ACTA UNIVERSITATIS LODZIENSIS
FOLIA OECONOMICA 211, 2007

*Rafał Wziątek**

OBIEKTOWA WALIDACJA DANYCH

1. WPROWADZENIE

Podczas projektowania aplikacji, wszędzie tam, gdzie zachodzi konieczność interakcji z użytkownikiem w celu pobrania od niego danych, pojawia się problem sprawdzenia ich poprawności. Szczegółowość sprawdzania danych na formularzu internetowym czy formatce ekranowej zależy od potrzeb i konkretnej sytuacji. Niejednokrotnie wystarczające jest stwierdzenie, czy dane pole nie jest puste (np. w przypadku pola do wprowadzania imienia lub nazwiska), a czasami zachodzi konieczność zbadania wpisu również pod względem jego poprawności (np. pole NIP bądź PESEL).

Walidacja danych może również wymagać odwołania się do bazy danych, usług sieciowych oraz innych, bardziej zaawansowanych mechanizmów. Często warunki sprawdzające łączy się w wyrażenia bardziej złożone, co czyni aplikację trudniejszą do analizy oraz stwarza niebezpieczeństwo popełnienia błędów.

Walidacja danych jest czasami traktowana jako element poboczny w projekcie. Niejednokrotnie dodaje się ją w końcowej fazie implementacji, jako „zło konieczne". Być może tworzenie mechanizmów walidujących dane nie jest nad wyraz ekscytującym zajęciem, zdecydowanie jednak wpływa na jakość oprogramowania. Możliwość wprowadzania błędnych danych lub niekontrolowanie wpisów w polach obowiązkowych skutkują błędnym działaniem systemu. Nie bez znaczenia jest też złe wrażenie użytkownika, dla którego działanie interfejsu aplikacji oraz jego wygląd jest miarą jakości oprogramowania[1].

Niniejszy tekst jest próbą spojrzenia na różne rodzaje i techniki walidacji danych. W części praktycznej przedstawione zostały w nim te najprostsze

* Mgr, asystent w Zakładzie Informatyki Ekonomicznej Uniwersytetu Łódzkiego.

[1] J. Cogswell, *Tworzenie użytecznego oprogramowania*, Wydawnictwo Mikom, Warszawa, luty 2005, s. 76.

oraz te bardziej zaawansowane. Zestawienie to oczywiście nie wyczerpuje w pełni tematu, gdyż możliwych sposobów walidacji jest nader wiele. Wybór danych metod lub metody walidacji zależy od konkretnego przypadku oraz upodobań projektanta aplikacji czy programisty. Przykłady napisane zostały w języku C#, ale analiza kodu w przypadku osób nie programujących w tym języku nie powinna stanowić problemu.

2. WALIDACJA DANYCH

Walidacja danych jest procesem mającym na celu zapewnienie aplikacji otrzymania poprawnych i dokładnych danych[2]. Potrzeba dokonania walidacji zachodzi prawie w każdym przypadku, gdy dane przyjmowane są bezpośrednio od użytkownika (np. za pomocą formularza) lub pochodzą z innych podsystemów (modułów bądź funkcji) aplikacji. Z tego punktu widzenia walidację danych możemy podzielić na związaną:

- z warstwą prezentacji,
- z warstwą logiki przetwarzania (reguły i logika biznesu),
- z bazą danych.

Częste walidacje wszelkiego rodzaju są charakterystyczną cechą techniki programowania defensywnego[3].

Poza tym, wyróżnić można kilka typów walidacji danych:

- sprawdzanie poprawności typów danych,
- sprawdzanie zakresów,
- sprawdzanie poprawności kodów/stałych,
- sprawdzanie złożonych kryteriów.

2.1. Sprawdzanie poprawności typów danych

W sytuacjach, gdy użytkownik wprowadził dane, zachodzi niejednokrotnie konieczność wykonania sprawdzenia poprawności typu danych. Jest to szczególnie istotne, gdy wprowadzone dane mają mieć charakter liczbowy. Walidacja w tym przypadku sprowadzać się będzie do stwierdzenia, czy podana wartość nosi znamiona wielkości liczbowej. Sprawdzenie takie można zrealizować np. w następujący, bardzo prosty sposób:

[2] http://msdn.microsoft.com/library/default.asp?url = /library/en-us/vsent7/html/vxcondatavalidation.asp, 2.02.2006 r.

[3] S. McConnell, *Programista doskonały*, Oficyna Wydawnicza LTP Sp. z o. o., Warszawa 2003, s. 110.

```
public bool IntegerValidator(string s)
        {
            try
            {
                int.Parse(s);
            }
            catch (System.FormatException fe)
            {
                return false;
            }
            return true;
        }
[...]
string sdata = „nieprawidłowy wpis";
if(!IntegerValidator(sdata))
        MessageBox.Show(„Podana wartość nie jest liczbą całkowitą!");
```

Przykład 1. Sprawdzenie poprawności typu danych

2.2. Sprawdzanie zakresów

Sprawdzanie zakresów dla wartości numerycznych występuje w większości przypadków, gdy interakcja z użytkownikiem sprowadza się do podania przez niego danych liczbowych. Czynnością poprzedzającą sprawdzenie zakresów może być stwierdzenie, czy podane wartości (wartość) są liczbami i czy nie występują w nich znaki niedozwolone.

Zakresy sprawdza się dla ustalonych wartości minimalnych i maksymalnych. W przypadku wartości nienumerycznych (np. znaków będących literami alfabetu), sprawdzenie polega na stwierdzeniu, czy kod podanego znaku występuje pomiędzy wartościami minimalną i maksymalną.

Do tej kategorii można też zaliczyć wszelkie sytuacje, gdy zachodzi potrzeba stwierdzenia, czy dane pole formularza (zawartość kontrolki) jest wypełnione.

2.3. Sprawdzanie poprawności kodów/stałych

Wszędzie tam, gdzie pojawia się konieczność wprowadzenia wszelkiego rodzaju kodów (np. pocztowych), symboli towarów, numerów referencyjnych czy innych, niezbędne jest zweryfikowanie ich poprawności. Sprawdzanie takie jest bardziej złożone i może polegać na:

– weryfikacji na podstawie specyfiki samego kodu (np. numer referencyjny towaru składa się z numeru oraz sumy kontrolnej),

– weryfikacji za pomocą odwołania się do bazy danych (np. sprawdzenie występowania danego towaru w rejestrze).

Dla drugiego przypadku odniesieniem nie zawsze musi być baza danych. Czasami, gdy zbiór danych jest mały, może to być plik lub nawet tablica z wartościami, przechowywana w pamięci.

Sprawdzanie poprawności kodów pod względem stopnia złożoności jest różne dla konkretnych przypadków. Czasami wystarczy przeszukać mały zbiór danych (np. w przypadku telefonicznych numerów kierunkowych miast), a czasami zbiór może być bardzo duży (np. baza danych ze wszystkimi adresami posesji w danym mieście czy województwie). W zależności od specyfiki kodu może zajść potrzeba skorzystania z właściwego algorytmu, służącego do jego weryfikacji. Jako przykład przedstawić można kwestię walidacji Numeru Identyfikacji Podatkowej (NIP).

Numer Identyfikacji Podatkowej zbudowany jest z dziesięciu cyfr. Pierwsze trzy cyfry oznaczają kod Urzędu Skarbowego, który wystawił numer. Wśród tych cyfr nie mogą występować zera, maksymalna dozwolona liczba z nich utworzona to 998, a najmniejsza to 111. Ostatnia, dziesiąta cyfra numeru jest cyfrą kontrolną, wyznaczoną za pomocą następującego algorytmu[4]:

1. Każda z cyfr NIP mnożona jest przez odpowiednią wagę.
2. Suma powyższych iloczynów poddawana jest operacji modulo przez 11. Otrzymana liczba stanowi cyfrę kontrolną NIP.

Poniżej przedstawiono przykładową implementację funkcji sprawdzającej NIP.

```
public bool NIPValidator(string nip)
      {
      //tablica wag, służąca do ustalenia cyfry kontrolnej
      byte[...] wagi  =  6,5,7,2,3,4,5,6,7};

      //usunięcie z numeru ewentualnych kresek („–")
      string str_nip = nip;
      while(str_nip.IndexOf('–')! = –1)
          str_nip = str_nip.Remove(str_nip.IndexOf('–'),1);

      //sprawdzenie, czy numer jest dziesięciocyfrowy
      if(str_nip.Length! = 10)
      {
          MessageBox.Show(„Numer nie jest dziesięciocyfrowy!");
```

[4] Oprac. własne na podstawie: http://pl.wikipedia.org/wiki/Nip, 2.02.2006 r.

```
    return false;
}

//sprawdzenie czy numer składa się z właściwych znaków
int i = 0;
while(i < str_nip.Length)
{
    if(!(str_nip[i] > = '0'&&str_nip[i] < = '9'))
    {
        MessageBox.Show(„W numerze występują niewłaściwe
        znaki!");
        return false;
    }
    i + +;
}

//sprawdzenie, czy numer urzędu nie jest równy 999
if(str_nip.Substring(0,3) = = „999")
{
    MessageBox.Show("Pierwszy człon numeru nie może być równy
    999!");
    return false;
{
//sprawdzenie czy w numerze urzędu nie występują zera
if(str_nip.Substring(0,3).IndexOf(„0")! = -1)
{
MessageBox.Show(„Wśród pierwszych trzech cyfr numeru nie może
być zer!");
return false;
}
//ustalanie cyfry kontrolnej
int tmp = 0;
for(i = 0; i < 9; i + +)
    tmp + = (int)char.GetNumericValue(str_nip[i])*wagi[i];

int cyfra_kontrolna = tmp%11;

//sprawdzenie, czy ostatnia cyfra nipu jest poprawną cyfrą kontrolną
if(char.GetNumericValue(str_nip[9])! = cyfra_kontrolna)
{
    MessageBox.Show(„Nieprawidłowa cyfra kontrolna!");
    return false;
```

```
        }
        return true;

        }
[...]

string s = „727-229-02-63";
if(NIPValidator(s))
        MessageBox.Show(„NIP jest poprawny");
else
        MessageBox.Show(„NIP jest błędny");
```

Przykład 2. Walidacja Numeru Identyfikacji Podatkowej

2.4. Sprawdzanie złożonych kryteriów

Ten rodzaj sprawdzania jest podobny do opisanego w punkcie 2.3, z tą różnicą, że tutaj dużo większy nacisk położony jest na logikę biznesową oraz informacje zawarte w bazie danych. Często też zachodzi w tym przypadku konieczność sprawdzenia wzajemnych relacji pomiędzy wprowadzonymi danymi. Walidacja tego rodzaju może wykorzystywać i łączyć różne rozwiązania technologiczne (np. usługi sieciowe, procedury składowe bazy danych).

Przykładem może tu być system służący do konfiguracji elementów sieci komputerowej. W takim przypadku wybranie przez użytkownika np. określonego systemu listw i kanałów kablowych determinuje dalsze wybory dotyczące sposobu montażu złącz, akcesorii i narzędzi. Z kolei wybór konkretnego, aktywnego elementu sieci danego producenta może wpłynąć na dalsze wybory użytkownika tak, aby spełnione zostały wymogi kompatybilności.

3. METODY WALIDACJI

W modelu „cienkiego" i „grubego" klienta metody walidacji danych są odmienne, szczególnie jeżeli chodzi o walidację związaną z warstwą prezentacji. Warstwa ta, w przypadku modelu „cienkiego klienta" (np. aplikacje internetowe) ma swoją specyfikę, która wymusza zastosowanie nieco innych metod walidacji danych.

3.1. Metody walidacji w modelu „cienkiego klienta"

Model „cienkiego klienta" charakteryzuje się małą mocą przetwarzania oraz ograniczeniami związanymi z dostępnymi zasobami. Wpływa to bezpośrednio na sposób walidowania danych związanych z warstwą prezentacji.

Walidację danych w modelu „cienkiego klienta" można podzielić na dwa rodzaje[5]:

- walidację po stronie klienta,
- walidację po stronie serwera.

3.1.1. Walidacja po stronie klienta

Po stronie klienta walidacja danych najczęściej sprowadza się do prostych czynności mających na celu sprawdzenie, czy:

- formularz został wypełniony,
- podano wartości z odpowiedniego przedziału,
- proponowane, nowo wprowadzone hasło spełnia kryteria odnośnie do wymaganej liczby znaków i ich rodzaju,
- adres e-mail zgodny jest z szablonem,
- w danym polu nie występują niedozwolone znaki,
- itp.

Bardziej złożone sprawdzenia są trudne lub wręcz niemożliwe do wykonania. Czasem jednak istnieje możliwość przeprowadzenia złożonej walidacji, lecz wtedy należy rozpatrzyć wszystkie za i przeciw, które przemawiają za tego typu rozwiązaniem. Należy odpowiedzieć sobie wcześniej na następujące pytania:

- czy ilość danych, która jest potrzebna do zrealizowania walidacji nie jest zbyt duża i czy nie wydłuży w znaczny sposób czasu załadowania się strony internetowej?
- czy złożoność obliczeniowa zadania nie obciąży nadto systemu klienta (przeglądarki internetowej)?

Walidacja danych po stronie klienta jest realizowana zwykle za pomocą języka skryptowego (np. JavaScript). Kod sprawdzający osadzony jest w widoku klienta i tam też jest wykonywany. Mimo całej wygody, związanej z możliwością dokonania prostych walidacji, należy pamiętać o tym, że interpreter języków skryptowych po stronie klienta może zostać wyłączony (np. ze względów bezpieczeństwa) lub kod, dostępny przecież w postaci jawnej, może być zmodyfikowany, np. w celu zakłócenia bądź uszkodzenia danych aplikacji działającej na serwerze. W związku z tym należy przeprowadzać podwójną walidację, wpierw po stronie klienta, następnie jeszcze

[5] D. Alur, J. Crupi, D. Malks, *Core J2EE Wzorce projektowe*, Wydawnictwo Helion, Gliwice 2004, s. 40–41.

raz po stronie serwera. Nasuwa się w takim przypadku pytanie, po co w ogóle prowadzić walidację po stronie klienta? Otóż interpreter języka skryptowego, działający w przeglądarce, jest w stanie wykonać kod bez odsyłania strony z powrotem do serwera w celu sprawdzenia. Dzięki temu możliwe jest natychmiastowe reagowanie na nieprawidłowe dane.

Narzędzia służące do tworzenia aplikacji internetowych mają często już gotowe mechanizmy do przeprowadzania walidacji po stronie klienta. W przypadku ASP.NET do dyspozycji jest kilka kontrolek, które są w stanie przeprowadzić taką walidację. Kontrolki te dokonają wspomnianej walidacji tylko wtedy, gdy przeglądarka użytkownika obsługuje DHTML. Należy też wspomnieć, że kontrolki walidacyjne w ASP.NET zawsze wykonują walidację po stronie serwera, natomiast walidacja po stronie klienta wykonywana jest jedynie w przypadku, gdy przeglądarka spełnia wyżej wymienione kryterium.

Kontrolki te działają w taki sposób, że gdy na formularzu któraś z nich wykryje błąd, natychmiast przerywane jest odsyłanie strony do serwera i wyświetlany jest komunikat o błędzie. Do dyspozycji w ASP.NET są następujące kontrolki walidacyjne[6]:

– RequiredFieldValidator – sprawdza, czy użytkownik wypełnił dane pole;

– CompareValidator – porównuje wpis w danym polu ze stałą lub z wartością z innej kontrolki, stosując przy tym całą gamę operatorów relacyjnych. Jednym z jej zastosowań jest sprawdzanie, czy użytkownik podał dwa razy identyczne wpisy podczas zmiany bądź nadawania hasła;

– RangeValidator – służy sprawdzeniu, czy wartość w danym polu jest z zadanego przedziału. Przedział może być określony dla wartości liczbowych, znaków oraz dat;

– RegularExpressionValidator – sprawdza, czy podany wpis spełnia założenia zdefiniowane w wyrażeniu regularnym. Umożliwia to walidację adresów e-mail, poprawności zapisu kodów pocztowych, numerów telefonu, numerów kart kredytowych oraz innych;

– CustomValidator – umożliwia stworzenie własnej reguły walidacyjnej;

– ValidatorSummary – udostępnia zbiorczą informację o wykrytych przez wszystkie kontrolki walidacyjne błędach znajdujących się na formularzu.

3.1.2. Walidacja po stronie serwera

Strona internetowa, zawierająca formularz do sprawdzenia, odsyłana jest do serwera. Tam też raz jeszcze powinny być sprawdzone pola, które rzekomo zostały zwalidowane po stronie klienta oraz dokonać można bardziej złożonej

[6] D. J. Reilly, *Designing Microsoft ASP.NET Applications*, Microsoft Press, Redmond, Washington, USA, s. 118–148.

oceny poprawności. W tej sytuacji nie ma już specjalnych przeciwwskazań związanych z ograniczeniami w zasobach, co do sposobu przeprowadzania walidacji. Jeżeli wykryte zostaną błędy, strona jest odsyłana wraz z informacją o nich z powrotem do klienta (tzw. *postback*).

3.2. Walidacja w modelu „grubego klienta"

Walidacja w aplikacjach modelu „grubego klienta" w wielu wypadkach korzystać może z podobnych technik, jak sprawdzanie po stronie serwera, omówione w punkcie 3.1.2. Oczywiście w tym przypadku warstwy prezentacji nie stanowi przeglądarka internetowa.

Warstwa prezentacji może w tym modelu przejąć część nawet bardziej złożonych zadań związanych z walidacją. Wbudowane mechanizmy wspierające walidację, jak np. zdarzenie *Validating* kontrolek Windows.Forms, ułatwiają proste kontrolowanie wpisów.

Pamiętać należy jednak, aby nie umieszczać zbyt dużej części kodu w warstwie prezentacji. Zgodnie z trójwarstwową architekturą aplikacji, warstwa prezentacji powinna zawierać minimum kodu związanego z przetwarzaniem logiki aplikacji[7]. Umieszczanie kodu sprawdzającego bezpośrednio w warstwie prezentacji nie będzie sprzyjać łatwości późniejszej konserwacji aplikacji, jej skalowalności oraz może być przyczyną powstawania kłopotliwych, trudnych do usunięcia i zlokalizowania błędów.

Przykłady pierwszy i drugi z poprzedniej części tekstu stanowiły proste rozwiązanie problemu walidacji. Dysponując powyższymi metodami, można dokonywać łatwej walidacji danych zarówno w obsłudze zdarzenia *Validating*, jak i np. w obsłudze zdarzenia przyciśnięcia przycisku zamykającego okno.

W celu łatwiejszego wykorzystania metod walidujących w aplikacji można zebrać je w klasie narzędziowej i uczynić je statycznymi. Późniejsze odwoływanie się do tych metod zaoszczędzi czas potrzebny na stworzenie aplikacji oraz pozwoli uniknąć przynajmniej częściowego powielania kodu. Nie rozwiązuje to jednak problemu łatwego ich użycia w przypadku, gdy dane pole (lub kilka pól, np. kontrolki do edycji tekstu) trzeba sprawdzić pod kątem kilku warunków. W dalszej części tekstu przedstawiona zostanie propozycja obiektowej metody, pozwalającej na elastyczne używanie i łączenie walidatorów.

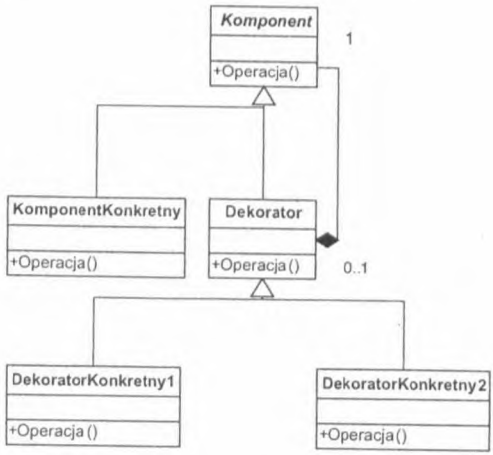
[7] C. Larman, *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design*, Prentice Hall PTR, NJ, USA 1998, s. 273–275.

4. WALIDACJA OBIEKTOWA

Poniższy przykład ilustruje jedno z możliwych obiektowych rozwiązań problemu walidacji zawartości kontrolek, będących częścią interfejsu użytkownika. Idea rozwiązania bazuje na enkapsulacji pojedynczych walidacji w postaci obiektów. Dokonuje się złożenia konkretnego walidatora z kontrolką w celu uzyskania żądanego sposobu sprawdzania. Gdy występuje konieczność przeprowadzenia kilku walidacji na jednej kontrolce, składa się obiekty walidatorów ze sobą. W momencie, gdy zachodzi potrzeba sprawdzenia warunków, kontrolka generuje zdarzenie *Validating*, a poszczególne walidatory dokonują sprawdzenia.

4.1. Wzorzec dekoratora

Zgodnie z definicją wzorzec dekoratora „dynamicznie dołącza do obiektów dodatkowe zobowiązania. Dekorator zapewnia elastyczną alternatywę dla tworzenia podklas w celu rozszerzania funkcjonalności"[8].



Rys. 1. Diagram wzorca dekoratora
Źródło: oprac. własne

Problem, który rozwiązuje dekorator polega na tym, że dekorowany obiekt ma już swoją funkcjonalność, a pojawia się potrzeba, aby rozszerzyć jej zakres o operacje wykonywane przed (lub po) dotychczasową działalnością[9].

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Wzorce Projektowe*, Wydawnictwa Naukowo-Techniczne, Warszawa 2005, s. 171.

[9] A. Shalloway, J. R. Trott, *Projektowanie zorientowane obiektowo. Wzorce projektowe*, Wydawnictwo Helion, Gliwice 2002, s. 198.

Zastosowanie wzorca stanowi też rozwiązanie problemu konfiguracji (w poniższym przypadku – sposobów i warunków walidacji) w czasie wykonania programu.

4.2. Przykład walidacji obiektowej

Przy tworzeniu przykładu skorzystano ze wspomnianego wyżej wzorca dekoratora. Różnica między „klasyczną" implementacją tego wzorca a przedstawioną poniżej polega na tym, że obiekty dekoratorów nie wywołują metody sprawdzającej kolejno od najbardziej zewnętrznego dekoratora do najbardziej wewnętrznego, aż po metodę dekorowanego komponentu. Tutaj kolejność jest odwrotna: to komponent dekorowany inicjuje wywołania metod dekoratorów poprzez wygenerowanie zdarzenia. Zdarzenia te obsługiwane są przez coraz to bardziej zewnętrzne, opakowujące obiekty dekoratorów. Każdy dekorator, po obsłużeniu we właściwy dla siebie sposób zdarzenia, sam generuje zdarzenie tego samego typu, jakie obsługiwał, wywołując tym samym pośrednio metodę walidującą w obiekcie go opakowującym.

Na potrzeby przykładu stworzono trzy klasy dekoratorów konkretnych, wywodzących się ze wspólnej klasy Validator. Dekoratory te to:

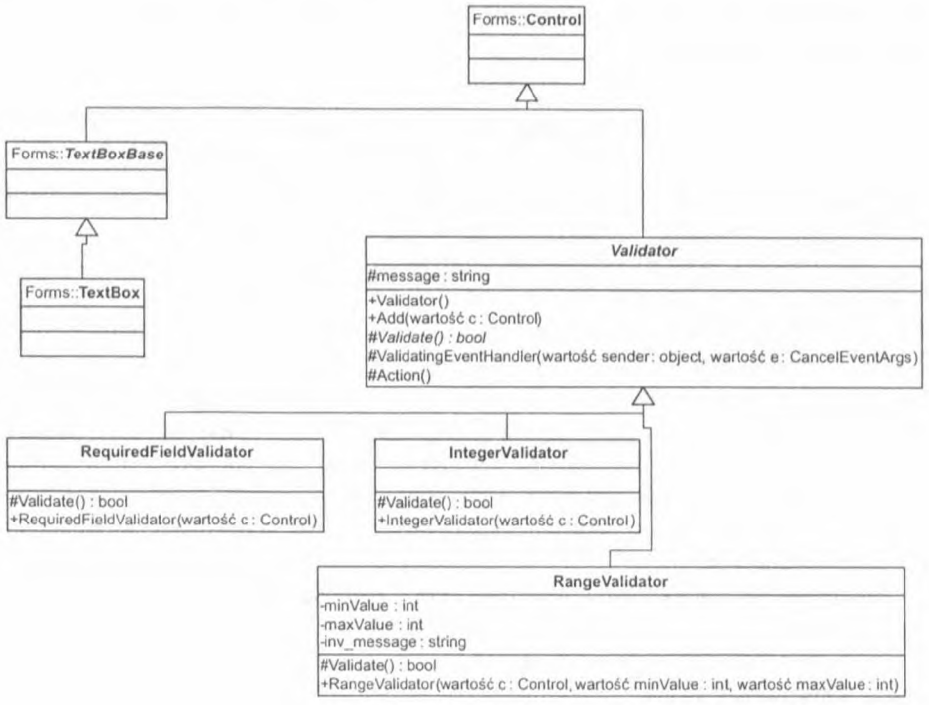
– RequiredFieldValidator (sprawdza, czy komponent ma ustawioną nie pustą właściwość Text);

– IntegerValidator (sprawdza, czy komponent jest liczbą całkowitą);

– RangeValidator (sprawdza, czy wartość przechowywana przez komponent mieści się w określonym przedziale).

Składanie, a właściwie powiązanie między obiektami w tym przykładzie, osiągnięto nie poprzez deklarowanie referencji do obiektu dekorowanego (komponentu) w klasie dekoratora, ale poprzez wykorzystanie delegacji w języku C#. Ciekawą konsekwencją tego rozwiązania jest to, że daną instancję dekoratora można wykorzystać do sprawdzania kilku komponentów. W związku z tym, można za pomocą jednej (lub kilku złożonych) instancji walidatora obsłużyć np. wszystkie kontrolki, które wymagają podobnych walidacji. Co więcej, do łańcucha walidatorów można „wpleść" kontrolkę w taki sposób, aby sprawdzana była tylko przez ich część. Dzieje się tak dlatego, że klasa Delegate w języku C# za pomocą funkcji Delagate.Combine() przyjmuje dwie delegacje i zwraca jedną, enkapsulującą obydwie. Dzięki temu delegacja może odwoływać się do więcej niż jednej funkcji i *de facto* obejmuje listę funkcji, które należy wykonać kolejno[10].

[10] E. Gunnerson, *Programowanie w języku C#*, Wydawnictwo Mikom, Warszawa, wrzesień 2001, s. 236.



Rys. 2. Diagram klas walidatorów
Źródło: oprac. własne

```
public abstract class Validator:System.Windows.Forms.Control
  {
      protected string message; //treść komunikatu o błędzie
      public Validator(){}

      public void Add(Control c)
      {
      c.Validating + = new
      System.ComponentModel.CancelEventHandler(ValidatingEventHandler);
      }

      //kryterium poprawności
      protected abstract bool Validate();

      //obsługa zdarzenia Validating
      protected void ValidatingEventHandler(object sender, System.Com-
      ponentModel.CancelEventArgs e)
```

```
        {
            this.Text = ((Control)sender).Text;
            if(!Validate())
            {
                Action();
                e.Cancel = true;
            }
            else
                this.OnValidating(e);
        {

        //akcja, która ma być wykonywana, gdy stwierdzony zostanie błąd
        protected virtual void Action()
        {
                MessageBox.Show(message);
        }
}

public class RequiredFieldValidator:Validator
{
        public RequiredFieldValidator(Control c)
        {
            message = „Brak wpisu!";
            Add(c);
        }
        protected override bool Validate()
        {
            if(Text  = = "")
            {
                return false;
            }
            return true;
        }
}

public class IntegerValidator:Validator
{
        public IntegerValidator(Control c)
        {
            message = „Podana wartość nie jest liczbą całkowitą!";
            Add(c);
        }
```

```
protected override bool Validate()
}
    try
    {
        int.Parse(Text);
    }
    catch
    {
        return false;
    }
        return true;
    }
}

public class RangeValidator:Validator
{
    private int minValue, maxValue;
    private string inv_message;
    public RangeValidator(Control c, int minValue, int maxValue)
{
    if(minValue > = maxValue)
        throw new System.ArgumentException(„Wartość minimalna
        nie może być większa od maksymalnej");
    inv_message = „Wartość powinna być z przedziału  < „ + min
    Value.ToString() + ";" + maxValue.ToString() + " > ";
    message = inv_message;
    this.minValue = minValue;
    this.maxValue = maxValue;
    Add(c);
}

protected override bool Validate()
{
    message = inv_message;
    int  v = 0;
    try
    {
        v = int.Parse(Text);
    }
    catch(System.FormatException fe)
    {
        message  =  fe.Message;
```

```
                return false;
            }
            if(!(v >= minValue && v <= maxValue))
            {
                return false;
            }
            return true;
        }
    }

[...]
//Poniższe TextBox-y mają być walidowane
private System.Windows.Forms.TextBox textBox1 = new TextBox();
private System.Windows.Forms.TextBox textBox2 = new TextBox();

//textBox1 ma być sprawdzany pod względem (dokładnie w takiej kolejności):
// 1. Czy jest wpisana do niego wartość?
// 2. Czy wpisana wartość jest liczbą całkowitą?
// 3. Czy wpisana wartość mieści się w zadanym przedziale?

RequiredFieldValidator requiredFieldValidator = new RequiredFieldVali-
dator(textBox1);
IntegerValidator integerValidator = new IntegerValidator(requiredField-
Validator);
RangeValidator rangeValidator = new RangeValidator(integerValidator,
0, 100);

//textBox2 będzie po poniższym zapisie sprawdzany przez już istniejące
instancje //walidatorów. Sprawdzany będzie, w kolejności:
//1. Czy wpisana wartość jest liczbą całkowitą?
//2. Czy wpisana wartość mieści się w zadanym przedziale?

integerValidator.Add(textBox2);
```

Przykład 3. Walidatory oraz ich wykorzystanie

5. PODSUMOWANIE

Dokładna walidacja danych jest nieodzownym elementem dobrze zaprojektowanej i wykonanej aplikacji. Odpowiednie zamodelowanie mechanizmów służących weryfikacji danych powoduje, że są one łatwe w użyciu oraz

elastyczne. Zaproponowany przez autora sposób obiektowej walidacji pozwala zmniejszyć nakład pracy związany z kontrolą poprawności danych. Rozwiązanie bazuje na wzorcu projektowym dekoratora, co czyni je również łatwiejszym do interpretacji i implementacji, a zastosowanie sprawdzonego mechanizmu wpływa na jego wartość praktyczną.

Rafał Wziątek

OBJECT-ORIENTED DATA VALIDATION

(Summary)

This article presents a wider view on data validation process. It shows theoretical background of problem and also gives a practical example of object-oriented approach to data validation.