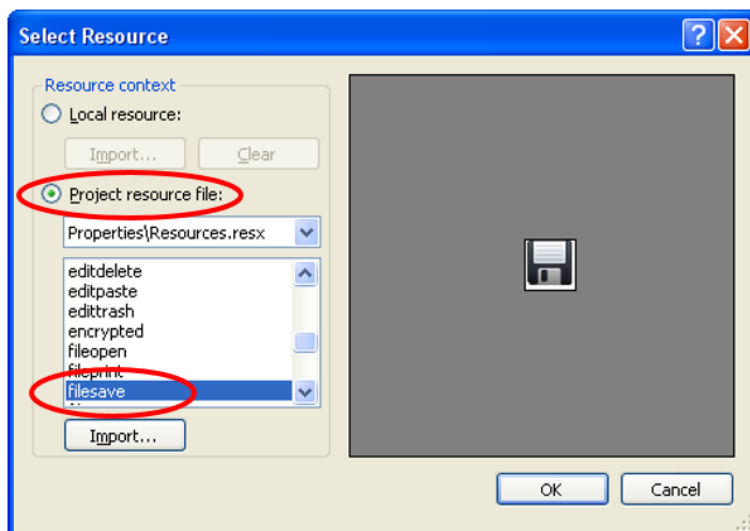
- Name** – nazwa w komponencie w programie
- Image** – obrazek wyświetlany obok opisu; Powinien być, jeżeli polecenie menu dostępne również przez pasek narzędzi, oczywiście wszędzie taki sam
- ShortcutKeys** – skrót klawiaturowy (nie mylić z *ShortcutKeyDisplayString* – opis skrótu)
- ShortcutKeyDisplayString** – właściwy opis pojawia się automatycznie po dodaniu skrótu przez właściwość *ShortcutKeys* (np. "Ctrl + O"), nie ma potrzeby zmieniać
- Checked** – symbol zaznaczenia
- CheckOnClick** – automatyczne włączanie i wyłączanie symbolu zaznaczenia – przekształca element menu w rodzaj *CheckBoxa* (nie należy używać, jeżeli dana opcja jest dostępna również inną drogą, np. przez pasek narzędzi)

Zasoby aplikacji

- Dwukrotne kliknięcie folderu *Properties* (właściwości), w oknie *Solution Explorer* (eksplorator rozwiązania) otwiera okno właściwości aplikacji
- Zakładka *Resources* (tj. zasoby) zawiera zasoby dostępne w całej aplikacji;
Po zmianie rodzaju zasobów ze *String* na *Images* można dodać obrazki

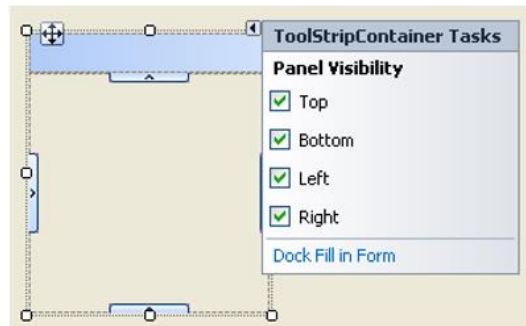


- Umieszczone w zasobach obrazki można umieszczać na wielu komponentach, np. w menu i na pasku narzędzi; Do wyboru obrazka z zasobów aplikacji lub wskazania pliku obrazka służy specjalne okno dialogowe:



ToolStripContainer – kontener na paski narzędzi

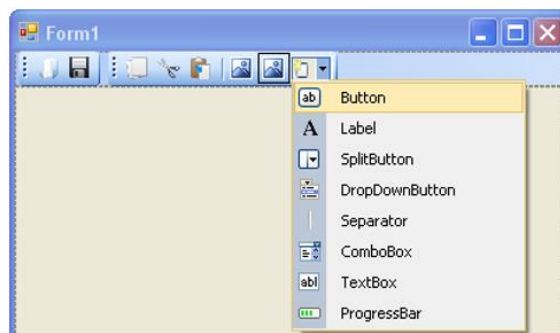
- Zwykle zajmuje całe okno (*Dock = Fill*)
- Składa się z czterech paneli ulokowanych przy krawędziach, przeznaczonych na paski narzędzi i menu, oraz centralnego panelu, w którym można umieścić główne komponenty aplikacji – w przypadku edytora *TextBox* lub *RichTextBox*



- Menu i paski narzędzi można swobodnie przemieszczać między czynnymi krawędziami kontenera, o ile właściwość *LayoutStyle* jest typu *StackWithOverflow*

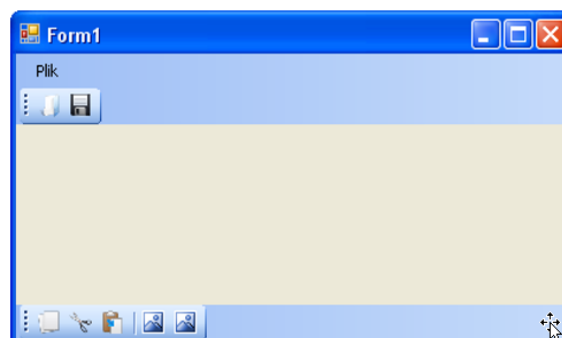
ToolStrip – pasek narzędzi

- Automatycznie lokuje się u góry okna (*Dock = Top*)
- Wyposażone w wygodny edytor, umożliwiający dodawanie elementów, najczęściej przycisków i separatorów, bezpośrednio na formularzu



Właściwości

- **ImageScalingSize** – rozmiar obrazków na przyciskach
- **LayoutStyle** – jeżeli ma wartość *StackWithOverflow*, to pasek można przenosić między aktywnymi krawędziami *ToolStripContainer*; a jeżeli ma wartość *Flow*, to nie można przenosić (brak jest stosownego uchwytu z lewej strony paska) – tak powinno się ustawiać paski umieszczone bezpośrednio na formularzu (bez *ToolStripContainer*)



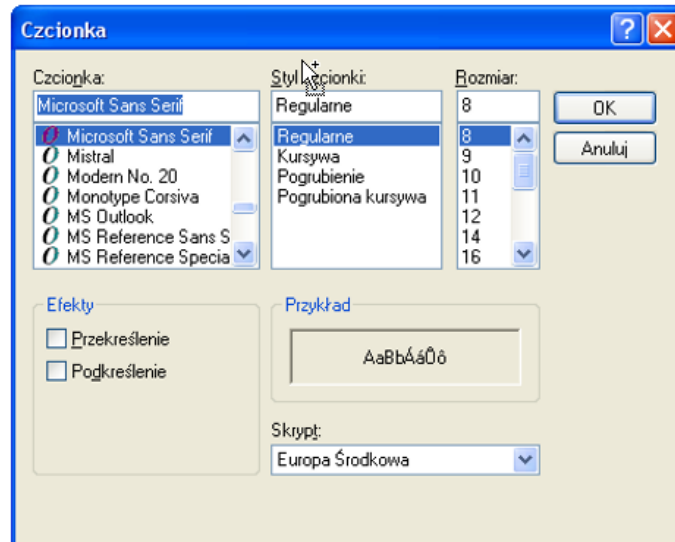
ToolStripButton – przycisk na pasku narzędzi

Właściwości

- **Image** – obrazek; można korzystać z zasobów aplikacji (jak w przypadku komponentu *ToolStripMenuItem*)
- **ImageTransparentColor** – kolor tła na obrazku, który będzie zamieniony na przezroczystość po wyświetleniu
- **AutoToolTip** – wyświetlanie podpowiedzi w "dymku" po wskazaniu myszą
- **ToolTipText** – treść podpowiedzi w dymku
- **Checked** – zaznaczony (tj. trwale wciśnięty); Pozwala traktować ToolStripButton jak pole wyboru (*Checkbox*) lub przycisk radiowy (*RadioButton*)
- **CheckOnClick** – automatycznie zmienia *Checked* po kliknięciu (nie wystarczy, jeżeli przycisk ma działać jak *RadioButton*)

FontDialog – Okno dialogowe "Czcionka"

- Komponent nie ma widocznej reprezentacji na oknie (podobnie pozostałe dialogi)
- Metoda *ShowDialog* wyświetla standardowe okno dialogowe systemu Windows zmiany czcionki; Właściwości komponentu określają jego wygląd i zachowanie



Właściwości

- ShowApply – wyświetla przycisk "Zastosuj"; trzeba obsługiwać zdarzenie *Apply*
- ShowEffects – wyświetla style (Pogrubienie, Pochylenie, ...)
- ShowColor – wyświetla komponent do zmiany koloru czcionki. Domyślnie wyłączony, ponieważ właściwość *Font* nie obejmuje koloru; Jeżeli włączony, to równoległe ze zmianą czcionki trzeba zmienić kolor
- FontMustExist – pozwala wybrać tylko istniejącą czcionkę
- **Font** – cechy czcionki:
 - proponowane (przed wyświetleniem okna),
 - wybrane przez użytkownika (po zamknięciu okna)
- **Color** – kolor czcionki:
 - proponowany (przed wyświetleniem okna),
 - wybrany przez użytkownika (po zamknięciu okna)

FontDialog – sposób użycia

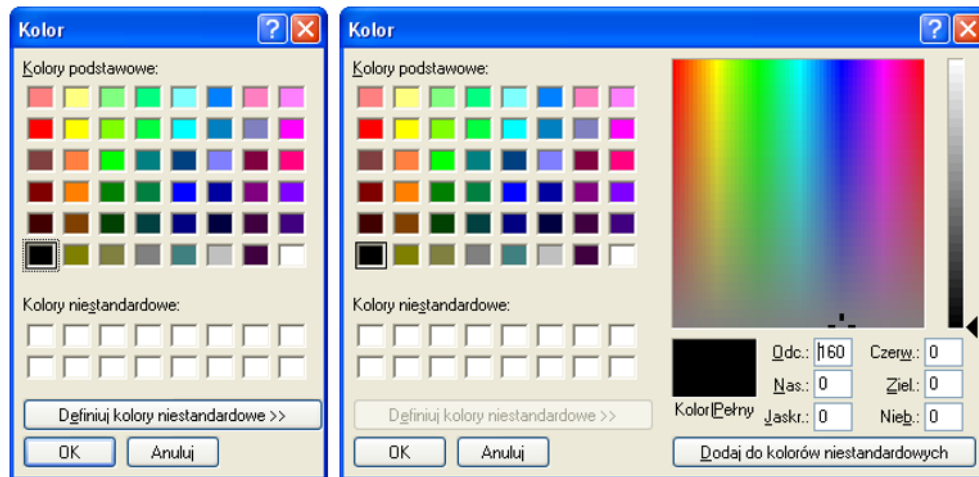
Najpierw należy przenieść cechy czcionki (i ew. koloru) z komponentu do okna dialogowego, następnie wyświetlić okno i koniecznie sprawdzić rezultat; Jeżeli jest *DialogResult.OK*, przenieść cechy czcionki z okna dialogowego do komponentu:

```
fontDlg.Font = textBox.Font;  
if (fontDlg.ShowDialog() == DialogResult.OK)  
{  
    textBox.Font = fontDlg.Font;  
}
```

W *RichTextBox* należy użyć właściwości *SelectionFont* oraz *SelectionColor*

ColorDialog – Okno dialogowe "Kolor"

- Komponent nie ma widocznej reprezentacji na oknie (jak pozostałe dialogi)
- Metoda *ShowDialog* wyświetla standardowe okno dialogowe systemu Windows zmiany koloru; Właściwości komponentu określają jego wygląd i zachowanie



Właściwości

- *FullOpen* – wyświetla całe okno
- *AllowFullOpen* – zezwala na odsłonięcie całego okna
- *SolidColorOnly* – pozwala wybrać tylko standardowe kolory (jeżeli *true*, to *AllowFullOpen* powinno być wyłączone)
- **Color** – kolor czionki:
 - proponowany (przed wyświetleniem okna),
 - wybrany przez użytkownika (po zamknięciu okna)

ColorDialog – sposób użycia

Najpierw należy przenieść kolor z komponentu do okna dialogowego, następnie wyświetlić okno i koniecznie sprawdzić rezultat; Jeżeli jest *DialogResult.OK*, przenieść kolor z okna dialogowego do komponentu:

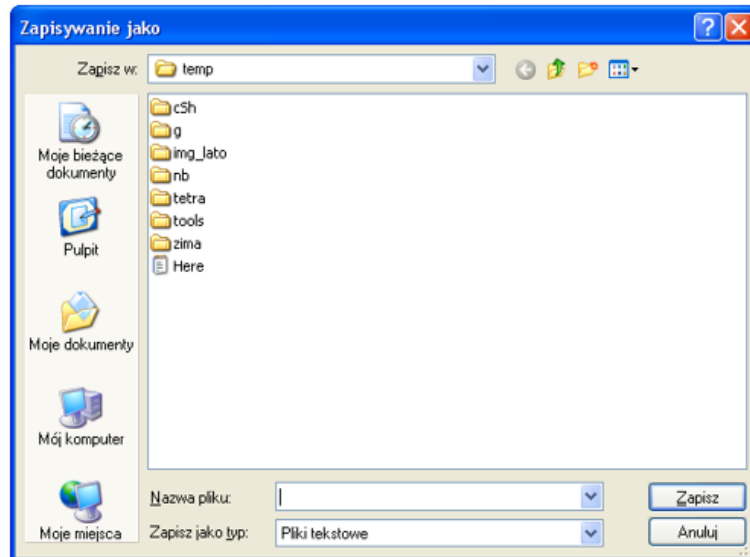
```
colorDlg.Color = textBox.ForeColor;
if (fontDlg.ShowDialog() == DialogResult.OK)
{
    textBox.ForeColor = colorDlg.Color;
}
```

W podobny sposób można zmienić kolor tła (*BackColor*).

W *RichTextBox* należy użyć właściwości *SelectionColor* oraz *SelectionBackColor*

SaveFileDialog – Okno dialogowe "Zapisywanie"

- Komponent nie ma widocznej reprezentacji na oknie (jak pozostałe dialogi)
- Metoda *ShowDialog* wyświetla standardowe okno dialogowe systemu Windows zapisywania pliku; Właściwości komponentu określają jego wygląd i zachowanie



Właściwości

- **Title** – tekst wyświetlany na pasku tytułu okna dialogowego
- **OverwritePrompt** – jeżeli plik istnieje, to wyświetla pytanie czy ma zostać nadpisany
- **CreatePrompt** – jeżeli plik nie istnieje, pyta czy utworzyć
- **CheckFileExists** – można wybrać tylko istniejący plik
- **CheckPathExists** – można wybrać tylko istniejący folder
- **SupportMultiDottedExtensions** – obsługa rozszerzeń w rodzaju .exe.manifest
- **ValidateNames** – nie można podać nazwy pliku z zabronionymi znakami (np. ":")
- **Filter** – filtr wyświetlanych plików Opis_1|Maska_1|Opis_2|Maska_2 ...
Pliki tekstowe|.txt|Pliki źródłowe C#|.cs|Wszystkie pliki|*.*
- **FilterIndex** – numer początkowo używanego filtru (numeracja od 1, nie od 0)
- **AddExtension** – czy do nazwy ma być dodawane rozszerzenie;
jeżeli tak, to dodawane jest rozszerzenie zależne od filtru
- **DefaultExt** – rozszerzenie domyślne (jeżeli nie wynika z filtru)
- **InitialDirectory** – katalog początkowy (jeżeli nie podany – katalog ostatnio używany do zapisywania pliku)
- **FileName** – kwalifikowana (z pełną ścieżką) nazwa pliku:
 - proponowana (przed wyświetleniem okna),
 - wybrana przez użytkownika (po zamknięciu okna)

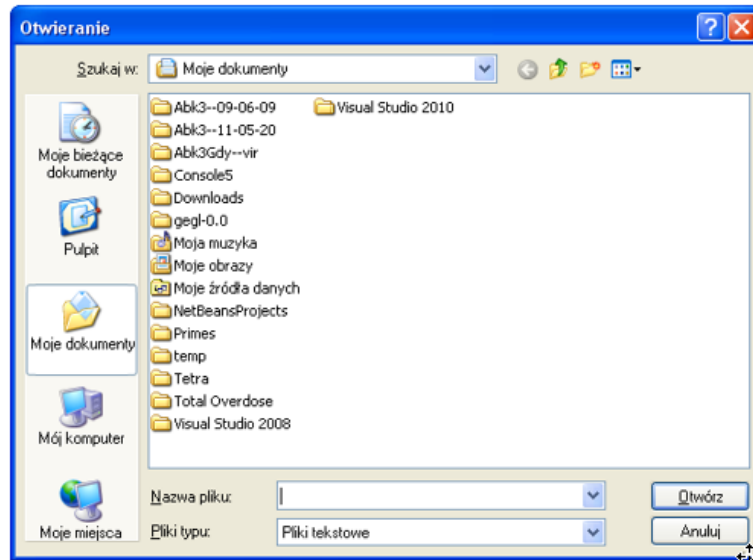
SaveFileDialog– sposób użycia

- Należy wyświetlić okno i koniecznie sprawdzić rezultat; Jeżeli rezultat jest *DialogResult.OK*, zapisać plik:

```
if (saveDlg.ShowDialog() == DialogResult.OK)
{
    // zapisanie pliku
}
```

OpenFileDialog – Okno dialogowe "Otwieranie"

- Komponent nie ma widocznej reprezentacji na oknie (jak pozostałe dialogi)
- Metoda *ShowDialog* wyświetla standardowe okno dialogowe systemu Windows otwierania pliku; Właściwości komponentu określają jego wygląd i zachowanie



Właściwości

- **Title** – tekst wyświetlany na pasku tytułu okna dialogowego
- **CheckFileExists** – można wybrać tylko istniejący plik
- **CheckPathExists** – można wybrać tylko istniejący folder
- **Multiselect** – możliwość wskazania kilku plików
- **ShowReadOnly** – pokazuje pole "Otwórz tylko do odczytu"
- **ReadOnlyChecked** – stan ww. pola
- **SupportMultiDottedExtensions** – obsługa rozszerzeń w rodzaju .exe.manifest
- **ValidateNames** – nie można podać nazwy pliku z zabronionymi znakami (np. ":")
- **Filter** – filtr wyświetlanych plików. Można dać po kilka masek w każdej kategorii
Pliki tekstowe|*.txt|Pliki www|*.htm; *.html|Wszystkie pliki|*.*
- **FilterIndex** – numer początkowo używanego filtra (numeracja od 1, nie od 0)
- **AddExtension** – czy do nazwy ma być dodawane rozszerzenie;
- **InitialDirectory** – katalog początkowy (jeżeli nie podany – katalog ostatnio używany do zapisywania pliku)
- **FileName** – kwalifikowana (z pełną ścieżką) nazwa pliku:
 - proponowana (przed wyświetleniem okna),
 - wybrana przez użytkownika (po zamknięciu okna)

OpenFileDialog– sposób użycia

- Należy wyświetlić okno i koniecznie sprawdzić rezultat; Jeżeli rezultat jest *DialogResult.OK*, wczytać plik:

```
if (openDlg.ShowDialog() == DialogResult.OK)
{
    // odczytanie pliku
}
```

TextBox – pole edycyjne

- Prostý edytor, odpowiednik systemowego Notatnika

Właściwości:

- **Dock** – dostosowanie rozmiaru i położenia (przyda się *Fill*)
- **Text** – wyświetlany tekst, dostępny do edycji
- **Multiline** – czy może zawierać więcej niż jedną linię tekstu
- **MaxLength** – maksymalna długość tekstu
- **WordWrap** – zawijanie tekstu
- **ScrollBars** – paski przewijania (domyślnie brak)
- **TextAlign** – wyrównanie tekstu
- **Font** – cechy czcionki
- **ForeColor** – kolor tekstu
- **BackColor** – kolor tła
- **Modified** – tekst został zmodyfikowany;
Uwaga! Automatyczna zmiana *Modified* tylko *false* → *true*, odwrotnie "ręcznie"

Zdarzenia:

- **TextChanged** – informuje o każdej zmianie tekstu (nie przyda się)
- **[-]** – brak jest zdarzenia informującego o zmianie zaznaczenia lub położenia kursora; Zamiast tego można obsługiwać wszystkie zdarzenia, których pośrednim skutkiem jest ww. zmiana, tj. *KeyDown*, *KeyPress*, *MouseDown*, *MouseUp*

Metody

- **Cut** – wytnij
- **Copy** – kopiuje
- **Paste** – wklej
- **SelectAll** – zaznacz wszystko
- **Undo** – cofnij (cofa tylko jedną, ostatnią czynność; kolejne użycie cofa cofnięcie)
- **Redo** – ponów
- **Clear** – wyczyść

Niemal wszystkie podstawowe operacje wykonują łatwe do użycia metody. Nie ma metod do odczytu i zapisu plików

Zmiana cech czcionki

Poszczególne cechy czcionki (np. rozmiar) są dostępne programowo tylko do odczytu. Aby zmienić tylko jedną cechę, należy utworzyć nowy obiekt *Font*, kopiując pozostałe cechy:

```
Font cf = textBox1.Font;  
textBox1.Font = new Font(  
    cf.FontFamily, 15.0f, cf.Style);
```

Najwięcej kłopotu sprawia właściwość *Style*. Może przyjmować wartości: *Regular*, *Bold*, *Italic*, *Underline* oraz *Strikeout*. Jest to typ wyliczeniowy z atrybutem *FlagsAttribute*, przez co na jego wartościach można wykonywać operacje binarne OR ("|") oraz XOR ("^"). Dla przykładu, następujący kod spowoduje przełączenie (włączenie, jeżeli było wyłączone, lub odwrotnie) pogrubienia:

```
Font cf = textBox1.Font;  
textBox1.Font = new Font(cf, cf.Style ^ FontStyle.Bold);
```

TextBox – odczyt i zapis plików

W bibliotece .Net jest kilka klas, które można do tego użyć. Np. klasa File dostarcza szereg statycznych metod, m.in. do tworzenia, kopiowania, usuwania, odczytywania i zapisu plików.

- `String ReadAllText (String path)`
Odczytuje zawartość pliku tekstowego i dostarcza w postaci pojedynczego łańcucha znaków; *path* to kwalifikowana nazwa pliku (tj. ze ścieżką dostępu), jaką dostarcza np. okno dialogowe OpenFileDialog

```
textBox.Text = File.ReadAllText(openDlg.FileName);
```

- `void WriteAllText(String path, String contents)`
Zapisuje do pliku (*path*) wielolinijkową zawartość (*contents*)

```
File.WriteAllText(saveDlg.FileName, textBox.Text);
```

RichTextBox – pole edycyjne RTF

- Edytor z obsługą formatu RTF (ang. *Rich Text Format*), odpowiednik WordPada; posiada niemal wszystkie właściwości, metody i zdarzenia komponentu *TextBox* oraz kilka dodatkowych

Właściwości:

- **Text** – wyświetlany tekst, dostępny do edycji
- **Multiline** – czy może zawierać więcej niż jedną linię tekstu
- **MaxLength** – maksymalna długość tekstu (2.147.483.647 zamiast 32.767 *TextBox*)
- **WordWrap** – zawijanie tekstu
- **ScrollBars** – paski przewijania (domyślnie automatyczne)

Brakuje kilku właściwości *TextBox* (*TextAlign*, *AcceptsReturn*), nie ma też menu kontekstowego. Zamiast tego są inne – ponieważ *RichTextBox* umożliwia różne formatowanie fragmentów tekstu. Właściwości specyficzne (niedostępne w *TextBox*) określają cechy zaznaczonego fragmentu lub tzw. punktu wstawiania (tj. pozycji kursora)

Cechy akapitów:

- **SelectionAlignment** – wyrównanie tekstu
- **SelectionBullet** – wypunktowanie akapitów
- **SelectionIndent**, **SelectionHangingIndent** – lewy margines
- **SelectionRightIndent** – prawy margines

Cechy tekstu:

- **SelectionBackColor** – kolor tła
- **SelectionColor** – kolor tekstu
- **SelectionFont** – czcionka (krój, wielkość, styl)

Zdarzenia:

- **TextChanged** – informuje o każdej zmianie tekstu (nie przyda się)
- **SelectionChanged** – informuje o zmianie obszaru zaznaczenia lub przesunięciu kursora; można wykorzystać do "sprzężenia zwrotnego" – zmiany stanu komponentów menu i paska narzędzi zależnie od cech zaznaczonego fragmentu tekstu

Metody:

Wszystkie znane z *TextBox*, a dodatkowo *RichTextBox* potrafi zapisywać i odczytywać pliki, zarówno w formacie RTF, jak i zwykłe pliki tekstowe

- **Cut** – wytnij
- **Copy** – kopiuj
- **Paste** – wklej
- **SelectAll** – zaznacz wszystko
- **Undo** – cofnij (obsługuje wiele czynności)
- **Redo** – ponów
- **Clear** – wyczyść
- **SaveFile** – zapisz do pliku
- **LoadFile** – wczytaj plik

RichTextBox – odczyt pliku

Metody *SaveFile* i *LoadFile* są przeciążone i mają po dwa warianty, np. dla *LoadFile*:

```
LoadFile(String name)
LoadFile(String name, RichTextBoxStreamType streamType)
```

Pierwszy wariant wczytuje plik w formacie RTF, drugi w formacie RTF lub tekstowym, zależnie od wartości argumentu *streamType*.

Jeżeli program ma zapisywać i wczytywać *wyłącznie* pliki RTF, to należy użyć pierwszego wariantu metod *SaveFile* i *LoadFile*, wówczas wystarczy dosłownie jedna linia kodu.

Zapisanie pliku:

```
richTextBox.SaveFile(savedIlg.FileName);
```

Wczytanie pliku:

```
richTextBox.LoadFile(openDlg.FileName);
```

Typ *RichTextBoxStreamType* jest typem wyliczeniowym, określającym sposób potraktowania pliku przy zapisie i odczycie. Ma dwie wartości:

- **RichText** – format RTF
- **PlainText** – zwykły tekst

Próba wczytania pliku nie-RTF jako RTF kończy się błędem; Do sprawdzenia typu pliku można wykorzystać klasę *Path* (która udostępnia metody statyczne) albo obiekt klasy *FileInfo* (udostępnia właściwości i metody – ale najpierw trzeba utworzyć obiekt tej klasy).

Klasa *Path* – analiza kwalifikowanej nazwy pliku

Metody klasy *Path* wymagają podania kwalifikowanej nazwy pliku (należy podać nazwę wybraną przez użytkownika w oknie dialogowym *OpenFileDialog* lub *SaveFileDialog*, dostępną we właściwości *FileName* tych komponentów):

- **GetExtension (String path)** – rozszerzenie (np. ".rtf", ".txt")
- **GetName (String path)** – nazwa z rozszerzeniem (np. "Plik.rtf")

Klasa *FileInfo* – informacja o pliku

Konstruktor wymaga podania kwalifikowanej nazwy pliku (należy podać nazwę wybraną przez użytkownika w oknie dialogowym *OpenFileDialog* lub *SaveFileDialog*, dostępną we właściwości *FileName* tych komponentów, analogicznie jak dla klasy *Path*).

Po utworzeniu obiektu *FileInfo* można odczytać szczegółowe informacje o pliku (np. *Name* – nazwa, *Extension* – rozszerzenie nazwy, *Length* – rozmiar, *Attributes* – atrybuty systemowe pliku, np. *Hidden* – ukryty) z właściwości tego obiektu oraz wykonywać różne operacje na pliku (np. *CopyTo* – kopiowanie, *MoveTo* – przeniesienie, *Delete* – usunięcie)

Implementacja wczytywania pliku, z rozróżnieniem na pliki RTF i tekstowe, i przy wykorzystaniu klasy *Path*, może być zrealizowana następująco:

```
if (Path.GetExtension(openDlg.FileName) == ".rtf")
{
    richTextBox.LoadFile(openDlg.FileName,
        RichTextBoxStreamType.RichText);
}
else
{
    richTextBox.LoadFile(openDlg.FileName,
        RichTextBoxStreamType.PlainText);
}
```

Lub w skróconym zapisie, z wykorzystaniem wyrażenia warunkowego:

```
RichTextBoxStreamType type;

type = Path.GetExtension(openDlg.FileName) == ".rtf" ?
    RichTextBoxStreamType.RichText :
    RichTextBoxStreamType.PlainText;

richTextBox.LoadFile(openDlg.FileName, type);
```

Ta sama operacja przy użyciu klasy *FileInfo*, w skróconym zapisie:

```
FileInfo info = new FileInfo (openDlg.FileName);

RichTextBoxStreamType type = info.Extension == ".rtf" ?
    RichTextBoxStreamType.RichText :
    RichTextBoxStreamType.PlainText;

richTextBox.LoadFile(openDlg.FileName, type);
```

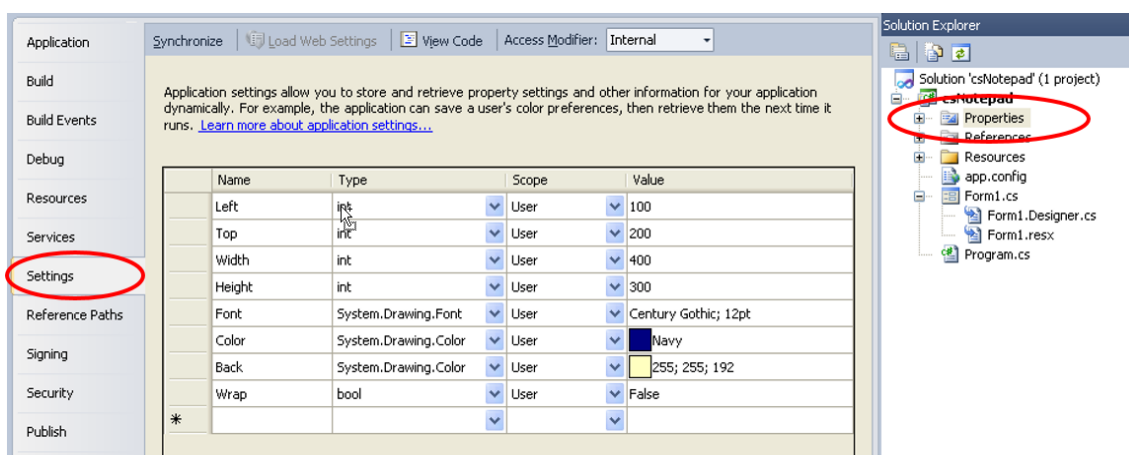
Ustawienia aplikacji

VS ma wbudowany mechanizm zarządzania ustawieniami - nie trzeba sięgać do bardziej złożonych i wymagających mechanizmów, takich jak pliki INI lub rejestr Windows

Komponent ustawień (ang. *Settings*) umożliwia dynamiczne zapisywanie i odtwarzanie dowolnych ustawień aplikacji oraz preferencji użytkownika. Można go wykorzystać do zapisania takich informacji, jak m.in.: rozmiar i położenie okna, opcje (np. zawijanie wierszy, widok pasków narzędzi, ...), ustawienia czcionki i kolorów itd.

Zdefiniowanie nazw, typów i domyślnych wartości

Nazwy, typy i wartości domyślne ustawień należy dodać w zakładce *Settings*. Wartości typów prostych można wpisać, dla typów w rodzaju *Font* lub *Color* jest dedykowany edytor.



Odczytanie i zastosowanie ustawień

W tym celu można wykorzystać konstruktor klasy formularza (nazywa się *Form1*, jak klasa formularza, i znajduje się na początku definicji klasy) albo (lepiej) funkcję obsługi zdarzenia *OnLoad* formularza, którą trzeba utworzyć w okienku właściwości, na zakładce *Events*.

Przy domyślnych ustawieniach, obiekt ustawień ma nazwę *Properties.Settings.Default*, natomiast nazwy ustawień są takie, jakie w zakładce *Settings* (obrazek powyżej)

W najprostszym przypadku wystarczy skopiować wartości ustawień do właściwości formularza lub komponentów – trzeba jednak pamiętać, aby były one zgodne ze stanem menu i pasków narzędzi – pokazuje to przykład opcji zawijania wierszy poniżej:

```
... Form1_Load(...)
{
    this.Left = Properties.Settings.Default.Left;
    this.Top = Properties.Settings.Default.Top;

    textBox.Font = Properties.Settings.Default.Font;
    textBox.ForeColor = Properties.Settings.Default.Color;

    textBox.WordWrap = Properties.Settings.Default.Wrap;
    zawijanieWierszyToolStripMenuItem.Checked =
        Properties.Settings.Default.Wrap;
}
```

Zapisanie ustawień

Tu najbardziej właściwe jest użycie funkcji obsługi zdarzenia *FormClosed* formularza, którą trzeba utworzyć w okienku właściwości formularza, na zakładce *Events*.

```
... Form1_FormClosed(...)
{
    Properties.Settings.Default.Left = this.Left;
    Properties.Settings.Default.Top = this.Top;

    Properties.Settings.Default.Font = textBox.Font;
    Properties.Settings.Default.Color = textBox.ForeColor;

    Properties.Settings.Default.Wrap = textBox.WordWrap;

    Properties.Settings.Default.Save();
}
```

Ważne! Aplikacja automatycznie wczytuje wszystkie ustawienia podczas uruchamiania, ale nigdy ich sama nie zapisuje – należy to wymusić używając metody *Save*.

Blokada zamknięcia programu

Zamknięcie programu – możliwe przyczyny

- Polecenie Plik/Zakończ
- Kliknięcie **x** na pasku tytułu
- Skrót klawiaturowy <Alt> + <F4>
- Zamknięcie systemu

Niezależnie od przyczyny, system wysyła do aplikacji komunikat nakazujący zakończenie działania. Objawia się to wystąpieniem *dwóch* zdarzeń:

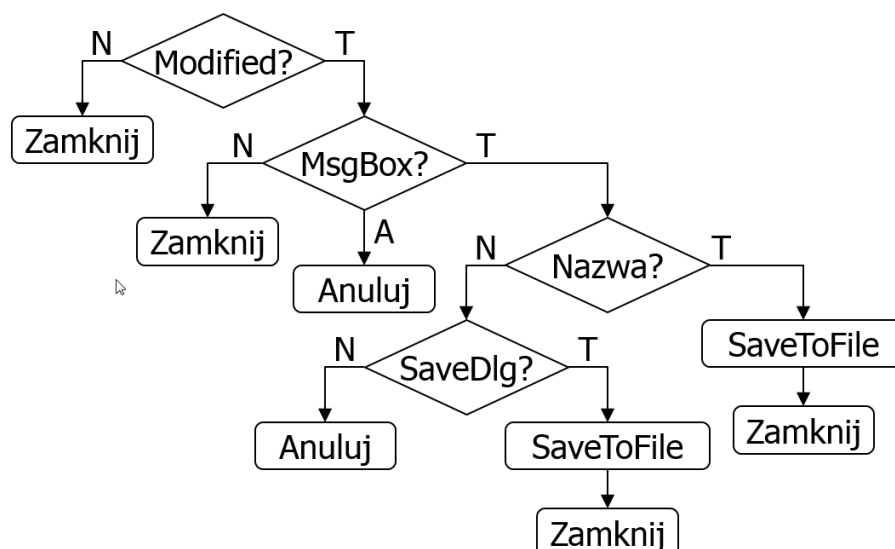
- Zdarzenie *OnFormClosing* (**pytanie** czy aplikacja może zostać zamknięta)
- Zdarzenie *OnFormClosed* (**informacja**, że aplikacja jest zamykana)

Jeżeli aplikacja odmawia zamknięcia (w tym celu należy ustawić właściwość *Cancel* argumentu zdarzenia na *true*, jak poniżej), to zdarzenie *OnFormClosed* nie wystąpi,

```
... FormClosing(object sender, FormClosingEventArgs e)
{
    if (textbox.Modified)
    {
        e.Cancel = true;
    }
}
```

Działanie edytora w zdarzeniu *OnFormClosing* zależy od kilku czynników:

- Czy dokument został zmodyfikowany od czasu utworzenia/otwarcia/zapisania – właściwość *Modified* komponentu *TextBox* (*RichTextBox*)
- Czy dokument posiada nazwę – dodatkowa zmienna, ustawiana po wykonaniu poleceń nowy/otwórz/zapisz jako
- Decyzje użytkownika – okno dialogowe *MessageBox* (co robić dalej?) – okno dialogowe *SaveFileDialog* (czy i gdzie zapisać plik?)



Blokada zamknięcia programu – wersja inteligentna

Implementacja pełnej wersji blokady jest dość skomplikowana: okno dialogowe daje użytkownikowi trzy możliwości (*Czy chcesz zapisać plik?* – *Tak, Nie, Anuluj*), a łącznie jest aż sześć możliwych zakończeń zdarzenia, co pokazuje diagram.

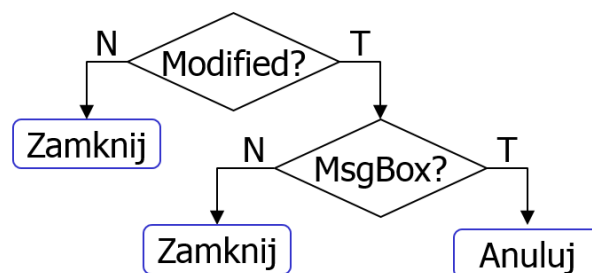
Blokada zamknięcia programu – warianty

Trzy stopnie trudności:

- Inteligentna – profesjonalnie (jw.),
- Półinteligentna – bez okna dialogowego zapisywania pliku,
- Ćwierćinteligentna – wymuszenie zapisania pliku przed zamknięciem.

Wersja półinteligentna

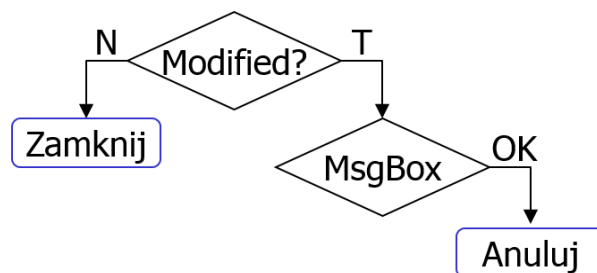
Jeżeli tekst został zmodyfikowany, to wyświetlane jest okno dialogowe *Czy chcesz zapisać zmiany?* z możliwymi odpowiedziami *Tak/Nie*. Odpowiedź *Tak* powoduje powrót do programu (*e.Cancel = true*), odpowiedź *Nie* – zamknięcie programu



Blokada zamknięcia programu – wersja półinteligentna

Wersja ćwierćinteligentna

Jeżeli tekst został zmodyfikowany, to wyświetlane jest okno dialogowe *Przed zamknięciem programu zapisz zmiany*, z jednym tylko przyciskiem, *Ok*, którego kliknięcie powoduje powrót do programu (*e.Cancel = true*).

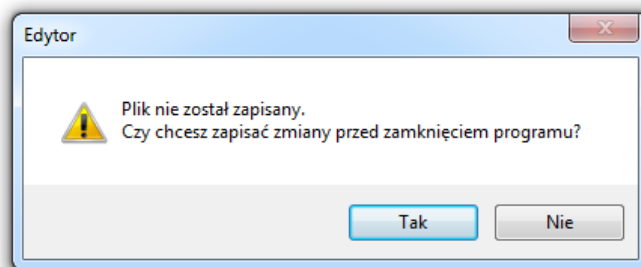


Blokada zamknięcia programu – wersja ćwierćinteligentna

Ważne! Podobne działania powinno się też realizować w reakcji na polecenia *Nowy* i *Otwórz*

Klasa MessageBox

Do wyświetlenia okna dialogowego z pytaniem i tylko kilkoma możliwymi odpowiedziami (*Czy chcesz zapisać plik?* – *Tak, Nie, Anuluj* albo *Czy chcesz zastąpić plik?* – *Tak, Nie* albo *Czy chcesz spróbować ponownie?* – *Ponów, Anuluj* itp.) zamiast tworzyć własny formularz, można użyć metody *Show* klasy *MessageBox*.



Metoda *Show* jest statyczna (więc nie trzeba tworzyć obiektu) i ma ponad dwadzieścia przeciążonych wersji, z różnymi kombinacjami argumentów. Sugerowana wersja:

```
DialogResult MessageBox.Show(  
    String text,  
    String caption,  
    MessageBoxButtons buttons,  
    MessageBoxIcon icon,  
    MessageBoxDefaultButton defButton)
```

- **text**, typu *String* – treść pytania,
- **caption**, typu *String* – tytuł okna (zazwyczaj nazwa programu macierzystego),
- **buttons**, typu wyliczeniowego *MessageBoxButtons* – zestaw przycisków do wyboru (*OK*, *OKCancel*, *AbortRetryIgnore*, *YesNoCancel*, *YesNo*, *RetryCancel*),
- **icon**, typu wyliczeniowego *MessageBoxIcon* – ikona wyświetlana obok komunikatu (*None*, *Question*, *Exclamation*, *Asterisk*, *Stop*, *Error*, *Warning*, *Information*)
- **defButton**, typu wyliczeniowego *MessageBoxDefaultButton* – przycisk domyślny (*Button1*, *Button2*, *Button3*)

Funkcja zwraca wartość typu *DialogResult*, którą należy traktować jako odpowiedź użytkownika na zadane pytanie. Warto użyć pomocniczej zmiennej do zapisania odpowiedzi, a sprawdzenie jej wartości przenieść do oddzielnej instrukcji, np.:

```
DialogResult result;  
  
result = MessageBox.Show(  
    "Czy chcesz zapisać zmiany",  
    "Edytor",  
    MessageBoxButtons.YesNo,  
    MessageBoxIcon.Exclamation,  
    MessageBoxDefaultButton.Button1);  
  
if (result == DialogResult.Yes)  
{  
    // działania dla TAK  
}
```

Strategia ucieczki od komplikacji

Jeżeli funkcja obsługi zdarzenia musi sprawdzać kilka warunków i wykonywać różne działania w różnych sytuacjach, kod takiej funkcji może mieć kilka poziomów zagnieżdżenia instrukcji, a dodając nowe funkcjonalności do programu często trzeba rozbudowywać funkcje napisane wcześniej. W takim przypadku sytuacji kod funkcji obsługi zdarzeń będzie się coraz bardziej gmatwał, co może doprowadzić do błędów. Aby uniknąć komplikacji kodu oraz potencjalnych błędów, można wykorzystać instrukcję *return*.

Weźmy za przykład taką strukturę funkcji obsługi zdarzenia:

```
if (warunek1)
{
    if (warunek2)
    {
        // warunki 1 i 2 spełnione
    }
    else
    {
        // warunek 1 spełniony, warunek 2 niespełniony
    }
}
else
{
    // warunek 1 niespełniony, warunek 2 nieistotny
}
```

Strategia ucieczki od komplikacji polega na użyciu instrukcji *return*, która powoduje natychmiastowe przerwanie wykonywania funkcji:

```
if (!warunek1)
{
    // warunek 1 niespełniony, warunek 2 nieistotny
    return
}

// jeżeli program dojdzie tu, to warunek 1 jest spełniony
// (gdyby było inaczej, to wcześniej wystąpi return)

if (warunek2)
{
    // warunki 1 i 2 spełnione
    return
}

// jeżeli program dojdzie tu, to warunek 1 jest spełniony
// a warunek 2 niespełniony
// (gdyby było inaczej, to wcześniej wystąpi return)

// warunek 1 spełniony, warunek 2 niespełniony
```

W efekcie zastosowania strategii ucieczki od komplikacji, zamiast dwóch zagnieżdżonych instrukcji *if-else* są dwie prostsze, niezagnieżdżone instrukcje *if*. Stosowanie tej strategii wymaga przemyślenia kolejności działań i, czasami, użycia operatora negacji (*!*).

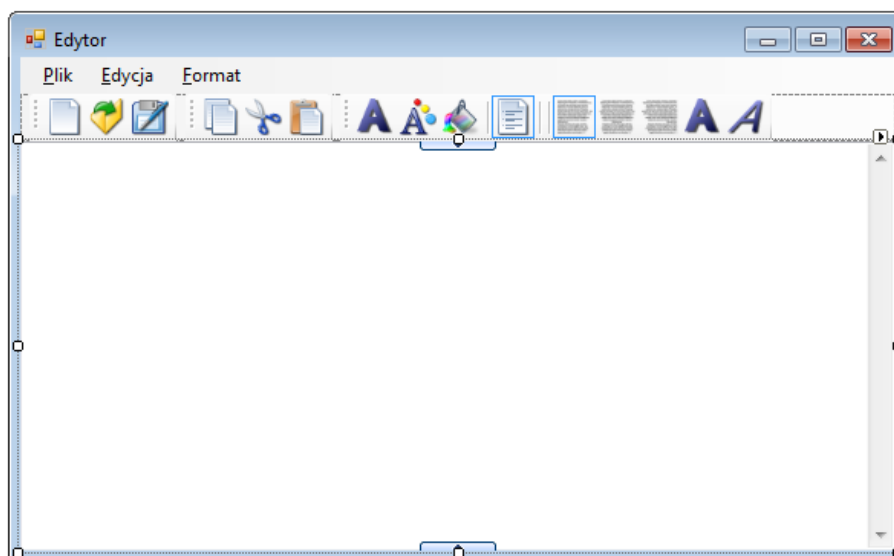
Zadania

Celem ćwiczenia jest opracowanie edytora, wyposażonego w menu i pasek narzędzi, umożliwiającego tworzenie, wczytywanie, edycję i zapisywanie plików tekstowych albo plików w formacie RTF, zależnie od wybranego komponentu. Do wyboru czcionek i kolorów oraz do wyboru nazwy pliku podczas zapisywania i wczytywania dokumentów, edytor powinien wykorzystywać standardowe okna dialogowe. Dodatkowo edytor powinien być wyposażony w funkcje zapisywania i odtwarzania ustawień (np. rozmiar i położenie okna) oraz blokadę zamknięcia, kiedy dokument nie zostanie zapisany przez użytkownika.

Podstawowe funkcje edytora

1. Okno i zasoby aplikacji

- Utworzyć nowy projekt, język C#, typ projektu Windows Forms,
- Umieścić na formularzu komponenty wg wzoru na rysunku poniżej: menu, kontener i dwa lub trzy paski narzędzi (albo jeden pasek narzędzi bez kontenera) oraz komponent do edycji tekstu, *TextBox* albo *RichTextBox*,
- Ustawić właściwości okna i komponentów: okno powinno mieć odpowiedni tytuł, kontener powinien mieć aktywne krawędzie górną i dolną i zajmować całe dostępne miejsce, podobnie komponent edycji tekstu,
- Do zasobów aplikacji dodać obrazki, do wykorzystania w menu i paskach narzędzi



2. Menu

- W pasku menu utworzyć potrzebne elementy, np.:
 - **Plik:** Nowy, separator, *Otwórz*, *Zapisz*, *Zapisz jako*, separator, *Zakończ*
 - **Edycja:** *Cofnij*, separator, *Kopiuj*, *Wytnij*, *Wklej*, separator, *Zaznacz wszystko*
 - **Format:** *Czcionka*, separator, *Kolor tekstu*, *Kolor tła*
 - **Widok:** *Zawijanie wierszy*, *Paski narzędzi*
- Wszystkie pozycje menu głównego i menu podrzędnych powinny mieć skróty klawiaturowe <Alt> + litera

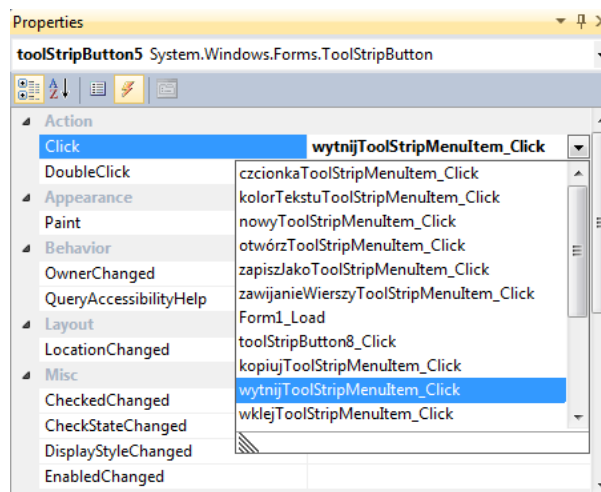
- Wszystkie pozycje menu podrzędnych *Plik* i *Edycja* powinny mieć dodatkowo skróty klawiaturowe <Ctrl> + litera. Należy użyć skrótów standardowych, znanych użytkownikom, np. *Otwórz*: <Ctrl> + O, *Wynij*: <Ctrl> + X.
- Wszystkie pozycje menu, które będą też dostępne przez pasek narzędzi, powinny mieć dodany obrazek – taki sam, jak na pasku narzędzi.

Wszystkie polecenia menu powinny mieć typowy układ, nazwy i skróty, tak aby koniecznością ich odkrywania nie drażnić użytkownika (tj. prowadzącego zajęcia).

3. Funkcje edycji tekstu

- Utworzyć funkcje obsługi zdarzeń dla wszystkich poleceń menu *Edycja* i wpisać stosowne instrukcje, np. dla *Cofnij* – `textBox.Undo()` lub `richTextBox.Undo()`
- Powiązać przyciski paska narzędzi z funkcjami obsługi zdarzeń, utworzonymi w poprzednim kroku dla menu.

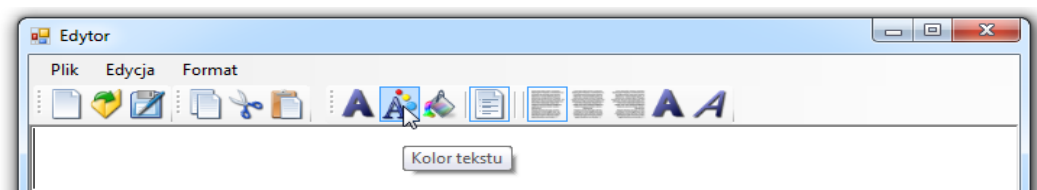
Ważne! Nie należy tworzyć nowej funkcji obsługi zdarzenia, jeżeli przycisk paska narzędzi ma wykonać dokładnie to samo, co polecenie menu. Zamiast tego należy zaznaczyć (pojedynczym kliknięciem) przycisk paska narzędzi, a następnie w okienku właściwości, na zakładce zdarzeń, wskazać już istniejącą funkcję utworzoną dla menu,



- Sprawdzić wszystkie polecenia edycji tekstu, w tym możliwość wywołania poleceń menu skrótami klawiaturowymi <Ctrl> + litera oraz <Alt> + litera
- Sprawdzić możliwość kopiowania tekstu pomiędzy edytorem a różnymi programami, w tym Notepad, Wordpad, Word i Visual Studio (jeżeli użyto *RichTextBox*, to należy sprawdzić również przenoszenie formatowania tekstu podczas kopiowania)

4. Paski narzędzi

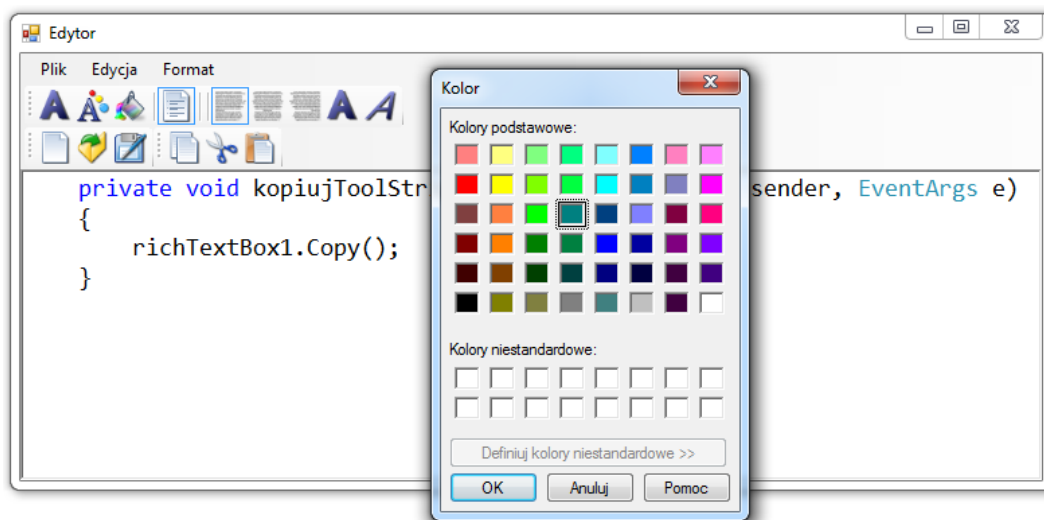
- We właściwościach wszystkich przycisków pasków narzędzi dodać opisy funkcji tych przycisków, we właściwości *ToolTipText*, albo wyłączyć wyświetlanie podpowiedzi przez właściwość *AutoToolTip*



Standardowe okna dialogowe

5. Czcionki i kolory

- Dodać do projektu komponenty okien dialogowych *FontDialog* i *ColorDialog*, ustawić odpowiednio ich opcje – np. czy można używać dowolnych kolorów czy tylko podstawowych, czy można zmieniać efekty czcionek itp.,
- Utworzyć funkcje obsługi zdarzeń dla poleceń menu *Czcionka*, *Kolor tekstu*, oraz *Kolor tła*; Dodać kod zmiany czcionki i kolorów,
Ważne! W przypadku komponentu *TextBox* należy korzystać z właściwości *Font*, *ForeColor* i *BackColor*, natomiast w przypadku *RichTextBox* – *SelectionFont*, *SelectionColor* i *SelectionBackColor*



- Powiązać przyciski paska narzędzi z funkcjami obsługi zdarzeń dla menu.
Ważne! Nie należy tworzyć nowej funkcji obsługi zdarzenia. Zamiast tego należy przypisać (w okienku właściwości) już istniejącą funkcję utworzoną dla menu
- Sprawdzić możliwość zmiany czcionki i kolorów przez menu i pasek narzędzi

Do dalszej części ćwiczeń należy wybrać zadanie 6 albo 7, zależnie od użytego komponentu.

6. Zapis i odczyt plików – komponent *TextBox*

- Dodać do projektu komponenty okien dialogowych *SaveFileDialog* i *OpenFileDialog*, ustawić odpowiednio ich opcje, a w szczególności:
 - **Filter** – pliki tekstowe (*.txt), pliki C# (*.cs), wszystkie pliki (*.*)
 - **DefaultExtension** – ".txt" (kropka txt)
- Utworzyć funkcje obsługi zdarzeń dla poleceń menu *Nowy*, *Zapisz jako*, oraz *Otwórz*; Dodać kod – czyszczenia zawartości komponentu (metoda *Clear*) oraz zapisywania i wczytywania pliku przy użyciu klasy *File*;
- Powiązać przyciski paska narzędzi z funkcjami obsługi zdarzeń, utworzonymi w poprzednim kroku dla menu.
- Sprawdzić możliwość zapisywania i wczytywania plików różnych typów, w tym plików tekstowych i plików źródłowych C# (można skopiować jeden z plików projektu Visual Studio z rozszerzeniem .cs na pulpit)

7. Zapis i odczyt plików RTF – komponent *RichTextBox*

- Dodać do projektu komponenty okien dialogowych *SaveFileDialog* i *OpenFileDialog*, ustawić odpowiednio ich opcje, a w szczególności:
 - **Filter** – tylko jeden rodzaj plików – pliki RTF,
 - **DefaultExtension** – ".rtf" (kropka rtf)
- Utworzyć funkcje obsługi zdarzeń dla poleceń menu *Nowy*, *Zapisz jako* oraz *Otwórz*; Dodać kod – czyszczenia zawartości komponentu (metoda *Clear*) oraz zapisywania i wczytywania pliku przy użyciu metod komponentu *RichTextBox*: *SaveFile* i *LoadFile*, **Ważne!** Drugi argument metod *SaveFile* i *LoadFile*, *streamType*, należy pominąć – zostanie wówczas użyty typ strumienia RTF
- Powiązać przyciski paska narzędzi z funkcjami obsługi zdarzeń, utworzonymi w poprzednim kroku dla menu.
- Sprawdzić możliwość zapisywania i wczytywania plików RTF, w tym otwierania w programie Word pakietu Office 365 plików utworzonych w edytorze i odwrotnie.

8. ** Zapis i odczyt plików różnych typów – komponent *RichTextBox*

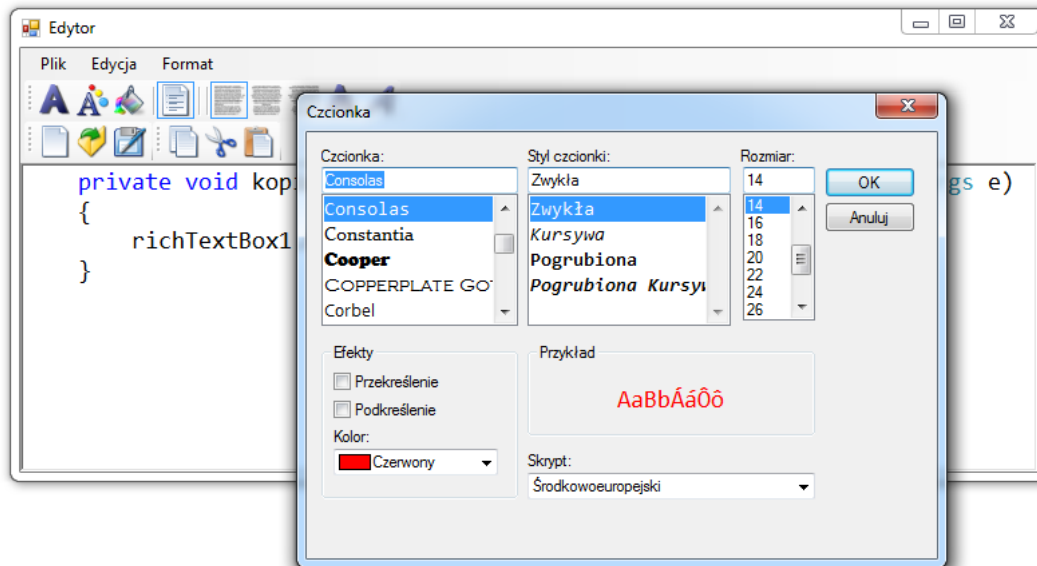
- Zmienić właściwość *Filter* komponentów *SaveFileDialog* i *OpenFileDialog*, dodając drugi typ plików, tj. pliki tekstowe:
 - **Filter** – pliki RTF ("*.rtf"), pliki tekstowe ("*.txt"),
 - **DefaultExtension** – ".rtf" (kropka rtf)
- Zmienić kod funkcji obsługi zdarzeń dla poleceń *Zapisz jako* oraz *Otwórz*, aby metody *SaveFile* i *LoadFile* używały odpowiedniego typu strumienia – *RichText* lub *PlainText*

9. ** Funkcja *Zapisz*

- W przestrzeni między funkcjami obsługi zdarzeń, najlepiej na początku klasy *Form1*, dodać deklarację zmiennej typu *String* na nazwę pliku o początkowej wartości w postaci pustego łańcucha znaków, np.:
`String fileName = "";`
- Wartość tej zmiennej należy aktualizować przy każdej operacji *Nowy*, *Zapisz jako* oraz *Otwórz*; Aby sprawdzić skuteczność tych działań, można dodawać nazwę pliku do tytułu aplikacji (właściwość *Text* formularza), np.:
`fileName = openFileDialog1.FileName;`
`this.Text = "Edytor " + fileName;`
- Utworzyć funkcję obsługi zdarzenia dla polecenia menu *Zapisz*; Dodać kod, który sprawdzi wartość zmiennej *fileName* i zależnie od wyniku sprawdzenia zapisze plik pod znaną już nazwą albo wywołać działania dla polecenia *Zapisz jako*; Programowe „kliknięcie” pozycji menu realizuje metoda *PerformClick*, np.:
`zapiszJakoToolStripMenuItem.PerformClick();`
Ważne! Jeżeli użyto komponentu *RichTextBox* oraz jest już zrealizowane zadanie 8, to podczas zapisywania pliku o znanej nazwie należy sprawdzić tę nazwę pod kątem typu pliku i użyć odpowiedniego typu strumienia, analogicznie jak w zadaniu 8

10. * Jednoczesna zmiana czcionki i koloru

- We właściwościach komponentu *FontDialog* włączyć wybór koloru czcionki,
- Zmodyfikować kod funkcji obsługi zdarzenia zmiany czcionki tak, aby cechy czcionki i kolor tekstu były zmieniane jednocześnie. Nie należy przy tym usuwać ani zmieniać istniejącej funkcji zmiany koloru tekstu, jedno drugiemu nie przeszkadza.



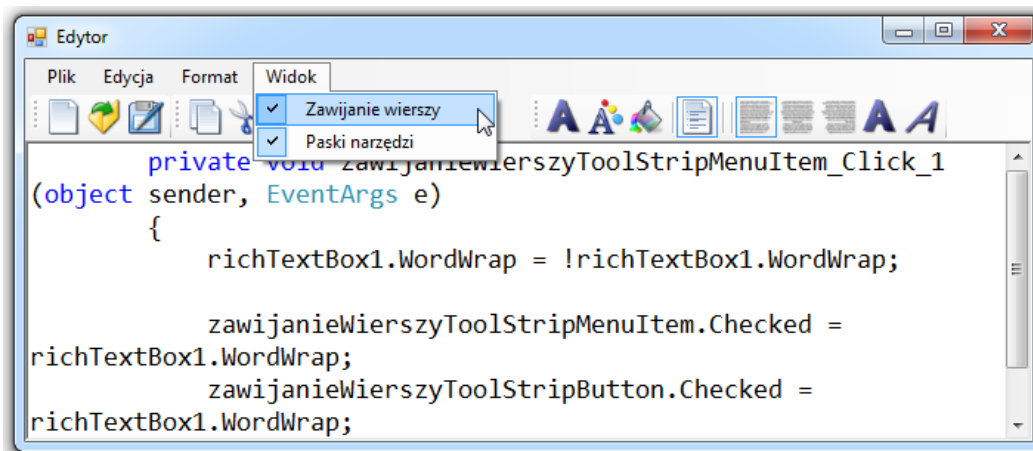
Ustawienia aplikacji

11. Ukrywanie pasków narzędzi

- Element *Paski narzędzi* z menu *Widok* przekształcić w *CheckBox* – ustawić właściwości *Check* oraz *CheckOnClick* na *true*,
- Utworzyć funkcję obsługi zdarzenia *Click* dla tego polecenia. Funkcja ta zostanie wywołana po zmianie wartości właściwości *Checked* – korzystając z tego faktu, wpisać instrukcję zmieniającą właściwość *Visible* wszystkich pasków narzędzi

12. * Zawijanie wierszy

- Włączyć symbol zaznaczenia elementu *Zawijanie wierszy* z menu *Widok* – ustawić właściwość *Check* na *true*, utworzyć funkcję obsługi jego zdarzenia *Click*,
- Dodać przycisk zawijania wierszy do paska narzędzi, ustawić jego właściwości *Check* na *true*, następnie powiązać przycisk z funkcją obsługi zdarzenia dla menu.
Ważne! Oba komponenty muszą mieć właściwość *CheckOnClick* ustawioną na *false*. Nie można polegać na automatycznej zmianie zaznaczenia, ponieważ zmiana nastąpiłaby tylko w jednym, klikniętym komponencie, a powinna nastąpić w obu.
- W funkcji obsługi zdarzenia zmienić wartość właściwości *WordWrap* na przeciwną, następnie zgodnie z jej wartością ustawić właściwość *Check* w menu i przycisku.
Ważne! W edytorze wykorzystującym komponent *TextBox* należy dodatkowo zmienić właściwość *ScrollBars* – tylko pionowy, gdy zawijanie wierszy jest włączone, albo oba w przeciwnym przypadku. Komponent *RichTextBox* nie wymaga takiej zmiany.



13. ** Wyrównanie tekstu

- Dodać przyciski wyrównania tekstu (do lewej, do środka, do prawej) do paska narzędzi, ustawić właściwość *Check* pierwszego z tych przycisków na *true*,
Ważne! Wszystkie te przyciski muszą mieć właściwość *CheckOnClick* ustawioną na *false*, ponieważ mają się zachowywać jak *RadioButton*, a nie jak *CheckBox*.
- Utworzyć funkcje obsługi zdarzenia *Click* dla przycisków wyrównania tekstu. W każdej z nich należy zmieniać wyrównanie tekstu edytora, ale też wartość właściwości *Check* przycisków – klikniętego na *true*, pozostałych dwóch na *false*.
Ważne! W przypadku komponentu *TextBox* należy korzystać z właściwości *TextAlign*, natomiast w przypadku *RichTextBox* – *SelectionAlignment*.

14. Zapisywanie podstawowych ustawień

- Otworzyć okno właściwości projektu i na zakładce *Settings* dodać nazwy, typy i wartości podstawowych ustawień aplikacji: rozmiar i położenie okna oraz czcionka, kolor tekstu i kolor tła edytora
Ważne! Jeżeli użyto komponentu *RichTextBox*, należy pominąć czcionki i kolory, ponieważ i tak zostanie to zignorowane podczas wczytywania ustawień. Zamiast tego można zmienić domyślną czcionkę i kolory, ale we właściwościach komponentu
- Utworzyć funkcję obsługi zdarzenia *Load* formularza, dodać kod który skopiuje wartości ustawień do właściwości formularza i komponentów edytora
- Utworzyć funkcję obsługi zdarzenia *FormClosed* formularza, dodać kod który skopiuje właściwości formularza i komponentów edytora do obiektu ustawień
Ważne! Należy pamiętać o wymuszeniu zapisu ustawień, do czego służy metoda *Save* obiektu ustawień *Properties.Settings.Default*

15. * Zapisywanie dodatkowych ustawień

- Na zakładce *Settings* właściwości projektu dodać ustawienia opcji programu –paski narzędzi i zawijanie wierszy, oba typu *bool*, o wartości domyślnej *true*,
- Uzupełnić kod funkcji obsługi zdarzenia *FormClosed* formularza, aby opcje programu były zapamiętywane – wystarczy skopiować wartości właściwości *Visible* paska narzędzi i *WordWrap* komponentu edytora do ustawień z poprzedniego kroku
- Uzupełnić kod funkcji obsługi zdarzenia *Load* formularza, aby opcje programu były odtwarzane podczas jego uruchamiania – w tym przypadku nie wystarczy skopiować wartości ustawień do właściwości *Visible* paska narzędzi i *WordWrap* komponentu edytora, należy dodatkowo ustawić wartość właściwości *Checked* elementów menu oraz przycisku paska narzędzi

16. *** Przyciski *Bold* oraz *Italic* – komponent *TextBox*

- Do paska narzędzi dodać przyciski *Bold* oraz *Italic*, ustawić ich właściwość *CheckOnClick* na *true* oraz utworzyć dla nich funkcje obsługi zdarzeń, zmieniające adekwatnie styl czcionki (wskazówki są w punkcie *Zmiana cech czcionki*),
- Uzupełnić kod obsługi zdarzenia dla polecenia menu *Czcionka*, aby przyciski stylu czcionki były odpowiednio ustawiane po zmianie czcionki; Przydadzą się właściwości *Bold* oraz *Italic* klasy *Font* (obie tylko do odczytu)
- Uzupełnić kod funkcji obsługi zdarzenia *Load* formularza, aby przyciski stylu czcionki były odpowiednio ustawiane podczas uruchamiania programu

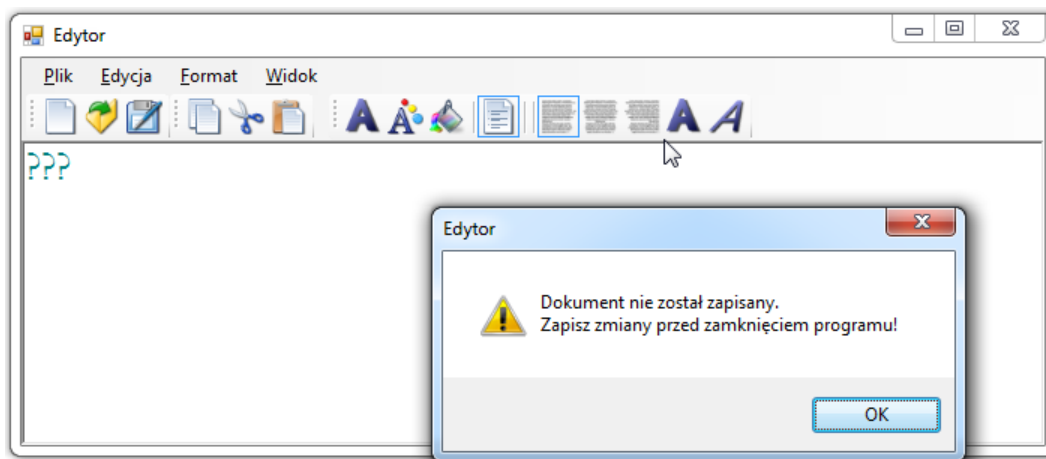
17. *** Przyciski *Bold* oraz *Italic* – komponent *RichTextBox*

- Do paska narzędzi dodać przyciski *Bold* oraz *Italic*, ustawić ich właściwość *CheckOnClick* na *true* oraz utworzyć dla nich funkcje obsługi zdarzeń, zmieniające adekwatnie styl czcionki obszaru zaznaczenia, *SelectionFont*, (wskazówki są w punkcie *Zmiana cech czcionki*),
- Uzupełnić kod obsługi zdarzenia dla polecenia menu *Czcionka*, aby przyciski stylu czcionki były odpowiednio ustawiane po zmianie czcionki; Przydadzą się właściwości *Bold* oraz *Italic* klasy *Font* (obie typu *Boolean*, tylko do odczytu),
- Utworzyć funkcję obsługi zdarzenia *SelectionChanged* komponentu *RichTextBox*, dodać kod zmieniający stan przycisków stylu czcionki odpowiednio do cech czcionki obszaru zaznaczenia, *SelectionFont*, kiedy kursor zostanie przeniesiony do inaczej sformatowanego fragmentu tekstu

Blokada zamknięcia programu

18. ** Blokada zamknięcia programu – ćwierćinteligentna

- We wszystkich miejscach programu, gdzie edytowany dokument staje się „niezmodyfikowany”, dodać instrukcję resetowania właściwości *Modified* komponentu edytora, *TextBox* lub *RichTextBox*:
 - po czytaniu pliku
 - po zapisaniu pliku
 - po utworzeniu nowego dokumentu
 - po wczytaniu ustawień programu, o ile zmieniana jest czcionka
- Utworzyć funkcję obsługi zdarzenia *FormClosing* formularza. Funkcja powinna sprawdzać wartość właściwości *Modified* komponentu edytora – jeżeli dokument jest zmodyfikowany, należy wyświetlić komunikat (tj. okno dialogowe, przy pomocy klasy *MessageBox*, z jednym przyciskiem, *Ok*) i zablokować zamknięcie programu,
- Podobną funkcjonalność należy dodać do obsługi poleceń menu *Nowy* i *Otwórz*
Ważne! Można nieco ułatwić sobie zadanie, stosując strategię ucieczki od komplikacji



19. *** Blokada zamknięcia programu – półinteligentna

- We wszystkich miejscach programu, gdzie edytowany dokument staje się „niezmodyfikowany”, dodać instrukcję resetowania właściwości *Modified* komponentu edytora, *TextBox* lub *RichTextBox*:
 - po czytaniu pliku
 - po zapisaniu pliku
 - po utworzeniu nowego dokumentu
 - po wczytaniu ustawień programu, o ile zmieniana jest czcionka
- Utworzyć funkcję obsługi zdarzenia *FormClosing* formularza. Funkcja powinna sprawdzać wartość właściwości *Modified* komponentu edytora – jeżeli dokument jest zmodyfikowany, należy wyświetlić okno dialogowe, przy pomocy klasy *MessageBox*, dając dwie możliwości wyboru – *Tak* i *Nie*. Jeżeli odpowiedź będzie *Tak*, należy zablokować zamknięcie programu, jeżeli *Nie*, aplikacja powinna zostać zamknięta
- Podobną funkcjonalność należy dodać do obsługi poleceń menu *Nowy* i *Otwórz*
Ważne! Można nieco ułatwić sobie zadanie, stosując strategię ucieczki od komplikacji