Interfejs graficzny i diagramy sekwencji w generacji prototypu

Przemysław Biecek, Zbigniew Huzar
Wydziałowy Zakład Informatyki, Wydział Informatyki i Zarządzania, Politechnika Wrocławska
biecek@im.pwr.wroc.pl zhuzar@ci.pwr.wroc.pl

Czy [...] my stwarzamy formę, czy ona nas stwarza?
--Witold Gombrowicz

Streszczenie

Przedstawiono metodę automatycznego generowania prototypu projektowanego systemu. Prototyp jest generowany na podstawie opisu interfejsu graficznego oraz na podstawie opisu zachowania obserwowanego na tym interfejsie. Przedstawiono reguły, które na podstawie tych opisów pozwalają na automatyczne wytworzenie prototypu aplikacji.

1. Wstęp

Możliwość łatwego wytworzenia i przebadania prototypu systemu w możliwie wczesnych fazach procesu wytwarzania oprogramowania w istotny sposób wpływa na jakość końcowego produktu. Przygotowanie prototypu zaleca wiele metodyk, np. USDP [JBR 1999], a niektóre narzędzia wspomagające wytwarzanie oprogramowania wychodzą naprzeciw temu postulatowi umożliwiając projektowanie interfejsu graficznego.

Przez prototyp rozumie się aplikację imitującą projektowany system ze względu na jeden lub wiele jego aspektów. Najczęściej chodzi o analizę i weryfikację wymagań funkcjonalnych, czasem też prototyp może służyć analizie wymagań niefunkcjonalnych, na przykład wydajnościowych.

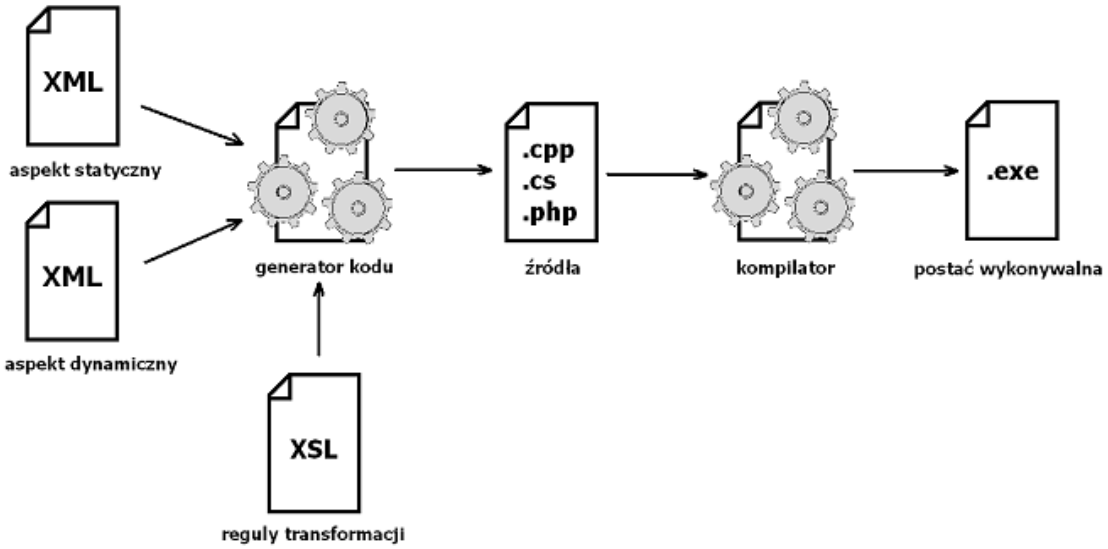
W rozdziale skupiamy się wyłącznie na wymaganiach funkcjonalnych. Zakłada się, że użytkownik ma kontakt z systemem poprzez interfejs graficzny aplikacji. Wymagania funkcjonalne są wyrażone przez interakcje użytkownika z systemem za pośrednictwem tego interfejsu.

Rozdział przedstawia metodę automatycznego generowania prototypu systemu, na podstawie wcześniej przygotowanych wymagań funkcjonalnych. Opis wymagań zawiera opis interfejsu graficznego w postaci diagramu klas oraz opisy interakcji w postaci zbioru diagramów sekwencji. Przedstawiona metoda generacji została wykorzystana w budowie prototypowego narzędzia, a następnie przetestowana na prostych przykładach studialnych.

W następnym podrozdziale wyjaśniono ideę generatora. Podrozdziały 3. i 4. zawierają formalne definicje składni opisu interfejsu (aspekt statyczny wymagań funkcjonalnych), diagramów sekwencji (aspekt dynamiczny wymagań funkcjonalnych). W podrozdziale 5. omówiono warunki spójności pomiędzy opisami obu aspektów. Podrozdział 6. przedstawia formalne reguły generacji wynikowego kodu prototypu. Ostatni podrozdział 7. stanowi podsumowanie rozdziału.

2. Struktura procesu generacji

Strukturę procesu generowania prototypu przedstawia rysunek 1. Ikony symbolizujące dokumenty z napisami to zbiory danych, pliki. Ikony symbolizujące dokument z kołami zębatymi to generatory, czyli aplikacje potrafiące na podstawie odpowiednich doku-mentów wejściowych wytworzyć dokument wyjściowy.



Rys. 1. Proces generowania prototypu

Zasadniczym elementem jest generator kodu, który na podstawie dokumentów tekstowych zapisanych w języku XML (*eXtended Markup Language*) [XML1998] produkuje program źródłowy w wybranym języku programowania.

Pierwszy dokument **aspekt statyczny** przedstawia wygląd graficzny okienek oraz kontrolek wraz z ich rozmieszczeniem i własnościami. Opisowi tego dokumentu jest poświęcony następny podrozdział.

Drugi dokument **aspekt dynamiczny** przedstawia zachowanie systemu. Zachowanie jest rozumiane jako zbiór sekwencji zmian zachodzących w graficznym wyglądzie aplikacji. Do opisu zachowania użyte zostały diagramy sekwencji zapisane jako dokument XML. Opis składni i semantyki tego dokumentu znajduje się w podrozdziale 4.

Oba dokumenty, stanowiąc wspólny opis wymagań funkcjonalnych, są powiązane węzłami spójności, których opis znajduje się w podrozdziale 5.

Dokument **reguły transformacji** jest również tekstowym, który zawiera szablony

generacji kodu zapisane w standardzie XSL *eXtensible Stylesheet Language*. Specyfikacja formatu XSL znajduje się w dokumencie [XSL1999]. Szablony w standardzie XSL mogą być wykonane dla różnych języków programowania. Wynikiem ich zaaplikowania do opisów aspektu statycznego i dynamicznego będzie kod źródłowy w wskazanym języku programowania przeznaczony na wskazaną platformę uruchomieniową.

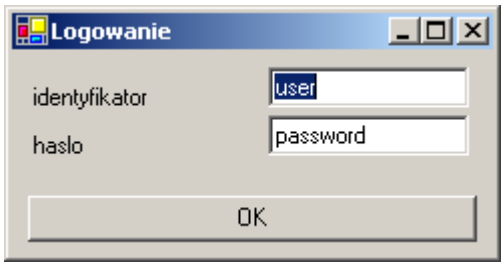
Dokumenty **aspekt statyczny** oraz **aspekt dynamiczny** stanowią dane wejściowe, **reguły transformacji** zawierają informację sterującą dla **generatora kodu**. Generatorem kodu może być dowolny parser potrafiący przetworzyć dokument XML z użyciem szablonów XSL.

Otrzymany kod źródłowy można jeszcze modyfikować lub, bez żadnej dodatkowej obróbki, przetworzyć kompilatorem otrzymując wykonywalną aplikację.

Przedstawiony schemat generacji aplikacji jest elastyczny, gdyż przy jednym ustalonym dokumencie **reguły transformacji** pozwala na przetworzenie dowolnego spójnego zestawu dokumentów opisujących statyczny i dynamiczny aspekt wymagań funkcjonalnych. Zmiana dokumentu **reguły transformacji** pozwala z kolei na generację aplikacji na odpowiednio wybraną platformę uruchomieniową.

3. Aspekt statyczny

Przez aspekt statyczny interfejsu użytkownika rozumie się informacje o reprezentacji graficznej interfejsu użytkownika. Taki opis można wyrazić w wielu postaciach.

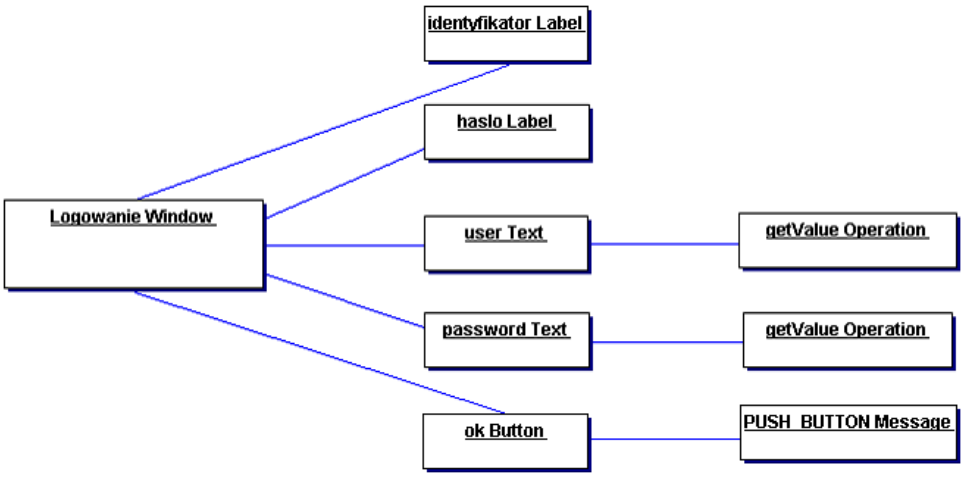


Rys. 2. Składnia konkretna aspektu statycznego

Rysunek 2. prezentuje przykładowe okienko aplikacji zapisane w konkretnej składni określonej dokumentem [WWW1999]. Opisem statycznego aspektu wymagań funkcjonalnych jest zbiór opisów wszystkich okien aplikacji.

Projekt interfejsu graficznego powstaje (zgodnie z zaleceniami USDP) już na etapie analizy wymagań. Projekt ten jest rozwijany równolegle do rozwoju pozostałej części aplikacji. W pracy zakłada się, że interfejs graficzny jest już zaprojektowany.

Z punktu widzenia procesu wytwarzania oprogramowania interesujące są dwie formy opisu aspektu statycznego. Pierwsza to opis wyrażony za pomocą diagramu obiektów.



Rys. 3. Instancja aspektu statycznego, zapis w postaci diagramu obiektów (dla czytelności nie zaznaczono wartości atrybutów dla poszczególnych obiektów)

Przykładowa postać takiego opisu znajduje się na rysunku 3. Zaletą tej formy opisu jest możliwość późniejszego jej wykorzystania przez narzędzia CASE (przykładowo do wytworzenia z niej szablonów klas docelowej aplikacji, lub w dalszym procesie rozwoju interfejsu graficznego).

```
- <StaticAspect project="Biblioteka" version="0.1">
  - <Window identifier="logowanie" name="Logowanie" size="192,85">
      <Label identifier="l_login" label="login" position="8,8" size="56,23" isActive="active" is
      <Label identifier="l_haslo" label="haslo" position="8,32" size="56,23" isActive="active"
      <Text identifier="t_login" value="-wpisz-login-" position="80,8" isMultiline="single" isA
      <Text identifier="t_haslo" value="-wpisz-haslo-" position="80,32" isMultiline="single"
      <Button identifier="b_loguj" label="Zaloguj" position="8,56" size="56,23" isActive="act
      <Button identifier="b_zamknij" label="Zamknij" position="80,56" size="56,23" isActive
    - <Menu identifier="l_menu" isActive="active">
      - <SubMenu identifier="l_plik" label="Plik" isCheckable="notcheckable" value="off" isAc
          <SubMenu identifier="l_zakoncz" label="Zakoncz" isCheckable="notcheckable" val
        </SubMenu>
      </Menu>
    </Window>
  + <Window identifier="w_czytelnik" name="widokCzytelnika" size="292,185">
  + <Window identifier="w_biblioteka" name="widokBibliotekarza" size="242,125">
  </StaticAspect>
```

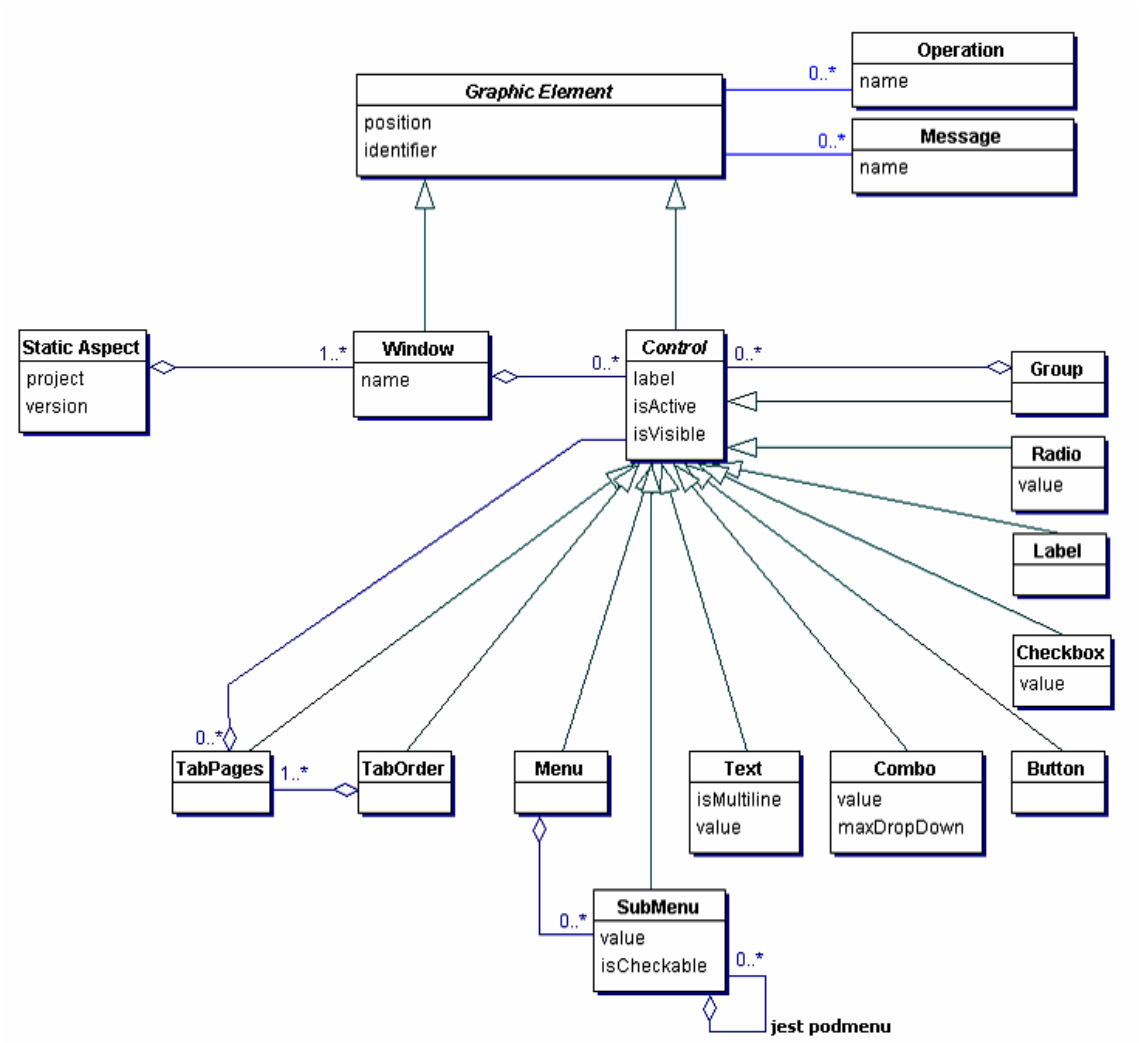
Rys. 4. Instancja aspektu statycznego, zapis w postaci pliku XML

Kolejną formą, która jest użyteczna z punktu widzenia generatora kodu, jest dokument tekstowy XML. Przykładowa zawartość takiego dokumentu jest przedstawiona na rysunku 4. Opisy w postaci diagramu obiektów i dokumentu XML są równoważne w tym sensie, że można je przekształcać wzajemnie jednoznacznie.

3.1. Składnia abstrakcyjna graficznej postaci interfejsu

Określmy składnie dla opisu aspektu statycznego. Rysunek 5. przedstawia składnię abstrakcyjną dla opisu interfejsu graficznego w postaci diagramu obiektów. Przykładem

zgodnego z nią diagramu jest diagram obiektów z rysunku 3. Składnia nie jest kompletna, dla czytelności część kontrolek pominięto, uwzględnione zostały najpopularniejsze kontrolki i ich atrybuty.



Rys. 5. Składnia abstrakcyjna aspektu statycznego

Znaczenie wybranych elementów tego opisu jest następujące:

StaticAspect – opis aspektu statycznego. Rozpatrywany opis składa się z opisu okien aplikacji reprezentowanych przez element **Window**.

Window – okno aplikacji. Przykładowe okno aplikacji prezentowane jest na rysunku 2. Każde okno posiada atrybut określający jego nazwę. Jest ona wyświetlana na górnej belce okna aplikacji.

Control – kontrolka systemowa. Kontrolką są wszystkie elementy graficzne interfejsu za wyjątkiem okna aplikacji. Kontrolka może przechowywać wartość *value* (obecna w klasach potomnych). Dodatkowe dwa wspólne dla wszystkich kontrolek atrybuty to *isActive* i *isVisible*. Określają one odpowiednio czy kontrolka jest aktywna, czyli czy można na niej wykonywać akcje, i czy kontrolka jest widoczna.

GraphicElement – każdy element interfejsu graficznego posiadający reprezentacje

graficzną. Inaczej mówiąc każdy byt, widoczny na ekranie komputera. Z takim elementem graficznym związane mogą być komunikaty *Message* (generowane na skutek akcji użytkownika) i operacje *Operation* (wywoływane na skutek działania prototypu). Każdy typ kontrolek, oraz okno aplikacji posiada zbiór dozwolonych komunikatów i zbiór dozwolonych operacji.

Message – reprezentuje możliwe komunikaty wysyłane od obiektu granicznego aplikacji do wnętrza danej aplikacji. Komunikaty zostają wysyłane wskutek akcji podejmowanych przez użytkownika. Przykładowo komunikat *PUSH_BUTTON* jest wysyłany, gdy na wskazany element graficzny użytkownik kliknie myszką. Taki, komunikat może spowodować wykonanie pewnego ciągu akcji opisanego za pomocą diagramu sekwencji.

Operation – reprezentuje możliwe operacje pozwalające na modyfikacje stanu obiektu graficznego (lub sprawdzenie wartości tego stanu). Przykładowo *getValue* zwraca wartość przechowywaną w danej kontrolce w atrybucie *value*.

Button, Text, Combo, **Checkbox**, **Radio, Menu** oraz inne kontrolki są pominięte. Są z nimi skojarzone standardowe komunikaty typu: *PUSH_BUTTON, VALUE_CHANGED*, *KEY_PRESSED, MENU_SELECTED*.

3.2. Opis składni interfejsu w postaci dokumentu XML

Do generatora kodu potrzebny jest opis interfejsu w postaci pliku XML. Język XML nie ma zdefiniowanego zestawu etykiet (*tag*) z określonymi znaczeniami. Etykiety są tworzone dowolnie, więc by określić strukturę informacji przechowywanej w dokumencie XML przygotowuje się dokument DTD (*Document Type Definition*). Dokument DTD jest uzupełnieniem składni dokumentu XML [WWW2001].

Na rysunku 6. znajduje się DTD opisu aspektu statycznego. Przykładowy opis jednego z okienek w postaci dokumentu XML był prezentowany wcześniej (rysunek 4).

```
    <!ELEMENT StaticAspect (Window+)>
    <!ATTLIST StaticAspect
    project CDATA #REQUIRED
        version CDATA "1.0">

    <!ELEMENT Window (Label*, Text*, Button*, Checkbox*, Radio*, Group*,
Combo*, TabPages*, Menu?)>
    <!ATTLIST Window
    identifier CDATA #REQUIRED
        name CDATA ""
        size CDATA ""
        position CDATA "">
    <!ELEMENT Label EMPTY>
    <!ATTLIST Label
        label CDATA ""
        identifier CDATA #REQUIRED
        size CDATA ""
        position CDATA ""
        isActive (active | inactive) "active"
        isVisible (visible | notvisible) "visible">
```

```
    <!ELEMENT Text EMPTY>
    <!ATTLIST Text
    identifier CDATA #REQUIRED
        size CDATA ""
        position CDATA ""
        isMultiline (single | multi) "single"
        value CDATA ""
        isActive (active | inactive) "active"
        isVisible (visible | notvisible) "visible">
    <!ELEMENT Button EMPTY>
    <!ATTLIST Button
    identifier CDATA #REQUIRED
        position CDATA ""
        size CDATA ""
        label CDATA ""
        isActive (active | inactive) "active"
        isVisible (visible | notvisible) "visible">
    <!ELEMENT Group (Label*, Text*, Button*, Checkbox*, Radio*, Group*,
Combo*, TabPages*)>
    <!ATTLIST Group
    identifier CDATA #REQUIRED
        size CDATA ""
        position CDATA ""
        label CDATA ""
        isActive (active | inactive) "active"
        isVisible (visible | notvisible) "visible">
    <!ELEMENT Menu (SubMenu*)>
    <!ATTLIST Menu
    identifier CDATA #REQUIRED
        label CDATA ""
        isActive (active | inactive) "active">
    <!ELEMENT SubMenu (SubMenu*)>
    <!ATTLIST SubMenu
    identifier CDATA #REQUIRED
        label CDATA ""
        isCheckable (checkable | notcheckable) "notcheckable"
        value (on | off) "off"
        isActive (active | inactive) "active">
]>
```

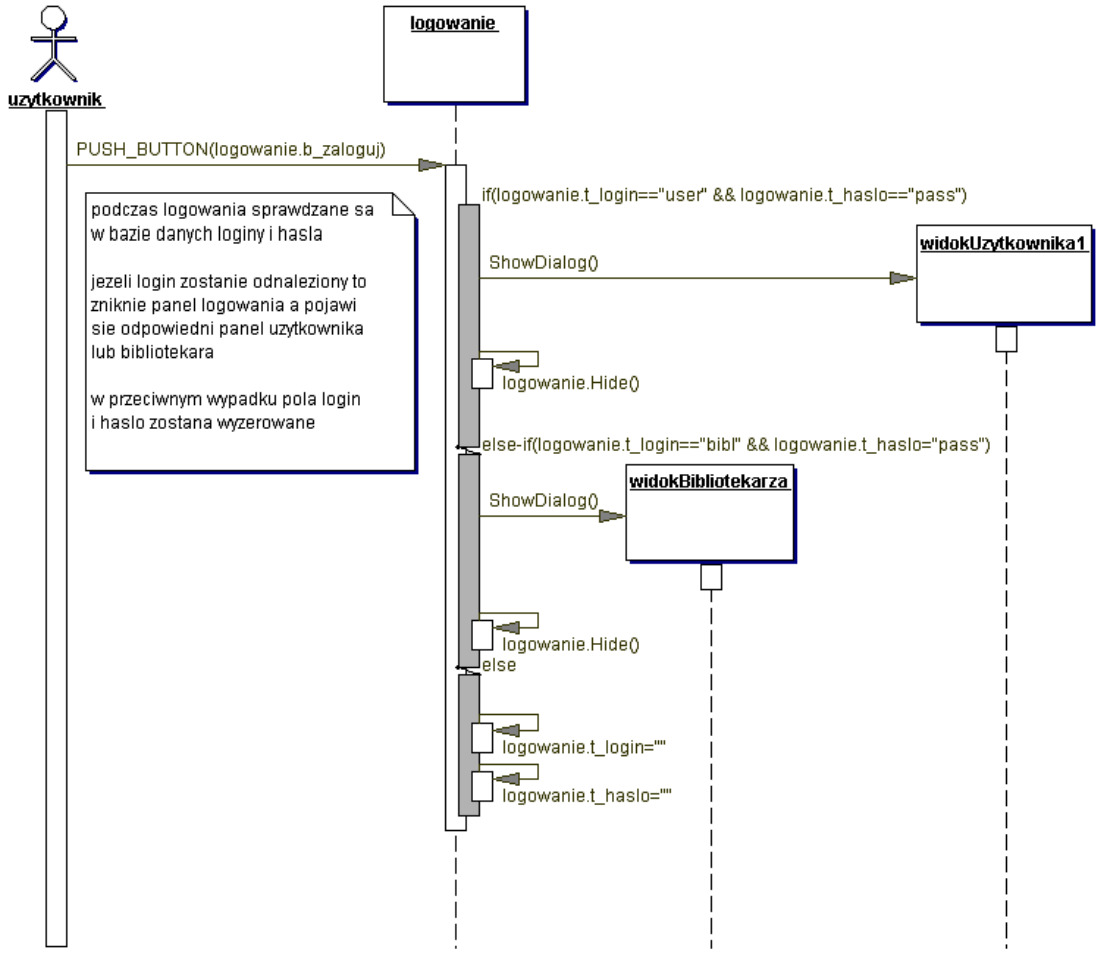
Rys. 6. Fragment DTD dla aspektu statycznego (nie wszystkie kontrolki zostały tutaj umieszczone ze względów na ograniczenie objętości)

4. Aspekt dynamiczny

Aspekt dynamiczny, czyli opis zachowania prototypu, można, podobnie jak aspekt statyczny, prezentować na wiele sposobów. Najbardziej naturalną formą opisu są (w proponowanym podejściu) diagramy sekwencji. Rysunek 7. prezentuje przykładowy diagram sekwencji opisujący reakcję systemu na przyciśnięcie przycisku *b_loguj*). Diagram ten jest zgodny z notacją przedstawioną w [UML2000] (jest zapisany w składni konkretnej). Diagramy zostały wykonane w programie *Together*. Każdy diagram sekwencji opisuje reakcję systemu na odebranie pojedynczego komunikatu od obiektu

granicznego. W ten sposób możemy określić jak system ma zareagować na określoną akcje użytkownika.

Graficzna postać diagramów jest czytelna dla analityka (projektanta), natomiast dla automatycznego generatora kodu przydatny jest zapis diagramów sekwencji w postaci dokumentu XML.



Rys. 7. Składnia konkretna aspektu dynamicznego

Na rysunku 8. jest przedstawiony dokument XML opisujący powyższy diagram sekwencji. Oba opisy niosą identyczną informację o zachowaniu systemu (są równoważne).

Każdy diagram sekwencji opisuje pojedynczą interakcję – odpowiedź systemu na jeden otrzymany komunikat. Taką odpowiedź systemu nie zawsze można dokładnie opisać korzystając z diagramów sekwencji o prezentowanej składni. W przypadku, gdy diagram sekwencji opisze tylko część akcji, które wykonać powinna docelowa aplikacja, warto opisać językiem naturalnym, co zostało pominięte (połączenia z bazą danych itp) a powinno być uwzględnione w docelowym systemie (takie informacje będą przydatne przy wykorzystywaniu diagramów sekwencji jako scenariusze testowe).

Przykładowe diagramy sekwencji – rysunki 6. i 7. – opisują reakcje systemu na odebranie komunikatu *PUSH_BUTTON* zgłoszonego przez kontrolkę *b_loguj*.

```
- <DynamicAspect project="Biblioteka" version="0.1">
  - <Diagram name="zaloguj">
      <Comment>podczas logowania sprawdzane sa w bazie danych loginy i hasla jezeli login zostanie odn
        pojawi sie odpowiedni panel uzytkownika lub bibliotekara w przeciwnym wypadku pola login i hasl
    + <ObjectsAndActors>
    - <InitMessage sender="uzytkownik" reciever="logowanie" instruction="PUSH_BUTTON" control="b_loguj">
      - <Activity instruction="if">
        - <Condition>
          - <And>
            - <Condition>
              - <Equal>
                  <fcontrol control="t_login" function="value" />
                - <value>
                    <Literal text="user" />
                  </value>
                </Equal>
              </Condition>
            + <Condition>
            </And>
          </Condition>
          <Message sender="logowanie" reciever="w_foremka" instruction="ShowDialog" />
          <Message sender="logowanie" reciever="logowanie" instruction="HideDialog" />
        </Activity>
      - <Activity instruction="else-if">
        + <Condition>
          <Message sender="logowanie" reciever="widokBibliotekarza" instruction="ShowDialog" />
          <Message sender="logowanie" reciever="logowanie" instruction="HideDialog" />
        </Activity>
      - <Activity instruction="else">
        - <Message control="t_login" sender="logowanie" reciever="logowanie" instruction="setValue">
          - <value>
              <Literal text="" />
            </value>
          </Message>
        + <Message control="t_haslo" sender="logowanie" reciever="logowanie" instruction="setValue">
        </Activity>
      </InitMessage>
    </Diagram>
```

Rys. 8. Instancja aspektu dynamicznego, zapis w postaci dokumentu XML (część węzłów została zwinięta)

4.1. Składnia abstrakcyjna opisu zachowania interfejsu

Rysunek 9. prezentuje składnię abstrakcyjną dla diagramu sekwencji. Znaczenie poszczególnych obiektów:

Diagram – każdy diagram sekwencji jest powiązany z jedną parą (komunikat, kontrolka), opisuje odpowiedź aplikacji na otrzymanie danego komunikatu z danej kontrolki.

Comment – opis oczekiwanej reakcji systemu za pomocą języka naturalnego.

Element – pomocniczy byt. Abstrakcja aktora, obiektu i okresu aktywności, czyli elementów mogących wysyłać lub odbierać wiadomości **message**.

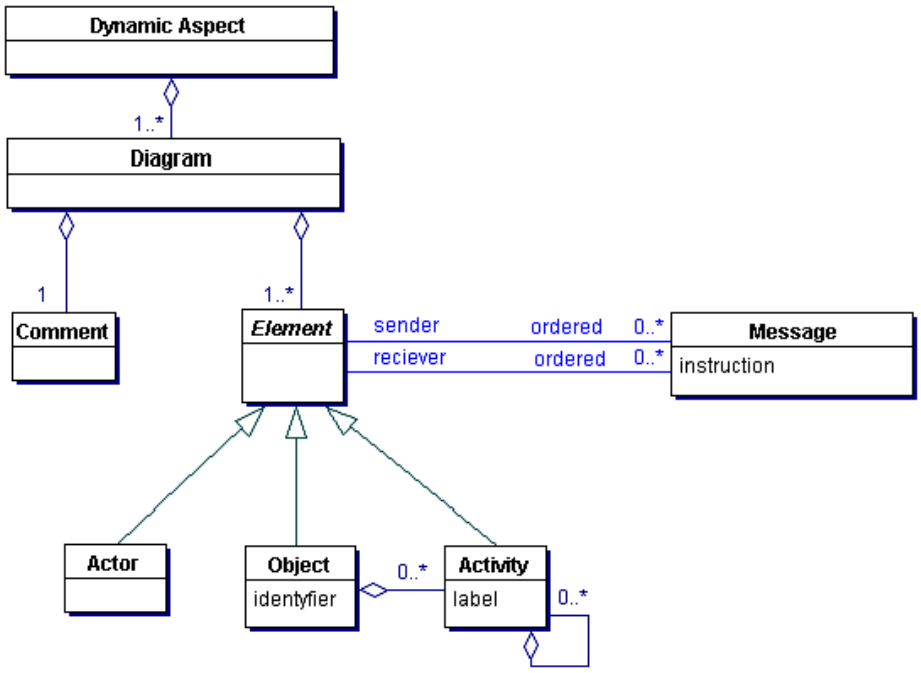
Message – wiadomość przesłana może być pomiędzy aktorem a obiektem, lub pomiędzy obiektem i obiektem. Z każdą wiadomością związana jest instrukcja opisująca rzeczywiste znaczenie tej wiadomości.

Jeżeli wiadomość przesyłana jest od aktora do obiektu, to jest to komunikat inicjujący diagram sekwencji. Opisuje on akcje użytkownika, po zaistnieniu, której wykonywane są sekwencje opisane na danym diagramie. Instrukcja zawiera informacje

o identyfikatorze kontrolki, która może wygenerować daną wiadomość, oraz o nazwie komunikatu związanego z tą kontrolką.

Jeżeli wiadomość przesyłana jest pomiędzy obiektami, to instrukcja ma zawsze postać *„ShowDialog"*. Oznacza ona, że w danej chwili powinno pojawić się na ekranie wskazane okno aplikacji (analityk nie musi się troszczyć czy to okno istnieje i tylko jest pokazywane, czy należy je wcześniej zainicjować).

Jeżeli wiadomość wysłana jest przez obiekt do samego siebie, to może mieć ona następujące znaczenia. W przypadku instrukcji *„Hide"* oznacza, że obecne okno aplikacji powinno zniknąć z ekranu. Inne instrukcje modyfikują wartości atrybutów *value*, *isActive*, *isVisible* lub innych atrybutów posiadanych przez dany element. Modyfikacja tych wartości powoduje adekwatną zmianę wyglądu graficznego.



Rys. 9. Składnia abstrakcyjna aspektu dynamicznego

Actor – Reprezentuje użytkownika.

Object – reprezentuje okno aplikacji, posiada identyfikator. Zakłada się, że istnieje okno opisane w aspekcie statycznym o identycznym identyfikatorze.

Activity – reprezentuje okres aktywności. Okresy te mogą być zagnieżdżone. Okresy aktywności mogą rozpoczynać się jedną z etykiet *if*, *else-if*, *else*, *while*. Etykiety te są powiązane z warunkiem logicznym. Etykieta ta określa gdzie kierowane jest przetwarzanie w chwili rozpoczęcia lub zakończenia okresu aktywności.

4.2 Opis składni zachowania w postaci dokumentu XML

Podobnie jak w przypadku opisu statycznego, poniższy dokument zapisany w standardzie DTD przedstawia składnię dokumentu XML przeznaczonego do opisu aspektu dynami-cznego.

```
<!DOCTYPE DynamicAspect [
    <!ATTLIST DynamicAspect
        project CDATA #REQUIRED
        version CDATA "1.0">
    <!ELEMENT Diagram (Comment?, ObjectsAndActors, InitMessage)>
    <!ATTLIST Diagram
        name CDATA #REQUIRED>
    <!ELEMENT Comment (#PCDATA)>
    <!ELEMENT ObjectsAndActors (Actor?,Object*)>
    <!ELEMENT Actor EMPTY>
    <!ATTLIST Actor
        identifier CDATA #REQUIRED>
    <!ELEMENT Object EMPTY>
    <!ATTLIST Object
        identifier CDATA #REQUIRED>
    <!ELEMENT InitMessage ((Activity|Message)*)>
    <!ATTLIST InitMessage
        sender CDATA #REQUIRED
        reciever CDATA #REQUIRED
        control CDATA #IMPLIED
        instruction (PUSH_BUTTON | VALUE_CHANGED) #REQUIRED>
    <!ELEMENT Message (value? , (Activity|Message)*)>
    <!ATTLIST Message
        control CDATA #IMPLIED
        sender CDATA #REQUIRED
        reciever CDATA #REQUIRED
        instruction (ShowDialog | HideDialog | setValue | setActive |
setUnActive | setVisible | setInVisible | setOn | setOff)  #IMPLIED>
    <!ELEMENT Activity (Condition?, (Activity|Message)*)>
    <!ATTLIST Activity
        instruction (if | else-if | else | while) #IMPLIED>
    <!ELEMENT Condition (And | Or | Not | isActive | isVisible | Equal |
Grater | GraterEqual | Lesser | LesserEqual | Difrent )>
    ]>
```

Rys. 10. DTD dla aspektu dynamicznego (pominięto tagi opisujące gramatykę wartości atrybutu instruction oraz label)

5. Warunki spójności opisów aspektu statycznego i dynamicznego

Opis aspektu statycznego jest związany więzami spójności z opisem aspektu dynamicznego. Aby generator mógł z tych opisów wyprowadzić kod źródłowy, wymaga się by spełnione były trzy warunki.

- Jeżeli w aspekcie dynamicznym występuje identyfikator w roli okna aplikacji lub kontrolki, to wymaga się by aspekt statyczny zawierał opis okna lub kontrolki o zadanym identyfikatorze. Na przykład, obecność w aspekcie dynamicznym obiektu o nazwie *widokUzytkownika1*, wymaga by aspekt statyczny opisywał okno o identyfikatorze *widokUzytkownika1*.
- Jeżeli w opisie dynamicznym znajduje się odwołanie do kontrolki, to identyfikator

kontrolki może być poprzedzony listą kontenerów (kontener to kontrolka zawierająca inną kontrolkę) zawierających daną kontrolkę, pod warunkiem, że lista ta zgodna jest ze strukturą zawierania kontrolki opisaną w aspekcie statycznym.

Na przykład, jeżeli aspekt statyczny informuje, że kontrolka ***przycisk1*** jest umieszczona w kontenerze ***panel1***, to w aspekcie dynamicznym odwołanie do tej kontrolki powinno wyglądać następująco: ***panel1.przycisk1***.

- Komunikaty wysyłane przez kontrolkę oraz operacje wykonywane na kontrolce powinny należeć do zbioru komunikatów lub operacji dozwolonych dla danej kontrolki.

Na przykład, kontrolka *Text* może wysyłać komunikaty *VALUE_CHANGED*, ale nie może wysyłać komunikatów *PUSH_BUTTON*.

6. Reguły generacji kodu

6.1. Koncepcja algorytmu generacji kodu źródłowego

Reguły są przygotowane w postaci szablonów XSL, czyli reguł przekształcania tekstu zapisanych za pomocą XML. Szablony takie umożliwiają przetworzenie drzewa z danymi umieszczonymi w węzłach zapisanego w formie XML na inne drzewo zapisane w formie XML. W przypadku generacji kodu źródłowego można przyjąć, że docelowe drzewo składa się tylko z jednego węzła-korzenia, który zawiera cały tekst programu. Drzewo wejściowe ma dużo bogatszą strukturę. Ponieważ są wykorzystywane opisy dwóch aspektów to korzeń drzewa wejściowego ma dwóch potomków. Jeden to węzeł typu *StaticAspect* a drugi typu *DynamicAspect*.

Generowanie kodu powinno być wykonywane w odpowiedniej kolejności. Najpierw na podstawie aspektu statycznego powinien być wygenerowany kod implementujący byty opisane w aspekcie statycznym. Następnie otrzymany tekst należy uzupełnić o elementy opisujące zachowanie.

Implementacja aspektu statycznego będzie wyglądała różnie w zależności od wybranego języka programowania i platformy uruchomieniowej. Wynikiem działania generatora może być zbiór klas, z których każda opisuje jedno okno aplikacji (tak może być w aplikacjach Javy, C#) lub zbiór plików HTML (tak może być w przypadku generowania prototypu na platformę WWW). Tak wygenerowany kod źródłowy jest już poprawny w sensie składni wybranego języka programowania.

Kod implementujący aspekt dynamiczny może mieć różne formy. Mogą to być metody w klasach odpowiadające za odbieranie określonych zdarzeń (tak może być w aplikacjach Javy, C#) lub mogą być to pliki, skrypty, uruchamiane lub interpretowane po nadejściu określonego żądania (tak może być w przypadku generowania prototypu na platformę WWW, żądania wysyła przeglądarka internetowa, a za uruchomienie skryptu odpowiedzialny jest serwer WWW).

Postać wynikowego kodu źródłowego może być różna. W poniższym podrozdziale

zostanie omówiony przykład generowania kodu źródłowego aplikacji przeznaczonej do uruchamiania w systemie Windows napisanej w języku C#. Każdy kod źródłowy składa się z trzech fragmentów. Jeden jest zależny od języka i musi występować w każdej aplikacji bez względu na to, czemu ona służy. Ten fragment opisuje używane przestrzenie nazw, zawiera informacje sterujące dla kompilatora itp. Operacje wspólne dla każdej aplikacji graficznej. Pozostałe dwa fragmenty kodu to reprezentacja aspektu statycznego i dynamicznego.

6.2. Przykładowe reguły dla języka C#

Pierwszy krok to generowanie kodu na podstawie aspektu statycznego. Fragment szablo-nu XSL odpowiedzialny za aspekt statyczny znajduje się na rysunku 11.

Ze względu na objętość kompletu reguł przedstawione są jedynie fragmenty pozwalające na wygenerowanie źródeł przedstawionych na rysunku 12.

W wyniku przetworzenia opisu statycznego otrzymujemy kod źródłowy, którego fragmenty znajdują się na rysunku 12. Bloki oznaczone etykietami powstały w wyniku przetworzenia jednego węzła XML. Szczegółowe opisy tych bloków znajdują się poniżej.

Blok P1 jest wspólny dla aplikacji okienkowych wytwarzanych w C#. Jest to wskazanie różnych przestrzeni nazw, z których korzystać będą elementy aplikacji.

Blok P2 to początek deklaracji klasy. Dla każdego okna aplikacji będzie przygotowana inna klasa odpowiedzialna za wygląd i zachowanie okna oraz kontrolek zawartych w tym oknie.

Blok P3 to deklarowanie zmiennych reprezentujących kontrolki. Każda kontrolka należąca do okna musi tutaj otrzymać typ oraz nazwę. Nazwą kontrolki będzie atrybut *identifier* z aspektu statycznego.

Blok P4 to standardowy fragment każdego okna graficznego. Służy inicjalizacji samego okna.

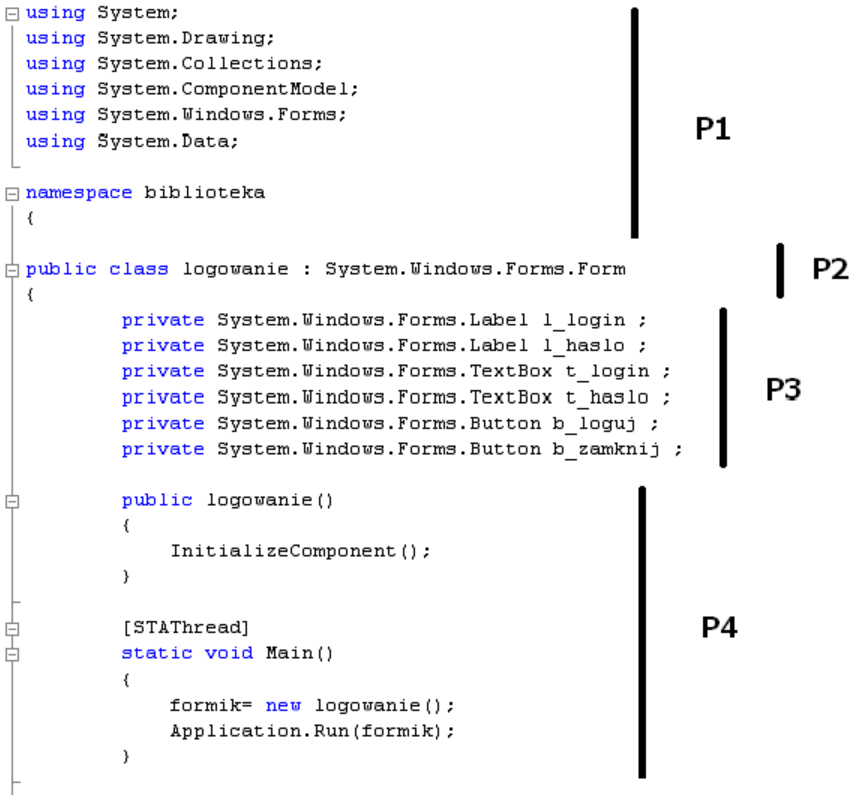
Blok G1 i G8 są wspólne dla każdego okna. W funkcji InitializeComponent() będą wykonane czynności niezbędne do określenia własności kontrolek, w tym również określeniu, które kontrolki odpowiadają na jakie komunikaty.

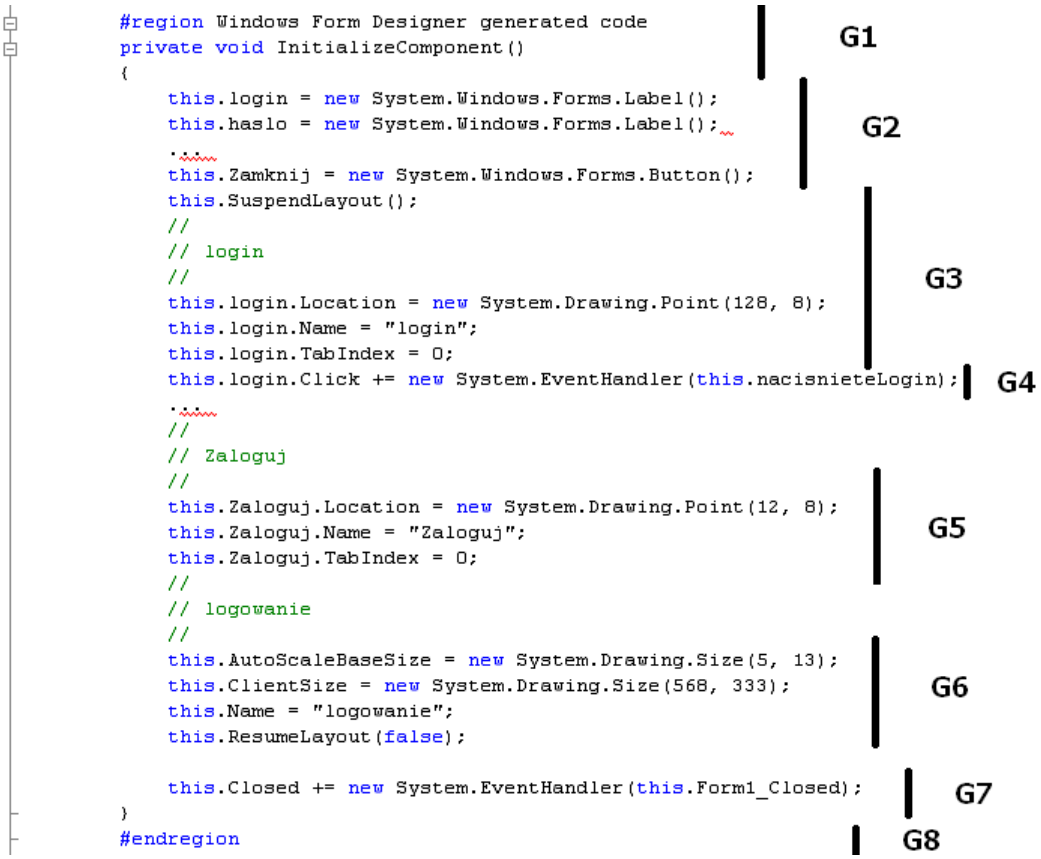
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html> <body> <pre>
    <xsl:apply-templates/>
  </pre> </body> </html>
</xsl:template>

<xsl:template match="StaticAspect">
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
```

```
using System.Windows.Forms;
using  System.Data;
namespace biblioteka { <xsl:apply-templates/> }
</xsl:template>

<xsl:template match="Window">
public class <xsl:value-of select="@ identifier "/> :
System.Windows.Forms.Form {
  <xsl:for-each select="Label">
    private System.Windows.Forms.Label <xsl:value-of select="@identifier "/>;
  </xsl:for-each>
  <xsl:for-each select="Text">
    private System.Windows.Forms.TextBox <xsl:value-of select="@identifier
"/>;
  </xsl:for-each>
  <xsl:for-each select="Button">
    private System.Windows.Forms.Button <xsl:value-of select="@identifier
"/>;
  </xsl:for-each>
    public <xsl:value-of select="@identyfikator"/>() {
        InitializeComponent();
    }
    protected override void Dispose( bool disposing ) {
        if( disposing && components != null)
            components.Dispose();
        base.Dispose( disposing );
    }
    #region Windows Form Designer generated code
    private void InitializeComponent() {
       <xsl:apply-templates/>
       this.ResumeLayout(false);
    }
    #endregion
}
</xsl:template>

<xsl:template match="Label">
    this.<xsl:value-of select="@identifier "/>
        = new System.Windows.Forms.Label();
    this.<xsl:value-of select="@identifier "/>.Location
        = new System.Drawing.Point(<xsl:value-of select="@position"/>);
    this.<xsl:value-of select="@identifier "/>.Name
        = "<xsl:value-of select="@text"/>";
    this.<xsl:value-of select="@identifier "/>.Text
        = "<xsl:value-of select="@text"/>";
</xsl:template>

<xsl:template match="Text">
    this.<xsl:value-of select="@identifier "/>
        = new System.Windows.Forms.TextBox();
    this.<xsl:value-of select="@identifier "/>.Location
        = new System.Drawing.Point(<xsl:value-of select="@position"/>);
    this.<xsl:value-of select="@identifier "/>.Name
        = "<xsl:value-of select="@text"/>";
    this.<xsl:value-of select="@identifier "/>.Text
        = "<xsl:value-of select="@text"/>";
</xsl:template>
```

```
<xsl:template match="Button">
    this.<xsl:value-of select="@identifier"/>
        = new System.Windows.Forms.Button();
    this.<xsl:value-of select="@identifier "/>.Location
        = new System.Drawing.Point(<xsl:value-of select="@poition"/>);
    this.<xsl:value-of select="@identifier "/>.Name
        = "<xsl:value-of select="@text"/>";
    this.<xsl:value-of select="@identifier "/>.Text
        = "<xsl:value-of select="@text"/>";
</xsl:template>
</xsl:transform>
```

Rys. 11. Fragment szablonu XSL przeznaczony do przetwarzania aspektu statycznego (Pogrubione elementy to fragmenty wynikowego kodu źródłowego, elementy bez pogrubienia to instrukcje sterujące przetwarzaniem)





Rys. 12. Fragment wygenerowanego automatycznie kodu

Blok G2 to inicjalizacja zmiennych reprezentujących kontrolki. Ten fragment powstanie z opisu aspektu statycznego.

Blok G3 i G5 musi pojawić się dla każdej kontrolki. Tutaj określane są własności kontrolek, ich nazwy, wartości początkowe przechowywanych danych, współrzędne położenia i inne atrybuty.

Blok G6 określa własności samego okna graficznego, rozmiar, położenie, nazwę, styl itp.

Blok G4 powstał w skutek przetwarzania aspektu dynamicznego. Taki fragment pojawia się dla każdego diagramu sekwencji, który jest powiązany z tym oknem. Diagram sekwencji jest przekształcany na funkcję obsługi zdarzenia. Treść funkcji znajduje się w klasie odpowiadającej oknu, którego kontrolka zainicjowała dany komunikat.

7. Podsumowanie

Opisana metoda wytwarzania prototypu została przetestowana na przykładowej aplikacji *Systemu Zarządzania Biblioteką*. Wygląd okien ukazujących się użytkownikowi zdefiniowano za pomocą *Microsoft Visual Studio.NET*. Za pomocą opracowanego parsera opis tych okien został zamieniony na postać XML. Wykorzystując pakiet *Together 4.2 Whiteboard Edition* przygotowano diagramy sekwencji opisujące zachowanie projektowanej aplikacji, następnie zostały one przekształcone na równoważny opis XML.

Oba te opisy zostały przekształcone z użyciem szablonów XSL w kod źródłowy. Kod ten został podzielony na pliki zawierające opisy pojedynczych klas. Następnie pliki te zostały poddane kompilacji i konsolidacji, w wyniku czego otrzymano gotową działającą aplikację. Aplikacja zachowywała się zgodnie z opisami, według których została wygenerowana. Wytwarzanie aplikacji od chwili podania dokumentów XML do parsera trwało kilkadziesiąt sekund. Aplikacja posiadała 7 różnych okien i zawierała opis obsługi 11. komunikatów.

Przy opisywaniu aspektu statycznego i aspektu dynamicznego najbardziej czasochłonne jest ustalenie wyglądu okien aplikacji i określenie zachowania aplikacji. Narysowanie już ustalonego okna aplikacji łącznie z ustawieniem właściwości jego kontrolek zajmuje kilka-kilkanaście minut. Czas ten oczywiście charakteryzuje się dużą zmiennością w zależności od stopnia zaawansowania użytkownika. Opis zachowania to jest to najczęściej prosty diagram opisujący zmianę zawartości jakiejś kontrolki i/ lub pokazania lub ukrycia okna aplikacji.

Prototypy wytworzone proponowaną metodą mają pewne ograniczenia związane z zakresem stosowania i z uproszczeniami w składni obu aspektów. Ograniczenie zakresu stosowania wynika z faktu, że nie każdą aplikacje można przedstawić w prezentowany sposób, aplikacje interaktywne typu gry, lub edytory tekstu będzie bardzo trudno opisać proponowaną metodą (łatwo opisać można aplikacje bazodanowe, oraz serwisy internetowe, a tego typu aplikacje stanowią znaczną część wytwarzanego obecnie oprogramowania).

Porównanie proponowanej metody z metodami opartymi o szybkie prototypowanie z użyciem aplikacji typu RAD (*Rapid Application Development*), przedstawiono w tabeli 1.

Tabela 1. Porównanie z innymi metodami szybkiego prototypowania

aspekt	użycie narzędzi typu RAD	użycie proponowanej metody
czas przygotowania prototypu	osoba znająca platformę, w której ma powstać prototyp potrafi szybko przygotować działający prototyp	osoba znająca ograniczenia i możliwości prezentowanych diagramów sekwencji potrafi szybko przygotować działający prototyp
przenaszalność	kod źródłowy dla danej platformy najczęściej nie nadaje się do przeniesienia na inną platformę	zmieniając reguły XSL otrzymujemy kod źródłowy przeznaczony na inną platformę, pozwala to na prezentowanie klientowi prototypów na różnych platformach
możliwość późniejszego wykorzystania	znikoma, kod może być nie udokumentowany (prototyp traktowany jest	potencjalnie wysoka, diagramy sekwencji można wykorzystać do weryfikacji zgodno-

	często „jednorazowo"), nie ma możliwości porównania źródeł prototypu i gotowej aplikacji ze względu na różnice w architekturze aplikacji	ści docelowego systemu z prototypem
możliwość zrównoleglenia prac	znikoma, ze względu na pracę na tych samych zbiorach, sensowne rozłożenie prac jest na tyle czasochłonne, że nie opłacalne	wysoka, poszczególne okna mogą być projektowane oddzielnie następnie łączone, dla już gotowego opisu interfejsu graficznego można niezależnie opisywać diagramami sekwencji zachowania interfejsu
możliwości funkcjonalne	znaczne, programista nie jest praktycznie ograniczony, jeżeli chodzi o możliwości wykorzystania gotowych bibliotek czy funkcji	ograniczone do prostych operacji powodujących zmiany w interfejsie graficznym
wymagane umiejętności	znajomość platformy, na którą generowany jest prototyp	znajomość składni i semantyki opisów obu prezentowanych aspektów

Największą zaletą przedstawionej metody jest możliwość szybkiego przygotowania prototypu, na dowolną platformę (niezależność od języka programowania). Istotna jest duża łatwość w zrównolegleniu prac nad przygotowywaniem prototypu oraz możliwość szybkiego późniejszego wykorzystania produktów ubocznych (diagramów sekwencji i opisu interfejsu). Te cechy sprawiają, że prezentowana metoda rzeczywiście usprawnia proces prototypowania.

Bibliografia

[GUI2002] *User Interface Style Guide*, http://www.pitb.gov.pk/standards/9-3GUI styl guide.pdf.

[SILV2001] Paulo Pinheiro da Silva, *User Interface Modelling with UML*, 2001, www.ksl.stanford.edu/people/pp/papers/ PinheirodaSilva_IMKB_2000.pdf.

[SILV2002] Paulo Pinheiro da Silva, *Object Modelling of Interactive Systems: the UML Approach*, 2002, www.ksl.stanford.edu/people/pp/papers/ PinheirodaSilva_PhD_2002.pdf .

[UML2000] *OMG Unified Modelling Language Specification version 1.4*, Listopad 2000, www.digilife.be/quickreferences/Books/ OMG UML Specification 1.4.pdf.

[USDP1999] I. Jacobson, G. Booch i J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[WWW1999] *Interfaced Systems International*, http://www.isii.com/.

[WWW2001] *DTD Specification of Intermediate XML Meta Model*, http://cic.vtt.fi/projects/ifcsvr/tec/VTT-TEC-ADA-03.pdf.

[XML1998] *Extensible Markup Language (XML) 1.0 W3C Recommendation*, 10-Feb-98, www.w3.org/TR/1998/REC-xml-19980210.

[XSL1999] *XSL Transformations (XSLT) Specification Version 1.0 W3C Working Draft*, 21 Apr 1999, www.w3.org/TR/xslt.