Złożenie pracy online:
2009-03-14 15:19:13
Kod pracy:
1065
Kod załącznika:
909

Artur Stawiarski
(nr albumu: 9438*INF/LIC)

Praca licencjacka

Komunikator internetowy z obsługą wtyczek

Instant messaging application with plugins support

Wydział: Informatyki

Kierunek: Informatyka

Specjalność: inżynieria oprogramowania

Promotor: dr Włodzimierz Moczurad

Nowy Sącz, 2013

Spis treści

Wstęp

Obecnie wciąż najpopularniejszą formą komunikacji w sieci Internet są czaty prowadzone przy użyciu programów zwanych komunikatorami. Jest bardzo dużo aplikacji tego typu, często pozwalających na komunikację więcej niż jednym protokołem. Niestety, liczba takich, które pozwalały by rozszerzać swoje możliwości za pomocą dodatkowych wtyczek jest bardzo mała. Nawet jeśli niektóre z nich mają budowę modułową, to albo nie umożliwiają użytkownikom tworzenia własnych dodatków albo ich API jest zbyt skomplikowane lub technologiczne przestarzałe. Jednym z lepszych komunikatorów, który umożliwia tworzenie własnych wtyczek jest Konnekt. Pomimo tego, iż kod samego komunikatora jest zamknięty i od dłuższego czasu nie rozwijany, a wtyczki pisze się w już nieco archaicznym Visual C++ / MFC powstało do niego bardzo dużo rozszerzeń.

Mammoth, jak nazwałem ten projekt jest platformą w oparciu o którą chcę stworzyć komunikator o idei podobnej do Konnekta. Aplikacja ta sama w sobie jest bezużyteczna, ale dzięki łatwemu tworzeniu wtyczek i dostarczeniu wszelkich mechanizmów potrzebnych komunikatorowi może stać się ciekawą pozycją dla entuzjastów, a być może i alternatywą dla zwykłych użytkowników. Dodatkowym atutem jest udostępnienie na zewnątrz pełnych źródeł, a dzięki temu, że jest napisana w popularnym języku C# na platformie .NET może się bardzo szybko rozwinąć. Obecne API jest wciąż ubogie, ale z czasem będzie roszerzane, aby możliwie najmniej ograniczało wyobraźnię deweloperów wtyczek.

I. Techniki i wzorce projektowe

Wszystkie projekty wchodzące w skład pracy (Komunikator i wtyczki) zostały napisane w C# .NET 2.0 za pomocą Microsoft Visual Studio 2008.

Najczęściej stosowanym wzorcem projektowym w mojej pracy jest Singleton. Polega on na zapewnieniu istnienia tylko jednej instancji danej klasy w całej aplikacji. W językach z rodziny C (C, C++, C#) polega to na ukryciu konstruktora obiektu i udostępnieniu statycznej metody (lub właściwości - C#) zwracającej skonstruowany obiekt. W językach umożliwiających zwracanie innych obiektów w konstruktorze (np. PHP 5) można to zrobić niejawnie.

Ze względu wielowątkowość aplikacji należało zastosować pewne techniki synchronizacji i ochrony zasobów. Język C# i framework .NET dostarczają wiele przydatnych do tego mechanizmów i klas. Jedną z zastosowanych jest monitor. Pozwala on na zablokowanie dostępu do bloku kodu na referencji do obiektu. Dopóki nie zostanie zwolniony, żaden wątek nie może do takiego bloku wejść. Jeżeli wywołanie i zamknięcie monitora znajduje się w tej samej funkcji to można skrócić zapis wykorzystując słowo lock:

```
lock(obj)
{
   //operacje chronione
}
```

jest rozwijany przez kompilator do

```
Monitor.Enter(obj);
try
{
   //operacje chronione
}
finally
{
   Monitor.Exit(obj);
}
```

Kontrolki w .NET dostarczają również mechanizm zabezpieczający przed odwoływaniem się do komponentów utworzonych w innym wątku niż aktualne wywołanie. Jedną z technik pozwalających to wykorzystać jest przekazywanie wywołania metody do wątku, który jest właścicielem danej kontrolki.

```
private delegate void SomeMethodCallback();
public void SomeMethod()
{
   if (InvokeRequired)
      Invoke(new SomeMethodCallback(SomeMethod));
   else
   //operacje na komponentach
}
```

Atrybut ***InvokeRequired*** zwraca ***true*** jeżeli wywołanie metody nie pochodzi z wątku, który jest właścicielem kontrolki. Metoda ***Invoke*** jest synchronicznym wywołaniem tej metody z odpowiedniego wątku. Jeżeli nie zależy nam na zwracanym wyniku metody lub nie chcemy aby aplikacja czekała na jej wykonanie możemy zamiast ***Invoke*** użyć ***BeginInvoke*** aby wywołać metodę asynchronicznie.

Utworzenie aplikacji w C# .NET obsługującej wtyczki nie jest bardzo skomplikowane. Poniżej znajduje się przykład, który został trochę uproszczony w stosunku do rozwiązania zastosowanego w kodzie komunikatora.

```
foreach (string fileName in Directory.GetFiles(FolderWtyczek))
{
   FileInfo file = new FileInfo(fileName);
   if (file.Extension.Equals(".dll"))
   {
      Assembly assembly = Assembly.LoadFrom(fileName);
      //pętla po wszystkich typach znajdujacych się w bibliotece .dll
      foreach (Type type in pluginAssembly.GetTypes())
      {
         if(!type.IsPublic || type.IsAbstract) continue;
         //jezeli można utworzyć instancję obiektu tej klasy
         Wtyczka wtyczka = (Wtyczka)Activator.CreateInstance(type);
         //operacje na obiekcie wtyczki
      }
   }
}
```

II. Obsługa wtyczek

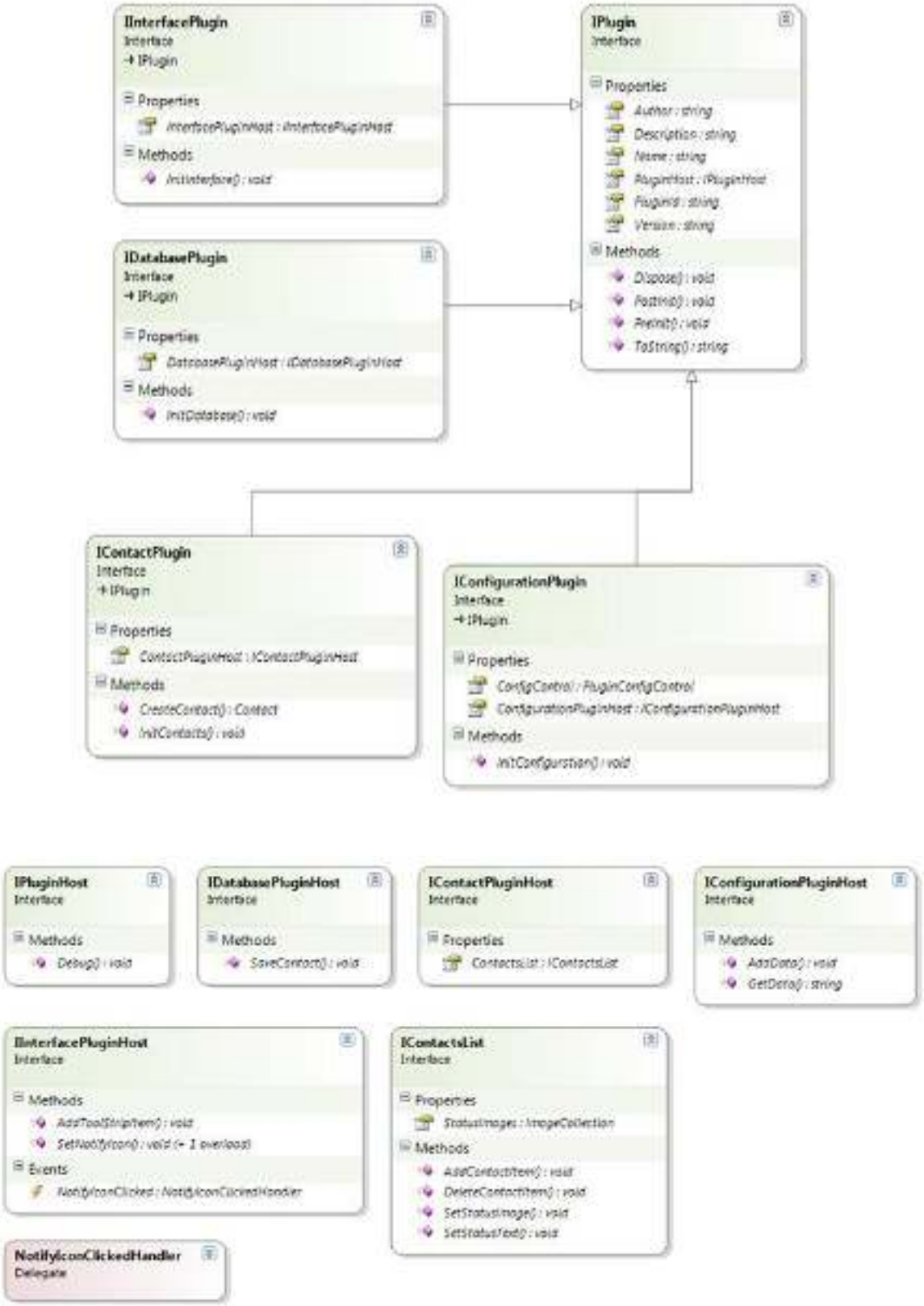
Z projektowego punktu widzenia obsługa wtyczek polega na opisaniu pewnych interfejsów przeznaczonych do implementowania przez wtyczki oraz tych opisujących elementy zaimplementowane w aplikacji. Interfejsy te, wraz z przydatnymi gotowymi klasami, powinny zostać wydzielone do osobnej biblioteki wykorzystywanej przez wtyczki i aplikację. Aby komunikator był jak najbardziej rozszerzalny interfejsy powinny zawierać jak najwięcej następujących elementów:

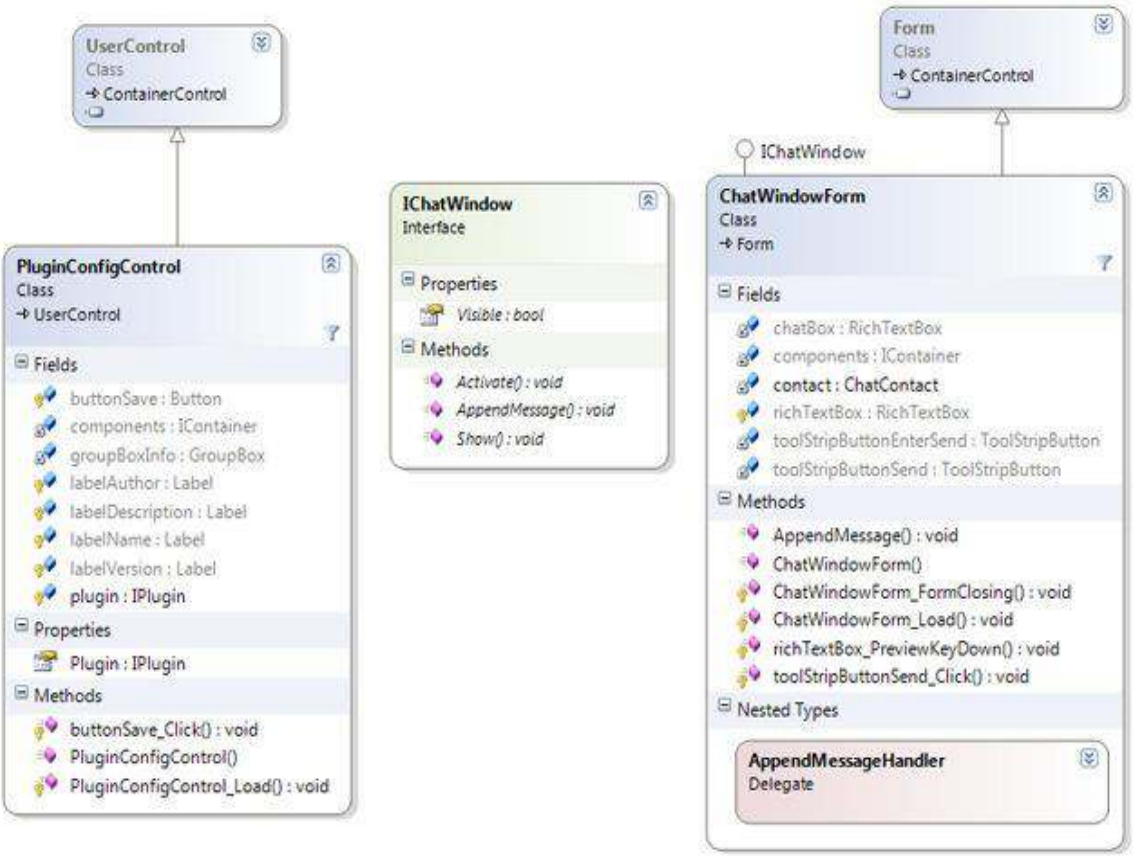
- metody rejestrujące komponenty takie jak dodatkowe menu, formatki itp.
- zdarzenia (`event`), do których wtyczka może się podpiąć
- metody zapewniające pewne funkcjonalności (operacje na lokalnej bazie, odświeżenie listy kontaktów itp.)

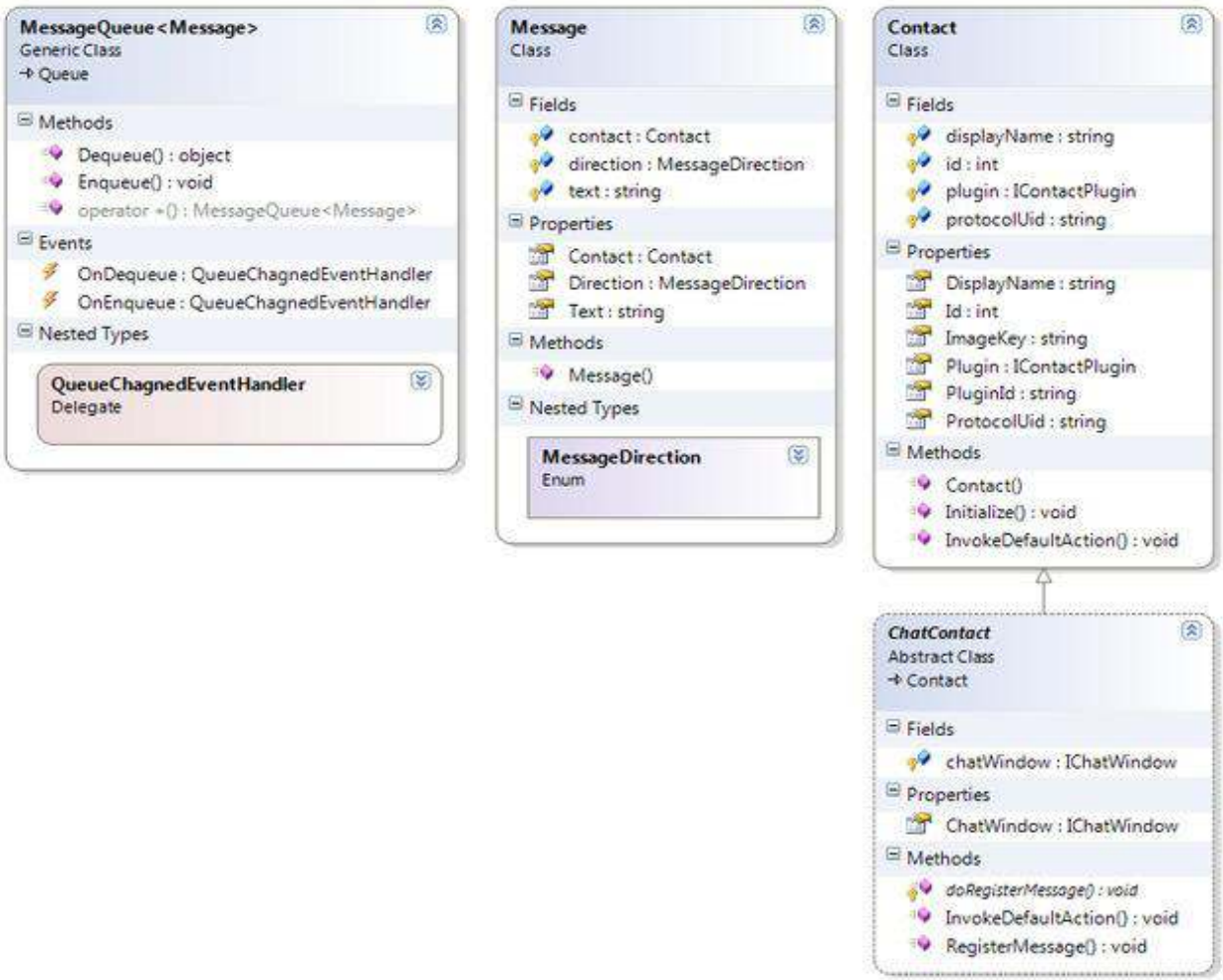
Przebieg działania aplikacji wygląda następująco:

1. Utworzenie instancji głownego obiektu komunikatora i jego inicjalizacja
2. Wczytanie wtyczek (pętla dla wszystkich .dll w katalogu z wtyczkami)
    a) Wczytanie biblioteki
    b) Odnalezenie odpowiednich klas wtyczek implementujących odpowiednie interfejsy
    c) Utworzenie instancji wtyczek i ich inicjalizacja
3. Rozpoczęcie działania aplikacji – w tym momencie wczytki już mają zarejestrowane swoje komponenty i działają praktycznie niezależnie od platformy wysyłając lub otrzymując od niej jedynie komunikaty na które mogą reagować.

Diagram klas biblioteki




UserControl
Class
ContainerControl
PluginConfigControl
Class
UserControl
Fields
buttonSave : Button
components : IContainer
groupBoxInfo : GroupBox
labelAuthor : Label
labelDescription : Label
labelName : Label
labelVersion : Label
plugin : IPlugin
Properties
Plugin : IPlugin
Methods
buttonSave_Click() : void
PluginConfigControl()
PluginConfigControl_Load() : void
IChatWindow
Interface
Properties
Visible : bool
Methods
Activate() : void
AppendMessage() : void
Show() : void
Form
Class
ContainerControl
IChatWindow
ChatWindowForm
Class
Form
Fields
chatBox : RichTextBox
components : IContainer
contact : ChatContact
richTextBox : RichTextBox
toolStripButtonEnterSend : ToolStripButton
toolStripButtonSend : ToolStripButton
Methods
AppendMessage() : void
ChatWindowForm()
ChatWindowForm_FormClosing() : void
ChatWindowForm_Load() : void
richTextBox_PreviewKeyDown() : void
toolStripButtonSend_Click() : void
Nested Types
AppendMessageHandler
Delegate



Tworzenie wtyczek

Oprogramowanie nowej wtyczki współpracującej z komunikatorem nie powinno sprawiać problemu. Dzięki zastosowaniu technologii .NET wtyczka może zostać napisana w dowolnym języku z tej platformy (C#, J#, VB#). W zależności od tego, co nasza wtyczka ma robić musimy zaimplementować odpowiednie interfejsy. Interfejs ***`IPlugin`*** jest podstawowy i każda wtyczka powinna implementować co najmniej jego. Dla wygody, interfejsy szczegółowe dziedziczą z podstawowego[1]. Do zarejestrowanych wtyczek zostaną podłączone przy ładowaniu odpowiednie hosty, na których będzie można wywoływać metody, podpinać się do zdarzeń czy też podpinać własne komponenty[2]. Dodatkowo, biblioteka dostarcza pewne klasy pomocnicze do bezpośredniego wykorzystania lub dziedziczenia. Dzięki nim będzie można utworzyć nową wtyczkę znacznie szybciej[3].

1 Patrz kod źródłowy: PluginInterfaces.cs
2 Patrz kod źródłowy: HostInterfaces.cs
3 Patrz kod źródłowy – klasy: Contact, Message, MessageQueue, ChatWindowForm, PluginConfigControl

Platforma komunikatora umożliwia aktualnie następujące operacje (poza inicjalizacją):

- Zdefiniowanie własnego rodzaju kontaktu i sposób jego tworzenia z zapewnionym zapisem do bazy, wtyczywaniem, inicjalizacją, prezentacją na liście kontaktów, uruchamianiem domyślnej akcji itp.; własna klasa zależnie od potrzeb może dziedziczyć z ***Contact*** , lub ***ChatContact*** jeżeli chcemy mieć zapewniony mechanizmy zwykle przydane przy implementacji protokołów komunikacji
- Ustawienie statusów i zmiana ikon kontaktów
- Ustawianie ikony aplikacji na pasku powiadomień
- Podpinanie się pod wydarzenie kliknięcie w ikonę na pasku powiadomień
- Rejestrację własnych lub podmianę istniejących ikon innych wtyczek
- Zapisywanie i odczytywanie danych z lokalnej konfiguracji
- Wykorzystanie istniejącego okna rozmowy, odziedziczenie z niego własnego lub stworzenie całkowicie odmiennego poprzez implementację odpowiedniego interfejsu
- Wykorzystanie istniejących klas ***Message*** i ***MessageQueue*** bardzo przydatnych przy protokołach komunikacji
- Wysyłanie komunikatów do wspólnej konsoli błędów

Poniższy kod jest przykładem najprostszej wtyczki, która zostanie wykryta i załadowana przez komunikator.

```
public class MojPlugin : IPlugin
{
   public String PluginId { get { return "moj_plugin"; } }
   public String Name { get { return "Moja Wtyczka"; } }
   public String Description { get { return "Moja pierwsza Wtyczka"; }}
   public String Author { get { return "Artur Stawiarski"; } }
   public String Version { get { return "0.1 alpha"; } }
   public override string ToString() { return Name; }

   private IPluginHost pluginHost;
   public IPluginHost PluginHost
   {
      get { return pluginHost; }
      set { pluginHost = value; }
   }
```

```
   public void PreInit()
   {
      //inicjalizacja wtyczki
   }

   public void PostInit()
   {
      //wykonanie dodatkowych operacji po pozostalych inicjalizacjach
   }

   public void Dispose()
   {
      //zamykanie wtyczki
   }
}
```

Jeżeli nasza wtyczka miałaby dostarczać nowy rodzaj kontaktu do listy użytkowników musi implementować interfejs ***IContactPlugin***. Zgodnie z opisem tego interfejsu do przykładu prostej wtyczki musimy dodać następujący fragment (wewnątrz klasy):

```
private IContactPluginHost contactPluginHost;
public IContactPluginHost ContactPluginHost
{
   get { return contactPluginHost; }
   set { contactPluginHost = value; }
}

//inicjalizacja czesci odpowiedzialnej za kontakty
public void InitContacts()
{
   //mozna tutaj np. zarejestrowac ikone dla naszego kontaktu
   ContactPluginHost.ContactsList.StatusImages.Add(
      "moj_plugin_ico", Properties.Resources.moja_ikona);
}

//metoda tworzaca i inicjalizujaca nowy kontakt naszej wtyczki
public Contact CreateContact()
{
   MojPluginContact contact = new MojPluginContact();
   contact.Plugin = this;
```

```
    return contact;
}
```

Jak widać w metodzie ***CreateContact***, tworzymy obiekt klasy naszego kontaktu i przypisujemy mu wtyczkę odpowiedzialną za jego zarządzanie. Oczywiście wcześniej musi istnieć taka klasa, która również nie musi być skomplikowana. Jedną z najważniejszych rzeczy, o których należy pamiętać przy tworzeniu wtyczki jest unikalność jej identyfikatora – ***PluginId***.

```
public class MojPluginContact : PluginInterface.Contact
{
   public override String ImageKey { get { return "moj_plugin_ico"; } }
   public override String PluginId { get { return "moj_plugin"; }

   //ta metoda zostaje uruchomiona podczas dwu-kliku
   //na kontakcie na liscie
   override public void InvokeDefaultAction()
   {
      Plugin.ContactPluginHost.ContactsList.SetStatusText(
         this, „wywolano domyslna akcje”);
   }

   //inicjalizacja kontaktu tuz po zaladowaniu z lokalnej bazy
   public override void Initialize()
   {
      base.Initialize();
      //dodatkowe inicjalizacje, np. zlecenie czego wtyczce this.Plugin
   }
}
```

III. Projekty

Komunikator

Jest to część właściwa projektu. Z aplikacji tej samej w sobie nie ma większego pożytku, jest jedynie platformą obsługującą wtyczki i udostępniającą możliwie najwięcej mechanizmów przydanych komunikatorowi. Zawiera prosty interfejs użytkownika, możliwość dodania i edycji zwykłego kontaktu i kilka komponentów. Dopiero dołączenie do niego dodatków w postaci wtyczek może sprawić, iż będzie użyteczny.

Wewnątrz tego projektu zostały zaimplementowane wszystkie interfejsy hostów wtyczek. Dzięki temu każda wtyczka poprzez implementację interfejsu wtyczki ma bezpośredni dostęp do elementów platformy. Oczywiście dostęp ten jest ograniczony do metod, zdarzeń, pól itp. zdefiniowanych w odpowiednich interfejsach.

Aplikacja składa się z nastepujących elementów:

1. PluginServices

   Klasa do zarządzania wtyczkami implementująca wzorzec Singleton. Wczytuje biblioteki z katalogu z wtyczkami, tworzy ich instancje w aplikacji oraz nimi zarządza (inicjalicjalizuje, udostępnia metody je zwracające, zamyka)

2. Database

   Klasa umożliwiająca komunikację z lokalną bazą danych. Również jest singletonem i udostępnia metody wykonujące konkretne zadania oraz umożliwiające wywoływanie dowolnych zapytań SQL.

3. Configuration

   Kolejny singleton tym razem obsługujący konfigurację aplikacji. Dzięki mechanizmowi serializacji klas w .NET stworzenie konfiguracji opartej o XML jest bardzo proste i wygodne.

4. MainWindow

   Element główny aplikacji. Na wzór wcześniej wspomnianego Konnekta ma maksymalnie prosty interfejs, który w przyszłości będzie mógł być modyfikowany za pomocą tzw. skórek. Z elementów wizualnych zawiera listę kontaktów oraz pasek narzędzi do którego wtyczki mogą dodawać swoje przyciski i menu. Od strony programistycznej implementuje większość interfejsów hostów wtyczek. Udostępnia metody do ustawiania ikony na pasku powiadomień, do wcześniej wspomnianej rejestracji elementów na pasku narzędzi oraz event kliknięcia w ikonę powiadomienia. Ta klasa podczas ładowania inicjalizuje obiekt

zarządzający wtyczkami oraz konfigurację.

5. ContactsList

   Do tego komponentu mają dostęp wtyczki implementujące interfejs ***IContactPlugin***. Udostępnia on metody do dodawania i usuwania kontaktów oraz ustawiania ich opisów i obrazków na liście. Jest on również odpowiedzialny za odpowiednie wywołanie domyślnej akcji przypisanej do danego kontaktu.

6. ConfigurationForm

   Panel konfiguracyjny jest miejscem w którym można konfigurować wszystkie dostępne opcje komunikatora i wtyczek. Jeżeli tylko wtyczka implementuje interfejs ***IConfigurationPlugin*** to jej formatka konfiguracyjna znajdzie się właśnie tam. Jedyne co należy zrobić to odziedziczyć komponent ***PluginConfigControl*** i dodać do niego swoje pola konfiguracyjne. W przeciwnym wypadku jedyne co się będzie wyświetlać to opis wtyczki.

7. EditContactForm

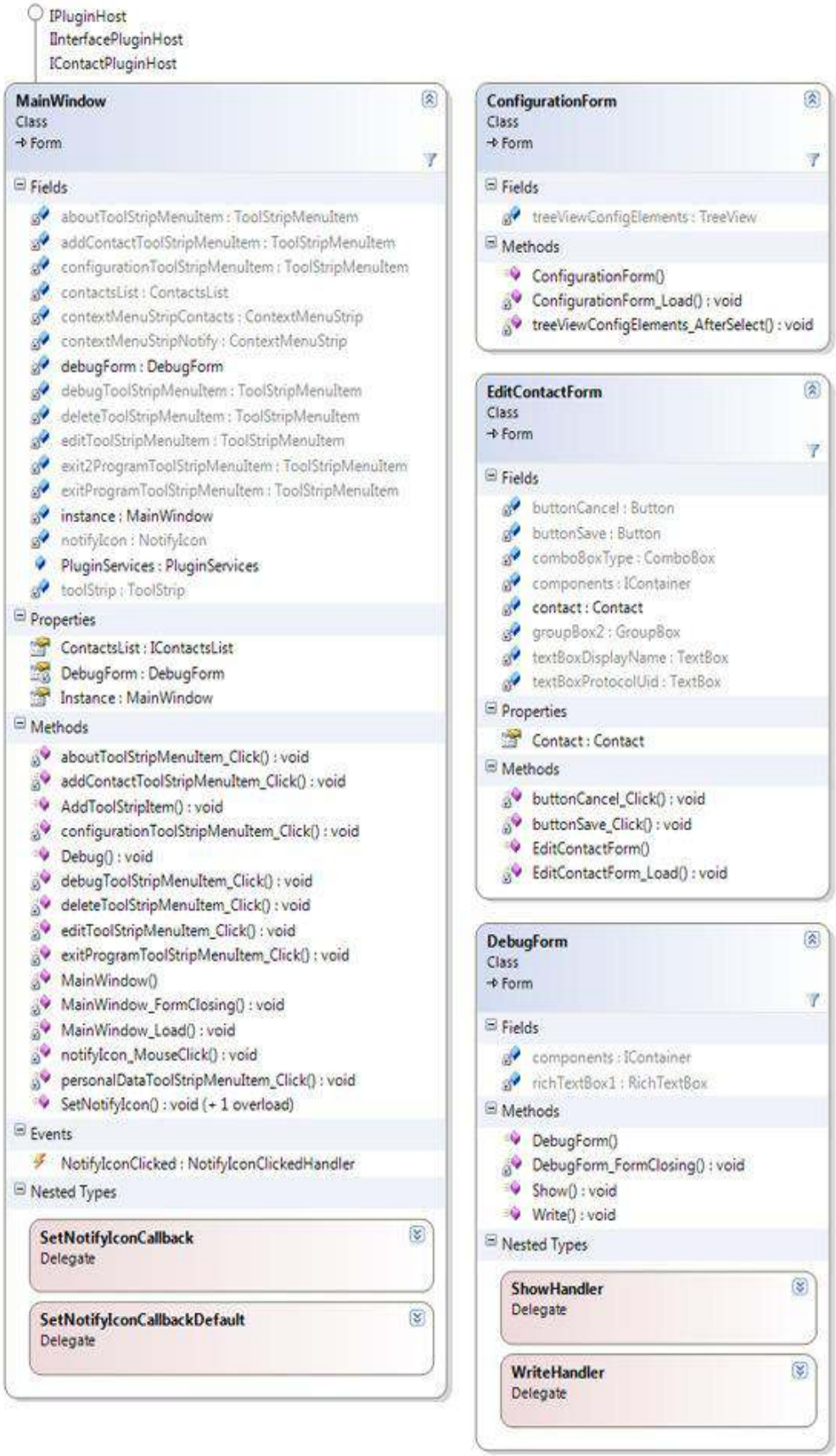
   Jest to stosunkowo prosta formatka edycji/dodawania kontaktu do listy. Ze względu na to, iż każda wtyczka implementująca interfejs ***IContactPlugin*** może dostarczyć własny rodzaj kontaktu (własną klasę dziedziczącą po ***Contact***) na liście wyboru typu kontaktu pojawiają się wszystkie wtyczki tego typu.

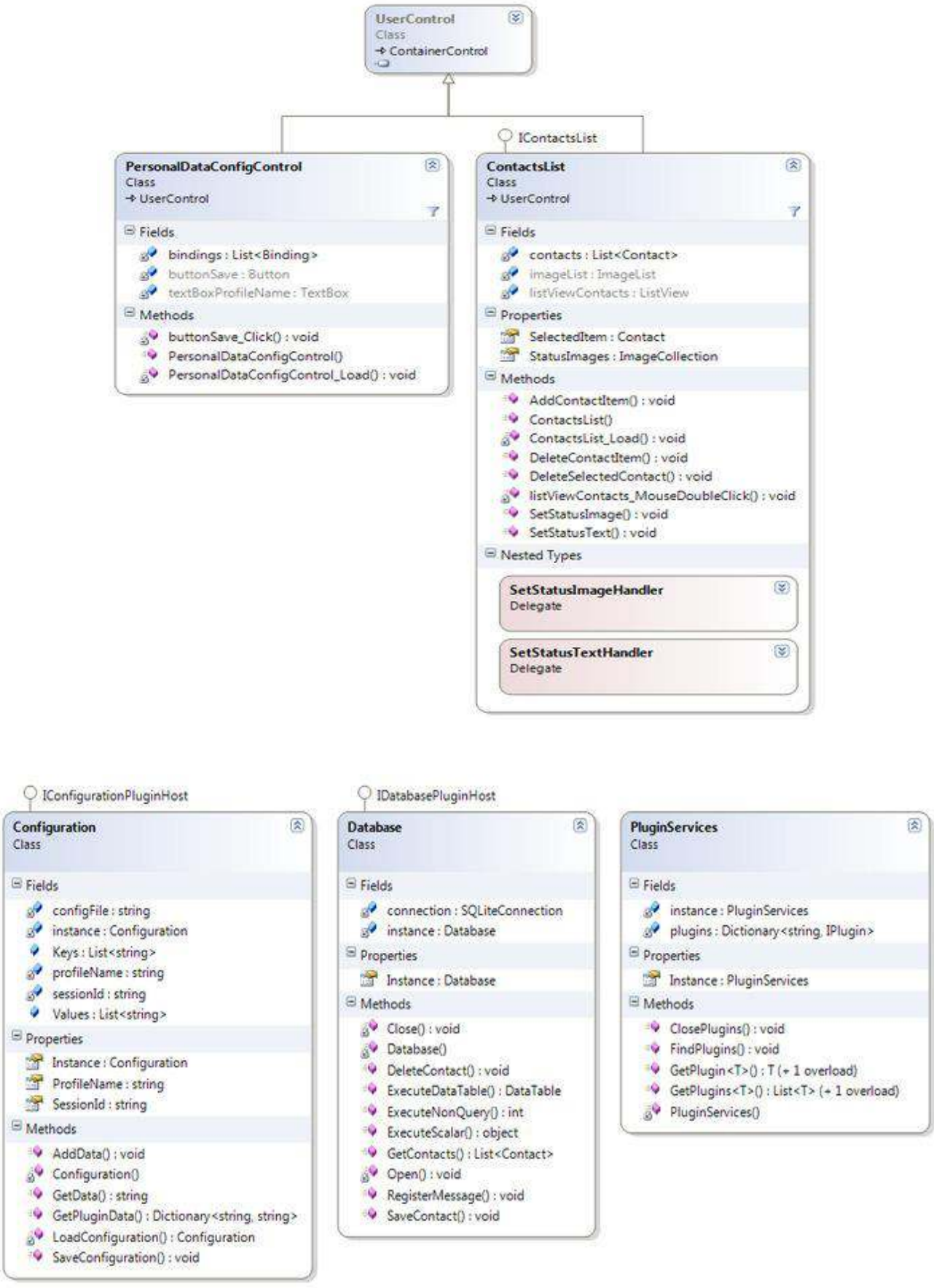
8. DebugForm

   Ta formatka zawiera jedynie duże pole tekstowe, które jest swego rodzaju logiem. Tutaj pojawiają się wszystkie informacje o działaniu wtyczki, które autor chciał pokazać (np. komunikaty o błędach). Ze względu na możliwość wywoływania jej metod w różnych wątkach, wszystkie jej uodstępnione metody zostały zabezpieczone sposobem opisanym w pierwszym rozdziale mojej pracy.

9. AboutForm

   Formatka z podstawowymi informacjami o programie.


IPluginHost
IInterfacePluginHost
IContactPluginHost
MainWindow
Class
→ Form
Fields
aboutToolStripMenuItem : ToolStripMenuItem
addContactToolStripMenuItem : ToolStripMenuItem
configurationToolStripMenuItem : ToolStripMenuItem
contactsList : ContactsList
contextMenuStripContacts : ContextMenuStrip
contextMenuStripNotify : ContextMenuStrip
debugForm : DebugForm
debugToolStripMenuItem : ToolStripMenuItem
deleteToolStripMenuItem : ToolStripMenuItem
editToolStripMenuItem : ToolStripMenuItem
exit2ProgramToolStripMenuItem : ToolStripMenuItem
exitProgramToolStripMenuItem : ToolStripMenuItem
instance : MainWindow
notifyIcon : NotifyIcon
PluginServices : PluginServices
toolStrip : ToolStrip
Properties
ContactsList : IContactsList
DebugForm : DebugForm
Instance : MainWindow
Methods
aboutToolStripMenuItem_Click() : void
addContactToolStripMenuItem_Click() : void
AddToolStripItem() : void
configurationToolStripMenuItem_Click() : void
Debug() : void
debugToolStripMenuItem_Click() : void
deleteToolStripMenuItem_Click() : void
editToolStripMenuItem_Click() : void
exitProgramToolStripMenuItem_Click() : void
MainWindow()
MainWindow_FormClosing() : void
MainWindow_Load() : void
notifyIcon_MouseClick() : void
personalDataToolStripMenuItem_Click() : void
SetNotifyIcon() : void (+ 1 overload)
Events
NotifyIconClicked : NotifyIconClickedHandler
Nested Types
SetNotifyIconCallback
Delegate
SetNotifyIconCallbackDefault
Delegate
ConfigurationForm
Class
→ Form
Fields
treeViewConfigElements : TreeView
Methods
ConfigurationForm()
ConfigurationForm_Load() : void
treeViewConfigElements_AfterSelect() : void
EditContactForm
Class
→ Form
Fields
buttonCancel : Button
buttonSave : Button
comboBoxType : ComboBox
components : IContainer
contact : Contact
groupBox2 : GroupBox
textBoxDisplayName : TextBox
textBoxProtocolUid : TextBox
Properties
Contact : Contact
Methods
buttonCancel_Click() : void
buttonSave_Click() : void
EditContactForm()
EditContactForm_Load() : void
DebugForm
Class
→ Form
Fields
components : IContainer
richTextBox1 : RichTextBox
Methods
DebugForm()
DebugForm_FormClosing() : void
Show() : void
Write() : void
Nested Types
ShowHandler
Delegate
WriteHandler
Delegate


UserControl
Class
ContainerControl
IContactsList
PersonalDataConfigControl
Class
UserControl
Fields
bindings : List<Binding>
buttonSave : Button
textBoxProfileName : TextBox
Methods
buttonSave_Click() : void
PersonalDataConfigControl()
PersonalDataConfigControl_Load() : void
ContactsList
Class
UserControl
Fields
contacts : List<Contact>
imageList : ImageList
listViewContacts : ListView
Properties
SelectedItem : Contact
StatusImages : ImageCollection
Methods
AddContactItem() : void
ContactsList()
ContactsList_Load() : void
DeleteContactItem() : void
DeleteSelectedContact() : void
listViewContacts_MouseDoubleClick() : void
SetStatusImage() : void
SetStatusText() : void
Nested Types
SetStatusImageHandler
Delegate
SetStatusTextHandler
Delegate
IConfigurationPluginHost
Configuration
Class
Fields
configFile : string
instance : Configuration
Keys : List<string>
profileName : string
sessionId : string
Values : List<string>
Properties
Instance : Configuration
ProfileName : string
SessionId : string
Methods
AddData() : void
Configuration()
GetData() : string
GetPluginData() : Dictionary<string, string>
LoadConfiguration() : Configuration
SaveConfiguration() : void
IDatabasePluginHost
Database
Class
Fields
connection : SQLiteConnection
instance : Database
Properties
Instance : Database
Methods
Close() : void
Database()
DeleteContact() : void
ExecuteDataTable() : DataTable
ExecuteNonQuery() : int
ExecuteScalar() : object
GetContacts() : List<Contact>
Open() : void
RegisterMessage() : void
SaveContact() : void
PluginServices
Class
Fields
instance : PluginServices
plugins : Dictionary<string, IPlugin>
Properties
Instance : PluginServices
Methods
ClosePlugins() : void
FindPlugins() : void
GetPlugin<T>() : T (+ 1 overload)
GetPlugins<T>() : List<T> (+ 1 overload)
PluginServices()

Wtyczka Gadu-Gadu

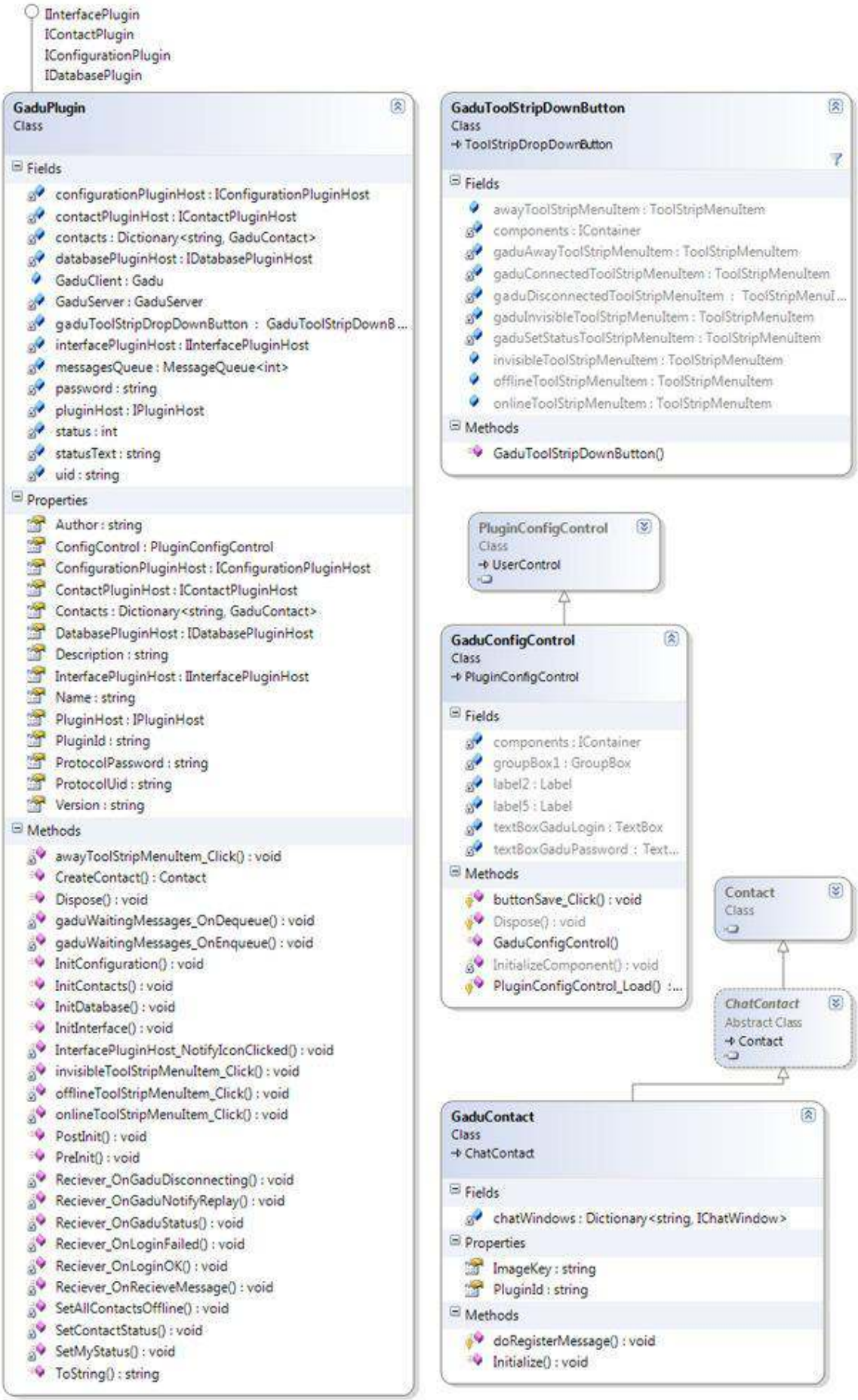
Wtyczka ta składa się z jednego obiektu głównego, implementującego następujące interfejsy:

- ***IInterfacePlugin***
- ***IContactPlugin***
- ***IConfigurationPlugin***
- ***IDatabasePlugin***

Jak widać jest już bardziej rozbudowana i wykorzystuje większość aktualnie dostępnych możliwości:

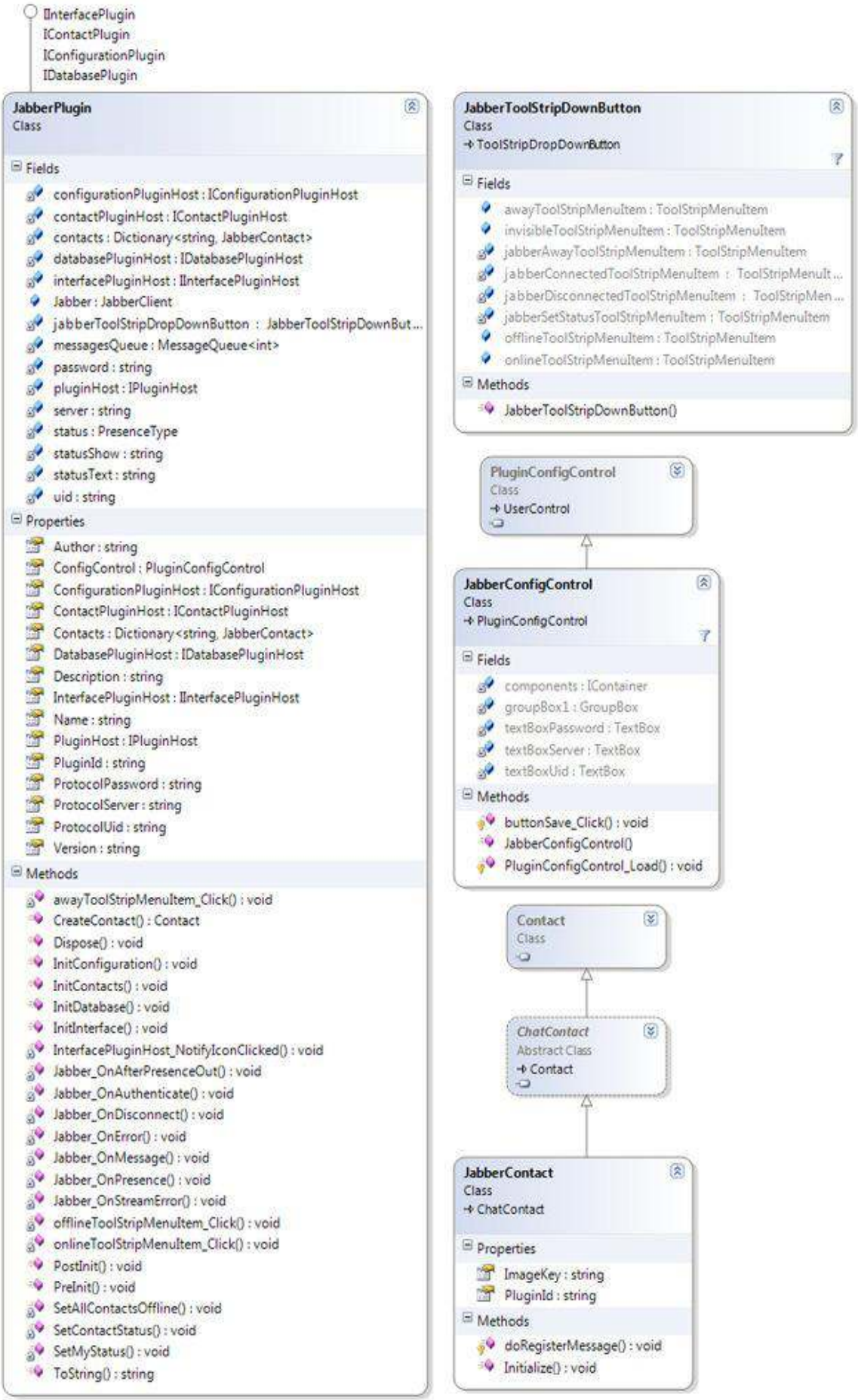
- Dodaje menu na pasku narzędzi
- Definiuje własny rodzaj kontaktów
- Przechowuje dane dotyczące obsługi protokołu w konfiguracji komunikatora
- Posługuje się lokalną bazą danych do zapisu danych kontaktu

Do komunikacji protokołem Gadu-Gadu została wykorzystana biblioteka DotGadu. Aktualnie dzięki tej wtyczce można ustawiać swoje statusy, sprawdzać statusy znajomych oraz wysyłać i odbierać wiadomości.


IInterfacePlugin
IContactPlugin
IConfigurationPlugin
IDatabasePlugin
GaduPlugin
Class
Fields
configurationPluginHost : IConfigurationPluginHost
contactPluginHost : IContactPluginHost
contacts : Dictionary<string, GaduContact>
databasePluginHost : IDatabasePluginHost
GaduClient : Gadu
GaduServer : GaduServer
gaduToolStripDropDownButton : GaduToolStripDownB...
interfacePluginHost : IInterfacePluginHost
messagesQueue : MessageQueue<int>
password : string
pluginHost : IPluginHost
status : int
statusText : string
uid : string
Properties
Author : string
ConfigControl : PluginConfigControl
ConfigurationPluginHost : IConfigurationPluginHost
ContactPluginHost : IContactPluginHost
Contacts : Dictionary<string, GaduContact>
DatabasePluginHost : IDatabasePluginHost
Description : string
InterfacePluginHost : IInterfacePluginHost
Name : string
PluginHost : IPluginHost
PluginId : string
ProtocolPassword : string
ProtocolUid : string
Version : string
Methods
awayToolStripMenuItem_Click() : void
CreateContact() : Contact
Dispose() : void
gaduWaitingMessages_OnDequeue() : void
gaduWaitingMessages_OnEnqueue() : void
InitConfiguration() : void
InitContacts() : void
InitDatabase() : void
InitInterface() : void
InterfacePluginHost_NotifyIconClicked() : void
invisibleToolStripMenuItem_Click() : void
offlineToolStripMenuItem_Click() : void
onlineToolStripMenuItem_Click() : void
PostInit() : void
PreInit() : void
Reciever_OnGaduDisconnecting() : void
Reciever_OnGaduNotifyReplay() : void
Reciever_OnGaduStatus() : void
Reciever_OnLoginFailed() : void
Reciever_OnLoginOK() : void
Reciever_OnRecieveMessage() : void
SetAllContactsOffline() : void
SetContactStatus() : void
SetMyStatus() : void
ToString() : string
GaduToolStripDownButton
Class
ToolStripDropDownButton
Fields
awayToolStripMenuItem : ToolStripMenuItem
components : IContainer
gaduAwayToolStripMenuItem : ToolStripMenuItem
gaduConnectedToolStripMenuItem : ToolStripMenuItem
gaduDisconnectedToolStripMenuItem : ToolStripMenuI...
gaduInvisibleToolStripMenuItem : ToolStripMenuItem
gaduSetStatusToolStripMenuItem : ToolStripMenuItem
invisibleToolStripMenuItem : ToolStripMenuItem
offlineToolStripMenuItem : ToolStripMenuItem
onlineToolStripMenuItem : ToolStripMenuItem
Methods
GaduToolStripDownButton()
PluginConfigControl
Class
UserControl
GaduConfigControl
Class
PluginConfigControl
Fields
components : IContainer
groupBox1 : GroupBox
label2 : Label
label5 : Label
textBoxGaduLogin : TextBox
textBoxGaduPassword : Text...
Methods
buttonSave_Click() : void
Dispose() : void
GaduConfigControl()
InitializeComponent() : void
PluginConfigControl_Load() :...
Contact
Class
ChatContact
Abstract Class
Contact
GaduContact
Class
ChatContact
Fields
chatWindows : Dictionary<string, IChatWindow>
Properties
ImageKey : string
PluginId : string
Methods
doRegisterMessage() : void
Initialize() : void

Wtyczka Jabber

Wtyczka do obsługi technologii Jabber (protokół XMPP) składa się z tych samych elementów, co wtyczka do Gadu-Gadu. Implementuje dokładnie te same interfejsy i zapewnia podobne możliwości. Do komunikacji protokołem XMPP została wykorzystana biblioteka Jabber-Net. Aktualnie dzięki tej wtyczce można ustawiać swoje statusy, sprawdzać statusy znajomych oraz wysyłać i odbierać wiadomości.


IInterfacePlugin
IContactPlugin
IConfigurationPlugin
IDatabasePlugin
JabberPlugin
Class
Fields
configurationPluginHost : IConfigurationPluginHost
contactPluginHost : IContactPluginHost
contacts : Dictionary<string, JabberContact>
databasePluginHost : IDatabasePluginHost
interfacePluginHost : IInterfacePluginHost
Jabber : JabberClient
jabberToolStripDropDownButton : JabberToolStripDownBut...
messagesQueue : MessageQueue<int>
password : string
pluginHost : IPluginHost
server : string
status : PresenceType
statusShow : string
statusText : string
uid : string
Properties
Author : string
ConfigControl : PluginConfigControl
ConfigurationPluginHost : IConfigurationPluginHost
ContactPluginHost : IContactPluginHost
Contacts : Dictionary<string, JabberContact>
DatabasePluginHost : IDatabasePluginHost
Description : string
InterfacePluginHost : IInterfacePluginHost
Name : string
PluginHost : IPluginHost
PluginId : string
ProtocolPassword : string
ProtocolServer : string
ProtocolUid : string
Version : string
Methods
awayToolStripMenuItem_Click() : void
CreateContact() : Contact
Dispose() : void
InitConfiguration() : void
InitContacts() : void
InitDatabase() : void
InitInterface() : void
InterfacePluginHost_NotifyIconClicked() : void
Jabber_OnAfterPresenceOut() : void
Jabber_OnAuthenticate() : void
Jabber_OnDisconnect() : void
Jabber_OnError() : void
Jabber_OnMessage() : void
Jabber_OnPresence() : void
Jabber_OnStreamError() : void
offlineToolStripMenuItem_Click() : void
onlineToolStripMenuItem_Click() : void
PostInit() : void
PreInit() : void
SetAllContactsOffline() : void
SetContactStatus() : void
SetMyStatus() : void
ToString() : string
JabberToolStripDownButton
Class
ToolStripDropDownButton
Fields
awayToolStripMenuItem : ToolStripMenuItem
invisibleToolStripMenuItem : ToolStripMenuItem
jabberAwayToolStripMenuItem : ToolStripMenuItem
jabberConnectedToolStripMenuItem : ToolStripMenuIt...
jabberDisconnectedToolStripMenuItem : ToolStripMen...
jabberSetStatusToolStripMenuItem : ToolStripMenuItem
offlineToolStripMenuItem : ToolStripMenuItem
onlineToolStripMenuItem : ToolStripMenuItem
Methods
JabberToolStripDownButton()
PluginConfigControl
Class
UserControl
JabberConfigControl
Class
PluginConfigControl
Fields
components : IContainer
groupBox1 : GroupBox
textBoxPassword : TextBox
textBoxServer : TextBox
textBoxUid : TextBox
Methods
buttonSave_Click() : void
JabberConfigControl()
PluginConfigControl_Load() : void
Contact
Class
ChatContact
Abstract Class
Contact
JabberContact
Class
ChatContact
Properties
ImageKey : string
PluginId : string
Methods
doRegisterMessage() : void
Initialize() : void

Wtyczka Checky

Jedynym interfejsem jaki implementuje klasa główna tej wtyczki jest ***IContactPlugin***. Jest to wystarczające aby zapewnić nowy rodzaj kontaktu, który okresowo wywołuje podany program, a zawartość standardowego wyjścia wstawia jako opis danego kontaktu na liście. W celu okresowego wywoływania wykorzystana została klasy ***AutoResetEvent***, ***TimerCallback*** oraz ***Timer***. Pierwsza służy do automatycznego uruchamiania i wznawiania działania wątku. ***TimerCallback*** jest jedynie delegatem, który będzie obsługiwał naszą metodę sprawdzającą. ***Timer*** natomiast zapewnia wątek, który w ustalonych odstępach czasu będzie uruchamiał metodę przekazaną jako parametr konstruktora delegata.