STUDIA INFORMATICA 2011
Volume 32 Number 2B (97)

Józef KABIESZ, Marcin MICHALAK, Sebastian IWASZENKO
Główny Instytut Górnictwa

# TWORZENIE INTERFEJSU UŻYTKOWNIKA NA PODSTAWIE SZABLONU *MiSS FRAMEWORK*

**Streszczenie**. Artykuł przedstawia rozwiązanie pozwalające na sprawne tworzenie zaawansowanych aplikacji okienkowych, opartych na komunikujących się ze sobą modułach. W ramach szablonu zdefiniowano zarówno mechanizm komunikacji w obrębie aplikacji, jak i pomiędzy różnymi aplikacjami. Opisano zarówno stosowane mechanizmy, jak i proces tworzenia aplikacji z wykorzystaniem frameworku. Artykuł kończy przykład zastosowania *MiSS Framework* do stworzenia prostej, okienkowej aplikacji bazodanowej.

**Słowa kluczowe**: framework, interfejs użytkownika, aplikacja bazodanowa, aplikacja modułowa

# *MISS FRAMEWORK* BASED USER INTEFACE DEVELOPMENT

**Summary**. This article describes a framework for window applications built from modules that communicate each other. With this framework the communication within the application and between application is possible. The description of used mechanisms and application development is also the part of the article. It ends with the example of the window application created on the basis of the *MiSS Framework*.

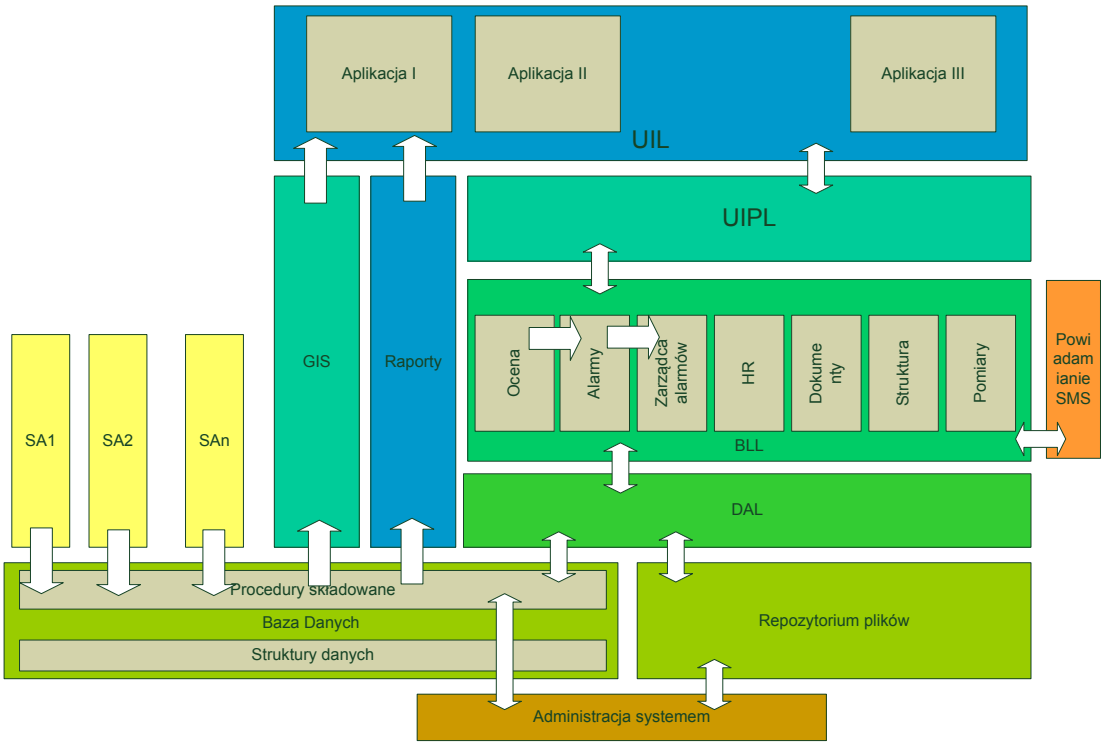
**Keywords**: framework, user interface, database application, modular application

## 1. Wstęp

Praca wydobywcza w kopalniach niesie za sobą ryzyko wystąpienia zdarzeń, wynikających z wielu zagrożeń naturalnych (np. tąpania, wybuchy metanu) czy też spowodowanych awarią sprzętu lub czynnikiem ludzkim. Większość stosowanych w górnictwie systemów monitorowania ogranicza swoją funkcjonalność do gromadzenia, wizualizowania w centrach dyspozycyjnych odpowiednich pomiarów i, w razie ich przekroczenia, do sygnalizowania

stanów alarmowych [1, 5, 7] (jako wyjątek można tu przywołać system HESTIA [6], który realizuje procedury kompleksowej metody oceny stanu zagrożenia tąpaniami [8]). Stąd też wynika potrzeba tworzenia systemów o bardziej zaawansowanej analizie bieżącego stanu bezpieczeństwa w kopalni na podstawie współwystępowania zagrożeń oraz ich wzajemnego oddziaływania na siebie [4].

System MiSS, którego dokładna koncepcja została wcześniej opisana w [3], jest przykładem systemu kompleksowego zarządzania zagrożeniami górniczymi. Schemat tego systemu przedstawia rys. 1.




Rys. 1. Architektura systemu MiSS
Fig. 1. The MiSS system architecture

W skład tego sytemu wchodzą moduły integracji z zewnętrznymi systemami akwizycji i przetwarzania danych (bloki SAi na rysunku), serwer bazy danych, rozbudowana warstwa biznesowa (wyznaczanie ocen poszczególnych zagrożeń, generowanie i przetwarzanie alarmów, obsługa dokumentacji itp.), moduł powiadamiania o sytuacjach alarmowych oraz warstwa interfejsu użytkownika. W części systemu odpowiedzialnej za kontakt z użytkownikiem wyodrębniono warstwę interfejsu użytkownika, której rolą jest bezpośrednia komunikacja z użytkownikiem i wizualizacja zgromadzonych oraz przetwarzanych w systemie danych. Ponadto, wyodrębniona została warstwa procesów interfejsu użytkownika. Pełni ona rolę zarządzania sposobem komunikacji pomiędzy elementami warstwy UI, zarządzania realizacją złożonych zadań za pośrednictwem UI oraz nadzoruje komunikację z warstwą logiki biznesowej. Ponieważ zakłada się, że użytkownicy będą korzystali z kilku rodzajów aplikacji ko-

munikujących się z komponentami systemu umieszczonymi na serwerach, wskazane jest przyjęcie wspólnego sposobu ich implementacji, a także wprowadzenie ujednoliconego sposobu komunikacji. Rozwiązanie takie powinno przyjąć formę zrębu architektonicznego (framework).

Niniejszy artykuł przedstawia implementację pewnej spójnej koncepcji tworzenia zaawansowanych aplikacji okienkowych. Ponieważ mechanizm ten powstał na potrzeby projektu *MiSS,* nazwany został *MiSS Framework*. Jest to ogół klas służących do tworzenia interfejsu użytkownika, komunikacji pomiędzy poszczególnymi modułami systemu oraz obsługi dostępu do danych. W kolejnych częściach pracy zidentyfikowano i przedstawiono problemy związane z tworzeniem interfejsu użytkownika, sprecyzowano wymagania, jakie powinno spełnić poszukiwane rozwiązanie, opisano funkcjonalność *MiSS Framework* oraz zaprezentowano jego zastosowanie do stworzenia przykładowej aplikacji okienkowej.

## 2. Charakterystyka problemu

Zasadniczym celem projektu „Informatyczny system wspomagania kompleksowego zarządzania zagrożeniami górniczymi" jest stworzenie oprogramowania wspomagającego pracę wybranych działów kopalni. System ten nazwano *MiSS*, jako akronim angielskiej nazwy *Mine Security System*. Ze względu na fakt, że w systemie będzie istniało wielu użytkowników korzystających z systemu w różnym zakresie, założono modułową budowę ostatniej warstwy systemu. Każdy moduł będzie realizował pewną część funkcjonalności systemu, związanej z interfejsem użytkownika, moduły z kolei będą łączone w aplikacje przeznaczone dla poszczególnych typów użytkowników.

Tworzenie złożonego interfejsu użytkownika, skomponowanego z zestawu współpracujących ze sobą kontrolek-widoków, wymaga zastosowania ujednoliconego, spójnego sposobu implementacji poszczególnych widoków oraz ich współpracy ze sobą oraz resztą systemu. Taka metoda powinna posiadać formę frameworku. Przeprowadzona analiza istniejących rozwiązań pokazała, że większość z nich dotyczy rozwoju aplikacji Web oraz oparta jest na technologiach innych niż .NET, której zastosowanie wynikało przyjętych założeń. Frameworki przeznaczone do tworzenia aplikacji okienkowych na podstawie technologii Windows Forms były albo rozwiązaniami wymagającymi zakupu licencji (TrueView, NConstruct), albo charakteryzowały się stopniem złożoności znacznie przewyższającym planowany sposób wykorzystania (Smart Client Software Factory). Przyjęto zatem, że w ramach tworzonego systemu opracowany zostanie również zestaw komponentów wspierający tworzenie aplikacji o cechach frameworku.

Przyjęto, że rozwiązanie problemu powinno spełniać następujące warunki:

- funkcjonalność interfejsu użytkownika jest realizowana przez moduły,
- każda aplikacja użytkownika udostępnia pewien zbiór modułów,
- układ graficzny poszczególnych aplikacji użytkownika jest identyczny,
- sposób funkcjonowania modułu nie zależy od aplikacji, w której został osadzony,
- wszystkie moduły komunikują się z serwerem systemu w jednakowy sposób,
- istnieje zdefiniowana forma komunikacji zarówno klienta z serwerem, jak i serwera z klientami,
- istnieje możliwość komunikacji pomiędzy poszczególnymi modułami w obrębie jednej aplikacji użytkownika,
- istnieje możliwość komunikacji pomiędzy poszczególnymi aplikacjami użytkownika.

## 3. Proponowane rozwiązanie – MiSS Framework

Podstawową ideą obiektowych języków programowania jest istnienie pojęć klasy i dziedziczenia [1]. Przez klasę rozumie się pewien jasno sprecyzowany byt programistyczny, zdefiniowany przez listę pól (wartości) i metod (funkcji, procedur) z określeniem ich dostępności z innych klas. Z kolei dziedziczenie to mechanizm pozwalający rozbudowywać funkcjonalność klas w taki sposób, że klasa B, dziedzicząca po klasie A, stanowi rozszerzenie jej funkcjonalności. Można zatem powiedzieć, że każdy kolejny poziom dziedziczenia wiąże się z utratą ogólności klasy na korzyść jej wyspecjalizowania.

Projekt MiSS jest tworzony m.in. na podstawie języka programowania C#, który posiada mechanizmy dziedziczenia. W ramach projektu wydzielono pewien zbiór klas tworzących *MiSS Framework*, odpowiedzialny właśnie za realizację wymienionych w poprzednim punkcie działań. Zasadniczym elementem frameworku są klasy związane z tworzeniem interfejsu użytkownika. Pozwalają one w prosty i powtarzalny sposób generować aplikacje dla różnych użytkowników, na podstawie różnych modułów. Kolejnym elementem frameworku są klasy związane z komunikacją pomiędzy elementami warstwy interfejsu użytkownika a serwerem systemu. W kolejnych punktach przedstawione zostaną poszczególne wymienione elementy.

### 3.1. Komponenty graficzne warstwy UIL

*MiSS Framework* dostarcza dwie klasy graficzne, na podstawie których tworzy się aplikację okienkową: kontrolkę użytkownika (*UcObject*) oraz formę aplikacji (*FObject*).

#### *3.1.1. UcObject*

*UcObject* to kontrolka bazowa dla poszczególnych kontrolek, które mają się składać na aplikację. Do jej zasadniczych elementów należy zaliczyć: pasek narzędzi, menu kontekstowe, „belkę tytułową", zestaw zdarzeń generowanych przez kontrolkę oraz obiekt bieżącego stanu kontrolki. Pasek narzędzi stanowi element udostępniany oknu głównemu aplikacji, co pozwala w trakcie pracy programu na dynamiczne przełączanie paska narzędzi w zależności od tego, która kontrolka użytkownika jest w danym momencie aktywna. Podobną funkcjonalność posiada menu kontekstowe – w chwili aktywacji danej kontrolki użytkownika jego elementy zostają dynamicznie osadzone w menu okna głównego aplikacji.

Każdy moduł tworzony w ramach systemu powinien dziedziczyć po klasie *UcObject*. Dzięki temu tworzona kontrolka jest w stanie poprawnie komunikować się z oknem głównym aplikacji oraz z pozostałymi kontrolkami użytkownika stworzonymi w ten sposób.

#### *3.1.2. FObject*

*FObject* to szablon formy głównej aplikacji okienkowej. Okno główne każdej aplikacji okienkowej, należącej do systemu, powinno dziedziczyć po tej właśnie klasie. Forma ta stanowi nie tylko kontener, w obrębie którego umieszczane są poszczególne kontrolki użytkownika (dziedziczące po *UcObject*). Bowiem w ramach klasy bazowej *FObject* dostarczono już także obsługę dynamicznego podpinania paska narzędzi z aktywnej aktualnie kontrolki użytkownika oraz podłączania menu kontekstowego kontrolki do menu głównego aplikacji. Klasa *FObject* odpowiedzialna jest także za obsługę komunikacji pomiędzy poszczególnymi kontrolkami użytkownika oraz za dostęp do danych i komunikację z serwerem.

### 3.2. Komunikacja w obrębie aplikacji

Komunikacja pomiędzy poszczególnymi kontrolkami użytkownika a formą główną aplikacji zachodzi dwukierunkowo i może być inicjowana zarówno przez kontrolki, jak i samą formę. Rozważmy dla przykładu dwie kontrolki „Wyrobiska" i „Oceny". Pierwsza z nich wyświetla istniejące w systemie wyrobiska kopalniane oraz posiada możliwość ich filtrowania. W każdym momencie pracy kontrolki tylko jedno wyrobisko posiada status „wybrane" („aktywne"). Druga kontrolka zajmuje się wyświetlaniem informacji o ocenach wyznaczonych dla konkretnego wyrobiska. Oczekiwalibyśmy, że zmiana aktywnego wyrobiska w kontrolce „Wyrobiska" spowoduje zmianę ocen wyświetlonych w drugiej kontrolce. Sekwencja przebiegu informacji (bez uwzględniania ewentualnych potrzeb odwołania do bazy danych znajdującej się na serwerze) wyglądać będzie następująco:

1. Użytkownik zmienia aktywne wyrobisko w kontrolce „Wyrobiska”, co powoduje wygenerowanie zdarzenia zmiany stanu tejże kontrolki.
2. Okno główne aplikacji obsługuje wspomniane zdarzenie przez odpytanie wszystkich kontrolek użytkownika (dziedziczących po *UcObject*) o ich stan i na tej podstawie buduje tzw. stan aplikacji.
3. Jedną ze składowych stanu kontrolki „Wyrobiska” jest informacja o identyfikatorze aktywnego wyrobiska.
4. Stan aplikacji przekazywany jest każdej z kontrolek frameworka. Stan ten jest przetwarzany w kontrolkach zależnie od ich funkcjonalności.
5. Kontrolka „Pomiary” odczytuje w obiekcie stanu aplikacji wartość identyfikatora aktywnego wyrobiska w kontrolce „Wyrobiska” i na tej podstawie dokonuje zmiany wyświetlanych informacji o wygenerowanych ocenach.

W sytuacji kiedy to forma aplikacji miałaby być inicjatorem komunikacji, zmianie ulegają pierwsze dwa punkty: punkt pierwszy nie jest związany ze zdarzeniem zmiany stanu kontrolki użytkownika, lecz z jakimś innym zdarzeniem (np. nadejściem komunikatu z serwera), natomiast w punkcie drugim stan aplikacji zostaje odpowiednio zmodyfikowany przez samą formę i dopiero potem rozgłoszony wśród wszystkich kontrolek użytkownika.

### 3.3. Komunikacja aplikacja-serwer

Komunikacja pomiędzy aplikacją a serwerem odbywa się za pomocą dwóch komponentów, zwanych dalej akcesorami. Pierwszy z nich, *DataAccessor*, służy wywoływaniu metod udostępnianych przez serwer oraz przesyłaniu informacji na serwer, natomiast drugi, *CommunicationAccessor*, służy jako odbiorca komunikatów otrzymywanych ze strony serwera. Funkcjonalność akcesorów można także określić na podstawie inicjatora komunikacji – jeśli komunikację rozpoczyna aplikacja użytkownika (pobranie bądź przesłanie danych), to komunikacja odbywa się za pomocą *DataAccessora*, natomiast, jeśli komunikacja zachodzi z inicjatywy serwera, to po stronie aplikacji okienkowej używany jest *CommunicationAccessor*.

#### *3.3.1. CommunicationAccessor*

*CommunicationAccessor* to komponent, którego zasadniczym zadaniem jest umożliwienie serwerowi przesyłania komunikatów dla aplikacji okienkowej. W obecnej wersji oprogramowania działa on w ten sposób, że podczas uruchomienia aplikacji użytkownika akcesor udostępnia usługę sieciową, której jest bezpośrednim hostem. Podstawową metodą usługi jest `PutMessage`. Serwer, chcąc przesłać komunikat do konkretnej aplikacji okienkowej, korzysta z uruchomionej na niej usługi i wspomnianej metody. Akcesor przechwytuje obsługę zdarze-

nia nadejścia komunikatu wygenerowanego przez usługę i przekazuje sterowanie, również przez wygenerowanie zdarzenia, do okna głównego aplikacji.

Przykładem wykorzystania tego typu komunikacji jest sytuacja, w której jeden z użytkowników dokonuje zmiany danych w systemie (modyfikacja bądź usunięcie). Kontrolka użytkownika, która dokonała tych operacji, jest w stanie powiadomić o tym fakcie jedynie inne kontrolki znajdujące się w obrębie tej samej aplikacji. Dopiero aplikacja może powiadomić serwer o takim zdarzeniu, a ten z kolei powinien przesłać do pozostałych aplikacji komunikat o konieczności odświeżenia odpowiednich danych. Można również rozważyć możliwość, by już samo wykonanie na serwerze operacji modyfikacji danych powodowało powiadamianie wszystkich aplikacji o zmianie danych, co nie zmienia sposobu informowania o tym fakcie.

Informację o tym, pod jakim adresem znajdują się usługi poszczególnych aplikacji, serwer otrzymuje od aplikacji podczas ich uruchamiania. Pełna informacja wysyłana jest przez aplikację drugim z wymienionych akcesorów, opisanym w kolejnym punkcie.

### *3.3.2. DataAccessor*

*DataAccessor* to komponent, który przeznaczony jest przede wszystkim do pobierania danych z pozostałych warstw systemu (rys. 1). W odróżnieniu od komponentu *CommunicationAccessor*, ten komponent nie musi przekazywać sterowania oknu aplikacji za pomocą zdarzeń. Dostęp do danych z użyciem *DataAccessora* odbywa się kolejno, w następujący sposób:

1. Kontrolka użytkownika generuje żądanie pobrania (modyfikacji, usunięcia) danych przez zdarzenie zmiany swojego stanu – bieżący stan to także żądanie wykonania operacji bazodanowych.
2. Okno główne aplikacji przetwarza stan aplikacji i natrafia na żądanie pobrania (modyfikacji itp.) danych.
3. Okno główne aplikacji wywołuje metodę akcesora, która jest pośrednim wywołaniem analogicznej metody po stronie serwera.
4. W przypadku metod zwracających dane, rezultat ich wykonania umieszczany jest w strukturze opisującej bieżący stan aplikacji (wcześniej te dane zwracane są do okna głównego przez akcesor) i przekazywany kontrolkom interfejsu użytkownika.

Dodatkową rolą komponentu jest zarejestrowanie uruchomionej aplikacji na serwerze. Rejestracja ta jest niezbędna z punktu widzenia komunikacji inicjowanej przez serwer, gdyż dostarcza informacji o lokalizacji usługi, przez którą serwer przesyła komunikaty do aplikacji użytkownika.

### 3.4. Komunikacja pomiędzy warstwami serwera i klienta

Za bezpośrednią komunikację pomiędzy aplikacjami okienkowymi a serwerem odpowiadają dwie usługi uruchomione po obu stronach. Po stronie serwera jest to usługa *UIPLService*, natomiast po stronie klienta *UserService*.

#### *3.4.1. Usługa serwera – UIPLService*

Usługa ta pełni wiele funkcji, które, ze względu na swoją specyfikę, można podzielić na kilka podgrup:

a) metody rejestracji nowych aplikacji klienckich,
b) metody dostępu do danych,
c) metody komunikacji z aplikacjami klienckimi.

Metody rejestracji wywoływane są przez aplikacje klientów w momencie uruchamiania tychże aplikacji. Jedną z ról serwera jest przechowywanie informacji o wszystkich uruchomionych aplikacjach klienta. W związku z tym, podczas rejestracji aplikacja użytkownika – a dokładniej usługa uruchomiona w procesie aplikacji użytkownika – przekazuje usłudze serwera komplet informacji niezbędnych do poprawnego wywoływania metod usługi klienta. Metody tej grupy pozwalają także na przeprowadzenie aktualizacji danych konfiguracyjnych, związanych z konkretną usługą klienta (np. zmiana numeru portu po stronie usługi użytkownika).

Druga grupa metod służy pośredniemu dostępowi do bazy danych. Wśród metod dostępu do danych należy wymienić zarówno metody odczytu danych, jak i ich modyfikacji (aktualizacji i usuwania).

Ostatnia grupa – w przeciwieństwie do dwóch poprzednich – służy do komunikacji od strony serwera w kierunku aplikacji okienkowych. Ponieważ serwer powinien mieć możliwość wymuszenia na aplikacji okienkowej, by ta odświeżyła dane, które wizualizuje (np. z powodu wygenerowania przez system nowej oceny), usługa serwera ma możliwość przesłania aplikacji okienkowej (a dokładniej – usłudze związanej z aplikacją okienkową) stosownego komunikatu.

#### *3.4.2. Usługa klienta – UserService*

Prosta usługa, uruchamiana w momencie startu aplikacji okienkowej. Jej jedynym zadaniem jest umożliwienie przyjmowania komunikatów ze strony serwera przez udostępnienie metody `PutMessage(string msg)`. W wyniku nadejścia wiadomości ze strony serwera, usługa generuje zdarzenie, którego obsługa odbywa się już w wyższej warstwie aplikacji okienkowej, jaką jest opisany już wcześniej *CommunicationAccessor*.

### *3.4.3. Komunikacja pomiędzy aplikacjami*

Realizacja komunikacji pomiędzy aplikacjami odbywa się za pośrednictwem serwera. Na chwilę obecną nie jest realizowana komunikacja bezpośrednia między poszczególnymi aplikacjami, a jedynie pomiędzy poszczególnymi typami aplikacji. Każda aplikacja posiada pewną funkcjonalność, która jest opisana zestawem znaczników. Komunikat wysyłany do pozostałych aplikacji posiada zamiast konkretnego adresata zestaw znaczników, opisujący, którego typu aplikacja powinna otrzymać dany komunikat. Ponieważ komunikacja odbywa się za pośrednictwem serwera, więc serwer, na podstawie znacznikowego opisu, decyduje, do których aplikacji ma zostać przesłany komunikat.

Ze względu na klasy i kontrolki graficzne, wykorzystane do budowy *MiSS Framework*, minimalnym wymaganiem programowym staje się .Net Framework 2.0. Oczywiście wymaganie to może zostać podniesione na etapie tworzenia aplikacji opartej na *MiSS Framework*.

## 4. Tworzenie i konfigurowanie aplikacji tworzonej na podstawie *MiSS Framework*

Tworzenie aplikacji okienkowej w architekturze *MiSS Framework* można podzielić na dwa zasadnicze etapy, które po części wiążą się z charakterem aplikacji. Pierwszy etap to tworzenie interfejsu użytkownika zarówno na poziomie kontrolek użytkownika (modułów), jak i na poziomie okna głównego aplikacji (integracja kontrolek użytkownika). Drugi etap to rozbudowa usług oraz akcesorów.

W przypadku gdy na podstawie *MiSS Framework* powstaje aplikacja niekorzystająca z serwera lub gdy z innych powodów nie przewiduje się komunikacji z aplikacją innej niż przez użytkownika, drugi etap konfigurowania aplikacji można pominąć.

### 4.1. Tworzenie interfejsu użytkownika

Zakładając, że kontrolka użytkownika (która oczywiście dziedziczy po klasie *UcObject*) została zaprojektowana (interfejs użytkownika) i wstępnie oprogramowana, aby kontrolka ta stała się elementem *MiSS Framework* należy dokonać następujących czynności:

Po stronie formy (okna głównego aplikacji):

1. Umieścić kontrolkę na formie (dziedziczącej po FObject).
2. Podpiąć po stronie formy obsługi trzech zdarzeń generowanych przez kontrolkę (aktywacja, deaktywacja i zmiana stanu).

3. W konstruktorze formy umieścić kod odpowiedzialny za poprawną inicjalizację kontrolki (sugerowane jest po inicjalizacji kontrolek wskazać kontrolkę aktywną).
4. Zapewnić poprawną analizę przetwarzania stanu nowo utworzonej kontrolki (zarówno samego stanu, jak i ewentualnych żądań dostępu do danych).

Po stronie kontrolki użytkownika:

1. Podpiąć dynamicznie pasek narzędzi kontrolki – ponieważ pasek narzędzi nie dziedziczy się w oczekiwany sposób („We intentionally decided to make ToolStrips read only on inherited forms to avoid potential problems due to visual inheritance. An inherited form does not know the properties of collections of its base form. Therefore if the collections are changed in the inherited form, potential repetition of data, and other problems occur. For this reason collections such as ToolStrip Items are read only.” [9]), należy zadbać o to, by kontrolka użytkownika posiadała swój własny pasek narzędzi. W klasie bazowej UcObject własność paska narzędzi zwraca null.
2. Wstawić dynamicznie menu kontekstowe – podobnie jak to ma miejsce w przypadku paska narzędzi, kontrolka użytkownika musi posiadać swoje własne menu kontekstowe, którego kopię zwraca przez własność.
3. Zapewnić poprawne udostępnianie stanu bieżącego kontrolki.
4. Zapewnić odpowiednie przetwarzanie przekazanego bieżącego stanu aplikacji.

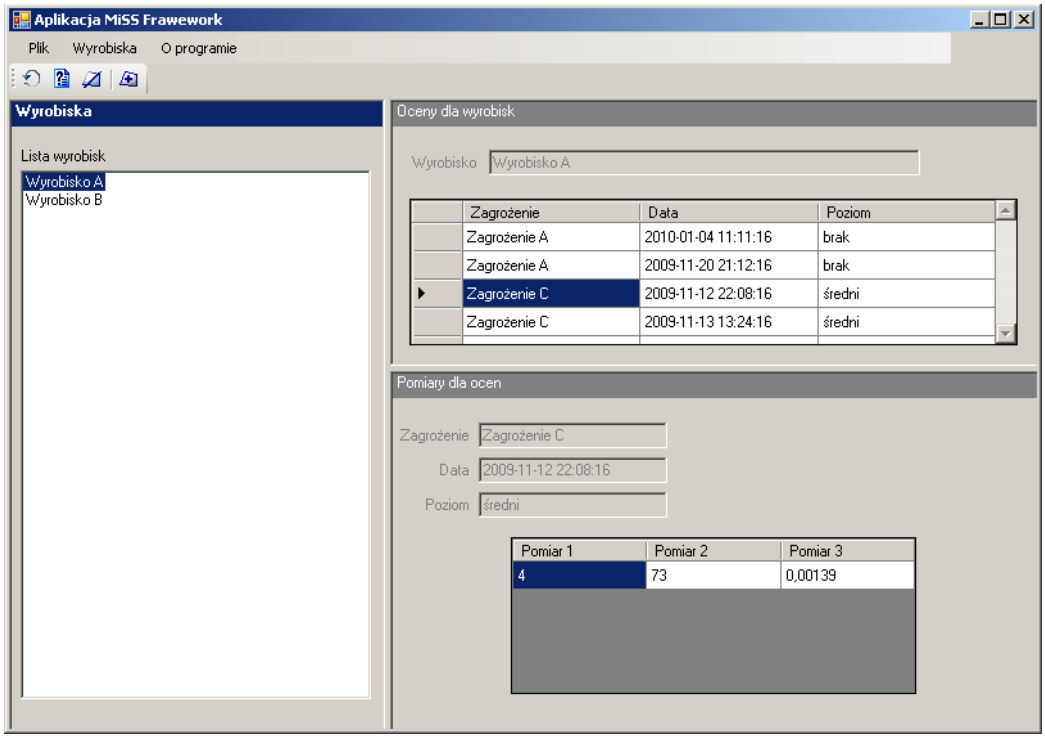
### 4.2. Rozbudowa usług serwera i klienta

Zakładając, że kontrolka użytkownika wymaga dostępu do danych (modyfikacja bądź jedynie odczyt), należy odpowiednio rozbudować usługi oraz *DataAcessor*. W przypadku pobierania danych, schemat rozbudowy jest następujący:

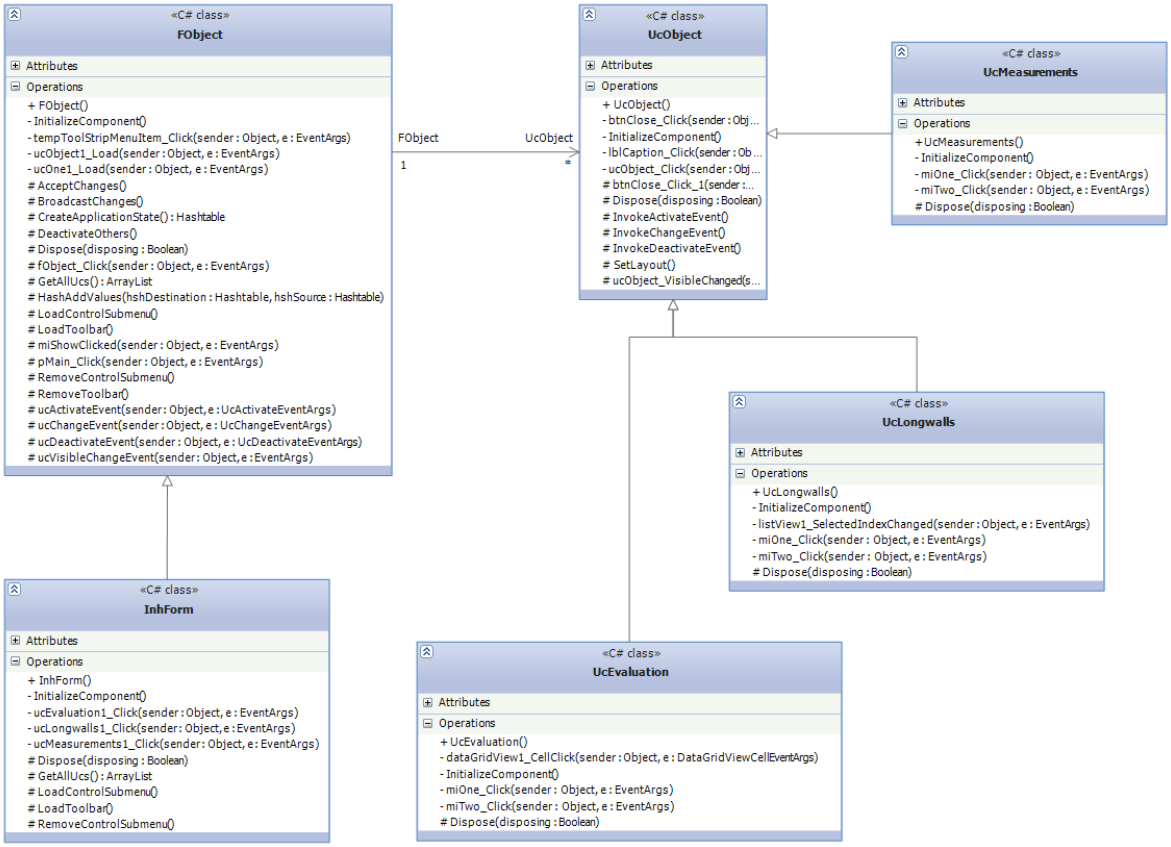
1. Wyposażyć usługę serwera w metodę pobierającą odpowiednie dane.
2. Wyposażyć *DataAccessor* w metodę zwracającą dane pobrane z serwera.
3. W stan kontrolki należy włączyć żądanie pobrania wskazanych danych.
4. W obsłudze stanu aplikacji (okno główne programu) należy, dla odpowiednich żądań poszczególnych kontrolek, wywołać odpowiednie metody *DataAcessora*.

## 5. Przykład zastosowania

Poniżej przedstawiono prostą aplikację okienkową opartą na *MiSS Framework*. Aplikacja (rys. 2) składa się z okna głównego, w którym osadzono trzy kontrolki użytkownika. Pierwsza z nich – opatrzona etykietą „Wyrobiska” – prezentuje listę wyrobisk istniejących w systemie.



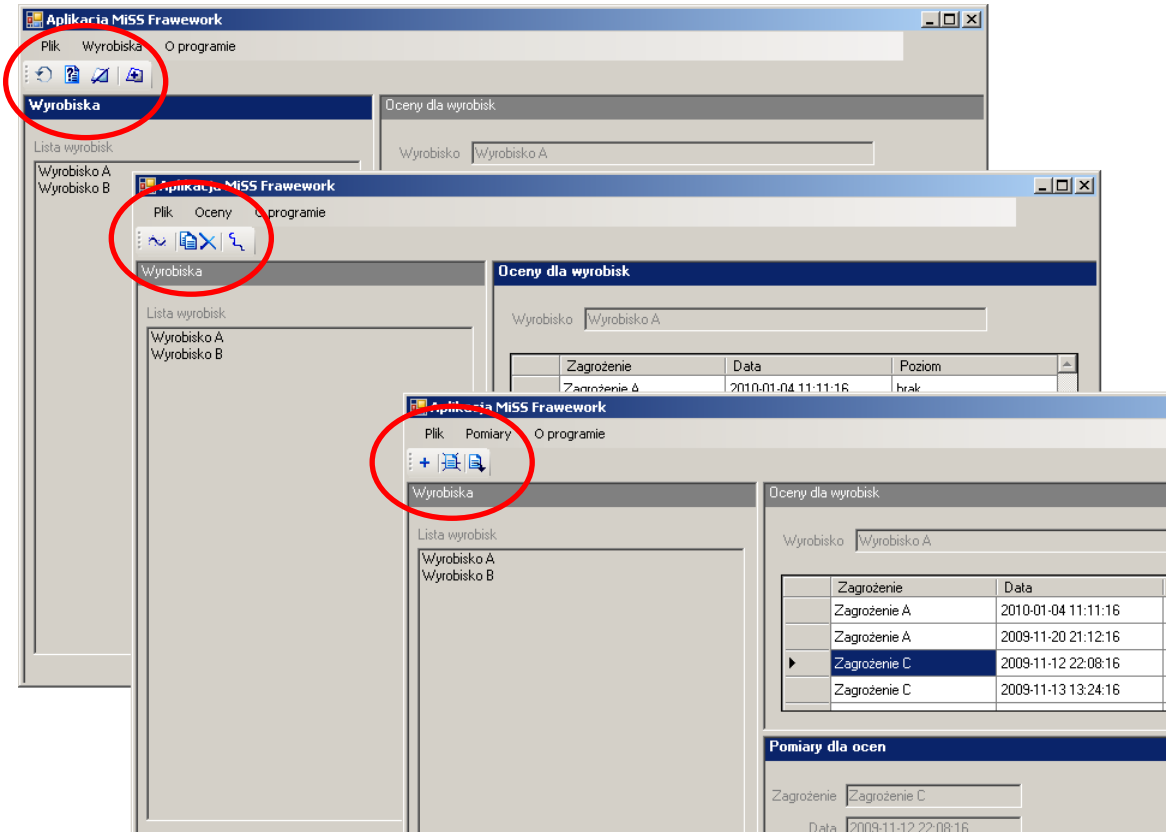

Rys. 2. Okno aplikacji opartej na *MiSS Framework*
Fig. 2. *MiSS Framework* based application window




Rys. 3. Diagram klas aplikacji opartej na *MiSS Framework*
Fig. 3. *MiSS Framework* based application class diagram

W zależności od aktualnie wybranego wyrobiska, druga kontrolka użytkownika – „Oceny dla wyrobisk" – prezentuje oceny trzech hipotetycznych zagrożeń wyznaczonych dla wskazanego wyrobiska. Każda ocena jest opisana przez zagrożenie, z którym jest związana, wyznaczony poziom zagrożenia oraz datę/czas wystawienia oceny. Po kliknięciu na dowolną z ocen, w kolejnej kontrolce – „Pomiary dla ocen" – przedstawiane są pomiary rzeczywiste, na podstawie których wypracowana została poszczególna ocena.

Implementacja oparta została na *MiSS Framework*. Diagram klas, wraz z klasami należącymi do frameworku, został przedstawiony na rys. 3. Zastosowanie *MiSS Framework* pozwala na sprawne skomunikowanie ze sobą poszczególnych kontrolek. Każda zmiana zaznaczonego elementu listy w kontrolce „Wyrobiska" czy też „Oceny dla wyrobisk", powoduje wywołanie zdarzenia zmiany stanu kontrolki. W reakcji na to okno główne aplikacji buduje aktualny stan aplikacji na podstawie stanu poszczególnych kontrolek. Przekazanie tych informacji kontrolkom powoduje z kolei, że każda z nich reaguje w przewidziany dla siebie sposób. I tak kontrolka „Ocena dla wyrobisk" wyświetla oceny wyznaczone dla aktualnie wybranego wyrobiska, a „Pomiary dla ocen" prezentują informacje, na podstawie których wyznaczono daną ocenę.




Rys. 4. Dynamicznie ładowany pasek narzędzi i menu (w zależności od aktywnej kontrolki)
Fig. 4. Dynamically loaded toolbar and menu (depending on the active control)

Ponadto, na rys. 4 przedstawiono kilka zrzutów ekranu aplikacji, wskazując przy tym, jak zmieniają się menu główne aplikacji oraz pasek narzędzi w zależności od aktywnej w danej chwili kontrolki.

## 6. Wnioski i perspektywy rozwoju

W pracy przedstawiono rozwiązanie wspomagające szybkie budowanie zestawu aplikacji okienkowych na podstawie wcześniej zaimplementowanych modułów. Stworzony na potrzeby realizacji projektu „Informatyczny system wspomagania kompleksowego zarządzania zagrożeniami górniczymi" zestaw klas zawiera zarówno komponenty interfejsu użytkownika, jak i elementy najwyższej warstwy systemu po stronie serwera, komponenty do komunikacji zarówno wewnątrz aplikacji okienkowej, jak i pomiędzy nią a serwerem.

Doświadczenia zebrane podczas pracy z frameworkiem pozwalają sformułować kilka niebanalnych wniosków, które powinny zostać uwzględnione podczas rozbudowywania funkcjonalności *MiSS Framework*. Jako najważniejsze elementy należałoby wymienić następujące:

1. Zagnieżdżone wywoływanie komponentów – na chwilę obecną żaden komponent wywołany z kontrolki dziedziczącej po *UcObject* nie staje się elementem frameworka. Oznacza to, że okno wywołane w ten sposób ani nie powiadomi innych okien (kontrolek użytkownika) o zmianie swojego stanu, ani też nie zareaguje na zmianę któregoś z nich. Należałoby wyposażyć *MiSS Framework* w nowy rodzaj kontenera analogicznego do *FObject*, który mógłby być wywoływany z kontrolek użytkownika, na nim umieszczane byłyby kontrolki dziedziczące po *UcObject*, sam zaś kontener posiadałby referencje do kontrolki, która go wywołała, i w ten sposób zapewniał powiązanie w ramach *MiSS Framework*.
2. Bezpośrednie adresowanie komunikatów – każdą zmianę stanu kontrolki można traktować jako komunikat przesłany do okna głównego aplikacji. Z kolei stan całej aplikacji jest komunikatem przesyłanym do każdej kontrolki – także i tej niezainteresowanej (broadcast). W celu usprawnienia komunikacji wewnątrz frameworku warto byłoby rozszerzyć komunikację o mechanizm powiadamiania tylko wybranych kontrolek, co wynikałoby z ich funkcjonalności.
3. Buforowanie danych – obecnie każde żądanie dostępu do danych, generowane przez kontrolkę użytkownika, przekłada się jednoznacznie na wywołanie odpowiedniej usługi serwera przez *DataAccessor*. Ponieważ poszczególne moduły mogą mieć potrzebę częstego pobierania danych bądź różne moduły mogą mieć potrzebę pobierania tych samych danych, warto byłoby rozważyć doimplementowanie mechanizmu pozwalającego na trzymanie kopii pobranych raz danych w aplikacji klienta do czasu ich zdezaktualizowania

się. Mechanizm ten powinien zostać dołączony do *DataAccessora*, jako że jest to klasa bezpośrednio zajmująca się pobieraniem danych dla aplikacji okienkowej. Wprowadzenie tej funkcjonalności mogłoby się przełożyć na szybkość działania aplikacji okienkowej oraz na częściowe odciążenie serwera.

Wypunktowane powyżej wnioski, a zarazem przyszłe cele rozwoju frameworka, wynikają bezpośrednio z doświadczeń zdobytych podczas tworzenia wersji prototypowej systemu MiSS. Tworzenie wersji produkcyjnej z pewnością odbędzie się o rozszerzoną wersję frameworku.

Artykuł napisano w ramach realizacji projektu *System wspomagania kompleksowego zarządzania zagrożeniami górniczymi*, UDA-POIG.01.03.01-24-048/08-00 z dnia 30.03.2008. Projekt realizowany w ramach Działania 1.3., Poddziałania 1.3.1., współfinansowany jest ze środków Europejskiego Funduszu Rozwoju Regionalnego w ramach Programu Operacyjnego Innowacyjna Gospodarka 2007-2013.

## BIBLIOGRAFIA

1. Bohosiewicz M., Jakubów A., Szarafiński M., Wasilewski S., Wojtas P.: Przegląd systemów monitorowania zagrożeń gazowych w polskich kopalniach. WUG: bezpieczeństwo pracy i ochrona środowiska w górnictwie, nr 10 (122), 2004.
2. Coad P., Yourdon E.: Analiza obiektowa. Oficyna Wydawnicza READ ME, Warszawa 1994.
3. Iwaszenko S., Sikora M., Michalak M., Długosz J.: Koncepcja systemu wspomagania kompleksowego zarządzania zagrożeniami górniczymi. Prace Naukowe GIG, Górnictwo i Środowisko, nr 4/2/2009, Katowice 2009, s. 83÷92.
4. Kabiesz J.: Charakterystyka skojarzonych zagrożeń górniczych w aspekcie ich oceny oraz doboru metod prewencji. Prace Naukowe GIG. Studia – Rozprawy – Monografie, nr 849, 2002.
5. Krzystanek Z., Dylong A., Wojtas P.: Monitorowanie środowiska w kopalni – system SMP-NT. Mechanizacja i Automatyzacja Górnictwa, nr 9 (404), 2004.
6. Sikora M.: System wspomagania pracy stacji geofizycznej – Hestia. Mechanizacja i Automatyzacja Górnictwa, nr 12/395, Katowice 2003.
7. Sikora M., Krzykawski D.: Zastosowanie metod eksploracji danych do analizy wydzielania się dwutlenku węgla w stacjach odwadniania kopalń węgla kamiennego. Mechanizacja i Automatyzacja Górnictwa, nr 6/413, Katowice 2005.

8. Zasady i zakres stosowania kompleksowej metody oceny stanu zagrożenia tąpaniami w zakładach górniczych wydobywających węgiel kamienny. Główny Instytut Górnictwa, Seria Instrukcje, nr 1, Katowice 1996.
9. https://connect.microsoft.com/VisualStudio/feedback/details/552017/visual-inheritance-of-a-toolstrip.

Recenzenci: Dr hab. inż. Adam Pelikant, prof. Pol. Łódzkiej
Dr inż. Łukasz Wyciślik

Wpłynęło do Redakcji 31 stycznia 2011 r.

**Abstract**

This article describes the solution for advanced window applications development. It defines a set of classes, interfaces and services called *MiSS Framework*. The name comes from the name of mining threats complex management support system: *Mine Security System*. The analysis of the system functionality (multiple user groups, common application architecture, communication between modules within the application and between applications etc.) led to the definition of a framework for user interface building.

This article contains definitions of *MiSS Framework* components (interfaces, classes, frazes and services), description of *MiSS Framework* based application development and example of simple application based on this technology.

**Adresy**

Józef KABIESZ: Główny Instytut Górnictwa, Plac Gwarków 1, 40-166 Katowice, Polska, Jozef.Kabiesz@gig.eu.
Sebastian IWASZENKO: Główny Instytut Górnictwa, Plac Gwarków 1, 40-166 Katowice, Polska, Sebastian.Iwaszenko@gig.eu.
Marcin MICHALAK: Główny Instytut Górnictwa, Plac Gwarków 1, 40-166 Katowice, Polska, Marcin.Michalak@gig.eu.