Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa inżynierska

OTWARTA PLATFORMA DLA MODUŁOWEGO TWORZENIA APLIKACJI

Mikołaj Dobski
Piotr Jessa
Maciej Kowalewski
Piotr Ślatała

Promotor
dr inż. Piotr Zielniewicz

Poznań, 2011 r.

Spis treści

1 Wprowadzenie **1**
1.1 Cel i zakres pracy . . . 1
1.2 Omówienie treści pracy oraz podział zadań . . . 2

2 Podstawy teoretyczne **5**
2.1 Specyfikacja zaleceń Composition Application Guidance . . . 5
2.2 Specyfika budowania frameworku . . . 5
2.3 Istniejące rozwiązania wspierające plug-in model . . . 7
2.3.1 Narzędzia dla platformy .NET . . . 8
2.3.2 Inne rozwiązania . . . 10
2.4 Architektura systemu Windows i platformy .NET . . . 10
2.4.1 Microsoft Windows . . . 10
2.4.2 .NET Framework . . . 11

3 Architektura rozwiązania **15**
3.1 Wymagania względem frameworka . . . 15
3.2 Zarys architektury platformy *Nomad* . . . 17

4 Moduły **19**
4.1 Badania poprzedzające implementację modułu. . . . 19
4.1.1 Możliwe koncepcje realizacji . . . 19
4.1.2 Badania praktyczne wydajności poszczególnych rozwiązań . . . 20
4.1.3 Implementacja . . . 22
4.2 Zarządzanie modułami . . . 22
4.3 Bezpieczeństwo modułów . . . 25
4.3.1 Podpisywanie *Assembly* .NET . . . 25
4.3.2 Podpisywanie manifestów *Nomad* . . . 26

5 Regiony – koncepcja modułowego interfejsu użytkownika **29**
5.1 Problem kompozycji interfejsu użytkownika . . . 29
5.2 Przegląd potencjalnych rozwiązań problemu . . . 29
5.2.1 Rozwiązania specyficzne dla aplikacji . . . 29
5.2.2 Rozwiązania oparte o regiony – wprowadzenie . . . 30
5.3 Przyjęte założenia i ograniczenia . . . 31
5.4 Użycie mechanizmu regionów . . . 31
5.5 Przykładowe adaptery . . . 32
5.6 Punkty rozszerzenia mechanizmu regionów . . . 33

6 Komunikacja międzymodułowa **37**
6.1 Komunikacja asynchroniczna . . . 37
6.1.1 Koncepcja Event Aggregatora . . . 38
6.1.2 Komunikacja asynchroniczna w środowisku wielodomenowym . . . 39
6.1.3 Komunikacja asynchroniczna w środowisku wielowątkowym . . . 41
6.1.4 Komunikacja asynchroniczna w środowisku z *GUI* . . . 41
6.1.5 Rzeczywista implementacja . . . 42
6.2 Komunikacja synchroniczna . . . 43
6.2.1 Wzorzec *ServiceLocator* . . . 43
6.2.2 Implementacja *ServiceLocator* . . . 43

7 Mechanizm uaktualnień modułów **45**
7.1 Koncepcja realizacji . . . 46
7.2 Implementacja . . . 46
7.2.1 Rodzaje aktualizacji . . . 46
7.2.2 Domyślna konfiguracja . . . 47

8 Rdzeń platformy - Nomad.Core **49**
8.1 NomadKernel jako punkt wejścia dla programisty . . . 49
8.2 NomadConfiguration . . . 49
8.3 Usługi udostępniane przez *Nomada* . . . 50
8.3.1 Listowanie załadowanych modułów . . . 50
8.3.2 Wsparcie dla wielojęzykowości . . . 50

9 Zapewnienie jakości produktu **53**
9.1 Testowanie *Nomada* . . . 53
9.2 Udostępnianie *Nomada* . . . 55

10 Zastosowanie wyników pracy **57**
10.1 Użycie *Nomada* . . . 57
10.2 Samouczki . . . 57
10.3 Narzędzia dostarczane wraz platformą . . . 58
10.3.1 Generowanie kluczy RSA . . . 58
10.3.2 Generowanie manifestów zgodnych z formatem *Nomad* w formacie XML . . . 58
10.4 Serwer modułów jako przykład aplikacji współpracującej z systemem Nomad . . . 59
10.4.1 Przykładowa implementacja strony klienckiej . . . 59
10.4.2 Przykładowa implementacja strony serwerowej . . . 60
10.4.3 Moduł Updater'a - graficzny moduł aktualizacji . . . 61
10.5 Przykładowa aplikacja prezentująca możliwości frameworka . . . 62
10.5.1 Zrzuty ekranu . . . 63
10.6 Przykłady praktycznego wykorzystania frameworka . . . 64

11 Podsumowanie oraz dalsze perspektywy platformy **65**
11.1 Założenia a wyniki implementacji . . . 65
11.2 Dalsze perspektywy rozwoju platformy . . . 65
11.2.1 *Sandboxing* modułów . . . 65
11.2.2 Proponowany mechanizm rozwiązywania zależności usługowych . . . 66
11.2.3 Aktywacja, deaktywacja modułów . . . 67
11.2.4 Definiowanie usług przez konfigurację . . . 68
11.3 Publikacja na zasadach wolnego oprogramowania . . . 68

Spis rysunków **69**

Spis tablic **69**

A Wykaz użytych technologii **71**
A.1 Biblioteki użyte do budowy platformy . . . 71
A.2 Narzędzia wspomagające testowanie . . . 71
A.3 Narzędzia wspomagające zarządzanie projektem . . . 71

B Wykaz haseł **73**

C Wykaz źródeł istniejących rozwiązań **75**

D Opis elementów zamieszczonych na płycie CD **77**
D.1 Elementy powiązane z pracą dyplomową . . . 77
D.2 Elementy powiązane z badaniami . . . 77
D.3 Elementy powiązane z wydaniem Nomada . . . 77

Literatura **79**

Rozdział 1

Wprowadzenie

Niniejsza praca porusza zagadnienia związane z budową narzędzi dla programistów, które ułatwiają tworzenie nowych aplikacji. W szczególności skupia się na *Microsoft .NET* jako bazie technologicznej, którą obecni programiści budujący oprogramowanie dla systemu *Microsoft Windows* wykorzystują. *Framework .NET*, stanowi kolejny poziom abstrakcji w tworzeniu oprogramowania – programista kreuje aplikację używając mechanizmów udostępnianych przez platformę *.NET*, abstrahując od warstwy sprzętowej czy interfejsu programistycznego systemu operacyjnego.

Wytwarzanie oprogramowania to dziedzina informatyki, nazwana Inżynierią Oprogramowania, która ma swój początek na przełomie lat pięćdziesiątych i sześćdziesiątych dwudziestego wieku. Przyczyną powstania nowej gałęzi było rosnące zapotrzebowanie przemysłu oraz sektora usług na oprogramowanie. Spełnienie rosnących wymagań klientów przekraczało w znaczący sposób możliwości niezorganizowanych zespołów, korzystających z wytworzonych przez siebie narzędzi. Im bardziej skomplikowane systemy informatyczne pragnęli budować programiści, tym bardziej zaawansowanych narzędzi potrzebowali, aby uczynić ten proces wydajnym i opłacalnym.

Od tego okresu minęło wiele lat, powstało wiele narzędzi dla informatyków – programistów czy też opracowań bardziej lub mniej naukowych omawiających problemy powiązane z inżynierią oprogramowania. Nie zmienił się jednak cel tych wszystkich zabiegów – skrócenie czasu produkcji nowego oprogramowania. Idące za owym uproszeniem procesu kreacji oszczędności czasu i pieniędzy stanowiły główną motywację do tak gwałtownego rozwoju tej dziedziny informatyki.

Obecnie na rynku dostępnych jest wiele różnych narzędzi przyspieszających wytwarzanie oprogramowania. Do tej grupy zaliczyć można biblioteki, zestawy predefiniowanych komponentów czy też frameworki – obszerne szkielety kodu dzięki którym programiści nie tracą czasu na żmudną implementację istniejących już rozwiązań. Takie „pomoce“ dostępne są także, w bardzo dużej liczbie, dla twórców wykorzystujących platformę *.NET*. Firma *Microsoft* rozwija platformę *.NET* konsekwentnie od 2002 roku. Do tej pory ukazała się czwarta wersja tego środowiska, a planowane już jest wydanie wersji piątej.

Interesujący jest jednak fakt, iż pomimo bogactwa bibliotek dla wyżej wspomnianej technologii bardzo trudno znaleźć kompleksowe rozwiązanie pozwalające stworzyć aplikację modułową. To jest taką, której wydzielone technicznie komponenty funkcjonalne można swobodnie uzupełniać, rozbudowywać, komponować wzajemnie, konfigurować.

Architektura takich aplikacji jest dużo bardziej skomplikowana niż programu, który modularności by nie posiadał. Poszczególne elementy mogą od siebie zależeć, wywoływać niekorzystne reakcje, czy też wymieniać się informacjami. Istnieje więc potrzeba stworzenia rozwiązania, które budowę takiej architektury zapewni i zaoferuje ją programistom w przystępny sposób.

1.1 Cel i zakres pracy

Celem pracy jest zaprojektowanie i zaimplementowanie otwartej platformy do modułowego tworzenia aplikacji w środowisku *.NET*. W ramach tego celu sformułowano poniższe wymagania:

- opracowanie architektury opartej na module, technicznej jednostce funkcjonalnej, pozwalającej na dynamiczne dołączanie się do aplikacji bądź odłączanie
- zapewnienie modułom możliwości komunikacji
- opracowanie mechanizmu ładowania modułów wraz z zależnościami

- opracowanie mechanizmu aktualizacji oraz przygotowanie przykładowego repozytorium modułów
- przygotowanie przykładowej aplikacji pokazującej możliwości opracowanej platformy
- przygotowanie dokumentacji technicznej

W ramach realizacji wyżej postawionych celów wykonano następujące zadania:

- Zapoznano się z *Microsoft Composite Application Guidance* stanowiącym zestaw wytycznych i wzorców dla wszystkich projektantów aplikacji o dużym stopniu złożoności.
- Zapoznano się oraz zbadano cechy istniejących na rynku rozwiązań do budowy modułowych aplikacji.
- Zapoznano się z literaturą z zakresu budowy *frameworków* programistycznych oraz literaturą z zakresu funkcjonowania platformy *.NET*
- Wykonano serię badań wydajnościowych mechanizmów komunikacji w ramach procesu w środowisku *.NET*
- Opracowano implementację przy użyciu technik *Test Driven Development* zapewniających pełne pokrycie testami wytworzonego oprogramowania.
- Przygotowano krótką serię samouczków obrazujących wykorzystanie poszczególnych elementów składowych platformy.

Nowo powstała platforma otrzymała nazwę roboczą *Nomad Framework*, będącą akronimem od angielskiego wyrażenia *.NET Open Modular Application Development Framework*, co w języku polskim oznacza otwartą platformę do modułowego budowania aplikacji.

1.2 Omówienie treści pracy oraz podział zadań

Struktura pracy jest następująca:

- Rozdział 2 – Stanowi przegląd istniejących rozwiązań, a także historycznych podejść do problemu aplikacji modułowych w kontekście używanej technologii. W rozdziale znajduje się też dokładne omówienie dokumentu *Microsoft Composite Application Guidance*, architektury systemu *Windows*, platformy *.NET*. Rozdział 2 opisuje również szczególne wymagania stawiane frameworkom.
- Rozdział 3 – Stanowi wprowadzenie do architektury zaproponowanego rozwiązania. Prezentuje poszczególne elementy składowe platformy *Nomad*, określa granice w których platforma wspiera programistę tworzącego w oparciu o nią aplikacje
- Rozdział 4 – Opisuje koncepcję modułu, wymagania jakie mu postawiono. W tym rozdziale jest także dyskusja wyboru odpowiedniej metody reprezentacji modułu poparta wynikami testów wydajnościowych mechanizmów *Frameworka .NET*.Rozdział opisuje także zaawansowane aspekty wprowadzenia modułu jako podstawy funkcjonalności, pochyla się nad zagadnieniem rozwiązywania zależności pomiędzy modułami oraz kwestiami bezpieczeństwa.
- Rozdział 5 – Opisuje rozwiązanie pozwalające na wspólną kompozycję graficznego interfejsu użytkownika przez wszystkie zainteresowane moduły.
- Rozdział 6 – Opisuje zaimplementowane w platformie *Nomad* metody wsparcia komunikacji pomiędzy modułami.
- Rozdział 7 – Opisuje zaprojektowane rozwiązanie mechanizmu aktualizacji.
- Rozdział 8 – Opisuje rdzeń platformy oraz prezentuje jak skonfigurować usługi jądra.
- Rozdział 9 – Prezentuje narzędzia wspierające rozwój projektu oraz zapewnienie odpowiedniej jakości produktu.
- Rozdział 10 – Opisuje istniejące zastosowania platformy *Nomad*. Rozdział omawia także, narzędzie dostarczone wraz z platformą.
- Rozdział 11 – Podsumowuje wykonaną pracę oraz kreśli dalsze perspektywy rozwoju platformy *Nomad*.

Podział zadań podczas pracy projektowej

Mikołaj Dobski:

- Badanie metod ograniczania uprawnień *AppDomains* środowiska *.NET*.
- Zaprojektowanie oraz zaimplementowane samouczków pokazujących w prostych scenariuszach wykorzystanie kodu *Nomad*.
- Zaprojektowanie i implementacja mechanizmów rozwiązywania zależności pomiędzy modułami.

Piotr Jessa:

- Zaprojektowanie, zaimplementowanie oraz integracja poszczególnych składowych systemu, architektury niskiego poziomu platformy *Nomad* w oparciu o dwie *domeny aplikacji*.
- Zaprojektowanie i zaimplementowanie mechanizmu aktualizacji.
- Implementacja systemu komunikacji synchronicznej – *Service Locator*.
- Przygotowanie przykładowego serwera działającego jako repozytorium modułów.
- Zaprojektowanie, zaimplementowanie i przeprowadzenie badań wydajnościowych mechanizmów komunikacji środowiska *.NET*.

Maciej Kowalewski:

- Zaprojektowanie oraz zaimplementowanie kompozycji interfejsu użytkownika przy użyciu koncepcji regionu.
- Zaprojektowanie architektury najwyższego poziomu.
- Zaprojektowanie oraz zaimplementowanie mechanizmu ładowania modułów.
- Przygotowanie mechanizmów automatycznego testowania budowanej aplikacji.

Piotr Ślatała:

- Zaprojektowanie i zaimplementowanie systemu komunikacji asynchronicznej – *EventAggregator*.
- Zaprojektowanie i implementacja mechanizmu podpisywania modułów.
- Przygotowanie przykładowej aplikacji stanowiącej kompleksowy pokaz możliwości platformy *Nomad*.
- Przygotowanie środowiska pracy – serwera ciągłej integracji, repozytorium kodu.
- Zaprojektowanie i implementacja wsparcia dla wielojęzykowości.

Rozdział 2

Podstawy teoretyczne

Tworzenie platformy *Nomad* poprzedzono lekturą literatury traktującej o narzędziach wspierających budowanie aplikacji modularnych. Przeanalizowano również dostępne na rynku już istniejące rozwiązania tego typu dla platformy *.NET*. Pozwoliło to zidentyfikować ich ograniczenia i zaprojektować *Nomada* jako narzędzie nie tylko powielające sprawdzone już rozwiązania, ale i adresujące problemy, których pozostałe platformy nie rozwiązują i przez to unikalne na rynku.

2.1 Specyfikacja zaleceń Composition Application Guidance

Spośród przeanalizowanej literatury, najpełniejszy opis problematyki modularności w *.NET* zawarty został w dokumencie *Composite Application Guidance* [Mic09]. Jest do dokument przygotowany oraz wydany przez firmę *Microsoft*. Porusza on m.in. kwestie ładowania modułów, kompozycji interfejsów użytkownika i komunikacji między modułami.

Composite Application Guidance zakłada, że aplikacja komponowana jest z wielu równorzędnych modułów. Każdy z modułów implementuje fragment logiki biznesowej oraz odpowiadający tej logice interfejs użytkownika. Moduły mogą być rozwijane niezależnie przez różne zespoły. Zgodnie z założeniami, jeden z modułów jest wyróżniony i zawiera niezbędną infrastrukturę – podstawowe elementy interfejsu użytkownika oraz logikę wspólną dla wszystkich modułów. Zależności między modułami powinny być ograniczane do minimum – tak, aby moduły potrzebowały jak najmniej informacji na temat szczegółów implementacji modułów, od których zależą.

W szczególności oznacza to, że moduły chcące umieścić własny widok (fragment graficznego interfejsu użytkownika) w oknie aplikacji nie muszą mieć bezpośredniego dostępu do tego okna – wystarczy, że mogą skorzystać z mechanizmu pozwalającego na wstawienie widoku w pewne nazwane miejsce (tzw. region). Ponadto, moduły reagujące na pewne zdarzenia (np. aktualizacja danych pochodzących z zewnętrznych źródeł, działania użytkownika) nie muszą koniecznie znać miejsca powstania tego zdarzenia. Wystarczy, że zgłoszą pewnej usłudze chęć otrzymywania informacji o zajściu określonych zdarzeń. Konsekwentne stosowanie tych mechanizmów pozwala na ograniczenie stopnia, w jakim moduły zależą od siebie (bez ograniczania funkcjonalności dla użytkownika końcowego). Tym samym upraszcza to tworzenie oraz utrzymanie aplikacji.

Jako, że zalecenia *Composite Application Guidance* w znacznym stopniu pokrywały się z zakresem niniejszej pracy, specyfikacja ta stała się głównym źródłem wykorzystywanym przez autorów. Należy jednak zauważyć, że *Composite Application Guidance* opisuje rozwiązania jedynie części postawionych problemów, stąd zaproponowane przez autorów rozwiązania nie są identyczne z rozwiązaniami zaimplementowanymi w referencyjnej implementacji zaleceń – *Prismie*.

2.2 Specyfika budowania frameworku

Konstrukcja frameworka znacząco różni się od budowy aplikacji, choćby ze względu na cykl życia platformy. Należy pamiętać, że frameworki powinny spełniać dwie cechy [KC08]:

1. Być potężne (oferować wiele możliwości).
2. Być łatwe w użyciu.

Zgodnie z nieformalną zasadą, 80% przypadków wykorzystania frameworku powinno być proste w realizacji, a 20% przypadków będzie ważne i nietrywialne. W myśl tej zasady, tworząc framework należy zachować dla programistów możliwość rozwiązania tych 20% przypadków [KC08].

Chcąc spełnić powyższe wymagania wyróżniono pewne techniki programistyczne, które zostały wdrożone w trakcie prac nad *Nomadem*. Wiele z nich znajduje zastosowanie w wykonywaniu dowolnych projektów informatycznych. Jednakże, w związku z charakterem opisywanego w pracy projektu – platformy programistycznej, postanowiono wypełnić opisywane dalej aspekty ze szczególną dbałością.

Zamrażanie *API*

Bazując na platformie programistycznej powstają inne aplikacje. Stąd zmiany *API* udostępnianego przez platformę muszą być wprowadzane bardzo rozważnie. Niepożądanym jest, aby aplikacje po aktualizacji bibliotek platformy przestawały działać. Jeśli zatem *API* ulega ciągłym modyfikacjom, to jednoznacznie oznacza to, że produkt nie jest jeszcze gotowy do publikacji. W takiej sytuacji nie należy uzależniać rozwoju innych produktów od prac nad platformą. Produkt powinno się udostępnić grupie docelowej dopiero w momencie, gdy *API* stabilizuje się, a wszystkie elementy są wyraźnie zdefiniowane. Od publikacji, *API* platformy staje się kontraktem wiążącym programistów. Powinno się jedynie dodawać nową funkcjonalność, bez modyfikacji istniejącego *API*. Elementy *API*, które producenci platformy najchętniej by usunęli, mogą wyłącznie deprecjonować i oznaczać atrybutem *obsolete*. W ten sposób definiujemy tak zwane „zamrażanie“ *API* platformy.

Projektowanie architektury

Kolejnym kluczowym elementem jest faza projektowania architektury oraz samych klas biorących w niej udział. Klasy powinny wyróżniać się możliwie najwyższą *spójnością (ang. cohesion)* [EFB04]. Powinny posiadać pojedyncze, najmniejsze możliwie, lecz spójne odpowiedzialności i realizować je w sposób maksymalnie niezależny od reszty systemu. Co więcej, realizowane zadania powinny budować kolejne poziomy abstrakcji – odpowiedzialność powinna być określona zadaniem, które chciałby zrealizować wywołujący, a nie sposobem implementacji tego zadania. Klasy wykorzystujące daną funkcjonalność mogą dzięki temu używać interfejsów opisujących tak powstałe abstrakcje, nie uzależniając się od konkretnej implementacji zadania. Tym samym, tak długo jak niezmieniony pozostaje kontrakt interfejsów, zmiany w dowolnym fragmencie architektury nie wywołują efektu domina w całym projekcie. Technika ta opisywana jest w literaturze jako *loose coupling* [EFB04]. Pełna wymienialność komponentów implementujących interfejsy jest możliwa do osiągnięcia dzięki stosowaniu się do *Liskov substitution principle* [RCM08].

Wsparcie dla początkujących użytkowników produktu

Głównym zadaniem wszelkiego rodzaju platform programistycznych jest ułatwienie pracy programisty, nie jej utrudnienie. Pisząc kod należy również mieć na uwadze by metody wykonywały pojedyncze zadania, były możliwie krótkie oraz posiadały nazwy pozwalające łatwo określić ich odpowiedzialność. Całe *API*, z którego programista może korzystać musi być dogłębnie udokumentowane. Kod mechanizmów działających wewnątrz platformy również należy opisać. Osoby chcące rozszerzyć czy podmienić te mechanizmy nie mogą tracić czasu na zastanawianie się jaką odpowiedzialność wypełnia dowolnym fragment kodu. Dlatego gotowy produkt musi być doskonale udokumentowany. Dokumentacja nie może być jednak jedynym źródłem wiedzy o platformie. Należy dostarczyć również samouczki (*tutorials*) opisujące wykorzystanie poszczególnych elementów architektury produktu, jak i przykładową zaawansowaną aplikację obrazującą wiele możliwości platformy.

Ewolucja projektu

Platforma programistyczna powinna również ewoluować wraz z rosnącym wykorzystaniem na rynku, zapotrzebowaniem na nowe funkcje czy optymalizację już zaimplementowanych elementów. Dlatego zgodnie ze świadomie zaprojektowaną architekturą, możliwie wiele elementów powinno być wymienialne (wszystkie wykorzystywane poprzez interfejsy). Powinny również istnieć punkty rozszerzenia produktu – *extension points*, dzięki którym można wzbogacić platformę o nową funkcjonalność bez ingerencji w kod źródłowy platformy. Tym samym produkt powinien być możliwie zamknięty na najniższym poziomie implementacji i jednocześnie możliwie otwarty w każdym punkcie architektury. Podąża to za *Open-closed principle* [RCM08].

Zapewnienie wysokiej jakości kodu wolnego od błędów

Programiści korzystający z gotowych platform potrzebują pewności, że rozwiązanie którym się wspierają jest pozbawiony błędów. Aby móc wydajnie pisać kod, powinni móc skupić się nad poprawnością własnego kodu, a nie zabezpieczać przed błędami i problemami platformy. Najprostszą metodą na zapewnienie im tej pewności jest zaprezentowanie zautomatyzowanych testów pokrywających w stu procentach kod oraz wszelkie funkcje oferowane przez platformę. W literaturze pojawia się wiele propozycji podziału testów na kategorie. W trakcie prac nad platformą *Nomad* przyjęto kategoryzacje zaprezentowaną w [Mes07]:

- testy jednostkowe (*Unit tests*) – poszczególne elementy frameworka są testowane w oderwaniu od innych elementów,
- testy integracyjne (*Integration tests*, zwane również *Component tests*) – weryfikowane są zintegrowane grupy funkcji, ich współpraca w proponowanej architekturze,
- testy funkcjonalne (*Functional tests* lub *Customer tests*) – testowane są całe funkcje systemu określające wybrane wymagania funkcjonalne.

Warto również podkreślić, że w przypadku platformy oferującej narzędzia do kompozycji graficznego interfejsu użytkownika, należy również wykonać testy tego mechanizmu (zachowanie okien, aktywowanie obszarów, klikanie w przyciski, itd.).

Testowanie artefaktów programistycznych nie jest jednak łatwym zadaniem. Doświadczenie pokazuje, że projekty, które posiadają fazę pisania testów dopiero po projektowaniu architektury i jej implementacji, wielokrotnie ją pomijają ze względu na brak czasu. Ów brak czasu wiążę się najczęściej z przedłużającą się fazą implementacyjną, w której programiści borykają się z poprawianiem wykrytych błędów.

Aby uniknąć tego rodzaju problemów zastosowano podejście *TDD (Test-driven development)* [Bec02]. Metoda ta zakłada tworzenie oprogramowania w poniższej pętli:

1. stworzenie testu sprawdzającego dodawaną funkcjonalność, który się nie udaje.
2. implementacja minimalnej ilości kodu spełniającego funkcjonalność by test się powiódł.
3. refaktoryzacja zamiplementowanego kodu.
4. powrót do kroku pierwszego.

Należy zauważyć, że w procesie tworzenia funkcjonalności wyróżnić można dwie takie pętle – zewnętrzną, w której tworzy się testy akceptacyjne (funkcjonalne) dla całej implementowanej funkcjonalności, oraz wewnętrzną, w której z pomocą testów jednostkowych stopniowo implementuje się wymagania zdefiniowane testem akceptacyjnym [SF09].

Dzięki zastosowaniu opisanej techniki powstaje prosty kod w bardzo dużym stopniu pokryty testami. Jeśli nie ma testów na jakąś funkcjonalność, produkt jej nie oferuje. Technika ta ułatwia również pielęgnowanie kodu produktu. Programista po wprowadzeniu jakichkolwiek zmian może szybko sprawdzić, czy cały produkt nadal zachowuje się zgodnie z wymogami.

Ponadto, zdefiniowanie z góry testów dla danego wycinka funkcjonalności pozwala na zaprojektowanie *API* z punktu widzenia użytkownika. Programista wie również, kiedy może uznać kod za gotowy – jest to moment, w którym „przechodzą" wszystkie testy akceptacyjne.

2.3 Istniejące rozwiązania wspierające plug-in model

Narzędzia wspierające tworzenie architektur typu *plug-in model* muszą oferować programistom możliwość utworzenia aplikacji oraz wtyczek do niej. Kluczowe są tutaj mechanizmy rejestracji oraz komunikacji aplikacji z wtyczkami. Owe wtyczki powinny stanowić możliwie niezależne jednostki funkcjonalne o wysokiej kohezji. Wydłużają one cykl życia aplikacji poprzez rozszerzenie jej funkcjonalności. *Plug-in model* dopuszcza również przypadek, w którym aplikacja główna spełnia jedynie funkcję rdzenia z interfejsem *GUI* definiując punkty rozszerzeń i sama w sobie nie dostarcza, żadnej logiki biznesowej. Dopiero wtyczki załadowane w aplikacji stanowią o jej zastosowaniu i wartości rynkowej.

2.3.1 Narzędzia dla platformy .NET

Dostępne są rozwiązania, które w pewnym stopniu pokrywają się z celami tej pracy. Zostaną one krótko zaprezentowane, w szczególności ich różnice względem *Nomada*, sugerujące biznesową potrzebę powstania produktu. Strony domowe przedstawionych projektów można znaleźć w dodatku C.

CLR Addin

Projekt koncepcyjny rozwijany przez programistów *Microsoftu*. Wprowadził do *.NET* model rozszerzenia *Add-in*. Narzędzie miało zapewnić ujednolicony sposób komunikacji dodatków z aplikacją, którą rozszerzają. W tym celu wyspecyfikowano strumień komunikacyjny, który łączył dodatki z aplikacją główną, zestawami widoków i adapterów poprzez kontrakty (interfejsy). Jest to „niskopoziomowe“ narzędzie, które nie oferuje kompleksowego i szybkiego tworzenia aplikacji modularnych. W żaden sposób nie wspiera także graficznej kompozycji interfejsu użytkownika.

Projekt był rozwijany do 2008, po czym włączony do *.NET Framework* jako przestrzeń nazw *System.Addin*. Dodatkowym skutkiem powstania narzędzia była motywacja do utworzenia opisanego dalej *MEF*.

MEF

MEF (*Managed Extensibility Framework*) to biblioteka wydana przez *Microsoft*. Podstawową funkcją oferowaną przez *MEF* jest mechanizm kontenera *IoC* (patrz dodatek B). Umożliwia bardzo łatwe rejestrowanie implementacji określonych typów (opisanych poprzez atrybuty) w kontenerze oraz ich pobieranie.

Najnowsza wersja została wydana jako integralna część platformy *.NET* w wersji 4.0 i jest dostępna jako *System.ComponentModel.Composition*.

Podobnie jak *CLR Addin*, *MEF* nie jest narzędziem kompleksowym, ułatwia jedynie rejestrowanie dodatków. Nieobecne są natomiast gotowe mechanizmy komunikacji, izolacji czy aktualizacji modułów.

PRISM

Prism to biblioteka stanowiąca zbiór klas rozwiązujących kilka problemów przedstawionych w niniejszej pracy. Została ona wydana przez *Microsoft*. Stanowi on referencyjną implementację zaleceń z *Composite Application Guidance* [Mic09] i ułatwia stosowanie ich we własnych projektach. *Prism* jest dostępny od roku 2008 i wciąż jest rozwijany.

Autorzy *Prisma* jasno wyspecyfikowali jego zastosowanie w następujących aplikacjach:

- wyświetlających wyniki z wielu różnych źródeł danych, integrując je w spójnym interfejsie użytkownika.
- wspierających moduły budowane niezależnie od innych modułów.
- intensywnie rozwijających się, dodające nowe widoki i funkcje.
- tworzonych przez wiele współpracujących zespołów programistycznych.
- wykorzystujących technologie *WPF* i *Silverlight* jako warstwę prezentacyjną.

Ponieważ zarówno *Nomad* jak i *Prism* oparte są na *Composite Application Guidance*, część problemów (np. kompozycję interfejsu użytkownika) rozwiązano w bardzo podobny sposób. W porównaniu do *Prisma*, *Nomad* oferuje szersze wsparcie dla modularności – zapewnia podstawową izolacje między modułami oraz zapewnia mechanizm automatycznych aktualizacji modułów. Autorzy pracy zdecydowali się natomiast nie zawierać w *Nomadzie* wielu elementów, które nie są bezpośrednio związane z modularnością, a które są zawarte w *Prismie* – np. bazowej implementacji wybranych wzorców prezentacji (*MVVM* czy *MVP*). Platforma *Nomad* nie wspiera również *Silverlighta*.

Mono.Addins

Framework bazujący na platformie *Mono* i umożliwiający tworzenie aplikacji z zestawami dodatków. Przy wykorzystaniu *Mono.Addins* powstają: aplikacja hosta (*Add-in host*), która wystawia swoje punkty rozszerzenia (*Extension points*) oraz podzespoły (*Addins*) grupujące dodatki (*Extension nodes*) i rejestrujące się w dostępnych punktach. Platforma opisuje dodatki poprzez definiowalne atrybuty typów lub pliki *XML* z manifestami całych *assemblies*. Możliwe jest również stworzenie rozszerzalnych bibliotek. Narzędzie dostarcza także mechanizmy zarządzania dodatkami w czasie uruchomienia. Zgodnie ze specyfikacją spełnia on również następujące funkcje:

- definiuje hierarchie dodatków (zależności pomiędzy dodatkami),
- umożliwia stworzenie rozszerzeń dostarczających jedynie dane,
- wspiera wzorzec opóźnionego ładowania (*lazy loading*) dodatków,
- oferuje dynamiczne włączanie i wyłączanie dodatków w trakcie działania aplikacji,
- pozwala na współdzielenie rejestrów dodatków pomiędzy aplikacjami oraz definiowanie ich lokalizacji,
- wspiera lokalizację dodatków,
- dostarcza API do analizy opisów dodatków w celu tworzenia narzędzi zarządzających dodatkami,
- oferuje bibliotekę zarządzania dodatkami, którą można dołączyć do dowolnej aplikacji w celu zarządzania dodatkami (włączania, wyłączanie, instalowanie nowych z on-line'owych repozytoriów).

Projekt jest cały czas rozwijany, wydany w stabilnej wersji w mają 2010 roku. Architektura projektu nie zawiera informacji o warstwie prezentacji danych, stąd nie wspiera w żaden sposób budowania graficznych aplikacji o spójnym *GUI*.

Plux.NET

Jest to kolejna biblioteka przeznaczona dla środowiska *.NET*. Projekt kładzie nacisk na utworzeni możliwie małej aplikacji definiującej jedynie miejsca przypięcia dodatków (sloty) poprzez określenie typów i parametrów interfejsów, które dodatki muszą implementować. Platforma sama znajduje rozszerzenia pasujące do otwartych slotów, ładuje je oraz informuje aplikację hosta, że slot posiada implementację. W tym momencie aplikacja hosta może podjąć działania integrujące dodatek z sobą – podpiąć zdefiniowany przez dodatek widok we własnym *GUI* oraz przekazać jego obsługę klasie dodatku.

Idea projektu powstała w 2006 jako produkt *CAP.NET* będący pracą naukową zespołu *Christian Doppler Laboratory for Automated Software Engineering* na Uniwersytecie Johannesa Keplera w Linz. W 2007 roku powstał pierwszy prototyp *Plux.NET* i od tego czasu platforma jest regularnie rozwijana. Na jej podstawie powstają publikacje dotyczące architektur modularnych. Pierwsze wersje platformy były niedostępne dla świata. W 2009 roku pojawiła się trzecia generacja narzędzia wspierająca już tworzenie *GUI*. Jednak dopiero po obronie pracy doktorskiej *Reincharda Wolfingera* w 2010 roku [Wol10] zostały udostępnione źródła projektu w wersji 0.3.

Wersja platformy opisana w w/w dokumencie oferuje:

- dynamiczne dodawanie i usuwanie dodatków bez konieczności przeprogramowania aplikacji i jej rekonfiguracji,
- modularną architekturę aplikacji z zależnościami opisanymi poprzez metadane,
- mechanizm wykrywania dostępnych modułów,
- mechanizm generowania aplikacji poprzez analizę zdefiniowanych punktów rozszerzeń oraz dostępnych ich implementacji,
- zdarzeniowy mechanizm informowania aplikacji hosta o wszelkich zmianach w aplikacji,
- wskazówki dla tworzenia graficznych modułów reagujących na zdarzenia jądra platformy,

- widgety związane na stałe ze slotami i automatycznie aktualizujące swój stan jako reakcja na rekonfigurację aplikacji.

Autorzy platformy przewidują dalszy jej rozwój. Poniższe problemy są nadal otwarte:

- wersjonowanie slotów,
- wsparcie dla testowania modułów opartych o architekturę *Plux.NET*,
- ograniczanie uprawnień modułów wpinających się w sloty,
- izolacja modułów w celu zapewnienia odporności aplikacji na awarie modułów.

Platforma obecnie jest w fazie rozwojowej, oferowana bez konkretnej licencji jak i żadnego wsparcia od strony producentów.

2.3.2 Inne rozwiązania

Rozwiązania nie powiązane bezpośrednio z zakresem funkcjonalności *Nomada*:

- AL Platform (*.NET*) – Narzędzie do tworzenia wirtualnego pulpitu, wewnątrz którego można uruchamiać tzw. „narzędzia“ – programy stanowiące o funkcjonalności całego pulpitu.
- .NET Based Add-in/Plug-in Framework with Dynamic Toolbars and Menus – Projekt już historyczny, powstały w 2006 roku jako alternatywa dla tworzenia aplikacji opartych o *MFC, ATL COM*. Prezentuje pierwsze próby ułatwienia tworzenia aplikacji modularnych w środowisku *.NET* z wykorzystaniem plików *XML*.
- Compact Plugs (*.NET*) – Współczesny projekt opierający się o ideę wtyczek, które wyniki swojej pracy mogą w czasie wykonania wstrzykiwać dowolnym innym wtyczkom.
- OSGI – Jest to specyfikacja budowania aplikacji modułowych w oparciu o technologie *Java*. Stanowi modelowy przykład jak powinna być opracowana platforma do tworzenia aplikacji. W oparciu o tą specyfikację powstały dwie implementacje, jedna z nich *Equinox* jest podstawą do budowy narzędzia *Eclipse*, druga rozwijana jest przez fundację *Apache* i została nazwana *Felix*. OSGI to przykład, że platforma do budowy aplikacji może być ważnym elementem decydującym często o wyborze danego środowiska do budowy docelowej aplikacji. Dla autorów *Nomad* stanowi także rodzaj wzoru, który chcieliby osiągnąć.

Podsumowanie cech istniejących rozwiązań

Jak widać istnieje wiele rozwiązań wspierających tworzenie modularnych aplikacji. Najbliższe *Nomadowi* są *Mono.Plugins* oraz *Plux.NET*. Jednakże wszystkie wymienione narzędzia nie oferują wsparcia na tak wysokim poziomie, jak robi to *Nomad*. W ten sposób platforma *Nomad* wpasowuje się w lukę na rynku. Jest narzędziem, które narzuca pewne wymagania twórcom aplikacji, ale przez to oferuje o wiele prostsze *API* oraz serię gotowych, działających mechanizmów, które w znaczny sposób przyspieszą tworzenie jak i prototypowanie wszelkich projektów.

2.4 Architektura systemu Windows i platformy .NET

Budowa produktu zawsze w bardzo silnym stopniu zależy wybranej grupy docelowej czy technologii. W przypadku platformy *Nomad* taką technologią jest system operacyjny *Microsoft Windows* i *Microsoft .NET Framework*. Poniżej zostanie zaprezentowana wybiórcza charakterystyka tych środowisk.

2.4.1 Microsoft Windows

System Windows jest powszechnie znanym produktem [NM10]. Jego wewnętrzna budowa opisana jest dość szczegółowo w literaturze, dla przykładu seria książek „Windows Internals” [MER09] oraz inne publikacje związane z wydawnictwem *Microsoft Press*. Projektowana biblioteka wprawdzie korzysta z systemu Windows, ale architektura systemu z punktu widzenia omawianej platformy *Nomad* nie ma dużego znaczenia. Istnieje jednak kilka elementów, o których warto wspomnieć, gdyż stanowią historyczne już rozwiązanie problemów przedstawionych w pracy.

Jednym z najważniejszym aspektów systemu operacyjnego jest izolacja procesów jako odrębnych środowisk wykonania programów. W Microsoft Windows każdy proces może posiadać osobny kontekst graficzny. Komunikacja między procesami (w dowolny sposób) podlega wielu restrykcjom – najistotniejszą grupą są restrykcje bezpieczeństwa oraz możliwe ograniczenia wydajnościowe [MER09].

Historia rozwoju oprogramowania dla tego systemu pokazuje, że możliwe było stworzenie aplikacji działających na zasadzie *plug-in model*. Przykładów jest bardzo wiele, duża część z nich to rozwiązania opracowywane przez firmy dla konkretnego produktu. Sztandarowym przykładem jest tutaj pakiet *Microsoft Office* w wersjach *2000* oraz *XP*. Wykorzystuje on bowiem system dodatków (z ang. *add-ins*) korzystając z modelu *COM* [Sup04].

System Windows dopuszcza, przy użyciu modelu *COM* (patrz dodatek B) budowanie modułowych aplikacji dzielących jeden kontekst graficzny. Technologia budowy aplikacji wykorzystująca binarny interfejs komponentów obiektowych oraz innych elementów *IPC* (patrz dodatek B) jest immanentną cechą jądra systemu Windows od bardzo wczesnych wersji. Ma status dojrzałego produktu, aczkolwiek w żaden sposób nie wykorzystuje technologii *.NET* z racji jej młodego wieku [MER09].

Z punktu widzenia projektowania platformy dla środowiska *.NET* skorzystanie z dobrze znanych mechanizmów modelu *COM* i jego wersji późniejszych, byłoby krokiem wstecz. Co więcej, istota działania środowiska uruchomieniowego *.NET Framework* sprawia, że wykorzystanie mechanizmów systemu operacyjnego byłoby przeszkodą utrudniającą pracę zarówno twórcom *Nomada* jak i potencjalnym użytkownikom.

2.4.2 .NET Framework

.NET Framework uruchamia kod programów, bibliotek przy użyciu środowiska uruchomieniowego *CLR* (patrz dodatek B), które jest włączone jako serwer obiektów *COM+* [Ric10].

Uruchamiając najprostszy program napisany w kodzie pośrednim zgodnym z *.NET*, początkowo zostaje wykonany bardzo niewielki fragment kodu binarnego, którego zadaniem załadownie środowiska uruchomieniowego *CLR*. To początkowe uruchomienie włącza już mechanizmy systemu operacyjnego odpowiedzialne za utworzenie procesu systemowego, który zostaje klientem serwera *CLR* – istnieje tylko jedna „instancja" środowiska uruchomieniowego *.NETu*. W nowym procesie automatycznie tworzona jest tzw. *domena aplikacji* (patrz B) oraz *wątek środowiska .NET* (patrz B). Następnie sterowanie zostaje przekazane do programu napisanego w języku pośrednim [Ric10].

Assembly

Podstawą działania platformy *.NET* jest *Assembly* (patrz dodatek B), która to jest „lepszym" odpowiednikiem programów oraz bibliotek binarnych. *Assembly* może zawierać w sobie również zasoby oraz metadane dotyczące typów [Ric10],[JA10].

Assembly zazwyczaj odpowiada jednemu plikowi, aczkolwiek specyfikacja *.NETu* przewiduje możliwość wieloplikowych *Assembly*. Jednoplikowe *Assembly* posiadają zwyczajowo rozszerzenie *dll* albo *exe* [Mic03a].

Warto zwrócić uwagę na oznaczenie *dll*, które może być mylnie utożsamiane z *Dynamic Link Library*, a więc biblioteką binarną ładowaną w sposób dynamiczny w trakcie wykonywania programu. *Assembly* o rozszerzeniu *dll* nie ma z nią nic wspólnego, jest to bowiem wysokopoziomowa odpowiedź na bolączki wynikające z niskopoziomowego charakteru klasycznych bibliotek *DLL* – znane potocznie jako *„DLL Hell"*. Więcej w [Ric10].

Ładowanie dodatkowych Assembly

Po załadowaniu do pamięci operacyjnej, *Assembly* przynależy do domeny aplikacji. Specyfikacja *Frameworka .NET* nie przewiduje możliwości wyładowania pojedynczej *Assembly* z pamięci. Jedyną możliwością jest wyładowanie całej *domeny aplikacji* [Ric10].

Ta cecha produktu firmy Microsoft w znaczącym stopniu utrudnia projektowanie aplikacji pozwalających na dynamiczną pracę z modułami. Jednocześnie może ona uprościć pewne scenariusze, w których to wyładowaniu mają podlegać kolekcje zależnych od siebie Assembly – zamiast ustalać poprawną kolejność wyładowań, z pamięci od razu usuwana jest cała partia.

Proces ładowania Assembly do pamięci jest dość skomplikowanym łańcuchem pomniejszych zdarzeń, które zostały dokładnie opisane w [Mic03a, MSDN].

Najważniejszą informacją z punktu widzenia budowy *Nomada* jest fakt, iż przeszukiwanie odbywa się domyślnie w folderze, w którym uruchomiona była aplikacja. Tzw. ścieżka przeszukiwania jest powiązana z domeną aplikacji i może być zmieniana w trakcie wykonywania programu [JA10].

W systemie Windows pewne biblioteki używane są nadzwyczaj często, jednym z najważniejszych przykładów jest *Assembly System.Core*, będąca swego rodzaju podstawą działania wszystkich programów napisanych w językach zgodnych z *.NET* – zawiera ona podstawowe definicje typów oraz funkcji systemu operacyjnego [Ric10].

Pomimo tego, iż *System.Core* rezyduje w folderze instalacyjnym *Frameworka .NET* to jest domyślnie dołączana do większości uruchamianych programów. Dzieje się to za sprawą *Global Assembly Cache* (patrz dodatek B). Mechanizm przeszukiwania *GAC* jest wbudowany w platformę *.NET* i dość silnie z nią związany – dodanie *Assembly* do niego wymaga dodatkowych czynności [Mic03a].

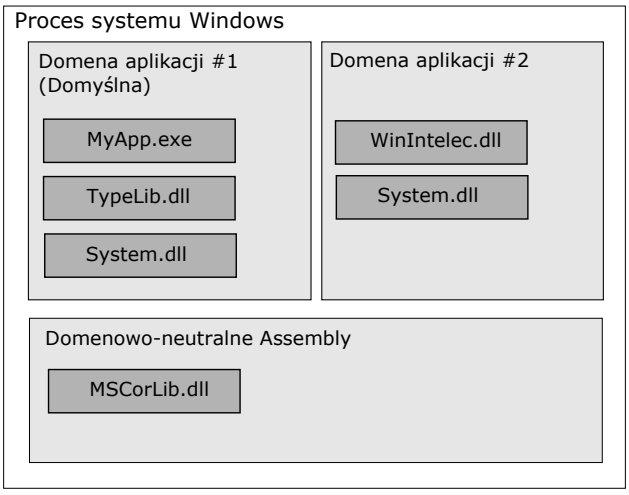
Mechanizm *GAC* rozwiązuje część problemów jakie stawia się platformie *Nomad* – sortowanie zależności referencyjnych oraz kwestie bezpieczeństwa. Niestety, jest on zbyt mało elastyczny oraz zbyt silnie ingerujący w system operacyjny by być efektywnie wykorzystanym nawet przez programy pisane w technologii *.NET*.

Domena aplikacji

W powyższym opisie wspomniano o *domenie aplikacji* jako o elemencie procesu systemu operacyjnego. Mechanizm ten jest specyficzny dla środowiska *.NET* i zgodnie z definicją – jest to logiczny kontener przeznaczony na *Assembly*. *Domena aplikacji* spełnia rolę podobną do procesu systemu operacyjnego, z tą różnicą że jest to element mechanizmów platformy *.NET*. Wprowadzenie dodatkowej warstwy umożliwiło autorom środowiska zapewnienie niezależnych od systemu operacyjnego mechanizmów izolacji, bezpieczeństwa [Ric10]

Domeny aplikacji wykorzystywane są intensywnie przez serwery *ASP.NET* oraz niekiedy przez aplikacje użytkowe.

Odpowiedzią środowiska *.NET* na brak możliwości wyładowania pojedynczej *Assembly* z pamięci jest możliwość wyładowania całej domeny aplikacji. Na jeden proces systemowy może przypadać wiele domen aplikacji. Każda z nich posiada własny kontekst bezpieczeństwa [JA10].



Rysunek 2.1: Domena aplikacji wewnątrz procesu systemu operacyjnego [Ric10]

Wątek środowiska .NET nie przynależy bezwzględnie do domeny aplikacji – może pod pewnymi warunkami przekraczać jej granicę. Warunki te opisane są przez technologię *.NET Remoting*, która leży u podstaw komunikacji intraprocesowej [Mic03a].

Możliwość przekraczania przez wątek granicy *domeny aplikacji* ma bardzo duże znaczenie dla projektowania *Nomada*. Pozwala ono mieć nadzieję na wykorzystanie mechanizmu *domen aplikacji* do realizacji platformy typu *plug-in model*.

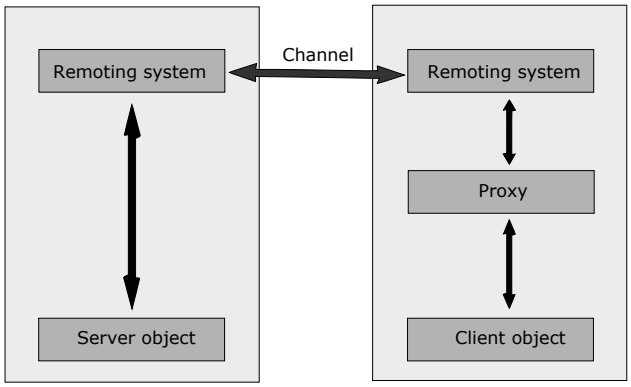
Komunikacja wewnątrzprocesowa

W przypadku komunikacji intraprocesowej, która jest obszarem zainteresowania platformy *Nomad*, istnieją dwa sposoby porozumiewania się:

- serializacja (z ang. serialization)
- marszalizacja (z ang. marshaling)

Serializacja to zamiana wymaganych informacji na surowy strumień bajtów i przesłania całości. Jest to rozwiązanie proste i skuteczne. Cechuje się jednak dość dużym narzutem wydajnościowym, im większy obiekt trzeba przesłać pomiędzy domenami tym gorzej. Nie należy do rozwiązań z rodziny *.NET Remoting*.

.NET Remoting jest technologią, w ramach środowiska *.NET* służącą do szeroko pojętej komunikacji. Rozwiązanie to wprowadza jednakowy interfejs do komunikacji niezależnie od konkretnej metody przesyłu danych, bazujący na interfejsie i zdalnym wywołaniu metod. Abstrahując od implementacji protokołów komunikacyjnych *.NET Remoting* stworzył podstawy do budowy wysokopoziomowych rozwiązań typu *WCF* (patrz dodatek B).



Rysunek 2.2: Działanie mechanizmu marshaling [Mic03a]

.NET Remoting jest jedną z metod pozwalających na komunikację pomiędzy *domenami aplikacji*. Bazując na rozwiązaniu zbliżonym do *RMI* (z ang. Remote Method Invocation – patrz dodatek B,[EFB04]) użytkownik zyskuje możliwość pracy na zdalnym obiekcie jakby znajdował się on lokalnie. W domenie macierzystej istnieje tylko jedna instancja obiektu marszalowanego, wszystkie inne zainteresowane domeny otrzymują jedynie informacje o interfejsie i położeniu oryginalnego obiektu. Informacje te potrzebne są do zbudowania przezroczystego proxy, które przejmie wywoływanie metod i komunikację. Ilość przesłanych informacji jest dużo mniejsza niż w przypadku serializacji. Poglądowo sytuację przedstawia rysunek 2.2.

Rozdział 3

Architektura rozwiązania

Pierwszym problemem, który należy rozwiązać, jest określenie wymagań względem platformy będących podstawą do zdefiniowania jej architektury. W typowych sytuacjach, wymagania te określają klienci. Klientami oprogramowania typu framework są najczęściej programiści. Dlatego, żeby je wyspecyfikować, w rolę klienta wcielili się promotor – dr. inż. Piotr Zielniewicz, grupa studentów ze specjalizacji Technologia Wytwarzania Oprogramowania oraz sami twórcy platformy.

3.1 Wymagania względem frameworka

Zdefiniowana grupa klientów produktu wyróżniła następujące zagadnienia, które platforma musi wspierać:

- Modułowość, ładowanie modułów dostarczanych przez:
  - osoby trzecie,
  - autorów aplikacji bazowej.
- Zaufanie wobec ładowanych modułów podpisanych za pomocą:
  - certyfikatów – w celu tworzenia zaufanych łańcuchów,
  - prostego mechanizmu opartego o klucz publiczny i prywatny.
- Aktualizacje:
  - możliwość aktualizacji automatycznej/sterowanej modułów,
  - mechanizm automatycznego przeładowania aplikacji po aktualizacji.
- Graficzny interfejs użytkownika:
  - współdzielenie fragmentów okien przez wiele modułów,
  - elastyczne zarządzanie fragmentami okien – dokładanie nowych elementów, korzystanie z już zdefiniowanych.
- Komunikacja międzymodułowa:
  - możliwość łatwej i asynchronicznej wymiany komunikatów pomiędzy różnymi modułami,
  - lokalizacja serwisów – znajdowanie dostawców usług definiowanych przez dowolne moduły.
- Wsparcie dla wielojęzykowości:
  - korzystanie z internacjonalizacji.

Powyższe wymagania wpasowują się jednak wyłącznie w kategorię wymagań funkcjonalnych. Dodatkowym wymaganiem jakie zostało postawione a nie jest wymienione w powyższej liście jest docelowa wersja platformy *.NET* – 3.5. W celu pełnej definicji zadań przedstawionych platformie *Nomad* należy je uzupełnić o szeroką gamę poniższych wymagań pozafunkcjonalnych.

Bezpieczeństwo

Nie można dopuścić by błędnie napisany moduł powodował awarię całej aplikacji. Programista musi mieć możliwość obsłużenia błędów oraz poinformowania pozostałych modułów o ponownym uruchomieniu aplikacji w celu zapisania swojego stanu i później odtworzenia go przy ponownym załadowaniu. Pożądanym jest również mechanizm ograniczenia modułom uprawnień, by programista bazujący na platformie mógł je świadomie im nadać.

Zarządzalność

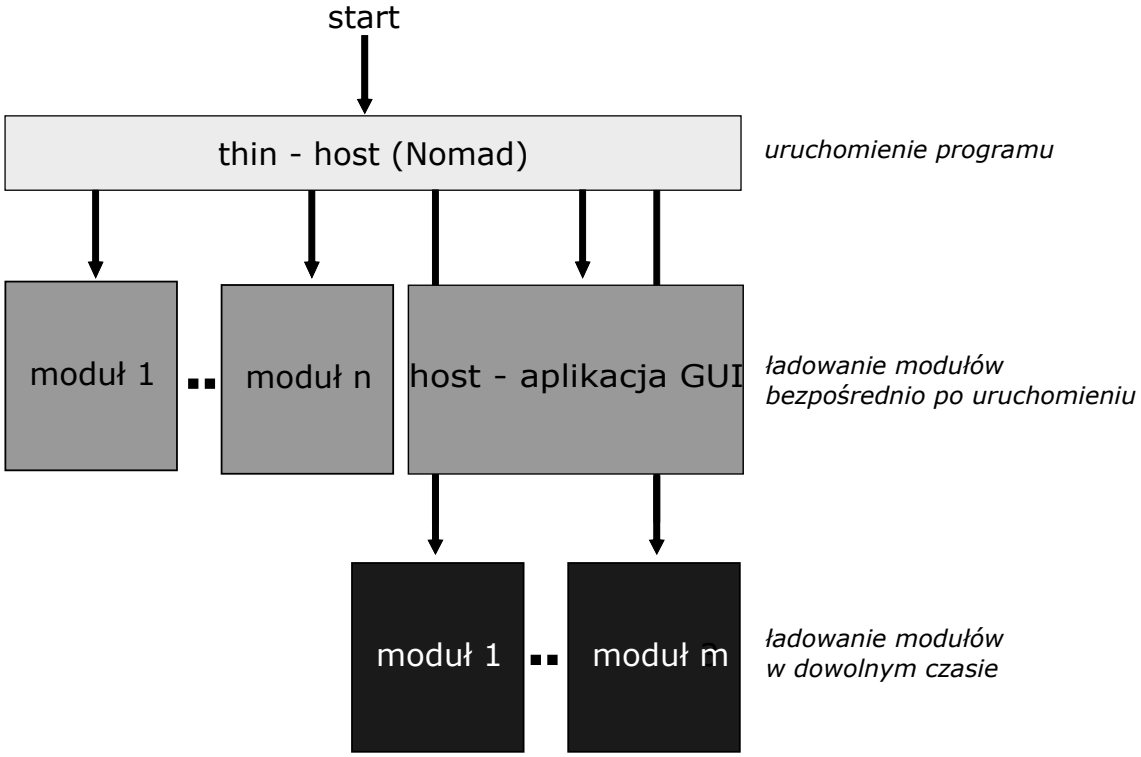
Kluczowym jest w aplikacji modularnej, by nie tylko ładowała ona zdefiniowany zestaw modułów w trakcie uruchomienia ale również pozwalała na modyfikację go w trakcie działania. Powinno odbywać się to na zasadzie dodawania nowych modułów z repozytorium. Niezbędny jest również mechanizm wyładowania modułów z pamięci operacyjnej celem ich aktualizacji.

Wydajność

Należy zachować kompromis pomiędzy izolacją modułów a ich wydajną współpracą. Platforma powinna ułatwiać współpracę modułów silnie ze sobą komunikujących się z minimalnym możliwym narzutem wydajnościowym wynikającym z mechanizmów bezpieczeństwa.

Wygoda obsługi

Platforma programistyczna musi ułatwiać pracę programiście a nie utrudniać ją poprzez skomplikowaną procedurę uruchomienia czy konfiguracji. Przez wyraźne nakreślenie granic odpowiedzialności poszczególnych komponentów rozszerzanie platformy powinno następować całkowicie niezależnie od reszty systemu. Stąd architektura sama w sobie powinna być możliwie modularna.



Rysunek 3.1: Architektura widoczna dla końcowego użytkownika platformy *Nomad*

3.2 Zarys architektury platformy *Nomad*

Powyższe wymagania wynikające z celów postawionych w pracy jak i wytycznych dotyczących ich realizacji pozwoliły stworzyć następującą architekturę *Nomada*.

Podstawą funkcjonowania platformy *Nomad* jest moduł. Przyjęto, że przez moduł będzie rozumiany *funkcjonalny fragment skompilowanego kodu przeznaczonego dla środowiska .NET w postaci Assembly, dostarczany przez programistę - klienta frameworku w celu rozszerzenia funkcji oferowanych przez aplikację.*

W pracy przyjęto, że zespół programistyczny – użytkownik *Nomada*, przygotowuje moduł *Shell* – szkielet kompozycji graficznego interfejsu użytkownika (*GUI*). Uzupełnia także, we własnym zakresie logikę biznesową, implementując ją w postaci modułów, bądź wcielając ją w moduł *Shell*. Moduł *Shell* stanowi pewnego rodzaju *hosta* – gospodarza dla pozostałych modułów, nie jest jednak uruchamiany bezpośrednio przez użytkownika.

W platformie *Nomad* wprowadzono bowiem koncepcję tzw. *thin-host* czyli bardzo krótkiego programu, którego zadaniem jest uruchomienie mechanizmów platformy oraz ewentualna ich konfiguracja. Dopiero ten program uruchamia moduł *hosta* oraz może załadować od razu inne moduły.

Przedstawiony na rysunku 3.1 sposób uruchomienia wynika z konieczności zapewnienia możliwości aktualizacji modułów oraz wsparcia dla aplikacji z graficznym interfejsem użytkownika.

Moduł *Shell* różni się względem innych modułów tym, iż tworzy wątek *GUI*. Wszystkie moduły, które używają *GUI*, muszą być zależne od *Shell*. Jeżeli moduły wykonują przetwarzanie niezależne od graficznego interfejsu użytkownika, nie są związane taką zależnością.

Architektura najwyższego poziomu

Platforma spełniająca wszystkie postawione wymagania jest złożonym projektem. Postępując zgodnie z zasadami budowania *frameworków* programistycznych wyróżniono pięć głównych aspektów funkcjonalności:

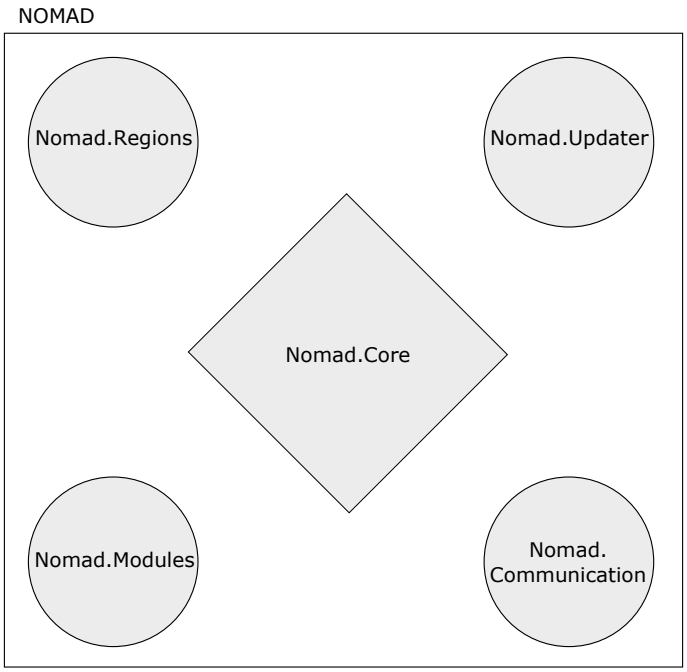
- rdzeń platformy – *Nomad.Core*
- zarządzanie modułami – *Nomad.Modules*
- środki komunikacyjne – *Nomad.Communication*
- mechanizm kompozycji interfejsu użytkownika – *Nomad.Regions*
- mechanizm aktualizacji – *Nomad.Updater*

Każdy z powyższych elementów zrealizowano jako przestrzeń nazw. Rysunek 3.2 przedstawia logiczne powiązanie pomiędzy tymi jednostkami. Centralnym ośrodkiem jest *Nomad.Core*, który pełni rolę integrującą oraz zapewnia dostęp do mechanizmów uruchamiających i konfigurujących *Nomada*. Przytoczone jednostki funkcjonalne omówiono w odpowiadających im rozdziałach.

Ograniczenia

Projektując *Nomada*, istniała potrzeba wprowadzenia pewnych ograniczeń, aby jasno zarysować podział odpowiedzialności pomiędzy *frameworkiem* a programistą go używającym.

- Tworzenie aplikacji opartej o *Windows Forms* jest ograniczone. Nie można wykorzystać wsparcia dla współdzielenia fragmentów okien (definiowanie regionów) – gdyż oparte są one o technologię *WPF*. Jednak pozostałe jednostki działają bez przeszkód.
- Mechanizm aktualizacji nie przewiduje dostarczenia wraz platformą narzędzi do aktualizacji *thin-hosta*. Związane jest to z mechaniką działania procesu uaktualniania oraz faktem, iż fragment kodu inicjalizujący całość aplikacji powinien służyć jedynie do uruchomienia platformy.
- Nie wspierana jest dynamiczna zmiana konfiguracji *frameworku*. Po wystartowaniu aplikacji wszelkie ustawienia konfiguracyjne dotyczące mechanizmów dostarczanych przez *Nomada* zostają zamrożone.



Rysunek 3.2: Architektura najwyższego poziomu platformy *Nomad*

Rozdział 4

Moduły

Moduły są najważniejszym elementem platformy *Nomad*. To właśnie dla nich powstała i dlatego cały proces tworzenia – od idei koncepcji do implementacji zaawansowanych mechanizmów wspierających sterowanie ich cyklem życia wymagał przemyślenia i dopracowania. Wiązało się to z przeprowadzeniem badań, syntetycznymi testami mechanizmów *.NET* oraz świadomą definicją jak i przetestowaną implementacją zaproponowanych rozwiązań.

4.1 Badania poprzedzające implementację modułu.

W systemie Windows istnieje szereg możliwości implementacji platformy *Nomad* zgodnie z zasadą *plug-in model*. Więcej informacji należy szukać w rozdziale 2. Każda z tych możliwości niesie ze sobą wady i zalety w kontekście uprzednio wymienionych kategorii. Dlatego decyzję o wybraniu konkretnej koncepcji realizacji modularności aplikacji opartych o platformę *Nomad* poprzedzono badaniem mechanizmów *.NET*, na których miała ona bazować.

4.1.1 Możliwe koncepcje realizacji

Poniżej zaprezentowano najbardziej istotne z możliwych rozwiązań, które spełniają minimalne wymagania postawione przed *frameworkiem*.

- moduł tożsamy z procesem, kernel tożsamy z drugim procesem
- moduł tożsamy z *domeną aplikacji*, rdzeń platformy tożsamy z osobną *domeną aplikacji*
- grupa modułów tożsama z *domeną aplikacji*, rdzeń platformy tożsamy z osobną *domeną aplikacji*

Moduł tożsamy z procesem systemu operacyjnego

Rozwiązanie to jest teoretycznym przykładem idealnej izolacji pomiędzy modułami a rdzeniem *Nomada*. Implementacja takiego rozwiązania jest jednak co najmniej trudna, ponieważ wsparcie systemu Windows dla budowy interfejsu użytkownika opartego o środowisko *.NET* jest silnie ograniczone [Mic03a]. Kwestia wydajności takiego rozwiązania jest sprawą dyskusyjną, można jednak domniemywać, iż system komunikacji międzyprocesowej byłby wąskim gardłem tak zbudowanego systemu, nawet przy wykorzystaniu istniejącego modelu komponentów binarnych (COM [Mic03a]).

Moduł tożsamy z domeną aplikacji

Jest to najbardziej „naturalne" rozwiązanie w środowisku *.NET*. Utożsamienie pojedynczego modułu z pojedynczą *domeną aplikacji* niesie bardzo wiele korzyści wynikających z właściwości samej domeny. Do największych zalet należą:

- izolacja – osiągnięta izolacja jest porównywalna z izolacją osiąganą przy użyciu procesu systemu operacyjnego [Ric10],
- bezpieczeństwo – system bezpieczeństwa środowiska *.NET* opiera się na *domenie aplikacji* ([Mic03a]), zastosowanie omawianego modelu modułowości pozwoliło by osiągnąć możliwości i elastyczność systemu bezpieczeństwa platformy *.NET* o ziarnie wielkości modułu [Ric10]

Największą wadą prezentowanego rozwiązania jest narzut wydajnościowy. Wynika on z tego, iż całość komunikacji pomiędzy modułami musiałaby być wykonywana przez mechanizm *.NET Remoting*. Wiązało by to się z koniecznością *serializacji* lub *marshallingu* wywołań metod pomiędzy modułami [Mic03a]. Więcej infromacji o *.NET Remoting* w dodatku 2.4.2.

Wszystkie moduły w jednej domenie aplikacji

Rozwiązanie wykorzystujące *domenę aplikacji* jako ziarno izolacji. W tym przypadku wszystkie moduły są grupowane w jedną *domenę aplikacji*. Drugą domenę stanowi domena rdzenia platformy.

Największą zaletą tego rozwiązania powinna być lepsza wydajność niż w rozwiązaniu wykorzystującym jedną domenę na moduł. Niestety wraz ze zwiększeniem „zasięgu" domeny poziom izolacji spada – wystarczy awaria w jednym z modułów aby wymusić przeładowanie wszystkich pozostałych. Również elastyczność rozwiązań dotyczących bezpieczeństwa jest znacznie niższa. W omawianym rozwiązaniu wszystkie moduły muszą mieć wspólną politykę bezpieczeństwa.

4.1.2 Badania praktyczne wydajności poszczególnych rozwiązań

Dwa ostatnie z przedstawionych rozwiązań uznano za konkurencyjne. Teoretycznie rozwiązanie utożsamiające moduł z *domeną aplikacji* ma przewagę nad rozwiązaniem z tylko dwiema domenami. W celu wyznaczenia lepszego rozwiązania w sposób ostateczny zostały przeprowadzone testy wydajności przytoczonych rozwiązań. Testy zostały przeprowadzone, ponieważ nie udało się odnaleźć wiarygodnych wyników, podsumowujących wydajność mechanizmów *.NET Remoting* w środowisku intraprocesowym. Jedyne znalezione wyniki, nie opisują w sposób wyczerpujący zagadnień istotnych z punktu widzenia projektantów *Nomada* [Abi09].

Cel prowadzenia testów wydajności i ich charakter

Pełne przetestowanie wspomnianych wcześniej koncepcji wymagałoby podwójnego zaimplementowania całości systemu dla każdej koncepcji osobno. W celu ograniczenia do minimum potrzebnej implementacji postanowiono wykonać pojedyncze testy obrazujące krytyczną różnicę między obydwiema koncepcjami – *szybkość komunikacji między domenami aplikacji w ramach jednego procesu systemu operacyjnego*. Cecha ta jest najbardziej znacząca, ponieważ w przypadku utożsamiania modułu z *domeną aplikacji* komunikacja pomiędzy modułami narusza granice domen. W przypadku kiedy by to naruszenie granic było bardzo kosztowne korzyści wynikające z tej koncepcji mogłyby zostać zniwelowane.

Wszystkie badania dotyczyły szybkości (czasu) wykonywania przedstawionych poniżej działań w przypadku kiedy to działanie przekraczało granicę domeny aplikacji oraz kiedy było przeprowadzane lokalnie. Ostatni przypadek stanowił *de facto* grupę kontrolną.

Dokładny opis przeprowadzonych testów wydajności

Kod badający jest zamieszczony i opisany w skrócie w dodatku D.2. Koncepcyjnie metody testowe dzielą się na trzy grupy.

Pierwszą grupę stanowi badanie czasu utworzenia obiektu. W środowisku *.NET* jest to działanie dość kosztowne przy czym w przypadku komunikacji międzydomenowej wiąże się z zestawieniem tzw. przezroczystego proxy (z ang. transparent proxy, marshalling - patrz rozdział 2.4.2). Spodziewano się więc długiego czasu działania. Warto zauważyć, iż tworzenie nowego obiektu pomiędzy domenami, w praktycznych i typowych zastosowaniach będzie wykonywane niezwykle rzadko. Grupę tę reprezentuje *Metoda 1*.

Drugą grupą testowanych metod są czynności polegające na wywołaniu przez domenę pierwszą metody z domeny drugiej. Przy czym przekazaniu ulegają parametry z domeny pierwszej. Jest to typowy przypadek jaki jest spodziewany podczas komunikacji między domenami – obrazuje on wiele mechanizmów działających podczas budowy złożonej aplikacji. Podczas testów różnicowaniu poddano obiekty przekazywane do wywołania: począwszy od braku argumentów przez argument prostego typu – *Int32*, po argument „złożony" będący serializowalną klasą składającą się z dwóch prostych pól (typy *String* i *Int32*). Grupę tą reprezentują kolejno *Metoda 2*, *Metoda 3* oraz *Metoda 4*.

Ostatni najbardziej złożony przypadek związany jest z trzema domenami. Analogicznie jak w poprzednich przypadkach zostaje wywołana metoda pomiędzy dwiema domenami. Jednak argumenty przekazane do zdalnego wywołania nie należą do domeny wywołującej, ale do innej –

trzeciej domeny. Taka sytuacja może występować dość rzadko w realnej aplikacji, aczkolwiek uznano, iż warto przeprowadzić test tego rodzaju. Grupę tą reprezentuje *Metoda 5.*

Podsumowanie badanych metod:

- *Metoda 1* – czas tworzenia nowego obiektu
- *Metoda 2* – czas wywołania metody bezargumentowej na obiekcie
- *Metoda 3* – czas wywołania metody z prostymi argumentami
- *Metoda 4* – czas wywołania metody ze złożonymi argumentami
- *Metoda 5* – czas wywołania metody z argumentami pochodzącymi z innej domeny

Wyniki badań

Testy wydajności rozwiązań zostały przeprowadzone na maszynie o konfiguracji:

- Intel Core 2 Duo T7300 - 2.0GHz
- 4GB pamięci RAM
- Windows 7 Professional w wersji 64 bitowej
- środowisko *.NET* w wersji 4.0 Extendend Profile

Podczas testów starano się zapewnić minimalne obciążenie maszyny innymi zadaniami. Niestety, specyfika budowy systemu Windows 7 uniemożliwia użytkownikowi pełną kontrolę nad systemem operacyjnym [MER09].

Zdefiniowano spojęcie badania. Na badanie składa się czas wykonania dziesięciu tysięcy wywołań wybranej metody (odpowiednio 1, 2, 3, 4, 5). Wynika to z faktu, iż jednokrotne wywołanie metody odbywa zbyt szybko – w czasie dążącym do zera. Dzięki temu wyeliminowano niedokładność pomiaru bardzo małych odcinków czasu. Każde z badań zostało powtórzone pięć razy. Na podstawie badań otrzymano *rezultaty*, które obrazują średnie i odchylenia standardowe każdego z badań. Optymalizator kodu *C#* został włączony celem uzyskania najlepszej wydajności w środowisku produkcyjnym.

Uśrednione wyniki wraz z odchyleniem standardowym prezentuje poniższa tablica.

Rezultaty badań	**W tej samej domenie**		**W osobnej domenie**	
	średnia [ms]	odchylenie [ms]	średnia [ms]	odchylenie [ms]
Rezultat 1	11,2	2,864	31790,2	1665,4859
Rezultat 2	116,4	4,037	650	21,564
Rezultat 3	227,8	12,438	800,8	31,586
Rezultat 4	242,8	5,02	1946,4	59,206
Rezultat 5	1148,4	35,67	5092	167,836

Tablica 4.1: Wyniki testów wydajnościowych

Podsumowanie wybranej koncepcji (wady i zalety)

Pokazane wyniki (tablica 4.1) pokazują, iż implementacja koncepcji, w której jednemu modułowi odpowiadała by jedna *domena aplikacji* jest niewydajna. Czas komunikacji, pomijając już tworzenie nowych instancji obiektów, jest około sześć razy dłuższy niż grupy kontrolnej. W świetle takiego stanu rzeczy wybrano rozwiązanie z dwiema domenami jako kompromis pomiędzy przyzwoitą wydajnością a elastycznością rozwiązania.

4.1.3 Implementacja

W technicznym odniesieniu do platformy moduł jest zawarty w *Assembly*, która posiada dokładnie jedną klasę implementującą interfejs *IModuleBootstraper*.

Moduły zgodne z zaprojektowaną platformą charakteryzują się pracą na zasadzie *odwrócenia sterowania (z ang. Inversion of Control) [Fow05]*. Programista *frameworka* uzupełnia jedynie wewnętrzne ciała funkcji. Funkcje jakie programista musi zaimplementować są zdefiniowane przy pomocy interfejsu *IModuleBootstraper*.

Listing 4.1: Definicja interfejsu IModuleBootstraper

```
public interface IModuleBootstraper{
  void OnLoad();

  void OnUnload();
}
```

Metoda *OnLoad* zostaje wywołana przez *Nomada* podczas ładowania modułu. Analogicznie metoda *OnUnload* zostaje wywołana przed odładowaniem modułu.

Praca na zasadzie *callback function* została wprowadzona w celu wymuszenia na użytkowniku platformy implementacji podanych powyżej dwóch metod. Wymuszenie osiągane jest przez mechanizm interfejsów środowiska *.NET* [Mic03a]. Ma na celu zwiększenie świadomości użytkownika *Nomada*, którego zadaniem jest explicite zaimplementowanie podanych metod, dzięki czemu może on (tj. użytkownik) uniknąć błędów. Doskonałym przykładem może być tutaj zapis przez moduł danych do pliku, tuż przed wyładowaniem.

4.2 Zarządzanie modułami

Centralnym punktem obsługi modułów w platformie *Nomad* jest klasa *ModuleManager*. Jest ona odpowiedzialna za ładowanie modułów oraz przechowywanie referencji do nich. Zadanie to wykonywane jest we współpracy z innymi obiektami, według poniższego schematu:

1. *ModuleManager* otrzymuje *IModuleDiscovery* za pomocą którego ma wyszukać moduły do załadowania.

2. *ModuleManager* filtruje moduły otrzymane w poprzednim kroku przy wykorzystaniu filtru zdefiniowanego w konfiguracji.

3. *ModuleManager* sprawdza wykorzystując *IDependencyChecker*, czy ładując znalezione moduły wszystkie zależności będą spełnione. Jeżeli nie, kończy przetwarzanie.

4. *ModuleManager* przy pomocy *IModuleLoadera* ładuje każdy moduł w kolejności wynikającej z sortowania topologicznego. W przypadku niepowodzenia ładowanie modułów jest przerywane.

Odpowiedzialności poszczególnych komponentów, z których korzysta *ModuleManager* zdefiniowano w następujący sposób.

ModuleDiscovery

Obiekt implementujący interfejs *IModuleDiscovery* odpowiedzialny jest za dostarczenie listy dostępnych modułów. Sposób, w jaki wyszuka dostępne moduły, pozostawiony został autorowi implementacji. Wraz z Nomadem dostarczane są trzy implementacje *IModuleDiscovery*:

- *SingleModuleDiscovery* – zapewnia „wykrycie" pojedynczego modułu. Wskazuje się ścieżkę do pojedynczego assembly'a, który ma zostać załadowany.

- *DirectoryModuleDiscovery* – po wskazaniu ścieżki do folderu wynajduje wszystkie moduły w nim się znajdujące. Możliwy jest wybór pomiędzy przeszukaniem tylko wskazanego folderu jak i wszystkich jego podrzędnych.

- *CompositeModuleDiscovery* – przyjmuje kolekcję *IModuleDisovery* i sumuje wyniki wszystkich *IModuleDiscovery* w nim zawartych. Implementuje wzorzec *Composite* [EFB04].

Dzięki zdefiniowaniu odpowiednich *ModuleDiscovery* (w szczególności *DirectoryModuleDiscovery*), konfiguracja *Nomada* nie musi być zmodyfikowana po przeprowadzeniu aktualizacji – w przypadku przeładowania aplikacji nowe i zaktualizowane moduły zostaną automatycznie wykryte przez mechanizmy.

Informacje o modułach zwracane przez *ModuleDiscovery* zawierają między innymi tak zwany *manifest modułu*.

Manifest modułu

Manifest modułu jest to plik *XML*, zawierający podstawowe informacje o module i jego wymaganiach. Manifest modułu tworzony jest przez generator manifestów (patrz 10.3.2).

Na moduł może składać się wiele plików. Część z nich to dodatkowe *Assemblies*, inne to np. konfiguracje modułu, zasoby graficzne, itp. W manifeście zawarte są informacje o wszystkich plikach wchodzących w skład modułu. W platformie *Nomad*, moduł jest opisywany także przez dodatkowe informacje, takie jak jego wystawca, wersja, zależności od innych modułów.
Do funkcji manifestu należy:

- identyfikowanie wersji modułu bez konieczności ładowania *Assembly* – przechowywanie informacji o numerze wersji modułu w pliku *XML* (patrz dodatek B) i możliwość odczytania jej na dowolnej platformie programowo-sprzętowej,
- weryfikowanie poprawności plików składowych modułu – sumy kontrolne wszystkich plików dostarczanych do modułu umożliwiające weryfikowanie oryginalności (spójności) plików,
- dostarczanie informacje o wystawcy – określenie klucza publicznego używanego do weryfikacja sygnatury manifestu,
- dostarczanie informacje o zależnościach – niezbędne przy ładowaniu modułów oraz współpracy z repozytorium.

Zawarcie w manifeście informacji o wersji modułu pozwala na przetwarzanie jej w aplikacjach nie opartych na *.NET*. Przykładem takiej aplikacji może być serwer repozytorium modułów. Sumy kontrolne plików pozwalają na weryfikacje, czy pliki należące do modułu nie zostały zmodyfikowane przez trzecią stronę. Weryfikacja ta odbywa się w szczególności przy instalacji aktualizacji pobranych z Internetu. Informacja o wystawcy pozwala na stwierdzenie, czy moduł jest zaufany czy nie, oraz wyświetlenie ewentualnego ostrzeżenia.

Kluczowa z punktu widzenia zarządzania modułami jest jednak lista zależności. W manifeście zawarte są nazwy wymaganych modułów oraz ich minimalne wersje. Na podstawie tych informacji mechanizm rozwiązywania zależności stwierdza, czy ładowana konfiguracja jest poprawna czy nie.

Zaufanie do informacji zawartych w manifeście (w szczególności do sum kontrolnych oraz informacji o wystawcach) możliwe jest dzięki zastosowaniu podpisu cyfrowego dla manifestu. Podpis manifestu został szczegółowo opisany w punkcie 4.3.2.

Manifesty dostępnych modułów przekazywane są przez manager modułów do *filtrów*.

ModuleFilter

Filtry definiują arbitralne ograniczenia. Analizują każdy spośród dostępnych modułów i na podstawie informacji zawartych w jego manifeście podejmują decyzję o dopuszczeniu lub zablokowaniu jego ładowania. Filtry przykładowo mogą blokować moduły o za dużym rozmiarze lub np. posiadające zabroniony format pliku.

Jednym z zaimplementowanych w *Nomadzie* filtrów jest *SignatureModuleFilter*. Odpowiada on za zweryfikowanie, czy moduł posiada prawidłową sygnaturę – jeżeli tak nie jest, blokuje ładowanie modułu. Dzięki zdefiniowaniu w Nomadzie filtra typu *CompositeModuleFilter* możliwe jest definiowanie kolekcji filtrów, które zostaną zastosowane.

Lista modułów po przefiltrowaniu trafia do mechanizmu zarządzania zależnościami między modułami

Zależności między modułami

W programach modularnych często występuje sytuacja, w której jeden moduł potrzebuje do działania usług dostarczanych przez inny moduł. O takiej sytuacji mówi się, że moduł *zależy*

od innego modułu. Zależność jest spełniona, jeśli moduł dostarczający wymagane usługi jest dostępny (może zostać załadowany). Aby zagwarantować poprawne działanie programu, należy zapewnić, że wszystkie zależności są spełnione oraz załadować moduły w takiej kolejności, aby moduły dostarczające usługi zostały załadowane przed modułami, które z tych usług korzystają.

Platforma *.NET* posiada wbudowany mechanizm rozwiązywania zależności między *Assembly*. Mechanizm ten zakłada, że *Assembly* zostanie załadowane dopiero wtedy, kiedy zostanie po raz pierwszy użyte. Możliwa jest sytuacja, w której załadowane zostanie *Assembly*, którego zależności nie są spełnione (brakuje jednego lub wielu *Assembly*, do których istnieją odwołania). Dopiero w momencie próby użycia klas pochodzących z niedostarczonych *Assembly* zgłoszony zostanie wyjątek, co zakłóci poprawną pracę programu. Takie zachowanie jest pożądane ze względu na wydajność działania platformy.

Aplikacja modularna nie powinna jednak dopuścić do sytuacji, w której brak jednego z wymaganych modułów (lub niepowodzenie jego ładowania) powoduje awarie całej aplikacji. Jednym z celów postawionych *Nomadowi* było stworzenie mechanizmu, który pozwalałby tak sterować ładowaniem modułów, aby opisana powyżej sytuacja nie występowała.

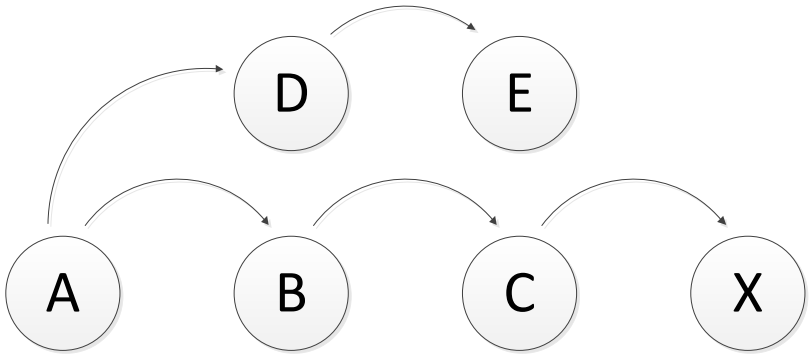
Mechanizm ten analizuje moduły i ustala, czy zależności wszystkich wybranych do załadowania modułów mogą być spełnione. Ponadto, ustala poprawną ze względu na zależności kolejność ładowania modułów. Również w sytuacji, gdy działającemu zestawowi modułów należy zainstalować aktualizacje, *Nomad* musi określić, czy po aktualizacji zestaw modułów wciąż będzie poprawny i możliwy do załadowania.

DependencyChecker

Klasa *DependencyChecker* definiuje algorytm, który odpowiada za sprawdzenie spójności podanej konfiguracji modułów (czy wszystkie zależności są we właściwy sposób spełnione). Dzięki zdefiniowaniu interfejsu(*IDependencyChecker*), programista może zaimplementować własny algorytm weryfikacji zależności – może np. w wybranej aplikacji zablokować możliwość pobrania i załadowania dwóch wykluczających się modułów. Wystarczy, że zaimplementuje wskazany interfejs i podmieni go w konfiguracji.

Zaimplementowany algorytm rozwiązywania zależności

Mechanizm *Nomada* wczytuje manifesty modułów i na ich podstawie buduje graf zależności. Następnie wykonywany jest na nim algorytm przeszukiwania w głąb *DFS* [THC01]. Dzięki temu identyfikujemy graf zależności jako acykliczny, skierowany (*DAG*) (patrz dodatek B) i tym samym listę wynikającą z jego sortowania topologicznego jako kolejność, w której należy załadować moduły.



Rysunek 4.1: DAG obrazujący przykładowy zestaw zależności

Dla grafu 4.1 poprawna kolejność ładowania modułów zgodna z algorytmen sortowania topologicznego to X,C,B,E,D,A.

Wspomniany algorytm umożliwia również wykrycie grafów cyklicznych, które prezentują błędy programistów modułów. *Nomad* zabezpiecza aplikację przed załadowaniem modułów cyklicznie

od siebie zależnych. Jednakże to programista jest odpowiedzialny za dostarczenie modułów z poprawnymi zależnościami.

Po ustaleniu kolejności, informacje o modułach przekazywane są kolejno do obiektu implementującego interfejs *IModuleLoader*.

ModuleLoader

Klasa implementująca interfejs *IModuleLoader* odpowiedzialna jest za:

- załadowanie *Assembly* zawierającego moduł,
- odnalezienie w ramach tego *Assembly* klasy *bootstrapera*,
- utworzenie obiektu *boostrapera*,
- wywołanie metody *OnLoad*

Ponadto, *ModuleLoader* powinien przechowywać referencje do wszystkich *bootstraperów*, które utworzył. W trakcie wyładowywania modułów, *ModuleLoader* odpowiedzialny jest za powiadomienie modułów – poprzez wywołanie metody *OnUnload*.

Implementacja *IModuleLoadera* zawarta w *Nomadzie* opiera się o kontener *IoC Castle Windsor*. Pozwala to twórcy modułu na pobranie usług, których potrzebuje ich moduł, poprzez umieszczenie ich jako parametrów konstruktora. Dopasowanie usług do parametrów realizowane jest przez kontener. Dzięki sortowaniu toplogicznemu modułów jest zagwarantowane, że w momencie tworzenia obiektu *bootstrapera* wszystkie wymagane przez ten obiekt usługi będą już zarejestrowane.

Domyślny *ModuleLoader* tworzy dla każdego modułu osobny subkontener. Oznacza to, że wszelkie rejestracje dokonane przez moduł na tym kontenerze będą wpływały jedynie na działanie tego modułu. Ponadto, dzięki zachowaniu relacji między głównym kontenerem a subkontenerem, dla modułu dostępne są wszystkie usługi zawarte w głównym kontenerze. Fakt ten wykorzystywany jest przez *ServiceLocatora* (patrz rozdział 6.2). W praktyce oznacza to, że twórca modułu może dowolnie wykorzystywać kontener użyty do tworzenia *bootstrapera*. Aby uzyskać referencje do tego kontenera, należy dodać do konstruktora parametr typu *IWindsorContainer*. Aby udostępnić usługę innym modułom, należy użyć *ServiceLocatora*.

Ze względu na wygodę użycia, *ModuleLoader* zapewnia wywołanie metod *OnLoad* oraz *OnUnload* w wątku *GUI*. Twórcy modułu mogą więc zakładać, że w metodach tych mogą ingerować w interfejs użytkownika (np. z wykorzystaniem opisanych w rozdziale 5 regionów). W przypadku modułów uruchamianych przed utworzeniem *GUI* metody zostaną wywołane w wątku ładującym moduły. Ponieważ jednak moduły te na pewno nie zależą od modułu zawierającego *GUI*, i tak nie mogą ingerować w interfejs użytkownika. Fakt niewykonania metod w wątku *GUI* nie powinien więc mieć znaczenia.

4.3 Bezpieczeństwo modułów

Postanowiono zbadać czy mechanizmy dostarczane w platformie *.NET* są wystarczające do zapewnienia integralności i autentyczności paczek zawierających moduły *Nomada*.

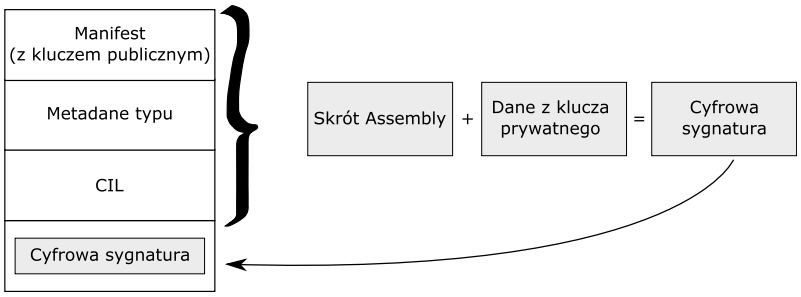
4.3.1 Podpisywanie *Assembly* .NET

Jednym z kroków, które programista może podjąć w czasie generowania *Assembly* platformy .NET jest jego podpisanie. Po wykonaniu tej czynności do wynikowego *Assembly* dołączany jest klucz publiczny i podpisany *skrót Assembly*. Wynikowy plik przedstawiony jest na rysunku 4.2 [Tro07].

Przedstawione rozwiązanie zostało wprowadzone do platformy *.NET* w celu zapewnienia tzw. „*Strong-Named Assemblies*". W języku polskim opisano to jako unikatowe nazwy (silnie nazwane) *Assembly*. Celem takiego rodzaju nazewnictwa jest możliwość jednoznacznej identyfikacji *Assembly* [Mic03a].

Na silnie nazwane *Assembly* składają się następujące cechy [Micc]:

- Zwykła nazwa tekstowa.
- Numer wersji.



Rysunek 4.2: Realizacja podpisów *Assembly* w .NET [Tro07]

- Opcjonalnie informacje o kulturze.
- Klucz publiczny i sygnatura.

Tak podpisane *Assembly* zapewnia, że wygenerowany plik jest globalnie unikatowy, to jest że spełni następujące wymagania [Micc]:

- Zagwarantuje unikalność *Strong-name* poprzez poleganie na parze kluczy (publicznym i prywatnym). Nikt nie może wygenerować tej samej nazwy *Assembly*, ponieważ nazwa wygenerowana z użyciem innego klucza prywatnego będzie inna.
- Zagwarantuje, że kolejne wersje wskazanego *Assembly* pochodzą od tego samego dostawcy – nikt nie może bowiem wygenerować podpisać ich tym samym kluczem prywatnym (klucz publiczny nie ulega zmianie).
- Zapewni, że zawartość *Assembly* nie uległa zmianie od czasu kompilacji.

Jak widać mechanizm podpisywania *Assembly* nie jest w stanie zapewnić weryfikacji integralności całego modułu (w skład którego wchodzą zarówno *Assembly* jak i inne zasoby – takie jak pliki graficzne, dźwiękowe, etc.). Nie zapewnia także możliwości weryfikacji dostawcy – np. z wykorzystaniem PKI (patrz dodatek B). Dlatego też w Nomadzie postanowiono podpisywać moduły za pomocą własnych mechanizmów.

4.3.2 Podpisywanie manifestów *Nomad*

Nomad oferuje dwa rozwiązania potwierdzania autentyczności modułu wykorzystujące podpisy cyfrowe:

- z wykorzystaniem infrastruktury klucza publicznego,
- algorytmem RSA, bez wykorzystania infrastruktury klucza publicznego.

Korzystając z weryfikacji autentyczności plików poprzez algorytm RSA klucz publiczny musi być dostarczony producentowi aplikacji innym kanałem. Podpisy wszystkich plików zawarte w manifeście są przeliczane w trakcie procedury ładowania modułu. Złośliwy kod, chcąc obejść mechanizm bezpieczeństwa, musiałby zablokować wykonywanie procedury ładowania i podmienić pliki już po ich weryfikacji.

Infrastruktura klucza publicznego

Wykorzystując tę opcję, programisty aplikacji nie interesuje pochodzenie modułu. Z jego punktu widzenia wszystkie moduły podpisane przez wystawców są bezpieczne i mogą być załadowane. Wystarczy, że ich autentyczność została potwierdzona przez zaufane (w systemie) centrum certyfikacji. Dodatkowo zaoferowano programistom możliwość zdefiniowania innych centrów certyfikacji rozpatrywanych jako zaufane (bez umieszczania ich w kontenerze systemowym).

Klucz publiczny i prywatny, RSA

Programista aplikacji może także zdefiniować wystawców zaufanych poprzez dostarczone przez nich klucze publiczne. Takie rozwiązanie upraszcza pewne scenariusze wynikające z zastoswania *PKI*. Istnieje mianowicie duże prawdopodobieństwo, że żaden z wystawców nie będzie miał możliwości uzyskania kwalifikowanego podpisu, którym mógłby podpisywać moduły. Wystarczy wtedy, by dostarczył programiście aplikacji swój klucz publiczny.

Należy mieć jednak na uwadze, iż nie będzie możliwości załadowania modułu nieznanego wystawcy, jeżeli jego klucz publiczny nie został dołączony do aplikacji. Dołączenie go do aplikacji oznacza konieczność wydania jej nowej wersji.

Typowe scenariusze bezpieczeństwa

Na podstawie wyżej opisanych mechanizmów można zrealizować następujące przypadku użycia:

- Brak weryfikacji podpisów – Programista aplikacji może zdefiniować, że każdy dostarczony moduł jest zaufany i nie wprowadzić żadnych ograniczeń. Takie podejście jest spotykane w przypadku wybranych projektów społecznościowych (np. *Eclipse*) – to użytkownicy filtrują moduły, którym ufają bądź nie.

- Wyłącznie potwierdzone przez zaufane centrum – Programista aplikacji dostarcza repozytorium zaufanych modułów. Domyślnie żadne moduły dostarczone przez programistów nie mogą być załadowane w aplikacji, nie pojawiają się w repozytorium. Dopiero po przejściu weryfikacji przez wydawcę aplikacji, moduły takie mogą zostać zaakceptowane jako potwierdzone przez dowolne kwalifikowane centrum.

- Wyłącznie zaufani wystawcy – zdefiniowany klucz publiczny. Definiujemy zaufanych wystawców za pomocą ich kluczy publicznych. Tylko moduły, które są podpisane przez zaufanego wystawcę mogą zostać załadowane.

- Podejście mieszane – Programista aplikacji definiuje, iż niektórzy dostawcy dostarczyli klucz publiczny, pozostali zaś korzystają z infrastruktury *PKI*.

Rozdział 5

Regiony – koncepcja modułowego interfejsu użytkownika

5.1 Problem kompozycji interfejsu użytkownika

Jednym z wymagań postawionych platformie *Nomad* jest udostępnienie modułom możliwości wyświetlania własnych elementów GUI w ramach okien aplikacji. Niniejszy rozdział zawiera przegląd potencjalnych rozwiązań oraz opisuje rozwiązanie zaimplementowane w platformie *Nomad*.

5.2 Przegląd potencjalnych rozwiązań problemu

Projektując rozwiązanie problemu autorzy zapoznali się z wybranymi rozwiązaniami problemu kompozycji interfejsu użytkownika.

5.2.1 Rozwiązania specyficzne dla aplikacji

Programiści decydujący się na takie rozwiązanie definiują zestaw punktów rozszerzenia (patrz dodatek B). Punkty te umożliwiają autorom rozszerzeń umieszczanie pewnych elementów interfejsu użytkownika, lub rozszerzanie istniejących. To autor aplikacji musi jednak z góry zdefiniować jakiego rodzaju elementy będą przyjęte przez każdy punkt rozszerzenia oraz w jaki sposób zostaną obsłużone.

Z jednej strony, rozwiązanie takie daje pełną kontrolę nad tym, co mogą modyfikować programiści wtyczek. Rozwiązanie umożliwia stworzenie pewnego rodzaju abstrakcji – autorzy rozszerzeń dostarczają opis elementów interfejsu użytkownika, a nie gotowe elementy interfejsu. Dzięki temu, rozszerzana aplikacja zachowuje kontrolę nad spójnym wyglądem aplikacji. Abstrakcje takie wymuszają również spełnienie wszystkich wymagań aplikacji. Ponadto, aplikacja może dostarczać dodatkowe usługi dla autorów wtyczek.

Przykładem zastosowania takiego rozwiązania problemu kompozycji interfejsu użytkownika jest pakiet Microsoft Visual Studio. Jednym z dostarczonych punktów rozszerzeń jest możliwość dodawania elementów do menu aplikacji. Host (czyli Visual Studio) wymaga jednak dostarczenia deskryptorów elementów, a nie gotowych elementów. Dzięki temu, pakiet może zweryfikować, czy programista wtyczki dostarczył wszystkich wymaganych informacji (tj. nazwy operacji, wyświetlanego użytkownikowi opisu oraz komendy wywoływanej w odpowiedzi na kliknięcie elementu). Wstawione w ten sposób elementy menu stają się automatycznie konfigurowalne – użytkownik może pokazywać, chować oraz przestawiać wszystkie elementy, nadawać im skróty klawiaturowe, dodawać do pasków narzędziowych, itd.. Ponadto, komendy stają się dostępne z poziomu makr, a także innych dodatków.

Z drugiej jednak strony, rozwiązanie to wprowadza ograniczenia. Wykorzystując punkt rozszerzenia autor wtyczki może dodać do interfejsu użytkownika wyłącznie to, co przewidział autor aplikacji. W przytoczonym powyżej przykładzie pakietu Microsoft Visual Studio, niemożliwe byłoby (z wykorzystaniem tego punktu rozszerzenia) dodanie do paska menu np. okienka wyszukiwania, znanego chociażby z przeglądarek internetowych. W takiej sytuacji, jeśli aplikacja nie przewidywałaby odpowiedniego punktu rozszerzenia, stworzenie wtyczki byłoby niemożliwe bez ingerencji twórców aplikacji.

Ze względu na indywidualny charakter tego rozwiązania, nie może ono być uogólnione. Przez to framework może wspierać tworzenie tego rodzaju rozwiązań jedynie w ograniczonym stopniu.

5.2.2 Rozwiązania oparte o regiony – wprowadzenie

Dwa ogólniejsze rozwiązania, zaprezentowane w *Composite Application Guidance* ([Mic09]), opierają się o koncepcję regionu. Logiczny region zdefiniowany został jako nazwany obszar interfejsu użytkownika, do którego dodawane mogą być widoki. To, co może być widokiem, jest definiowane przez kontrolkę, do której przypisany został region. W ogólności może to być dowolny element interfejsu użytkownika.

Rysunek 5.1: Graficzne przedstawienie koncepcji regionu

Jeden region skupia w sobie logicznie powiązane widoki (np. wszystkie otwarte edytory, wszystkie elementy menu). Programista aplikacji udostępniając region nie musi deklarować precyzyjnie w którym miejscu okna zostaną wyświetlone te widoki, ani w jaki sposób dokładnie będą prezentowane. Powinien natomiast określić ogólnie jakiego rodzaju będzie to region, nie uwzględniając szczegółów implementacyjnych. Może również zdefiniować pewne wymagania, które muszą spełnić widoki dodawane do regionu (np. konieczność implementacji określonego interfejsu).

Takie rozwiązanie pozwala rozdzielić wygląd aplikacji od logicznych powiązań między widokami. Ponadto, jest to rozwiązanie ogólniejsze niż zaprezentowane powyżej. Wadą tego rozwiązania jest utrata pełnej kontroli nad wyglądem i zachowaniem interfejsu użytkownika.

Composite Application Guidance przytacza 2 rodzaje wykorzystania mechanizmu regionów – View injection oraz View discovery. Rozwiązania te różnią się jedynie sposobem dodawania widoków do regionów.

View injection - wstrzykiwanie widoków

W tym rozwiązaniu na programiście modułu spoczywa obowiązek utworzenia widoku oraz dodania go do regionu. Programista ma pełną kontrolę nad tym, kiedy i do którego regionu dodany zostanie widok. Rozwiązanie to wymaga jednak, aby w momencie dodawania region już istniał. Utrudnia to sytuacje, w której programista chciałby dodać określony widok do każdego regionu spełniającego pewne warunki (np. każda instancja edytora tekstu mogłaby zostać wzbogacona o widok wyszukiwarki). Aby dodać widok, programista musiałby obsługiwać odpowiednie zdarzenie (o ile takowe jest publikowane) i w odpowiedzi na nie tworzyć i dodawać widok.

View discovery - odkrywanie widoków

View discovery opiera się na deklaratywnym określaniu widoków, które mają zostać dodane do określonych regionów. To framework powinien zapewnić, że niezależnie od momentu utworzenia regionu, widoki zostaną dodane do każdego spełniającego warunki regionu. W najprostszej postaci (opisanej w [Mic09]) programista modułu deklaruje, że do każdego regionu o określonej nazwie ma zostać wstawiony widok określonego typu. Aby zapewnić większą elastyczność, należałoby jednak

zapewnić możliwość wpłynięcia na sposób tworzenia tego widoku (np. poprzez przekazanie metody fabrycznej, a nie samego typu).

Rozwiązanie to upraszcza niektóre scenariusze, jednak kosztem utraty pełnej kontroli nad momentem dodania widoku do regionu.

5.3 Przyjęte założenia i ograniczenia

Ponieważ założeniem platformy *Nomad* jest dostarczanie rozwiązań ogólnych, autorzy pracy zdecydowali się na implementację mechanizmu *View Injection*. Jak opisano powyżej, rozwiązania specyficznie nie dają się w łatwy sposób uogólnić. Rozwiązanie *View Discovery* jest natomiast mniej ogólnie niż *View Injection* – nie zapewnia pełnej kontroli nad momentem dodania widoków do regionu.

Poniżej zaprezentowano przyjęte założenia odnośnie działania mechanizmu *View Injection*:

- aplikacja może zdefiniować wiele regionów,
- regiony muszą mieć unikalną w obrębie całej aplikacji nazwę,
- aby dodać widok do regionu, wystarczy znać jego nazwę, oraz ew. dodatkowe wymagania stawiane widokom,
- mechanizm powinien wspierać GUI napisane w *Windows Presentation Foundation*.

5.4 Użycie mechanizmu regionów

Centralnym punktem mechanizmu regionów w platformie *Nomad* jest obiekt *RegionManagera*. Zapewnia on dostęp do wszystkich utworzonych regionów oraz możliwość tworzenia nowych. Obiekt *RegionManagera* powinien zostać utworzony przez programistę aplikacji, tworzącego moduł *Shell* (zawierający interfejs użytkownika). Również programista aplikacji decyduje jakie typy regionów będą obsługiwane (poprzez dołączenie wybranych *IRegionAdapterów*) oraz jakie elementy interfejsu użytkownika powiązane będą z regionami (tworząc odpowiednie regiony). Programiści modułów również mogą tworzyć własne regiony.

Tworzenie RegionManagera

Aby utworzyć *RegionManagera*, należy wybrać fabrykę, która zostanie użyta do tworzenia nowych regionów. Jeśli programista zdecyduje się na wybór dostarczonej wraz z platformą *Nomad* fabryki, będzie musiał również utworzyć i dostarczyć wybrane *IRegionAdaptery*. Programista udostępnia *RegionManagera* innym modułom, rejestrując go w *ServiceLocatorze*.

Listing 5.1: Tworzenie RegionManagera

```
var regionManager = new RegionManager(new RegionFactory(GetRegionAdapters()));
serviceLocator.Register(regionManager);

private IRegionAdapter[] GetRegionAdapters() { ... }
```

Definiowanie regionów

Programiście aplikacji udostępniono dwie metody definiowania regionów:

- Imperatywnie, w kodzie programu
- Deklaratywnie, w kodzie XAML interfejsu użytkownika

Dodanie regionu z poziomu kodu odbywa się poprzez wywołanie metody *RegionManager.AttachRegion* (fragment kodu 5.2).

Listing 5.2: Dodanie regionu z poziomu kodu

```
regionManager.AttachRegion("regionName", regionHost);
```

Przy czym:

- regionName to nazwa regionu
- regionHost to kontrolka, która ma prezentować widoki dodane do regionu

Aby dodać region z poziomu kodu XAML, należy nadać wartość tzw. attached property *RegionManager.RegionName*. Należy pamiętać, że WPF wymaga, aby zadeklarować najpierw odwołania do przestrzenii nazw. Sposób wykorzystania attached property pokazuje fragment kodu 5.3.

Listing 5.3: Dodanie regionu z poziomu XAML

```
<Window (...)
  xmlns:Regions="clr-namespace:Nomad.Regions;assembly=Nomad">
  <Grid>
    <TabControl
      Regions:RegionManager.RegionName="mainTabs" />
  </Grid>
</Window>
```

Proces tworzenia regionów

Oba przytoczone fragmenty kodu uruchamiają proces tworzenia regionu. Proces ten zaczyna się w obiekcie *RegionManagera*, który sprawdza, czy nazwa regionu jest unikalna. Następnie żądanie utworzenia regionu jest przekazywane do *IRegionFactory*, które wyszukuje pasujący *IRegionAdapter* i jemu zleca utworzenie regionu. Każdy *IRegionAdapter* definiuje, jaki typ kontrolki obsługuje. *IRegionFactory* wyszukuje najbardziej dopasowany adapter na podstawie typu kontrolki, do której przypisany ma być region. Przez dopasowanie rozumiemy zgodność typów – wspieranego przez adapter oraz typu kontrolki do której przypisany ma być region. Wybierany jest adapter wspierający najbardziej specyficzny typ.

Wstawienie widoku do regionu

Aby wstawić widok do regionu należy najpierw pobrać odpowiedni region z *RegionManagera*, za pomocą metody *GetRegion*. Należy zauważyć, że metoda *GetRegion* zgłosi wyjątek, jeśli region o żądanej nazwie nie istnieje. Użycie tej metody jest równoznaczne z deklaracją wymagania - moduł pobierający region wymaga, aby ten region istniał. Aby umożliwić modułom wcześniejsze sprawdzenie, czy region istnieje, zaimplementowano metodę *ContainsRegion*.

Gdy uda się pobrać region, dodanie widoku odbywa się za pomocą metody *IRegion.AddView*. W celu usunięcia widoku należy wywołać metodę *IRegion.RemoveView*. Ponadto, widoki można również aktywować (metoda *IRegion.ActivateView*) i deaktywować (*IRegion.DeactivateView*). Dokładne znaczenie aktywności widoku jest definiowane w zależności od typu kontrolki z którą powiązany jest region.

Wykorzystanie wzorców MVVM, MVP i innych

Użycie mechanizmu regionów nie blokuje, ani nie wymusza użycia wzorców projektowych takich jak Model-View-ViewModel czy Model-View-Presenter. Programista modułu może w dowolnym zakresie wykorzystywać wybrany przez siebie wzorzec. Mechanizm regionów wymaga jednak, aby dostarczono mu obiekt widoku, a nie *Presentera* czy *ViewModelu*. Platforma nie wymaga, aby wybrany wzorzec był stosowany spójnie we wszystkich modułach – każdy programista może dokonać innego wyboru.

5.5 Przykładowe adaptery

Wraz z platformą *Nomad* przygotowane zostały przykładowe *RegionAdaptery*.

TabControl

Kontrolka typu *TabControl* może zostać wykorzystana np. do wyświetlenia wielu otwartych dokumentów. Adapter przystosowujący tego rodzaju kontrolkę do współpracy z mechanizmem regionów, wykorzystuje poniższe adaptery:

- SynchronizeItemsBehavior
- SynchronizeActiveItemsBehavior
- ActiveAwareBehavior
- TabControlHeaderTemplateSelectorBehavior

Taki wybór zachowań powoduje, że każdy widok dodany do regionu jest prezentowany jako jedna zakładka. Aktywnym widokiem będzie ten widok, który odpowiada obecnie wyświetlanej zakładce. Widoki mogą uzyskać informacje o wybraniu przez użytkownika zakładki lub jej opuszczeniu – wystarczy, że zaimplementują interfejs *IActiveAware*. To pozwala np. na wyświetlenie albo ukrycie przycisków na pasku narzędziowym, które dotyczą tylko określonych typów dokumentów. Ponadto, programista dodający widok może ustawić tytuł zakładki – implementując interfejs *IHaveTitle*. Programista aplikacji musi dostarczyć jednak szablon o nazwie „Nomad_ItemWithTitle".

StatusBar

Adapter dla kontrolek typu *StatusBar* synchronizuje kolekcje widoków (zachowanie *SynchronizeItemsBehavior*). Znaczenie aktywacji/deaktywacji widoku nie zostało zdefiniowane.

ToolBarTray

Adapter dla kontrolek typu *ToolBarTray* synchronizuje kolekcje widoków. Wymaga jednak, aby każdy z widoków był typu *ToolBar*. Niespełnienie tego wymagania spowoduje zgłoszenie wyjątku (*InvalidOperationException*) przy próbie dodania widoku do regionu. Znaczenie aktywacji/deaktywacji widoku nie zostało zdefiniowane.

5.6 Punkty rozszerzenia mechanizmu regionów

Zaimplementowane rozwiązanie pozwala na użycie jedynie wybranych elementów mechanizmu, z pominięciem pozostałych. Ponadto, możliwe jest zastąpienie wybranych elementów własnymi implementacjami. Autorzy platformy pozostawili użytkownikom:

1. możliwość stworzenia *RegionAdapterów* oraz *RegionBehaviorów* dla dowolnego typu kontrolki,
2. możliwość stworzenia własnego typu *Regionu*,
3. możliwość zastąpienia algorytmu wyboru *RegionAdaptera*,
4. możliwość zastąpienia całości procesu tworzenia *Regionu*,
5. możliwość wykorzystania tylko mechanizmu tworzenia regionów, z pominięciem *RegionManagera*.

Tworzenie *RegionAdapterów* oraz *RegionBehaviors*

Aby w pełni wykorzystać mechanizm regionów, programista powinien przygotować dopasowane do potrzeb aplikacji *IRegionAdaptery*. Zadaniem *IRegionAdaptera* jest utworzenie odpowiedniego regionu oraz powiązanie widoku z regionem. Powiązanie to może przybrać dowolną postać, np:

- utworzenie powiązania (*Binding*) między kolekcją widoków regionu i kolekcją elementów wyświetalnych w kontrolce (*ItemSource*),
- synchronizacja widoków aktywnych regionu i wybranych zakładek w kontrolce typu *TabControl*,
- przekazanie informacji zwrotnej do widoku, kiedy widok zostanie aktywany/deaktywowany.

Z racji tego, że część takich powiązań jest współdzielona przez *RegionAdaptery*, każdy spójny fragment powiązania stworzono jako osobną klasę – *IRegionBehavior*. Platforma *Nomad* nie wymusza korzystania z tych klas przy tworzeniu własnych *RegionAdapterów*. Mogą one jednak ograniczyć duplikację w kodzie i pomóc w zapewnieniu spójności funkcjonalnej.

Opis IRegionBehavior

Obiekt *IRegionBehavior* odpowiedzialny jest za mały, ale spójny wycinek powiązania między regionem a hostującą go kontrolką. Wraz z platformą *Nomad* dostarczono zbiór zaimplementowanych zachowań:

ActiveAwareBehavior – jeśli widok implementuje interfejs *IActiveAware*, to przekazywana jest mu informacja o tym, że został aktywowany/deakywowany (np. wybrano zakładkę lub ją opuszczono). Zachowanie to musi działać w powiązaniu z zachowaniem synchronizującym elementy aktywne (np. *SynchronizeActiveItemsBehavior*)

SynchronizeActiveItemsBehavior – synchronizuje listę aktywnych widoków między regionem a kontrolką. Kontrolka musi być typu *Selector* lub *MultiSelector*. Zachowanie to zapewnia, że kolekcja *IRegion.ActiveViews* zawierać będzie te i tylko te elementy, które są wybrane w kontrolce. Synchronizacja następuje w dwie strony. Wykorzystując to zachowanie można np. aktywować z poziomu kodu pewną zakładkę w kontrolce *TabControl*

SynchronizeItemsBehavior – łączy kolekcję *IRegion.Views* z właściwością *ItemsSource* kontrolki typu *ItemsControl*.

SynchronizeToolbarsBehavior – synchronizuje kolekcję *IRegion.Views* z listą pasków narzędziowych kontrolki typu *ToolBarTray*

TabControlHeaderTemplateSelectorBehavior – ustawia kontrolkom typu *TabControl* dostarczony przez platformę *ItemTemplateSelector*. Obiekt ten pozwala na wybranie szablonu zakładek. Jeśli widok implementuje interfejs *IHaveTitle* to wybrany zostanie szablon o nazwie „Nomad_ItemWithTitle". Dostarczenie tego szablonu jest odpowiedzialnością programisty aplikacji. Musi się on znaleźć w zasobach kontrolki powiązanej z regionem lub elementu nadrzędnego. W ramach szablonu można odwoływać się (poprzez *Binding*) do właściwości widoku - w tym przypadku do właściwości *Title*. Jeśli tytuł podawany przez widok może się dynamicznie zmieniać, widok powinien implementować interfejs *INotifyPropertyChanged* i zgłaszać odpowiednie zdarzenia.

Te zachowania mogą zostać uzupełnione przez programistę aplikacji, przy tworzeniu własnych *RegionAdapterów*.

Typy regionów

Wraz z platformą *Nomad* dostarczono 2 implementacje regionów – *SingleActiveViewRegion* oraz *MultipleActiveViewsRegion*. *SingleActiveView* wymusza, aby w danym momencie aktywny był tylko jeden widok. Jest to przydatne np. przy kontrolkach typu *TabControl*. Drugi typ umożliwia, aby wiele widoków było aktywnych. Programista aplikacji może stworzyć własne, bardziej rozbudowane implementacje. Mogą one dodatkowo np. sprawdzać typy dodawanych widoków czy też zgłaszać zdarzenia. Aby móc skorzystać z własnego typu regionu konieczne będzie również zaimplementowanie własnego *IRegionAdaptera*.

Algorytm wyboru *RegionAdaptera*

Dostarczona w platformie *Nomad* domyślna fabryka regionów (*RegionFactory*) tworzy regiony używając do tego celu obiektów *IRegionAdapter*. Dla każdego widoku wybierany jest dokładnie jeden adapter, który odpowiedzialny jest za utworzenia regionu i powiązanie regionu oraz widoku. Domyślny algorytm wyszukuje adaptery na podstawie typu kontrolki – wyszukuje najbardziej specyficzny adapter. Programista aplikacji może jednak zastąpić ten algorytm, tworząc podklasę klasy *RegionFactory* i nadpisując metodę *FindAdapterFor*. Przykładowym rozszerzeniem może być wybieranie adaptera na podstawie implementowanych przez widok interfejsów.

Zastąpienie procesu tworzenia *Regionu*

Programista aplikacji nie jest ograniczony do tworzenia regionów przy pomocy *RegionAdapterów*. Implementując interfejs *IRegionFactory* można w dowolny sposób zdefiniować proces tworzenia *Regionów*. Obiekt implementujący ten interfejs należy przekazać do konstruktora *RegionManagera*.

Pominięcie RegionManagera

Jak opisano powyżej, *RegionManager* implementuje mechanizm *View injection*. Możliwe jest jednak wykorzystanie jedynie dostarczonego mechanizmu tworzenia *Regionów* (tj. fabryki regio-

nów, adapterów oraz typów regionów) i zaimplementowanie innych mechanizmów – np. *View discovery*.

Rozdział 6

Komunikacja międzymodułowa

Jeden z elementów funkcjonalnych platformy stanowi komunikacja między modułami (z której programista może korzystać w możliwie prosty sposób). Rozważono dwa przypadki: komunikację synchroniczną oraz asynchroniczną. Aby mówić o komunikacji należy najpierw wyróżnić jej dwie strony: nadawcę komunikatów oraz ich odbiorcę. Zwykle są to dwa różne moduły, może jednak zajść konieczność komunikacji między elementem platformy a modułem i taki jej rodzaj również jest możliwy. Dlatego też wyróżniono dwa rozwiązania, które choć podobne, znacznie różnią się od siebie.

Komunikacja międzymodułowa została oparta w dużej mierze na wiedzy pozyskanej z publikacji Martina Fowlera [Fow05] [Fow04].

6.1 Komunikacja asynchroniczna

Komunikację asynchroniczną dobrze prezentują wzorce programistyczne typu *Publish-Subscribe*. Jedną z jego specjalizacji jest wzorzec *Observer-Listener* ([RCM08]). Nie zapewnia on jednak elastyczności, którą platforma powinna oferować - gdyż zgodnie z jego założeniami, ktoś musiałby zgłosić obiekt słuchający do obiektu publikującego, ponadto obiekt publikujacy musiałby być definiowany przez moduł.

Dlatego też, względem komunikacji asynchronicznej, sprecyzowaliśmy następujące wymagania:

- Moduł może w dowolnym momencie publikować dowolny *typ* komunikatu. Moduły mogą definiować nowe *typy* komunikatów, mogą również publikować już zdefiniowane *typy*. Wszystkie moduły, które na wskazany *typ* się zapisały, otrzymają go. Jeżeli nikt nie oczekuje danego typu, komunikat nie zostanie dostarczony do nikogo, a przetwarzanie nie zostanie zatrzymane.
- Dowolny moduł może zapisać się na otrzymywanie określonego *typu* komunikatów – takie podejście pozwala na jednoznaczne zdefiniowanie odbiorców, pozwala także na odseparowanie logiki modułu od klas dostarczających zdarzenia. Co więcej, pozwala to modułowi zapisać się na wiele różnych zdarzeń (wielu różnych typów). W przypadku, gdy nikt danego typu nie publikuje, nigdy nie nastąpi zdarzenie związane z odbiorem komunikatu.
- W momencie zapisywania się na *typ* komunikatu, subskrybent *może* wyspecyfikować w jakim wątku chce otrzymać zdarzenie (czy w wątku *GUI*, czy w wątku domyślnym, czy też w osobnym wątku zapewniającym separację długotrwałych operacji). Jest to w szczególności ważne dla zdarzeń, których obsługa wykonuje operacje w interfejsie graficznym (takie operacje można wykonać wyłącznie z wątku *GUI*).
- Moduł może zrezygnować z subskrypcji komunikatów danego *typu*.

Jednocześnie, korzystający z tej formy komunikacji muszą zapewnić, iż:

- Obsługa zdarzenia odbędzie się w skończonym (możliwie krótkim) czasie. Wynika to z faktu, iż doręczenie wielu komunikatów w ramach jednej publikacji może odbywać się sekwencyjnie.
- Obsługa zdarzenia nie zakończy się błędem (to jest: nie zostanie rzucony wyjątek poza obręb metody). Mechanizm dostarczania komunikatów nie będzie przechwytywał takich wyjątków (ze względu na separację odpowiedzialności), gdyż mogłoby to zagrozić poprawności działania aplikacji.

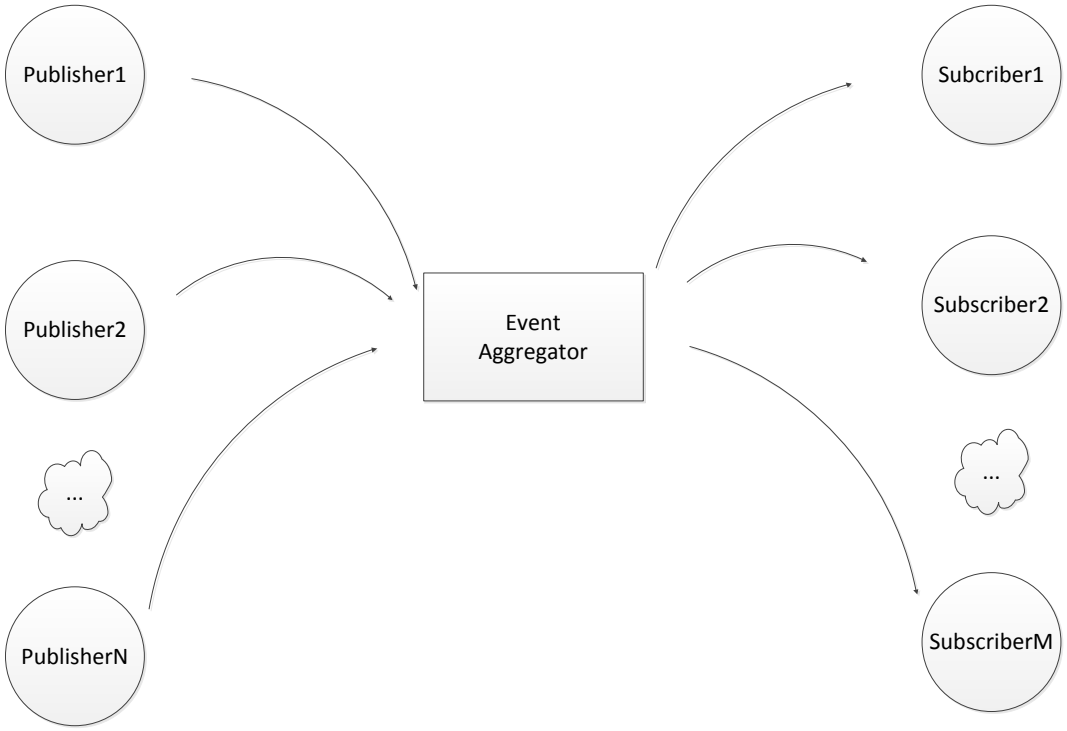
- Obsługa zdarzenia będzie bezpieczna z punktu widzenia wielowątkowości. Subskrybent modułu nie może zakładać, że komunikaty będą docierały sekwencyjnie (z wyjątkiem sytuacji, gdy subskrybent zdecydował się otrzymywać komunikaty w wątku *GUI*).

- Publikacja zdarzenia może zablokować wątek publikujący na chwilę (czas zależny od wykonania metod subskrybentów). Jeżeli publikacja ma nastąpić w sposób nieblokujący, należy wykorzystać wątek z puli wątków (patrz dodatek B).

6.1.1 Koncepcja Event Aggregatora

Rozwiązanie spełniające powyższe wymagania zostało przedstawione na poglądowym rysunku 6.1. Jak za chwilę zostanie wykazane, w przypadku bieżącej architektury z dwiema domenami aplikacji implementacja zgodna z wskazanym rysunkiem nie będzie wystarczająca.

Istnieje jeden centralny obiekt do którego obserwatorzy są rejestrowani. Aby rozróżnić klientów, muszą oni jednak podać typ komunikatów, na jaki się rejestrują.



Rysunek 6.1: Poglądowy rysunek publikacji i subskrypcji komunikatów

EventAggregator to centralny punkt wymiany komunikatów. Wszystkie moduły korzystają z tej samej instancji aggregatora, który musi zapewniać bezpieczeństwo wątków (to znaczy, że wiele wątków może wykonywać na nim operacje równolegle). By zapewnić, że wszystkie moduły korzystają z tej samej instancji *EventAggregatora* postanowiono, że w ramach jednej domeny aplikacji istnieje wyłącznie jeden *EventAggregator*. W świetle wiedzy, iż wszystkie moduły działają w jednej domenie aplikacji oznacza to, że korzystają z tego samego *EventAggregatora*.

Zgodnie z rysunkiem 6.1, aggregator zdarzeń jest bytem samodzielnym – nie posiada zależności od żadnego modułu. Nie musi nawet wiedzieć jakiego typu komunikaty będzie publikował. Aby móc skorzystać z aggregatora zdarzeń, moduł musi znać jedynie *EventAggregatora* (który zawarty jest w platformie) oraz *typ* komunikatu, który chce przesłać lub otrzymać.

Przykład realizacji wymiany komunikatów (subskrypcja w listingu 6.1, publikacja w listingu 6.2).

Listing 6.1: Zapisanie się na typ komunikatu

```
eventAggregator.Subscribe<MessageType>(payload => HandlePayloadAction);
```

Listing 6.2: Opublikowanie komunikatu komunikatu

```
eventAggregator.Publish(sentPayload);
```

Pojęcia

Wprowadzono następujące pojęcia w celu ujednolicenia opisu problemów:

- *Komunikat* – definiuje zestaw informacji przekazywanych w ramach zdarzenia. Definiowany jest przez dowolny moduł. Jego rodzaj określany jest za pomocą *typu*. Komunikat może być zarówno typem wartościowym jak i referencyjnym. Komunikaty pochodzące z jądra platformy dziedziczą z *NomadMessage*. Są ponadto serializowalne, gdyż mogą być przesyłane do domeny modułów. Żaden komunikat domeny modułów nie musi być serializowalny.
- *Subskrypcja* – w kodzie reprezentowana przez *Ticket*. Jest to zobowiązanie *IEventAggregatora* do dostarczania komunikatów danego typu do danego odbiorcy (subskrybenta). Z subskrypcji można zrezygnować wywołując na niej metodę *Dispose*.
- *Subskrybent* – w zależności od kontekstu – klasa dokonująca subskrypcji (i zarządzająca subskrypcją), lub metoda bądź delegat który będzie otrzymywał komunikaty wskazanego *typu*.
- *Publikujący* – dostawca komunikatów wskazanego *typu*.

6.1.2 Komunikacja asynchroniczna w środowisku wielodomenowym

W środowisku wielodomenowym, a takie jest konstruowane w Nomadzie (rozdział 4.1.2), sytuacja jednak nie wygląda w taki prosty sposób. Wynika to z faktu, iż w platformie *.NET* nie każdy komunikat może opuścić ramy domeny aplikacji. Dokładnie opisano to w rozdziale 2.4.2. Jedynie komunikaty które są serializowalne, bądź są specjalizacją *MarshalByRefObject* mogą być przesłane pomiędzy domenami.

Wprowadzono więc następujące wymagania i ograniczenia do platformy:

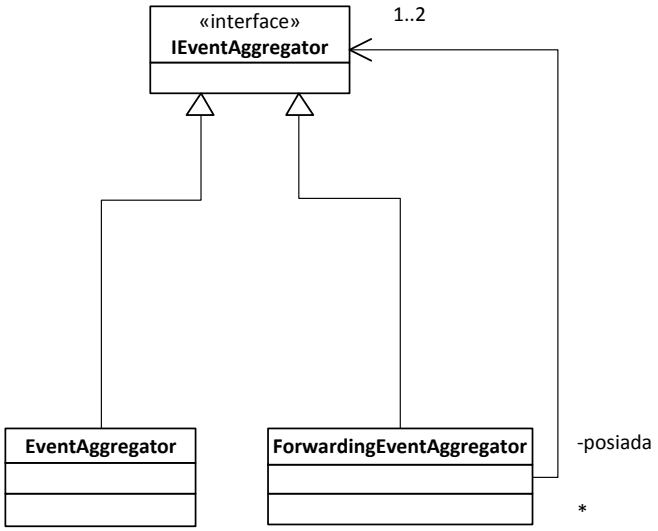
- Każda domena posiada własnego *EventAggregatora*.
- Wszystkie komunikaty przesyłane pomiędzy domenami muszą być serializowalne.
- Dopuszczamy jedynie możliwość przesłania komunikatów z domeny jądra do domeny modułów. Nie można przesłać żadnego komunikatu z domeny modułów do domeny jądra.
- Wszystkie komunikaty wysyłane w domenie jądra (przez składowe jądra) muszą być serializowalne, tak, aby mogły przekroczyć granice domeny aplikacji.

W domenie jądra utworzony został *ForwardingEventAggregator* oraz *EventAggregator*. Rolą *ForwardingEventAggregatora* jest przekazanie komunikatów z domeny jądra do dwóch właściwych *EventAggregatorów*: z domeny modułów i z domeny jądra. Diagram obrazujący relacje pomiędzy *IEventAggregatorami* przedstawiono na rysunku 6.2. Sam *EventAggregator* odpowiada jedynie za publikację i obsługę subskrypcji komunikatów z własnej domeny (tej, do której przynależy).

W domenie modułów utworzono wyłącznie *EventAggregatora*, gdyż zachowaniem niepożądanym byłaby publikacja komunikatów z domeny modułów do domeny jądra.

Takie rozwiązanie zapewnia, że żadne subskrypcje nie muszą być przesyłane pomiędzy domenami (gdyż każda domena posiada autonomicznego *EventAggregatora* utrzymującego subskrypcje). Natomiast komunikat, który musi być przekazany pomiędzy domenami, przesyłany jest wyłącznie raz (od *ForwardingEventAggregatora* jednej domeny do *EventAggregatora* drugiej). Każdy z *EventAggregatorów* obsługuje już autonomicznie własną domenę.

Jak wspomniano, w domenie modułów rzeczywiste funkcje spełnia już tylko standardowy *EventAggregator*. Dzieje się tak, gdyż po pierwsze nie mamy gwarancji, że przesyłane komunikaty z domeny modułów będą serializowalne (a taki jest jeden z warunków przesłania ich pomiędzy



Rysunek 6.2: Diagram klas prezentujących standardowy EventAggregator oraz ForwardingEventAggregator

domenami). Po drugie, zgodnie z wymaganiami architektury – niepożądane byłoby dostarczanie komunikatów z domeny modułów do domeny jądra.

Reasumując, w momencie publikowania komunikatu w domenie jądra, za pomocą *IEventAggregatora* jądra, podejmowane są następujące kroki (przez *ForwardingEventAggregator*):

1. Przekaż komunikat do *EventAggregatora*, który opublikuje go (komunikat) w domenie jądra.
2. Przekaż komunikat do *EventAggregatora* domeny modułów, przekraczając granicę domeny aplikacji (to jest: serializując komunikat). Publikacją komunikatu w domenie modułów zajmie się *EventAggregator* domeny modułów.

W momencie publikacji komunikatu w domenie modułów, obsługą zajmuje się już wyłącznie *EventAggregator* tej domeny. Wygląda to w następujący sposób:

1. Opublikuj komunikat do wszystkich subskrybentów danego typu w domenie modułów.
2. Zakończ działanie.

Na podstawie powyższej specyfikacji działania należy odpowiedzieć na pytanie, w jaki sposób *ForwardingEventAggregator* uzyska dostęp do *IEventAggregatora* domeny modułów. Otóż, aby przekroczyć granicę domeny modułów do domeny jądra *EventAggregator* musiałby zostać zserializowany, bądź musiałby dziedziczyć z *MarshalByRefObject*. Przyjmijmy, że *EventAggregator* jest serializowalny. Oznacza to, że jego wszystkie subskrypcje również musiałyby być serializowalne. Gdyby było to możliwe, należało by rozpatrzeć dwa przypadki. Pierwszy, gdy *EventAggregator* domeny modułów jest przesyłany do domeny *jądra* w momencie tworzenia *ForwardingEventAggregatora*. Okazuje się, że zachowanie spójności pomiędzy przesłanym *EventAggregatorem* a tym, który pozostał w domenie modułów nie jest trywialne. Drugi przypadek, rozwiązujący problem zachowania spójności, to przesyłanie w momencie każdej publikacji *EventAggregatora* z domeny modułów do domeny rdzenia. Wiązałoby się to jednak z przesyłaniem pomiędzy domenami dużej ilości informacji (i serializacją dużej liczby elementów, co niekorzystnie wpłynęłoby na zużycie zasobów pamięci i procesora). Rozważania te są jednak wyłącznie hipotetyczne ponieważ ani typ *Action*, ani *Delegate* nie są serializowalne. Wynika z tego jednoznacznie, że niemożliwym byłoby dokonania serializacji subskrypcji. Co więcej, zdarzenia publikowane w przesłanym za pomocą

serializacji *EventAggregatorze* byłyby publikowane w domenie rdzenia a nie w domenie modułów, co całkowicie wyklucza taką realizację.

Z powyższego analizy wynika jednoznaczny wniosek: *EventAggregator* musi dziedziczyć z *MarshalByRefObject* aby mógł pokonać granice domeny aplikacji. Co więcej, dzięki temu wszelkie akcje wykonane na tak przesłanym *EventAggregatorze* zostaną wykonane w macierzystej (dla niego) domenie aplikacji. Same subskrypcje (ani typ *Action* ani *Delegate*) nie będą musiały być przesłane, więc nie będą musiały być serializowane.

6.1.3 Komunikacja asynchroniczna w środowisku wielowątkowym

Należy pamiętać, że *Nomad* może być wykorzystywany w aplikacjach wielowątkowych. Dlatego też obsługa komunikatów musi swoją architekturą wspierać wielowątkowość, to jest pozwalać wielu wątkom wykonywać na niej operacje w tym samym czasie. Co więcej, wymiana komunikatów jest funkcją krytyczną z punktu widzenia systemu, toteż powinna zajmować możliwie mało czasu. Oznacza to, że struktury wykorzystane do implementacji *EventAggregatora* powinny być możliwie szybkie, a sekcje krytyczne (ziarna blokad) możliwie małe.

Aby spełnić te wymagania, *EventAggregator* wykorzystuje w swojej implementacji słownik (*Dictionary*), w którym przechowywane są kolekcje subskrypcji (*ticketów*) do wykonania dla określonego typu. Uzyskanie dostępu do elementu słownika dzieje się w czasie „bliskim O(1)" [Mic03b]. Utworzenie nowej kolekcji w słowniku następuje dość rzadko (tylko w przypadku subskrypcji na typ, na który nikt jeszcze się wcześniej nie subskrybował) i ma złożoność $O(n)$ (jeżeli trzeba zwiększyć rozmiar słownika, gdzie n to pojemność słownika), w średnim przypadku jednak $O(1)$.

Zasubskrybowanie się na komunikat określonego typu ma więc zwykle złożoność $O(1)$.

Publikacja komunikatu określonego typu ma złożoność $O(1) * O(n)$, gdzie n to ilość subskrybentów komunikatu danego typu. Złożoność ta wynika z faktu, iż metoda każdego subskrybenta musi zostać wykonana. Oczywiście, złożoności metod wykonywanych przez subskrybentów są różne i nie dotyczą one bezpośrednio złożoności operacji *EventAggregatora*.

6.1.4 Komunikacja asynchroniczna w środowisku z *GUI*

Jedną z funkcji oferowanych przez *EventAggregator* jest dostarczanie komunikatów w wybranym wątku. W szczególności może to być wątek graficznego interfejsu użytkownika. Biorąc pod uwagę przytoczone wcześniej założenia – platforma nie wymaga do działania żadnej aplikacji graficznej. Doskonale sprawdzać się może w środowisku konsolowym, czy działając nawet jako usługa systemowa (patrz dodatek B). W takich okolicznościach nie istnieje tak zwany wątek *GUI* wymagania więc wobec komunikatów w nim dostarczane zostają osłabione – należy zapewnić jedynie sekwencyjność ich wykonania.

Aby jednak aggregator zdarzeń mógł uzyskać dostęp do wątku *GUI*, musi on zostać zdefiniowany. W celu pozostawienia elastyczności w zakresie warstwy prezentacji, zdefiniowany został interfejs *IGuiThreadProvider* któremu przekazywane są akcje do wykonania w wątku *GUI*. W zależności od wykorzystania konkretnej warstwy prezentacji (czy będzie to np. *Windows Forms* czy *WPF*) inne implementacje interfejsu będą wykorzystane przez agregator zdarzeń.

Szczegółowa implementacja tego rozwiązania zakłada wykorzystanie klasy [EFB04] implementującej interfejs *IGuiThreadProvider*, delegującej wszystkie akcje do jednego ze zdefiniowanych w platformie dostawców wątku *GUI*:

- NullGuiThreadProvider – wykorzystywany w aplikacjach konsolowych i innych, przed zainicjalizowaniem właściwego wątku *GUI*. Zapewnia sekwencyjne wykonanie wszystkich akcji (taka bowiem jest gwarancja dawana przez agregator zdarzeń dla modułów, które korzystają z wątku *GUI* – że wykonanie akcji będzie sekwencyjne).

- WpfGuiThreadProvider – wykorzystywany w aplikacjach opartych o technologię WPF. Deleguje odpowiednie akcje do właściwego wątku Dispatchera (patrz słowniczek B).

- LazyWpfThreadProvider – sprawdza czy jest aktualnie uruchomiona aplikacja WPF, jeżeli nie to deleguje zadania do *NullGuiThreadProvidera*, jeżeli tak to zadania zostaną wykonane we właściwym dispatcherze.

Należy podkreślić, iż z punktu widzenia architektury aplikacji tworzonej przy wykorzystaniu *Nomada*, w pierwszej fazie uruchomienia nie ma w ogóle wątku *GUI*. Dopiero załadowanie modułu

aplikacji spowoduje utworzenie wątku interfejsu użytkownika i od tego momentu moduły będą mogły wykonywać zdarzenia w oknach. Takie podejście jest zgodne z wymaganiami stawianymi przez framework – aby aplikacje mogły korzystać między innymi z regionów, muszą być zależne od modułu aplikacji – co oznacza, że w praktyce będą załadowane dopiero *po* module aplikacji – czyli klasyczny wątek *GUI* będzie już dla nich dostępny.

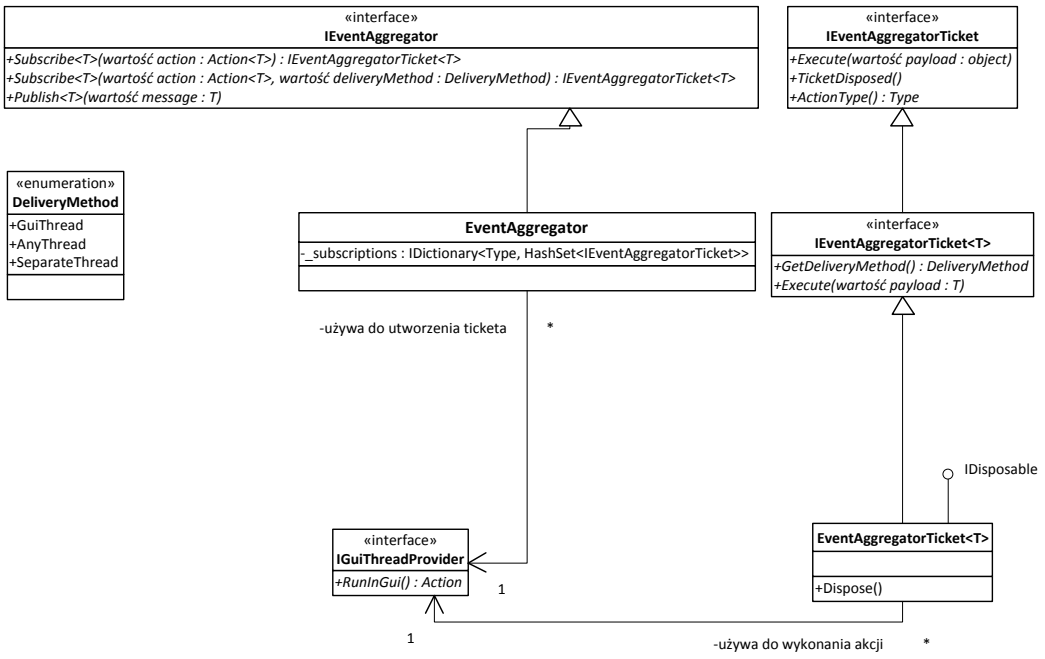
6.1.5 Rzeczywista implementacja

Implementacja *EventAggregatora* przedstawiona jest na rysunku 6.3. Pominięto szczegóły związane z rozwiązaniem problemu wielu domen, gdyż niepotrzebnie komplikowałoby to diagram.

Procedura dodania nowej subskrypcji przebiega następująco:

1. Utwórz *IEventAggregationTicket* dla danej akcji oraz wybranego sposobu jej dostarczenia.
2. Dodaj utworzoną subskrypcję (*ticket*) do kolekcji odpowiadającej typowi komunikatu (*_subscriptions*).
3. Zwróć ticket subskrypcji.

Gdy wskazany moduł będzie chciał zrezygnować z subskrypcji, musi po prostu wykonać metodę *Dispose* na otrzymanym tickecie.



Rysunek 6.3: Diagram klas przedstawiający poglądowo *EventAggregator*

Jak wynika z diagramu 6.3, użytkownik może wybrać sposób dostarczenia komunikatu. Domyślnie odbywa się to w tym samym wątku, w którym następuje publikacja. W takim przypadku obsługa komunikatu powinna być możliwie krótka. Komunikat może być także dostarczony w wątku *GUI*, lub w nowym wątku (tworzonym wyłącznie na potrzebę obsługi komunikatu).

W architekturze *EventAggregatora* to *ticket* odpowiada za dostarczenie komunikatu. Wykonanie metody *Publish* przebiega w uproszczeniu w następujący sposób:

1. Pobierz kolejkę *subskrypcji* (*ticketów*) dla wskazanego typu komunikatu.
2. Wykonaj metodę *execute* na każdym tickecie subskrypcji.

Wykonując metodę na tickecie, sprawdzane jest w jakim wątku ma zostać dostarczony komunikat. Jeżeli ma być to wątek interfejsu użytkownika, akcja wykonywana jest przez *IGuiThreadProvider*. Jeżeli ma być to osobny wątek, to zadanie umieszczane jest w puli wątków środowiska *.NET*. Jeżeli ma być to dowolny wątek, to akcja jest wykonywana w tym samym wątku, w którym zostało wykonane wywołanie akcji subksrypcji (*ticketa*).

6.2 Komunikacja synchroniczna

Poprzednie sekcja obszernie wyczerpuje temat komunikacji asynchronicznej. Drugi jej rodzaj, to komunikacja oparta na synchronicznych wywołaniach – wykonanie metody na odbiorcy blokuje nadawcę do czasu zakończenie przetwarzania u odbiorcy.

Projektując *Nomada* zdecydowano się na model komunikacji synchronicznej w postaci serwisów o znanym interfejsie. Uzasadnieniem tej decyzji jest długa lista bibliotek i gotowych rozwiązań, które z z sukcesem implementują ten model [Mic09].

Znając interfejs – czyli abstrakcyjny opis API – danego serwisu możemy próbować wykonać metody tego interfejsu nie wiedząc nic o implementacji stojącej za tym interfejsem. Podobny model funkcjonuje np. w technologii *COM* [MER09].

W przytoczonym modelu powstaje problem, odnajdywania istniejącej, aczkolwiek nieznanej implementacji serwisu, którego interfejs jest jedynie znany klientowi.

6.2.1 Wzorzec *ServiceLocator*

Najbardziej znaną koncepcją rozwiązującą problem odnajdywania nieznanej implementacji znanego interfejsu jest wzorzec *ServiceLocator*, pełen opis znajduje się w bibliotece firmy Sun [Net02].

Wzorzec ten definiuje klasę „lokatora serwisu" (z ang. Service Locator), której jedynym celem jest połączenie interfejsu serwisu i jego konkretnej implementacji. Mechanika tego połączenia jest ukrywana przed użytkownikiem *ServiceLocatora*. Klient lokatora jest jedynie świadom dwóch metod:

- Register – zarejestruj dostawcę serwisu (nie mówiąc nic o sobie)
- Resolve – pobierz serwis (nie precyzując dostawcy)

Bardziej obrazowe wyjaśnienie znajduje się u [Fow02].

6.2.2 Implementacja *ServiceLocator*

Implementacja wzorca *ServiceLocator* jest w platformie *Nomad* niemal książkowa. Domyślnie utworzona klasa *ServiceLocator* jest implementacją interfejsu *IServiceLocator*, który w stu procentach odpowiada przytoczonemu wzorcowi:

Listing 6.3: Rejestracja serviceProvider jako dostarczyciela usługi InterfaceType

```
serviceLocator.Register<InterfaceType>(serviceProvider);
```

Listing 6.4: Pobranie obiektu implementującego żądany serwis InterfaceType

```
service = serviceLocator.Resolve<InterfaceType>();
```

Powyższe listingi tego nie pokazują, ale serwis – obiekt *serviceProvider* jest w rzeczywistości obiektem klasy dziedziczącej po interfejsie *InterfaceType* (patrz listing 6.3).

Domyślnie zaimplementowany w platformie *Nomad ServiceLocator* charakteryzuje parę cech nie występujących w oryginalnym wzorcu:

- Wykorzystano kontener *IoC* Castle Windsor jako bazę działania lokatora.
- Ograniczono możliwość wyrejestrowania serwisu. Usunięcie serwisu z bazy dostępnych jest możliwe jedynie w przypadku przeładowania domeny aplikacji.
- Jeden serwis może być zarejestrowany tylko raz.
- Domyślnie dostarczany *ServiceLocator* nie jest bezpieczny w kwestii wielowątkowości.

Każda z powyższych cech ma swoje uzasadnienie.

Kontener *IoC* jest używany także w innych częściach platformy *Nomad*. Umożliwia to projektantom przygotowanie wygodniejszej obsługi np. kwestii ładowania modułów. Zostało to dokładniej opisane w 4.2.

Implementacja domyślna nie pozwala na usunięcie raz już zarejestrowanego serwisu, ponieważ w ten sposób klient frameworku, a w szczególności autor modułów, jest chroniony przed błędami wynikającymi z przedwczesnego wyrejestrowania serwisu. W obecnej sytuacji, autor modułu może

założyć, że skoro uzyskał serwis nie musi już martwić się czy przypadkiem go nie utraci. Takie rozwiązanie poza poczuciem bezpieczeństwa, jest wydajniejsze – nie ma potrzeby sprawdzania dostępności serwisu przed każdym wywołaniem metody na nim.

Podobną argumentację ma także kolejna z cech zaimplementowanego *ServiceLocatora*. W celu uniknięcia niepotrzebnych komplikacji z wyborem dostarczyciela usługi, przewidziano jedynie możliwość jednokrotnego zarejestrowania serwisu.

Domyślna implementacja nie zapewnia bezpieczeństwa w kwestii wielowątkowości (z ang. thread safety), gdyż zapewnienie niejawnej obsługi wielu wątków antagonizowało by z modelem komunikacji synchronicznej. Użycie *ServiceLocatora* w sytuacji wielowątkowej jest możliwe przez autora modułu, musi on jednak zapewnić synchronizację ręcznie.

Rozdział 7

Mechanizm uaktualnień modułów

Platforma wspierająca modułowe budowanie aplikacji powinna oferować gotowe mechanizmy weryfikacji aktualności wersji załadowanych modułów. W przypadku wykrycia dostępności aktualizacji możliwym scenariuszem może być przeprowadzenie automatycznej, przymusowej aktualizacji lub udostępnienie mechanizmów pozwalających programiście aplikacji na kontrolowane przeprowadzenie uaktualniania.

Taka kontrolowana aktualizacja dość często przybiera postać scenariusza, w którym to użytkownik otrzymuje możliwość wyboru modułów, które chciałby posiadać. Zazwyczaj oczekuje się wyświetlenia przez aplikację listy możliwych aktualizacji oraz możliwości wyboru poszczególnych elementów w sposób adekwatny do stylu i poziomu odbiorców aplikacji.

Nomad wspiera powyższe przypadki, wykorzystuje do tego wewnętrzny element rdzenia platformy nazwany *Nomad.Updater*. Pod pojęciem aktualizacji rozumiane są jakiekolwiek zmiany w strukturze modułów – czy to instalacja nowego dodatku, czy też zmiana jego wersji.

Problemy związane z aktualizacją

Proces aktualizacji, choć wydaje się łatwy, niesie ze sobą kilka problemowych zagadnień. Przede wszystkim, aktualizacja rozpoczyna się od stanu spójnego aplikacji (wszystkie zależności są spełnione) i powinna w takim stanie aplikację pozostawić. Wbudowany w platformę *Nomad* mechanizm zapewniania spójności odbiega od mechanizmów występujących w systemach zarządzania bazami danych.

Przede wszystkim, nie zapewnia odporności na zdarzenia zewnętrzne – podczas awarii zasilania stan spójny nie zostanie zachowany. Uzyskamy jedynie informacje, że aktualizacja nie powiodła się.

W momencie uruchamiania aktualizacji platforma sprawdza, czy zestaw modułów osiągnięty po zakończeniu aktualizacji będzie spójny – to jest, czy wszystkie zależności modułów będą spełnione. Co więcej, aktualizację można przeprowadzić dopiero, gdy wszystkie moduły zostaną pobrane.

Nomad definiuje pojęcie źródła (repozytorium) modułów w postaci interfejsu *IModulesRepository*, który odpowiada za komunikację z serwerem repozytorium. Wraz z platformą dostarczany jest przykładowy serwer wykorzystujący ASP MVC (patrz 10.4.1).

Można jednak wykorzystać dowolny inny protokół i formę dostarczania pakietów. Serwer może być wykonany w oparciu o strukturę katalogów, z wykorzystaniem *PHP* (patrz dodatek B) lub dowolnego innego mechanizmu. Tworząc alternatywny serwer, należy dostarczyć implementację interfejsu *IModulesRepository*.

Jak już zostało wspomniane wcześniej, użytkownicy końcowi chcą oglądać moduł aktualizacji przystosowany do użytkowanej aplikacji. Dlatego właśnie postanowiono, że fragment platformy odpowiadający za mechanizm uaktualniania będzie pozbawiony interfejsu graficznego. Powoduje to zwiększenie elastyczności rozwiązania wbudowanego w platformę, gdyż każdy programista może w dowolny sposób przygotować elementy wizualne odpowiadające użytkownikowi.

W ramach narzędzi i przykładów wykorzystania platformy *Nomad* zaimplementowano przykładowy interfejs graficzny dla aktualizatora. Więcej informacji na temat przygotowanego modułu znajduje się w 10.4.3.

7.1 Koncepcja realizacji

Proces aktualizacji został podzielony na trzy etapy:

1. Sprawdzenie aktualizacji.
2. Przygotowanie aktualizacji.
3. Przeprowadzenie aktualizacji.

Dodatkowo abstrakcyjny aktualizator powinien opisywać swój stan oraz sygnalizować koniec procesu aktualizacji wykorzystując mechanizmy *AutoResetEvent* (więcej informacji w dodatku B).

Przedstawione powyżej cechy elementu aktualizującego stanowią interfejs *IUpdater*, definiujący zachowanie się każdego mechanizmu aktualizacji. Wprowadzony on został w celu zwiększenia elastyczności oraz potencjalnych zmian jakie mogą się pojawiać już po opublikowaniu platformy.

7.2 Implementacja

Jednym z wymagań określonych wobec mechanizmu uaktualnień jest automatyczne przeładowanie aplikacji po zakończeniu aktualizacji. Można ten cel osiągnąć na wiele sposobów:

- Uruchomić osobny proces, który odpowie za przeprowadzenie aktualizacji i uruchomienie aplikacji po zakończeniu tego procesu – rozwiązanie dosyć proste w realizacji, ale wykorzystując mechanizmy oferowane przez *.NET Framework* można je zrealizować inaczej.
- Wyładować domenę modułów i w domenie kernela przeprowadzić podmianę plików modułów – takie rozwiązanie nie restartuje całej aplikacji, a przeładowuje jedynie wszystkie moduły. Niesie ze sobą jedno ograniczenie – nie można zaktualizować w ten sposób samej aplikacji kernela. W domenie rdzenia platformy nie powinny jednak rezydować żadne biblioteki związane z aplikacją.

W platformie *Nomad* zaimplementowano koncepcję drugą, wykorzystującą domeny aplikacji. Przygotowany element nosi nazwę *NomadUpdater* – następne podrozdziały odnoszą się już tylko do tej konkretnej implementacji. Uzyskanie dostępu do elementu aktualizującego odbywa się z wykorzystaniem *Service Locatora* (patrz 6.2). Taki sposób dostępu powoduje, że programista aplikacji może świadomie wykonać proces aktualizacji w osobnym wątku. Dzięki temu interfejs graficzny pozostanie responsywny w trakcie aktualizacji, a aplikacja nie będzie wyglądała na „zawieszoną".

Zaimplementowany aktualizator uzupełnia zdefiniowane w interfejsie *IUpdater* metody w następujący sposób:

1. Sprawdzenie aktualizacji – wykorzystanie *IModulesRepository* do otrzymania możliwych uaktualnień.
2. Przygotowanie aktualizacji – wykorzystanie *IModulesRepository* do ściągnięcia poszczególnych aktualizacji oraz *IDependencyChecker* do sprawdzenia czy po aktualizacji stan aplikacji nadal będzie poprawny.
3. Przeprowadzenie aktualizacji. – wykorzystanie *IModulePackagera* do przeprowadzenia aktualizacji.

Wszystkie wymienione powyżej dodatkowe składniki są elementami konfiguracji platformy *Nomad* i mogą być w dowolny sposób zmienione lub rozszerzone. Zapewnia to domyślnej implementacji elementu aktualizującego elastyczność, która powinna wystarczyć w większości zastosowań.

7.2.1 Rodzaje aktualizacji

Aktualizacja automatyczna

Najprostszy przypadek pozwala na wykonanie aktualizacji w pełni automatycznie. Po wywołaniu metody sprawdzającej dostępność zostają wybrane wszystkie uaktualniania, a *Nomad* przeprowadza pełną aktualizację – przechodząc przez kolejne trzy etapy. Informacje o wyniku aktualizacji dostępne są jako stan obiektu aktualizatora. W czasie przeprowadzania aktualizacji automatycznych domyślna implementacja interfejsu *IUpdater* nie wysyła żadnych wiadomości, poza informacją o błędach na etapie sprawdzania i przygotowania aktualizacji.

Aktualizacja manualna

W przypadku ustawienia w konfiguracji platformy opcji manualnego trybu aktualizacji, klient (czyli moduł sterujący pracą aktualizatora) po kolei wywołuje trzy metody odpowiadające trzem etapom uaktualniania. Wyniki wywołań są zwracane jako wiadomości za pośrednictwem *IEventAggregatora* – pozwala to na przeprowadzenie aktualizacji asynchronicznie.

Przekazywane wiadomości to:

1. *NomadAvailableUpdatesMessage* – informacja zawierająca listę dostępnych manifestów modułów, które zostały zgłoszone przez *IModulesRepository*.
2. *NomadUpdatesReadyMessage* – informacja zawierająca listę manifestów, które zostały przygotowane (zazwyczaj jest to synonim słowa pobrane)

Moduł obsługujący aktualizacje, subskrybując te wiadomości może w pełni kontrolować przepływ procesu uaktualniania. W przypadku błędów w którymś z powyższych etapów, zostanie zwrócona wiadomość o błędzie oraz zrzut stosu wykonania.

Przesłaniu między domenami ulegają tylko manifesty modułów, a nie same moduły. Takie działanie poprawia wydajność komunikacji oraz zwiększa bezpieczeństwo – dostęp do paczki ma tylko moduł aktualizatora.

7.2.2 Domyślna konfiguracja

Domyślna implementacja *IUpdater*, jaka jest dostarczona razem z platformą *Nomad* umożliwia swobodną konfigurację wielu elementów. Istnieją jednak elementy niezmienne:

- każdy błąd (wyjątek) jaki się pojawi podczas procesu aktualizacji zostaje przechwycony i przekształcony na wiadomość dla *IEventAggregator*'a
- *Nomad.Updater* posiada pole typu *Nomad.UpdaterState* opisujące stan w jakim się on znajduje
- jeżeli błąd wystąpi podczas procesu aktualizacji i po wyładowaniu domeny modułów, to aktualizator przyjmie stan *Invalid* oraz udostępni informacje o przyczynie błędu
- aktualizator umożliwia wywołanie metod jedynie w przypadku, kiedy jego stan na to pozwala – instancja aktualizatora jest bezpieczna w kontekście wielu wątków

Domyślna konfiguracja aktualizatora przewiduje następujące elementy:

- *IModulesRepository* – nie jest przewidziana domyślna klasa implementująca dostęp, programista musi zdefiniować wykorzystywaną implementacje *explicite*.
- *IModulePackagera* – wykorzystywana jest klasa implementująca archiwa w postaci paczek formatu *ZIP* – więcej o formacie w dodatku B
- *IModuleFinder* – wykorzystywana jest domyślna implementacja, szukająca modułów w folderze aplikacji na podstawie otrzymanego manifestu
- *UpdaterType* – domyślnie ustawiony jest ręczny tryb aktualizacji

Ograniczenia implementacji

Koncepcja elementu aktualizującego jako nieodłącznej części rdzenia platformy okazała się problematyczna w kwestii wyładowania domeny aplikacji. Wątek, wywołując ostatnią metodę podczas aktualizacji, musi jak najszybciej opuścić domenę rdzenia, inaczej środowisko *.NET* zgłosi wyjątek.

W domyślnej implementacji, archiwa zip do czasu przeprowadzenia aktualizacji przechowywane są w pamięci w formie spakowanej. W przypadku dużej ilości modułów o dużych rozmiarach może to stanowić pewne ograniczenie, wymagające implementacji mechanizmu aktualizacji oddzielnie.

Domyślna implementacja – *NomadUpdater* oferuje jedynie „wysokopoziomowe" możliwości śledzenia postępu aktualizacji. Przekłada się to na informacje o przygotowaniu uaktualnienia lub nie. W przypadku, kiedy pożądane byłyby bardziej szczegółowe informacje (np. o stopniu zaawansowania pobierania plików) użytkownik platformy może przygotować własną implementację *IModulesRepository*. Jej zadaniem byłoby utworzenie warstwy komunikacyjnej dostarczającej

szczegółowych informacji o postępie. Do tego celu może oczywiście skorzystać z *IEventAggregatora*.

Rozdział 8

Rdzeń platformy - Nomad.Core

Aby odnieść sukces, dowolne narzędzie programistyczne musi posiadać proste i przejrzyste *API*. Dlatego w *Nomadzie* postanowiono wyróżnić klasę *NomadKernel* jako punkt wejścia dla programistów.

8.1 NomadKernel jako punkt wejścia dla programisty

Najprostszy scenariusz użycia zakłada, że autor aplikacji *thin-host*:

1. utworzy obiekt klasy *NomadKernel*, oraz
2. poda lokalizację, w której znajdują się moduły do załadowania.

Powyższa procedura pokazuje oczywiście najprostszy możliwy scenariusz inicjalizacji platformy.

NomadKernel, jako jądro platformy, oferuje wszelkie możliwe akcje związane z operowaniem na modułach:

- ładowanie
- odładowywanie
- listowanie załadowanych

Operacja ładowania korzysta z mechanizmu *IModuleDiscovery* opisanego w rozdziale 4.2. Programista może zlecić ładowanie wszystkich modułów jednym poleceniem, bądź rozbić na etapy – zgodnie z upodobaniem.

NomadKernel umożliwia jedynie odładowanie wszystkich modułów, co wynika z ograniczenia *AppDomains* i architektury *.NET* wyjaśnionego w rozdziale 2.4.2 oraz 4.1.1.

Programista *thin-hosta* jak i dowolnego modułu może pobrać z *NomadKernela* informację o wszystkich modułach załadowanych w aplikacji. Poprzez obiekt *NomadKernel* programista *thin-hosta* ma także dostęp mechanizmów komunikacyjnych *Nomada*.

Tworząc platformę poprzez *NomadKernel* programista ma możliwość wyboru implementacji wszystkich mechanizmów wewnętrznych *Nomada*.

8.2 NomadConfiguration

NomadConfiguration umożliwia programiście elastyczną konfigurację frameworka. Zbiera wiele rodzajów ustawień, które pozwalają na zdefiniowanie zasad współpracy *Nomada* z modułami, repozytoriami, itp.

Wraz z *Nomadem* dostarczana jest podstawowa konfiguracja, która wystarcza do realizacji najprostszych przypadków użycia platformy. Programista może dokonać w niej dowolnych zmian do momentu, aż nie zostanie ona *zamrożona* [ang. *frozen*]. Od tego momentu każda zmiana uznawana jest za niedozwoloną i powoduje zgłoszenie wyjątku. Podejście takie ma za zadanie nie tylko zapewnić bezpieczeństwo całej platformy ale też chronić programistę przed nieświadomie popełnionymi błędami (np. zmiana zasad weryfikacji podpisów modułów nie powoduje weryfikacji modułów już załadowanych).

Na elementy konfiguracji *Nomada* składają się:

- DependencyChecker (typu IDependencyChecker) – definiuje algorytm, który odpowiada za sprawdzenie spójności podanej konfiguracji modułów.
- ModulesDirectoryPath – katalog, w którym wyszukiwane są *Assemblies* modułów. Domyślnie jest to podkatalog „modules" w katalogu aplikacji.
- ModuleFilter (typu IModuleFilter) – filtry podejmują decyzję na podstawie informacji o *jednym* module i mogą zablokować jego ładowanie.
- ModuleFinder (typu IModuleFinder) – lokalizacja plików powstających za pomocą mechanizmu aktualizacji.
- ModulePackager (typu IModulePackager) – to mechanizm odpowiadający za rozpakowanie pobranej aktualizacji do wskazanego katalogu (w szczególności katalogu modułów).
- ModuleRepository (typu IModuleRepository) – odpowiada za komunikację z serwerem repozytorium. W domyślnej konfiguracji nie jest ustawiona, dlatego mechanizm aktualizacji nie jest dostępny.
- UpdaterType – definiuje czy aktualizacje odbywają się automatycznie (po rozpoczęciu), czy każdy krok programista musi potwierdzić.
- SignatureProvider – dostarcza informacji o wybranym trybie podpisywania paczek z modułami *Nomad*.

Każdy z powyższych elementów może być według potrzeb rozszerzany. Możliwe jest również łączenie mechanizmów dostarczanych z Nomadem z własnymi mechanizmami np. module repository, który wykorzystuje repozytorium *Nomada*, oraz inny protokół.

8.3 Usługi udostępniane przez *Nomada*

Nomad poza wszelkimi mechanizmami zdefiniowanymi w wymaganiach oferuje również rozszerzalny zestaw usług dostarczanych przez rdzeń platformy i dostępny poprzez *ServiceLocator*. Obecnie zaimplementowane zostały dwie opisane poniżej usługi.

8.3.1 Listowanie załadowanych modułów

Każdy moduł po poprawnym załadowaniu do aplikacji zostaje zapisany na liście dostępnych modułów. Poprawność danych zawartych w tej liście jest odpowiedzialnością klasy wypełniającej kontrakt *IModuleLoader* – bezpośrednio związanej z ładowaniem modułów. Jest to jednakże klasa wewnętrzna platformy. Użytkownik *Nomada* ze względu na enkapsulację nie ma do niej dostępu.

Istnieje jednak wysokie prawdopodobieństwo, że dowolny moduł będzie potrzebował wiedzy o dostępnych w aplikacji modułach. Taka sytuacja może się wydarzyć chociażby, gdy dowolny moduł będzie chciał sprawdzić dostępność konkretnej usługi znając nazwę modułu dostarczającego ją przed żądaniem jej z *ServiceLocatora*. Kluczowym jest również dostarczenie tej informacji poprzez komunikację synchroniczną. Dzięki temu odpowiada ona stanowi faktycznemu aplikacji.

Dlatego właśnie utworzono interfejs *ILoadedModulesService*. Jego jedyną odpowiedzialnością jest wydobycie z wnętrza platformy listy załadowanych modułów i w bezpieczny sposób przekazanie jej żądającemu klientowi.

8.3.2 Wsparcie dla wielojęzykowości

.NET Framework sam w sobie oferuje pewne wsparcie dla wielojęzykowości. Nie oferuje jednak zarządzania wielojęzykowością w ramach modułów, nie oferuje także możliwości szybkiej podmiany fraz w aplikacjach *WPF*. W Internecie można znaleźć wiele różnych rozwiązań oferujących rozszerzenia dla *WPFa*, jednak ze względu na ich możliwości jak i ograniczenia licencyjne nie można było ich dołączyć do *Nomada*. Zdecydowano się więc na implementację wsparcia dla wielojęzykowości w ramach tworzonej platformy.

Nomad oferuje wsparcie wyłącznie dla ciągów tekstowych, jednak bez problemu można przystosować go do wykorzystania dowolnego typu danych (typu *object*). Podstawowym założeniem była łatwa integracja z *WPFem* w postaci rozszerzenia do *Xamla*.

Drugim założeniem była możliwość pochodzenia tłumaczeń z dowolnego źródła (niekoniecznie z *Resource'ów* aplikacji). Użytkownik ma możliwość pobierania tłumaczeń na bieżąco bezpośrednio z bazy danych, xmla bądź dowolnego innego źródła. Jest to możliwe, gdy użytkownik dokona własnej implementacji interfejsu *IResourceSource* na który składa się wyłącznie jedna metoda: *Retrieve(nazwa_zasobu)*. Dzięki temu można w łatwy sposób zaadaptować dowolne źródło danych.

Zakłada się także, że każdy moduł może dostarczać własne źródła lokalizacyjne (czy to zasoby, czy inne implementacje) – podczas pobierania tłumaczeń należy je także uwzględnić.

Pożądanym zachowaniem jest wykorzystanie kultury wątku *UI* jako źródła informacji na temat lokalizacji językowej aplikacji. Dlatego też programista nie specyfikuje menadżerowi zasobów, w jakim języku chce otrzymać wskazany tekst.

Architektura

Na architekturę rozwiązania składają się następujące elementy:

1. *ResourceProvider* – umożliwia dodawanie nowych źródeł zasobów w określonych językach oraz pobieranie odpowiednich tłumaczeń. Podczas pobierania język jest automatycznie wyznaczany na podstawie kultury wątku. Istnieje wyłącznie jedna instancja *ResourceProvidera* w domenie aplikacji.

2. *IResourceSource* – źródło zasobów w określonym (jednym) języku. Dla każdego języka mogą być utworzone osobne źródła. Istnieje także możliwość posiadania jednego źródła i wtedy to ono dokonuje decyzji, z którego języka w danym momencie skorzystać.

3. *ResourceExtension* – rozszerzenie składni języka *XAML*, umożliwiające wykorzystanie zasobów bezpośrednio w definicji *UI*.

4. *TranslationSource* – wartość zwracana przez *ResourceExtension*. Zapewnia aktualizacje zawartości elementów graficznych dzięki wykorzystaniu mechanizmu wiązania (*Binding*).

Implementacja

Aby dostarczyć tłumaczenia własnego modułu, programista musi zgłosić (najlepiej w trakcie ładowania modułu) swoich dostawców tłumaczeń dla poszczególnych kultur. Proces dodania tłumaczeń przy wykorzystaniu *ResourceManagera* dostarczanego z *.NETem* przedstawia listing 8.1.

Listing 8.1: Dodanie tłumaczeń

```
resourceProvider.AddSource("en-GB",
        new ResourceManagerResourceSource(resourceManager));
```

Wraz z Nomadem dostarczany jest *ResourceManagerResourceSource*, który dostarcza między innymi zasoby tekstowe (ale także dowolne inne – graficzne, dźwiękowe) z wykorzystaniem *ResourceManagera .NET* czerpiącego dane z plików *resx*.

Schemat UML rozwiązania przedstawiony jest na rysunku 8.1.

Pobieranie tłumaczeń w kodzie obrazuje listing 8.2. W celu pobrania zasobów z poziomu *XAML-a* zaimplementowano *MarkupExtension* którego działanie prezentuje listing 8.3.

Listing 8.2: Pobieranie tłumaczeń w kodzie

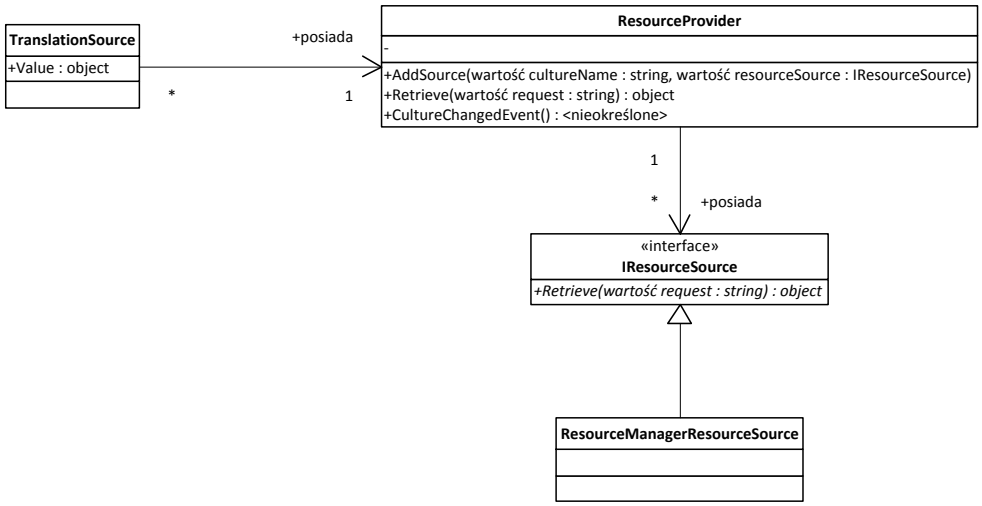
```
var text = resourceProvider.Retrieve("NAZWA_ZASOBU");
```

Listing 8.3: Pobieranie tłumaczeń w *XAMLu*

```
<Button Content="{Internationalization:Resource NAZWA_ZASOBU}"></Button>
```

Dzięki zastosowaniu wiązania *WPF* natychmiast po zmianie kultury wątku następuje w aplikacji odświeżenie tłumaczeń i wyświetlenie nowych.

Aby zmienić język aplikacji, dowolny moduł musi opublikować zdarzenie *NomadCultureChangedMessage*, w którym wskazuje nową docelową kulturę aplikacji.



Rysunek 8.1: Diagram klas lokalizacyjnych

Rozdział 9

Zapewnienie jakości produktu

W celu zapewnienia jakości produktu końcowego, w trakcie prac nad platformą położono szczególny nacisk na dwa aspekty:

- testowanie kodu oraz funkcji *Nomada*
- udostępnianie *Nomada* w postaci gotowego rozwiązania

9.1 Testowanie *Nomada*

Jak już zostało opisane w rozdziale 2.2, testowanie jest bardzo ważnym aspektem budowy platform programistycznych. Podczas budowy wyróżniono trzy kategorie testów, zgodne w wcześniej wprowadzonym podziałem:

- testy jednostkowe – *Unit Tests* – zapewniające poprawność kodu napisanego przez autorów platformy. W szczególności poprawność działania pojedynczych klas w izolacji od ich zależności.
- testy integracyjne – *Integration Test* – w teoretycznym ujęciu testy jednostkowe, których zadaniem jest sprawdzenie oczekiwanej współpracy pomiędzy poszczególnymi elementami. Działają jednak one jednak na „nieco wyższym poziomie" niż klasyczne testy jednostkowe sprawdzające kod w izolowanym otoczeniu, dlatego zostały wyróżnione.
- testy funkcjonalne – *Functional Tests* – testy wykorzystujące publiczne API *Nomada* jako jedyne dostępne. Testują platformę w dokładnie taki sposób w jaki będzie używana przez programistów.

Testowanie jest skuteczne jedynie wtedy, kiedy pokrywa znaczącą część funkcjonalności – im więcej testów, tym więcej przypadków rozpatrzonych. W trakcie implementacji platformy *Nomad*, w duchu metodyki *Test Driven Development* starano się uzyskać maksymalne możliwe pokrycie testami z kategorii *Unit Tests* oraz *Functional Tests*.

Takie działanie gwarantuje, że wszystkie najbardziej prawdopodobne scenariusze użycia platformy opisane są testami z kategorii *Functional Tests*. Kategoria *Integration Tests* została wprowadzona w celu jasnego podziału pomiędzy testami wynikającymi z przypadków użycia a scenariuszami rozpatrywanymi wewnętrznie w platformie.

Podczas budowy platformy *Nomad* wykorzystano narzędzie do pomiaru „pokrycia kodu testami" (ang. *Code coverage*) – czyli metryki określającej, jaka część napisanego kodu będzie przetestowana podczas automatycznych testów. Istnieje wiele możliwości pomiaru tego parametru. W projekcie wykorzystano program *dotCover* firmy *JetBrains*, szacujący wartość tej metryki w oparciu o wykonane instrukcje. Program ten jest dostępny za darmo dla projektów typu Open-Source oraz instytucji edukacyjnych. Wszystkie testy biblioteki *Nomad* pokrywają 81% instrukcji według wyżej opisanego sposobu wyznaczania pokrycia kodu.

Nazwane kategorie pojawiają się w kodzie *frameworka* jako atrybuty opisujące testy. Umożliwia to sterowanie automatem włączającym testy według następującego algorytmu:

1. Uruchom *Unit Tests*.
2. Jeśli błąd to zakończ.

3. Uruchom *Integration Tests*.
4. Jeśli błąd to zakończ.
5. Uruchom *Functional Tests*.
6. Jeśli błąd to zakończ.

Taki mechanizm ogranicza ilość informacji jakie docierają do testera. Pozwala mu także skupić się na ważniejszym aspekcie w pierwszej kolejności. Kiedy zawiedzie test z kategorii *Unit Tests*, są bardzo małe szanse, że pozostałe kategorie uzyskają pomyślny wynik.

Budowanie testów

Pełne przetestowanie produktu, zwłaszcza w kategorii testów *Functional Tests*, jest bardzo trudne. *Nomad* jako platforma do budowy aplikacji prawie zawsze potrzebuje do działania conajmniej jednego modułu. Przetestowanie jak największej ilości przypadków wymaga stworzenia wielu modułów, często realizujących zupełnie różne zadania. Stworzenie takich modułów *statycznie* (przy użyciu tradycyjnych narzędzi do kompilacji kodu) byłoby bardzo czasochłonne i utrudniałoby pracę z pozostałymi fragmentami kodu platformy.

W platformie *Nomad* zastosowano podejście dynamiczne, polegające na tworzeniu odpowiedniego modułu tuż przed wykonaniem testu i usunięciu go tuż po wykonaniu. Wykorzystano mechanizmy środowiska *.NET* pozwalające na dynamiczną generację kodu podczas wykonania (*CodeDOMProvider*). Do zalet takiego rozwiązania należy elastyczność – tester może tuż przed testem zdefiniować jakie działania ma przeprowadzić moduł, zdefiniować jego kod. Wadą jest wydłużenie czasu działania testów związane z wykonywaniem operacji zapisu na dysk oraz konieczność stworzenia mechanizmu dynamicznej kompilacji modułu.

Testowanie *GUI*

Znaczną część unikatowości platformy *Nomad* stanowi wsparcie dla graficznej kompozycji interfejsu użytkownika. Baza testów funkcjonalnych nie byłaby kompletna bez testowania akcji wykonywanych przez użytkownika na graficznych kontrolkach, czyli bez przetestowania warstwy *GUI*. Dlatego w ramach zaimplementowanych testów znajduje się duża grupa testów sprawdzających działanie mechanizmów *Nomad.Regions* przy wykorzystaniu narzędzia do testowania *GUI* – biblioteki *White*.

Testy akceptacyjne

Rolę testów akceptacyjnych pełnią częściowo testy z kategorii *Functional Tests*. Jednak w przypadku budowy narzędzia programistycznego, a zwłaszcza biblioteki wykorzystywanej przez aplikacje najlepszymi kandydatami na testy akceptacyjne są działające programy wykorzystujące *Nomada*. W ramach projektu wykonano szereg przykładowych małych aplikacji oraz jedną dużą. Więcej informacji znajduje się w rozdziale 10.2.

Narzędzia wspomagające zapewnianie jakości produktu

Podczas prac nad projektem wykorzystano szereg narzędzi umożliwiających przeprowadzenie testowania jak i znacznie je przyspieszające. Adresy internetowe oraz dokładne wersje wszystkich przytoczonych tutaj narzędzi znajdują się w dodatku A.2.

NUnit

Narzędzie odpowiadające za uruchamianie testów platformy. W jego skład wchodzi biblioteka *.NET* która umożliwia pisanie testów oraz środowisko uruchomieniowe testów. *NUnit* pozwala na wydzielenie poszczególnych kategorii testów i w zależości od potrzeb uruchomienie odpowiedniego zestawu.

Moq

Biblioteka, której zadaniem jest wspomaganie pisania testów – pozwala bowiem na tworzenie tzw. „mocków". Obiekty *Mock* umożliwiają testowanie zachowań i jednocześnie ograniczają powtarzanie kodu testów w przypadku podobnych testów ([Mes07]). Pozwalają one między innymi na tworzenie obiektów, które implementują pewien określony interfejs – programista musi podać tylko fragment jego implementacji, niezbędny do przeprowadzenia testów. Możliwe jest również określanie tzw. oczekiwań. Programista może zdefiniować, że pewne metody powinny być wywołane z argumentami spełniającymi pewne warunki. Brak wykonania metod skutkuje niepowodzeniem testu. Pozwala to testować niebezpośrednie wyjścia testowanej funkcjonalności.

Project White

Stanowi platformę do automatyzacji testów interfejsu użytkownika. Innymi słowy, odpowiada za przeprowadzanie testów na najwyższym poziomie – uruchamia bowiem aplikacje, które korzystają z elementów frameworka i symuluje zachowanie użytkownika (klikanie myszką, klawiaturą). Po przeprowadzeniu sekwencji operacji weryfikowana jest zgodność rezultatu z oczekiwaniem.

Biblioteka *White* nie działa poprawnie w środowisku 64-bitowym. W związku z błędami w kodzie odpowiedzialnym za wywołanie metod *WinAPI* konieczne jest uruchamianie testów w trybie 32-bitowym. Sama testowana aplikacja może działać natomiast w trybie 64-bitowym. W przypadku platformy *Nomad* testy tworzą testowaną aplikacje w ramach własnego procesu, więc zarówno testy jak i aplikacja muszą działać w trybie 32-bitowym. Osiągnięto to, korzystając z 32-bitowego programu *nunit-console-x86.exe*, zamiast *nunit-console.exe*, który uruchamia się w trybie 64-bitowym na 64-bitowych systemach operacyjnych.

Warto nadmienić, iż początkowo zamiast *White'a* planowano korzystać z *Wipflasha*. Projekt ten był jednak w bardzo wczesnej fazie rozwoju i dostarczana funkcjonalność okazała się niewystarczająca.

9.2 Udostępnianie *Nomada*

Platforma jako produkt powinna być łatwo dostępna. *Nomad* został wykonany zgodnie z koncepcją ciągłej integracji (z ang. *Continuous Integration*, CI) – koncepcją budowy oprogramowania małymi krokami oraz nieustającej integracji już opracowanych, niekoniecznie kompletnych rozwiązań. Serwer CI służy także, za ostateczną instancję weryfikującą poprawność przygotowanego kodu. Do budowy pełnowartościowego serwera potrzeba narzędzi do zarządzania:

- samym serwerem CI
- procesem budowy i testowania oprogramowania
- kodem źródłowym
- dokumentacją

Poniżej omówiono narzędzia wykorzystane podczas budowy platformy *Nomad*. Więcej informacji o każdej z przedstawionych technologii znajduje się w dodatku A.3.

Psake

Narzędzie automatyzujące proces budowania aplikacji oparte o *PowerShell*. Pozwala definiować dowolne zadania oraz zależności na zadaniu. Następnie sortuje zadania według wymagań dotyczących zależności – tak, by wykonując zadanie wszystkie jego zależności były spełnione.

W *Nomadzie* wyróżniono ogólne kroki budowania aplikacji, gdzie każdy kolejny jest zależny od poprzedniego:

- Przygotowanie struktury katalogów.
- Skompilowanie *Nomada*.
- Uruchomienie zestawu testów w kolejności *Unit, Integration, Functional*.
- Wygenerowanie dokumentacji.

W przypadku, gdy którekolwiek z zadań nie zostanie wykonane, następne nie są już wykonywane, a proces budowy zostaje przerwany z błędem.

Hudson Continuous Integration Server

Jest serwerem ciągłej integracji, który odpowiada za dostarczanie nowych wersji platformy oraz zapewnienie integralności (kompletności i poprawności w kontekście automatycznego wykonania kompilacji i testów aplikacji) głównej gałęzi repozytorium.

Po każdej zmianie w repozytorium (w gałęzi *staging*) uruchamiana jest tak zwana szybka budowa platformy, na którą składa się skompilowanie platformy i uruchomienie testów jednostkowych. Jeżeli wskazane czynności zakończą się powodzeniem, zmiany migrowane są do gałęzi głównej.

Raz dziennie uruchamiany jest kompletny proces budowania platformy. Jest on kilkukrotnie dłuższy niż szybka kompilacja, gdyż wykonuje również testy integracyjne i funkcjonalne oraz buduje dokumentację. Efektem tego zadania jest powstanie kompletnej paczki z platformą. Struktura uzyskanej paczki wygląda następująco:

- *Nomad* – pliki wyjściowe platformy.
  - bin – zawiera binarne pliki *Nomada*.
  - doc – zawiera dokumentację *Nomada*.
- Tools – zawiera narzędzia dodatkowe dostarczane z Nomadem (np. kreator manifestów czy narzędzie do podpisywania modułów).

Microsoft Sandcastle

Narzędzie to generuje dokumentację *Nomada* na podstawie kodu opisanego odpowiednio ustrukturalizowanymi komentarzami. Generuje pliki *CHM*, w których znajduje się opis wszystkich klas i interfejsów systemu. Wygenerowana dokumentacja przypomina stylem dokumentację *MSDN*. Poruszanie się po niej powinno więc być intuicyjne dla programistów *.NET*.

SCM git

Git to rozproszony system kontroli wersji. Pozwala na przechowywanie informacji o wszystkich zmianach w kodzie i cofnięcie się bądź porównanie określonych wersji. Ponadto, pozwala programistom na umieszczanie w nim zmian bez posiadania dostępu do internetu – każdy z programistów posiada własną, pełną kopię repozytorium na której operuje. Wyróżniony został jednak centralny serwer, do którego, mówiąc terminologią gita, „wpychane" są zmiany z lokalnych repozytoriów. Z tego centralnego serwera pozostali użytkownicy uzyskują aktualny kod.

Na potrzeby *Nomada* utworzono dwie gałęzie w repozytorium:

- master – to gałąź, w której znajduje się najnowszy i stabilny (kompilujący się i przechodzący testy).
- staging – to gałąź, której stabilność może być zachwiana w przypadku niepoprawnych zmian. W niej programiści umieszczają swoje zmiany.

Przenoszenie stabilnego kodu z gałęzi *staging* do gałęzi *master* realizuje automatycznie serwer *Hudson* po każdym udanym uruchomieniu procesu budowy aplikacji.

Rozdział 10

Zastosowanie wyników pracy

Opracowana platforma jest bazą stanowiącą przyczynek do budowy złożonych aplikacji. Bezproduktywnym byłoby wydanie środowiska, którego możliwości nie są w żaden sposób zaprezentowane. Niniejszy rozdział opisuje zbudowane w oparciu o platformę rozwiązania, czy to przygotowane przez samych autorów czy studentów specjalności Technologie Wytwarzania Oprogramowana w ramach współpracy. W rozdziale znajduje się także, krótkie wprowadzenie dotyczące użycia platformy.

10.1 Użycie *Nomada*

W celu zastosowania platformy należy do budowanej aplikacji dołączyć bibliotekę *Nomad* wraz ze wszystkimi zależnościami (biblioteki *Castle.Core*, *Castle.Windsor* oraz *Ionic.Zip.Reduced*). Każdy moduł mający współpracować z taką aplikacją musi posiadać także referencje na przygotowywany produkt. Zgodnie z założeniami projektu, *Nomad* jest udostępniany dla *.NET Framework* w wersji 3.5.

W przypadku chęci współpracy z modułami skompilowanymi w wersji nowszej niż 3.5 wymagana jest rekompilacja *Nomada* w wersji docelowej – wymagają tego mechanizmy środowiska *.NET*. Proces kompilacji można wywołać przez wykonanie skryptu *runBuild.bat*, który znajduje się w głównym katalogu ze źródłami.

10.2 Samouczki

Jedna z podstawowych zasad tworzenia frameworka mówi o konieczności pisania przykładowych aplikacji wykorzystujących poszczególne jego elementy [KC08]. Wynika to z faktu, że dzięki temu programista frameworka może wcielić się w rolę klienta i zobaczyć jak API, które zaprojektował może być w praktyce wykorzystywane. Dlatego też wraz z biblioteką *Nomada* dostarczane są przykłady (tak zwane samouczki, *tutorials*), które prezentują poszczególne elementy funkcjonalne platformy. Każdy z nich prezentuje wybraną funkcjonalność *Nomada*, której wykorzystanie jest w nich dogłębnie opisane w komentarzach. Powstały następujące prezentujące następujące elementy:

1. ładowanie i odładowanie modułów – *thin-host*, najprostszy możliwy moduł – *Hello Nomad*
2. komunikację synchroniczną – *thin-host*, moduł dostarczający usługę, moduł żądający usługi
3. komunikację asynchroniczną – *thin-host*, moduł publikujący zdarzenia, moduł subskrybujący
4. tworzenie aplikacji *WPF* i aktualizacje – *thin-host*, moduł *Shell*, moduł dostarczający widok dla regionu, moduł aktualizatora

Samouczki przedstawiające komunikację międzymodułową zostały dodatkowo wykonane zgodnie z zaleceniami opisanymi w poniższym akapicie.

Zależność usługowa – *ServiceDependencies*

W wersji *Nomada* dostarczonej do pracy korzystanie z mechanizmów komunikacyjnych w przypadku ładowania wszystkich modułów jednym *IModuleDiscovery* wymaga dostosowania się do pewnych zaleceń przy oprogramowywaniu *IModuleBootstraper*.

W komunikacji asynchronicznej poprzez *Event Aggregator* wyróżniamy moduł słuchający zdarzeń – *Subscriber* oraz moduł publikujący zdarzenia – *Publisher*. By żadne zdarzenie nie zostało przez moduł *Subscriber* pominięte musi się na nie zapisać w metodzie *OnLoad()* wywoływanej w trakcie ładowania. Moduł *Publisher* może wymagać natomiast pewności, że wszyscy potencjalni słuchacze jego zdarzeń są już zapisani. Stąd rozpoczęcie publikowania poleca się odsunąć do otrzymania zdarzenia *NomadAllModulesLoadedMessage*.

Sytuacja odwraca się w przypadku komunikacji synchronicznej. Moduł *ServiceProvider* musi od razu przy ładowaniu zarejestrować się w *ServiceLocatorze* jako dostawca usługi. Odbiorca – *ServiceResolver* może jej żądać dopiero otrzymawszy informacje, że wszystkie moduły zostały załadowane. W przeciwnym wypadku może wystąpić wyjątek nieodnalezienia dostawcy, który ze względu na swoje zależności dopiero oczekuje na załadowanie.

W obu powyższych przypadkach moduły zarówno dostawcy jak i odbiorcy usług muszą posiadać referencję na *Assembly* zawierający interfejs komunikacyjny oraz dostarczać go w swoich paczkach.

Istnieje alternatywne rozwiązanie opisanych powyżej problemów. Zostało ono zaprezentowane w rozdziale 11.2.2.

W celu zaprezentowania (i przetestowania) możliwości frameworka jako całości powstała również większa aplikacja wykorzystująca je wszystkie.

10.3 Narzędzia dostarczane wraz platformą

W celu pełnego wykorzystania możliwości platformy dostarczane są wraz z nią narzędzia, które umożliwiają generowanie podpisów dla modułów.

10.3.1 Generowanie kluczy RSA

Wraz z *Nomadem* dostarczany jest generator kluczy RSA. Generuje on parę kluczy (prywatny i publiczny) o zadanej długości. Argumenty *KeyGenerator.exe*:

- Ścieżka do pliku wynikowego, zawierającego klucz publiczny i prywatny.
- (opcjonalny) Ścieżka do pliku wynikowego, zawierającego wyłącznie klucz publiczny (który można przekazać dalej).
- (opcjonalny) Długość wygenerowanego klucza w bitach – domyślnie 1024.

Wygenerowane za pomocą tego narzędzia klucze można wykorzystać do podpisywania modułów (generatorem manifestów).

Ponieważ generowanie kluczy o odpowiednim poziomie bezpieczeństwa wykracza poza temat niniejszej pracy, zastosowano rozwiązanie dostarczane wraz z *frameworkiem .NET*: *RSACryptoServiceProvider*, które to odpowiada za wygenerowanie kluczy i przekształcenie ich do odpowiedniego *XMLa*. Generator kluczy dla *Nomada* wykorzystuje bezpośrednio tę klasę [Mic03a] i sam nie odpowiada w żaden sposób za jej logikę.

10.3.2 Generowanie manifestów zgodnych z formatem *Nomad* w formacie XML

Immanentnym elementem modułu jest manifest go opisujący. Nie można bez niego załadować modułu, ani pobrać go z repozytorium. Dokładny opis manifestu znajduje się w rozdziale 4.2, tu wystarczy tylko nadmienić, iż manifest zawiera wszystkie informacje niezbędne do sprawdzenia integralności modułu (także z punktu widzenia jego zależności).

Sam manifest to struktura, która może być generowana automatycznie – wystarczy mu bowiem dostarczyć minimum informacji:

- Zestaw plików które wchodzą w skład modułu – w szczególności wystarczy w jednym folderze umieścić komplet plików jednego modułu.
- Lista modułów tworzących zależności.

Kolejnymi argumentami *KeysGenerator.exe* są więc:

1. typ podpisu – do wyboru RSA (bez łańcucha) lub PKI.

2. ścieżka do pliku xml z kluczem prywatnym wystawcy / potwierdzającego.
3. ścieżka do katalogu ze wszystkimi plikami modułu.
4. nazwa pliku stanowiąca jednocześnie assembly z *IModuleBootstraper*. Stanowić ona będzie jednocześnie nazwę modułu.
5. nazwa wystawcy / producenta modułu.

Ponadto, generator manifestów wykorzystuje plik *„dep.conf"*, w którym w kolejnych liniach podawane są ścieżki do modułów od których wskazany moduł jest zależny. W momencie uruchomienia sprawdzana narzędzia sprawdzane są wersje zależnych modułów i niezbędne dane zapisywane są w manifeście.

Wynikiem działania tego programu są dwa pliki:

- Manifest modułu.
- Podpis (sygnatura) manifestu modułu.

Można zauważyć, że program ten łączy tak naprawdę dwie odpowiedzialności – wystawcy modułu (manifestu) oraz podpisującego go. Dzięki połączeniu tej odpowiedzialności upraszczany jest scenariusz, w którym to twórca modułu jednocześnie go podpisuje (robi to w jednym kroku). Jeżeli instytucja podpisująca moduł nie jest tworzącą moduł to istnieje mniejsze ryzyko, że ktoś umieści manifest w którym zawrze fałszywe informacje – bo pełną kontrolę nad zawartością manifestu posiada instytucja go tworząca i weryfikująca.

W ten sposób możliwe jest, by firma A wyprodukowała produkt, a instytucja zaufana, po zweryfikowaniu produktu (np. czy nie narusza wymogów bezpieczeństwa) podpisała swoim zaufanym certyfikatem wskazany moduł (pozostawiając oryginalnego producenta).

10.4 Serwer modułów jako przykład aplikacji współpracującej z systemem Nomad

W ramach budowania projektu *Nomad* powstał serwer pełniący rolę repozytorium modułów. Repozytorium to jest *przykładową* implementacją odpowiadającą za zaprezentowanie możliwości wbudowanego w platformę mechanizmu aktualizacji. Jak już zostało wspomniane w rozdziale 7 współpraca pomiędzy platformą *Nomad* a jakimkolwiek repozytorium wymaga dostarczenia kompatybilnej implementacji *IModulesRepository* z tym serwerem.

10.4.1 Przykładowa implementacja strony klienckiej

W *Assembly* platformy *Nomad* dystrybuowana jest także paczka klas, które stanowią przykładową implementację *IModulesRepository*. Przygotowane repozytorium umożliwia współpracę z serwerem udostępniającym moduły za pośrednictwem protokołu HTTP. Adres serwera jest parametrem, co umożliwia zastosowanie tej przykładowej klasy jako rzeczywistego mechanizmu w realnej aplikacji.

Uproszczenia

Zdecydowano, iż przykładowa implementacja nie powinna wprowadzać żadnych nadmiarowych mechanizmów – na przykład do sprawdzania poprawności danych czy sygnalizacji ilości pobranych danych – gdyż mogły by one zaburzyć odbiór idei stojącej za *IModulesRepository*. W realnym zastosowaniu warto by zastanowić się nad wprowadzeniem takich mechanizmów.

Informacja o dostępnych aktualizacjach

W zaimplementowanym modelu serwer wystawia plik XML zawierający:

- listę manifestów modułów jakie są dostępne na serwerze
- adres konkretnego modułu

Poniżej znajduje się fragment takiego pliku:

Listing 10.1: Fragment tego pliku zawierającego informacje o aktualizacjach

```
<WebAvailablePackagesCollection xmlns...>
  <AvailablePackages>
    <WebModulePackageInfo>
      <Url>/Modules/1</Url>
      <Manifest>
        <Issuer>TES_ISSUER_COMPILER</Issuer>
        <ModuleName>ModuleWithDependency</ModuleName>
        <ModuleVersion>
          <Build>0</Build><Major>0</Major><MajorRevision>0</MajorRevision>
          <Minor>0</Minor><MinorRevision>0</MinorRevision><Revision>0</Revision>
        </ModuleVersion>
        <ModuleDependencies>
          <ModuleDependency>
            <ModuleName>DependencyModule2</ModuleName>
            <MinimalVersion>...</MinimalVersion>
            <ProcessorArchitecture>MSIL</ProcessorArchitecture>
          </ModuleDependency>
        </ModuleDependencies>
        <SignedFiles>
          <SignedFile>
            <FilePath>\DependencyModule2.dll</FilePath>
            <Signature>...</Signature>
          </SignedFile>
          ...
        </SignedFiles>
      </Manifest>
    </WebModulePackageInfo>
    ...
  </AvailablePackages>
</WebAvailablePackagesCollection>
```

Przesyłanie aktualizacji

Domyślna implementacja przesyła paczki w formacie *ZIP* spakowane przy użyciu biblioteki *DotNetZip*. W ramach paczki umieszczane są pliki niezbędne do działania aktualizowanego modułu. Wymagana jest obecność *Assembly* oraz manifestu platformy *Nomad*. W paczce może twórca modułów umieścić również dowolną ilość dodatkowych plików. Paczka po pobraniu przechowywana jest pamięci – nie jest zapisywane na dysk do czasu wykonania aktualizacji.

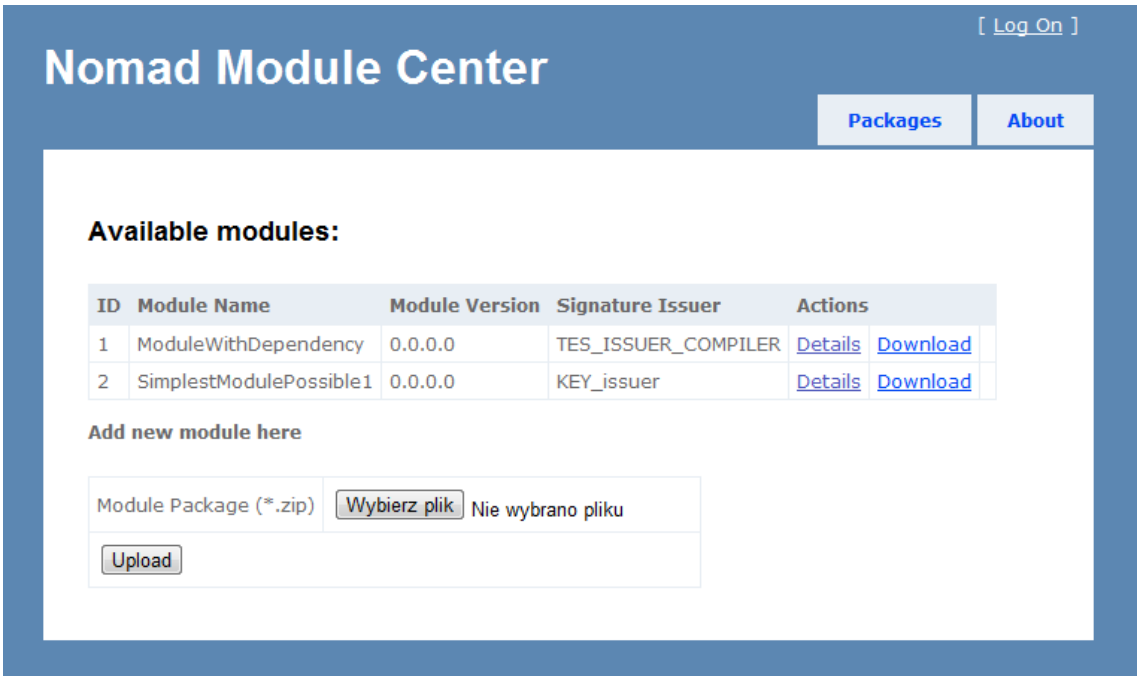
10.4.2 Przykładowa implementacja strony serwerowej

Serwer wykonano w technologii *ASP.NET MVC2*. Poza wystawianiem odpowiedniego pliku *XML* opisującego dostępne aktualizacje oraz samych aktualizacji, program ten stanowi przykład implementacji większego repozytorium wykorzystującego bazę danych jako mechanizm przechowywania danych. Ukazuje również webowy interfejs użytkownika pozwalający na zarządzanie dostępnymi na serwerze modułami.

Serwer posiada następujące możliwości:

- Dodawanie, usuwanie, ściąganie modułów za pomocą internetowego interfejsu użytkownika. Wygląd głównej strony pokazuje rysunek 10.1
- Zaimplementowany system uprawnień przy wykorzystaniu szablonu klas *MembershipProvider* – ograniczający możliwości niezalogowanego użytkownika serwisu tylko do przeglądania repozytorium.
- Prosta walidacja poprawności paczki – serwer nie przyjmie paczek w formacie innym niż *ZIP* oraz nieposiadających obowiązkowo jednego *Assembly* i manifestu platformy *Nomad* go opisującego.
- Elastyczny mechanizm przechowywania informacji w bazie danych przy wykorzystaniu mechanizmów odwzorowania relacyjno-obiektowego *LinqToSql*.

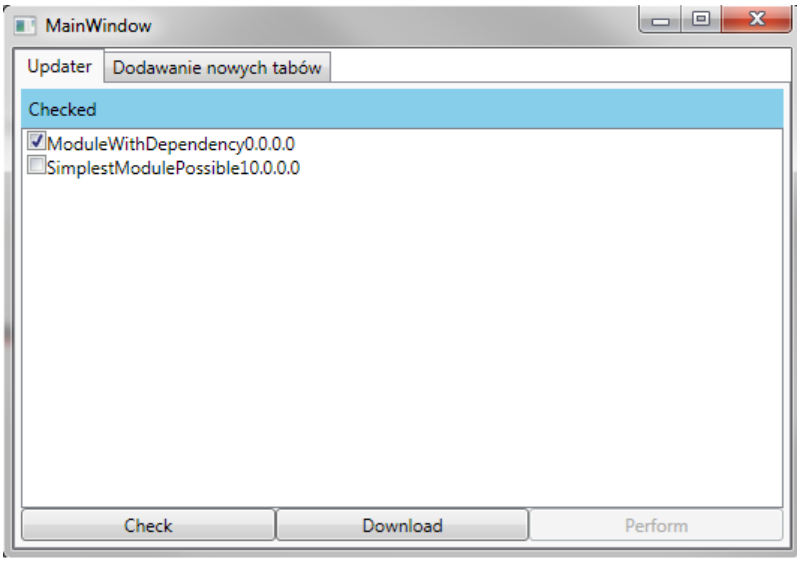
Przykładowy serwer pomimo swojej prostoty jest doskonałym pokazem możliwości platformy *Nomad* oraz platformy *ASP.NET MVC2* jako środowiska do tworzenia aplikacji webowych przyjaznych metodyce *Test Driven Development*.



Rysunek 10.1: Widok głównej strony przygotowanego serwera

10.4.3 Moduł Updater'a - graficzny moduł aktualizacji

W ramach jednego z dostarczonych samouczków zaprezentowano wykorzystanie rozwiniętej platformy do budowy graficznego modułu obsługującego *NomadUpdater*. Jest to przykładowy moduł, którego zadaniem jest zaprezentowanie obsługi mechanizmów aktualizacji oraz pokazanie sposobu na graficzne budowanie aplikacji. Graficzny interfejs użytkownika modułu prezentuje poniższy rysunek 10.2.



Rysunek 10.2: Interfejs przygotowanego modułu graficznego aktualizatora

10.5 Przykładowa aplikacja prezentująca możliwości frameworka

W celu zaprezentowania integracji wszystkich mechanizmów oferowanych przez *Nomada* postanowiono utworzyć aplikację prezentującą pełną funckjonalność platformy.

Wymagania względem aplikacji

Wyróżniono dla niej następujące wymagania:

- Wykorzystuje *EventAggregatora* do przesyłania komunikatów – w tym celu zdefiniowany będzie moduł odpowiadający za wybranie pliku z dysku, oraz moduły które nasłuchują komunikatów odnośnie wybranego pliku z dysku, otwierając te typy plików, których obsługę implementują.
- Użyje *ServiceLocatora*
- Prezentuje interfejs użytkownika przy wykorzystaniu regionów:
  - Programista aplikacji definiuje regiony, z których programiści modułów mogą korzystać. Wykorzystane kontrolki:
    - ToolbarTray
    - Toolbar
    - StatusBar
    - TabControl
    - ItemsControl
  - Programista modułu definiuje region, z którego korzystać będzie inny moduł.
  - Zawartości regionów zmieniają się kontekstowo w zależności od wybranej w danym momencie zakładki w TabControl.
- Pozwala na przetwarzanie w tle i informuje o zakończeniu przetwarzania.
- Prezentuje załadowane moduły.
- Umożliwia zmianę języka.

Należy jednak podkreślić, iż zadaniem tej aplikacji jest prezentacja możliwości frameworka (wykorzystanie z punktu widzenia użytkownika frameworka – programisty), a nie dostarczenie rozwiązania konkretnego problemu (wykorzystanie z punktu widzenia klienta).

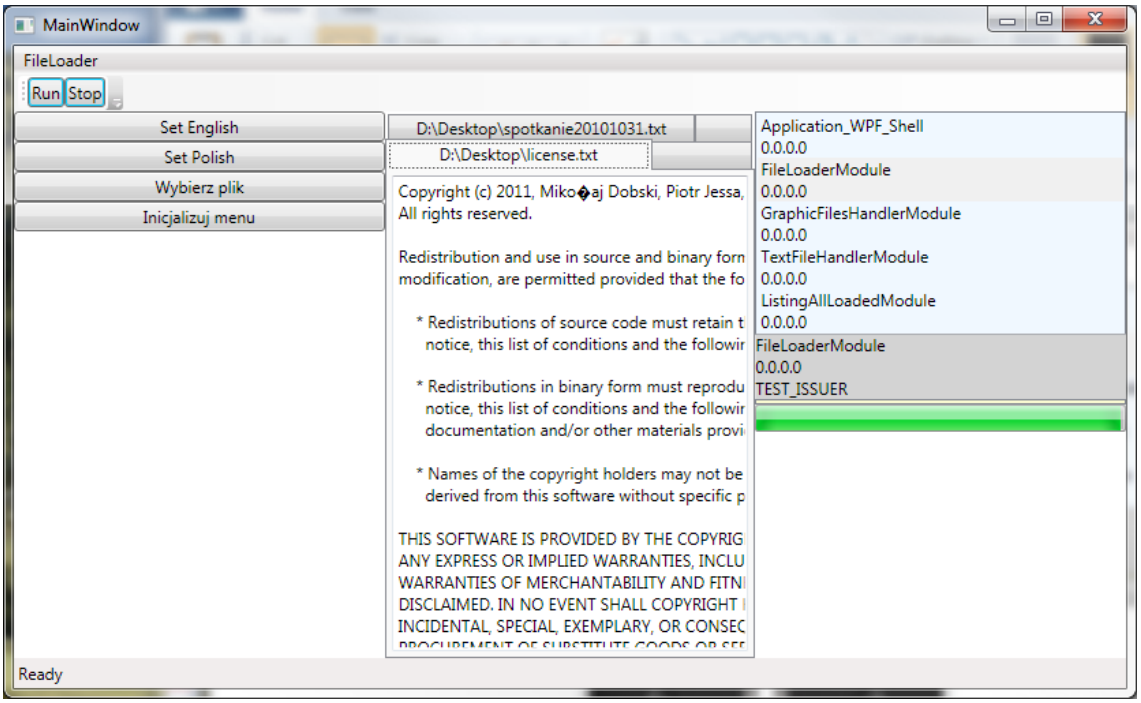
Implementacja

Żeby spełnić powyższe wymagania, sprecyzowano następującą aplikację:

- Moduł wybierający plik z dysku – po wskazaniu pliku na dysku wysyła komunikat za pomocą *EventAggregatora* o wybranym pliku.
- Moduł otwierający graficzny plik z dysku – po otrzymaniu komunikatu o wybranym pliku, jeżeli wybrany plik jest plikiem graficznym, pojawia się jako nowa zakładka w głównym regionie aplikacji.
- Moduł otwierający plik tekstowy z dysku – po otrzymaniu komunikatu o wybranym pliku, jeżeli wybrany plik jest plikiem tekstowym, pojawia się jako nowa zakładka w głównym regionie aplikacji.
- W przypadku zmiany zakładki zmieniany jest stan w pasku stanu, zmieniany jest także zestaw toolbarów w zależności od typu aktualnie wyświetlanej informacji.
- Po wskazaniu folderu, moduł odpowiedzialny za przetwarzanie w tle co 100ms będzie zgłaszał postęp, po 100 cyklach (10 sekundach) przetwarzanie zostanie zakończone. W zależności od stanu przetwarzania zmieniać się będzie wartość paska postępu oraz stan w status barze.
- Po kliknięciu przycisku następuje cykliczna zmiana języka aplikacji – angielski, polski...
- Moduł wybierający plik z dysku tworzy nowe poddrzewo menu „Obsługiwane pliki”, w którym moduł graficzny i tekstowy umieszczają swoje informacje „About”.
- Zaprezentuje listę modułów załadowanych i ich zależności.

10.5.1 Zrzuty ekranu

Rysunek 10.3: Interfejs przygotowanej aplikacji po polsku



Rysunek 10.4: Interfejs przygotowanej aplikacji po angielsku

Źródła opisanej aplikacji znajdują się na dołączonej płycie CD.

10.6 Przykłady praktycznego wykorzystania frameworka

Zadaniem studentów ze specjalizacji *Technologia Wytwarzania Oprogramowania* były skorzystanie z *Nomada*, stworzenie w pełni funkcjonalnej aplikacji i dostarczenie twórcom platformy sugestii zmian, rozwoju, czy wykrytych błędów.

Rozdział 11

Podsumowanie oraz dalsze perspektywy platformy

Ten rozdział podsumowuje wyniki przeprowadzonych prac oraz porusza zagadnienia związane z publikacją platformy *Nomad*. Omówione zostaną również perspektywy dalszego rozwoju projektu.

11.1 Założenia a wyniki implementacji

Głównym celem postawionym budowanej platformie *Nomad* było zapewnienie kompleksowego wsparcia do budowy aplikacji modułowych w oparciu o zalecenia *Microsoft Composite Application Guidance*. Zadanie to w aspekcie wymagań funkcjonalnych zostało w pełni zrealizowane. Świadczy o tym seria aplikacji zbudowanych przy wykorzystaniu mechanizmów oferowanych przez opracowany produkt. Znacząca ilość testów funkcjonalnych wypełnia zalecenia dotyczące przetestowania produktu przed udostępnieniem go szerszemu gronu odbiorców.

W ramach powstawania projektu wiele wymagań pozafunkcjonalnych kontrastowało ze sobą. Spełnienie ich wszystkich jest oczywiście problemem optymalizacji wielokryterialnej i nie istnieje sposób, aby wypełnić je wszystkie w stu procentach. Skoncentrowano się na zapewnieniu maksymalnej wygody użytkowania oraz wydajności, rezygnując częściowo z bezpieczeństwa i izolacji. Problem ten dokładnie opisują przeprowadzone badania w rozdziale 4.1.

W ramach pracy, oprócz wymaganej dokumentacji technicznej, przygotowano zestaw narzędzi oraz samouczków wykraczający poza założenia przewidziane w początkowej fazie projektu. Motywacją do tego działania była chęć przygotowania produktu o możliwie najmniejszej „pracy wejścia", jaką musiałby wykonać programista chcący skorzystać z oferowanej platformy. Podjęte działania mogą zaowocować szerszym zainteresowaniem ze strony potencjalnych użytkowników.

11.2 Dalsze perspektywy rozwoju platformy

Budowa platformy programistycznej jest niekończącym się procesem. W przypadku wydania projektu na licencji wolnego oprogramowania rozwój jest właściwie nieunikniony. Już podczas fazy projektowania *Nomada* powstały pomysły, rozwiązania, których implementacja znacząco wykraczała poza zakres niniejszej pracy. Najciekawsze proponowane rozszerzenia oraz dalsze perspektywy rozwoju ukazuje niniejszy podrozdział.

11.2.1 *Sandboxing* modułów

Założenia

Zaproponowano, by platforma *Nomad* oferowała mechanizm nadawania ograniczeń domenie modułów. Domyślna konfiguracja tworzyłaby domenę modułów z uprawnieniami równoważnymi uprawnieniom domeny zawierającej *NomadKernel*. Programista mógłby jednak w prosty sposób stworzyć swój *PermissionSet* i przekazać go do *NomadConfiguration*. Dzięki temu w pełni, świadomie wprowadzałby uprawnienia dla modułów w najniższym możliwym ziarnie bezpieczeństwa.

Proponowane uprawnienia:

- brak możliwości tworzenia nowych *AppDomain*

- dostęp jedynie do serwera repozytorium.
- brak intearkcji z systemem plików bądź dostęp tylko do wydzielonego jego obszaru

Implementacja

Z powodu, iż *Nomad* jest *Assembly* działającym w obu domenach aplikacji, należałoby zapewnić by posiadał on pełne uprawnienia niezależnie od ustawionych dla domeny modułów.

Dlatego *Nomad* jak i wszystkie biblioteki, z których korzysta, miałyby być dostarczane jako *Strongly named* i w ten sposób przekazane przy tworzeniu domeny ograniczonej.

Powyższe wymagania, opisane w artykule dotyczącym tworzenia *sanboxowanego* kodu [Micb], zostały zrealizowane w badawczej wersji kodu. Jednakże, aby ograniczone moduły mogły bez przeszkód korzystać z mechanizmów *Nomada*, wymagane jest by *Nomad* jako *Assembly* oznaczony został atrybutem: *AllowPartiallyTrustedCallersAttribute (APTCA)*. Zasadę jego działania opisuje artykuł [Micd].

Kolejny etap implementacji prezentowanej funkcjonalności nie mógł jednak zostać wykonany. Zaistniałą sytuację doskonale opisuje dokument [Mica]. Wynika z niego, że jeśli w łańcuchu wywołań dowolnej metody w *Assembly* oznaczonym powyższym atrybutem istnieje kod działający z ograniczonym zaufaniem, to musi ona wywoływać metody wewnątrz własnego *Assembly* bądź z innego *Assembly* koniecznie oznaczonego tym atrybutem. W przeciwnym razie następuje naruszenie mechanizmu zabezpieczeń platformy *.NET*.

Bezpieczeństwo stosowania *APTCA*

Należy mieć jednak na uwadze, że mechanizm *APTCA*, poprzez wyłączenie mechanizmów bezpieczeństwa *.NET*, pozwala na wiele nadużyć. Stąd decyzja o oznaczeniu nim swojej biblioteki, która dzięki *StronglyName* posiada w systemie pełne uprawnienia, wymaga dogłębnego audytu bezpieczeństwa kodu. Dziurawa biblioteka może w ten sposób dostarczyć złośliwemu kodowi furtki do systemu w środowisku ograniczonym. Dlatego właśnie autorzy *Castle Windsor* nie biorą na siebie takiej odpowiedzialności i nie oferują wersji produktu z atrybutem *APTCA*.

Zależność od *CastleWindsor*

Architektura *Nomada* jest oparta na kontenerze *IoC CastleWindsor*. Biblioteka ta wywoływana jest bardzo często przez wszelkie mechanizmy platformy. Komunikacja synchroniczna poprzez *ServiceLocator*, a także mechanizm sterowania cyklem życia modułów w swoich podstawach zawierają wspomniany kontener. Ładowanie modułów czy też rejestrowanie/żądanie usług przez moduły na końcu łańcucha wywołań odwołuje się właśnie do *Castle Windsora*. Zatem zawsze istnieje opisany wcześniej łańcuch.

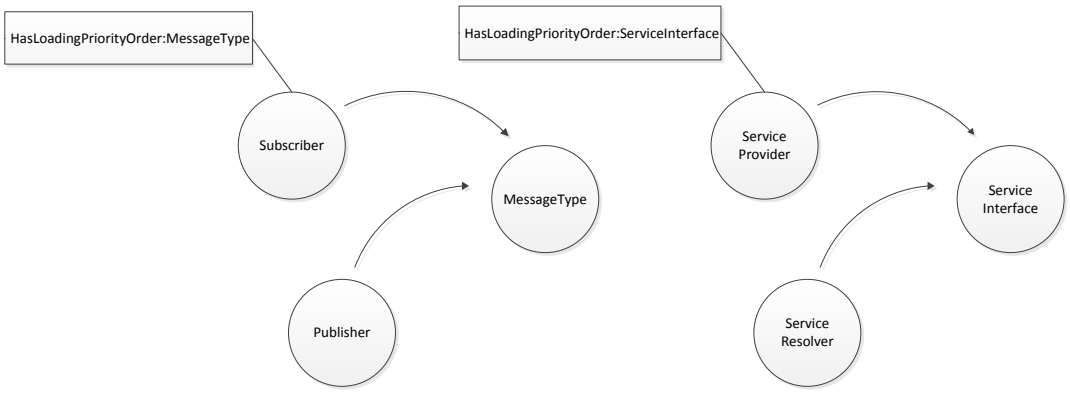
Tym samym niemożliwym jest skorzystanie z funkcjonalności *Castle Windsor* w środowisku ograniczonym i zrealizowanie mechanizmu sandboxingu domeny modułów. Kontener *IoC Castle Windsor* jest jedynym produktem wykorzystywanym przez *Nomada*, którego nie można łatwo wymienić. Realizacja funkcjonalności ograniczania uprawnień wymagałaby wymiany go i zaadaptowania rdzenia platformy do alternatywnego produktu. Z tego powodu, opisywana funkcjonalność została pozostawiona jako proponowany element rozwojowy.

Perspektywy *.NET4.0*

Warto również wspomnieć, że *Nomad*, jako oparty o *NET 3.5* cały czas korzysta z mechanizmu zabezpieczeń zdefiniowanego w wersji 2.0 platformy *.NET*. Platforma w wersji 4.0 wprowadza nowy mechanizm (tzw. *Transparency*), który może dostarczyć alternatywnego rozwiązania opisanego problemu. Jednym z wymagań postawionych autorom *Nomada* było jednak wykorzystanie *.NET* 3.5, dlatego też nie było możliwe wykorzystanie nowego mechanizmu.

11.2.2 Proponowany mechanizm rozwiązywania zależności usługowych

Poniżej przedstawiono jedną z alternatywnych koncepcji rozwiązujących problem zależności usługowych pomiędzy modułami. Kluczowym jej elementem jest atrybut klas *HasLoadingOrderPriority*. Zgodnie z rysunkiem 11.1 należy oznaczyć nim klasą główną modułu (implementującą *IModuleBootstraper*) oraz wpisać do niego listę wszystkich interfejsów komunikacyjnych, względem których jest modułem typu:



Rysunek 11.1: Atrybut priorytetu ładowania modułu.

- *Subscriber* lub
- *ServiceProvider*.

W obrębie listy modułów zawierających referencję na wybrany interfejs komunikacyjny tylko oznaczone moduły otrzymują priorytet w załadowaniu przed wszystkimi innymi. Mechanizm wymaga zatem, by programista modułu *explicite* podał w miejscu atrybutu, z których usług zamierza korzystać. W powstałej sytuacji nie istnieje jednak potrzeba odwlekania początku publikowania zdarzeń czy znajdowania dostawcy usługi do momentu załadowania wszystkich modułów. Odpowiednią kolejność ładowania zapewnia w danym rozwiązaniu *Nomad*.

Proponowana implementacja mechanizmu opierającego się atrybut *HasLoadingOrderPriority* wymaga sporego nakładu pracy. Należy m.in. uzupełnić poniższą logikę *Nomada*:

1. generator manifestów – wpisywanie listy klas w atrybucie do manifestu przy odpowiednich zależnościach
2. alternatywna implementacja interfejsu *IDependencyChecker* – rozpoznawanie listy w manifeście i posortowanie wg niej modułów.

Podstawowym problemem jest właśnie implementacja algorytmu sortującego zależności. Oczywistym wydaje się rozszerzenie algorytmu sortowania topologicznego wskazującego konieczną kolejność ładowania modułów ze względu na zależności modułowe. Wynikającą z niego listę należy posortować następnie według opisywanych *ServiceDependencies*.

Może się jednak okazać, że zależności modułowe oraz usługowe się wykluczają i nie istnieje lista modułów do załadowania spełniająca wszystkie wymagania.

W przypadku sporej liczby modułów odczuwalny staje się wówczas wzrost złożoności ładowania modułów. Sortowanie sprawdza dodatkowe warunki oraz wprowadza dodatkowe iteracje po całej liście modułów. Ciąg znaków w atrybucie opisujący klasy interfejsów komunikacyjnych nie jest w żaden sposób generowany automatycznie ani weryfikowalny przez standardowe edytory kodu platformy *.NET*. Generator manifestów musi w trakcie parsowania listy typów w atrybucie weryfikować je z referencjami modułu. Wprowadza to kolejny narzut wydajnościowy.

Realizacja powyższej koncepcji jak widać wymaga dalszych badań i testów. Stąd została pozostawiona programistom *Nomada* jako potencjalny *extension-point* (patrz dodatek B) platformy.

11.2.3 Aktywacja, deaktywacja modułów

Standardowy model platformy nie przewiduje możliwości „wyłączenia modułu" oraz ponownego włączenia. Jednak taka funkcjonalność może zostać z powodzeniem zaimplementowana przy wykorzystaniu istniejącej infrastruktury klas. Programiści mogą rozszerzyć zadania stawiane modułowi przy użyciu dziedziczenia z interfejsu *IModuleBootstraper* oraz przygotowania odpowiedniego mechanizmu uruchomianego jako serwis, który by zajmował się obsługą działań związanych z tymi czynnościami.

Mechanizm aktywacji spotykany w rozwiązaniach konkurencyjnych nie został zaimplementowany w *Nomadzie*, ponieważ nie byłby on zbudowany na najniższej warstwie platformy. Do działania nie potrzebuje wsparcia mechanizmów ładowania czy dostępu do *Nomad.Core*. W środowisku *.NET* nie ma innej możliwości deaktywacji modułu niż poinformowanie modułu, że ma się wyłączyć. Nadal jednak pozostanie on załadowany w pamięci. Reakcja na informacje o wyładowaniu modułu zależna będzie od implementacji dostarczonej przez twórcę modułu.

Z pojęciem deaktywacji wiąże się usunięcie przygotowanego przez moduł *View* z graficznego interfejsu użytkownika. Istnieje jednak wiele możliwości implementacji takiego mechanizmu – wyłączenie kontrolki, ukrycie, zniszczenie. Mechanizm ten musiałby współgrać z każdą szatą graficzną co przy dowolności budowy aplikacji w technologi *WPF* jest niemożliwe. Dlatego właśnie ten element *Nomada* został świadomie pozostawiony programiście.

11.2.4 Definiowanie usług przez konfigurację

W obecnej wersji platforma posiada możliwość dodania usług powiązanych z rdzeniem platformy przez dziedziczenie z klasy *NomadKernel* i przeciążenie odpowiednich metod. Rozwiązanie to jest skuteczne, ale niestety nie jest wygodne.

Istnieje możliwość rozszerzenia mechanizmu usług o klasę *installera*, czyli ogólnego interfejsu pozwalającego programiście definiować aktywne usługi w ramach konfiguracji. W takim przypadku jedyne, co musiałby wykonać programista, to przygotowanie odpowiedniej klasy implementującej interfejs *installera* i umieszczenie jej w ramach konfiguracji.

W ramach rozwoju platformy należało by przebudować nieznacznie klasy związane z *Nomad.Core* aby umożliwić maksymalnie swobodne definiowanie usług.

11.3 Publikacja na zasadach wolnego oprogramowania

Jednym z najważniejszych aspektów jakie wytyczono projektowi, było wydanie opracowanej platformy na licencji wolnego oprogramowania. Całość rygorystycznych mechanizmów budowy projektów opisanych w rozdziale 9 była opracowana z myślą o wydaniu produktu. Planowane jest wydanie platformy *Nomad* na licencji *New BSD* w wersji trzeciej.

Spis rysunków

2.1 Domena aplikacji wewnątrz procesu systemu operacyjnego [Ric10] 12
2.2 Działanie mechanizmu marshaling [Mic03a] 13

3.1 Architektura widoczna dla końcowego użytkownika platformy *Nomad* 16
3.2 Architektura najwyższego poziomu platformy *Nomad* 18

4.1 DAG obrazujący przykładowy zestaw zależności 24
4.2 Realizacja podpisów *Assembly* w .NET [Tro07] 26

5.1 Graficzne przedstawienie koncepcji regionu 30

6.1 Poglądowy rysunek publikacji i subskrypcji komunikatów 38
6.2 Diagram klas prezentujących standardowy EventAggregator oraz ForwardingEventAggregator 40
6.3 Diagram klas przedstawiający poglądowo *EventAggregator* 42

8.1 Diagram klas lokalizacyjnych 52

10.1 Widok głównej strony przygotowanego serwera 61
10.2 Interfejs przygotowanego modułu graficznego aktualizatora 61
10.3 Interfejs przygotowanej aplikacji po polsku 63
10.4 Interfejs przygotowanej aplikacji po angielsku 63

11.1 Atrybut priorytetu ładowania modułu. 67

Spis tablic

4.1 Wyniki testów wydajnościowych 21

Dodatek A

Wykaz użytych technologii

A.1 Biblioteki użyte do budowy platformy

Poniższe biblioteki są niezbędne do działania *Nomada*:

- Castle Windsor (wersja 2.5.0.2108) – `http://www.castleproject.org/container/`
- DotNetZip (wersja 1.9.1.5) – `http://dotnetzip.codeplex.com/`
- NLog (wersja 2.0.0.0) – `http://nlog-project.org/`

A.2 Narzędzia wspomagające testowanie

Poniżej znajduje się wykaz użytych bibliotek oraz narzędzi wspomagających testowanie:

- NUnit (wersja 2.5.5.10112) – `http://www.nunit.org/`
- Moq (wersja 4.0.0.0) – `http://code.google.com/p/moq/`
- Project White (wersja 0.2.00) –`http://white.codeplex.com/`
- WipFlash (wersja 0.0.0.1) – `http://code.google.com/p/wipflash/`
- dotCover (wersja 1.0.120) – `http://www.jetbrains.com/dotcover/`

A.3 Narzędzia wspomagające zarządzanie projektem

Poniżej przedstawiona jest lista wykorzystanych narzędzi:

- Psake – `https://github.com/JamesKovacs/psake`
- Hudson – `http://hudson-ci.org/`
- SandCastle – `http://sandcastle.codeplex.com/`
- git – `http://git-scm.com/`

Dodatek B

Wykaz haseł

Poniżej zaprezentowano skróty oraz hasła często pojawiające się w pracy.

- domena aplikacji (z ang. AppDomain) – jest to logiczny kontener przeznaczony na *Assembly* występujący w ramach procesu systemu operacyjnego [Ric10]. Również klasa *.NET* umożliwiająca kontrolę nad tym kontenerem,
  `http://msdn.microsoft.com/en-us/library/w124b5fa.aspx`

- Assembly – w środowisku *.NET* określa kolekcję typów i zasobów, które tworzą logiczną jednostkę funkcjonalności. Ma postać kodu pośredniego zaopatrzonego w manifest oraz metadane o typach. Zazwyczaj przybiera formę pliku o rozszerzeniu *dll* albo *exe*.
  `http://msdn.microsoft.com/en-us/library/ms973231.aspx`

- AutoResetEvent – mechanizm synchronizacji wbudowany w platformę *.NET*. Posiada działanie podobne do semafora binarnego.
  `http://msdn.microsoft.com/en-us/library/system.threading.autoresetevent.aspx`

- WCF (z ang. Windows Communication Foundation) – API środowiska *.NET* wprowadzone w wersji 3.0, służące do budowy aplikacji „zorientowanych sieciowo". Jest to API udostępniające wysokopoziomowe mechanizmy komunikacji przez co przyjazne programiście.
  `http://msdn.microsoft.com/en-us/library/ms735119.aspx`

- RMI (z ang. Remote Method Invocation) – mechanizm umożliwiający zdalne wywoływanie metod na obiektach, wprowadzony w technologii *Java*. Stanowi rodzaj obiektowych mechanizmów *RPC* (Remote Procedure Calls).
  `http://download.oracle.com/javase/6/docs/api/java/rmi/package-summary.html`

- CLR (z ang. Common Language Runtime) – środowisko uruchomieniowe *.NET Framework*.
  `http://msdn.microsoft.com/pl-pl/netframework/cc511286`

- GAC (z ang. Global Assembly Cache) – mechanizm w systemie Windows, którego zadaniem jest umożliwienie udostępniania bibliotek pomiędzy wieloma aplikacjami opartymi na platformie *.NET*. Umieszczając *Assembly* w GAC uzyskuje się efekt współdzielonej biblioteki dostępnej z poziomu środowiska *.NET*, bez potrzeby umieszczania jej w ścieżce przeszukiwania.
  `http://msdn.microsoft.com/en-us/library/yf1d93sz.aspx`

- Punkt rozszerzenia (z ang. Extension point) – Fragment kodu celowo pozostawiony przez programistę autora w celu późniejszego rozszerzania możliwości napisanego wcześniej kodu.

- COM (z ang. Component Object Model) – technologia firmy Microsoft umożliwiająca elementom aplikacji na komunikację. Powstała na początku lat 90-tych. Komponenty działają na zasadzie serwer – klient i komunikują się poprzez binarne interfejsy zestawione przez system operacyjny.
  `http://www.microsoft.com/com/default.mspx`

- DAG (z ang. directed acyclic graph) – skierowany acykliczny graf. Jest to graf skierowany, który nie posiada skierowanych cykli.

- Framework – szkielet do budowy aplikacji, zazwyczaj w postaci bibliotek. `http://pl.wikipedia.org/wiki/Framework`
- IPC (z ang. Inter Process Communication) – ogólna nazwa na zestaw mechanizmów systemu operacyjnego, wspierających komunikację pomiędzy procesami. W systemie Microsoft Windows do tej grupy należą takie technologie i narzędzia jak: schowek, COM, gniazda, RPC, nazwane potoki itd.
`http://msdn.microsoft.com/en-us/library/aa365574.aspx`
- wątek środowiska *.NET* – jest to wątek, którego zarządzaniem zajmuje się maszyna wirtualna (czyli CLR w przypadku *.NET*). Tego rodzaju wątki emulują wielozadaniowość środowiska abstrahując od implementacji sprzętowej czy systemu operacyjnego. W przypadku systemu firmy Microsoft nieznany jest sposób implementacji mechanizmu odwzorowania wątków zarządzalnych na wątki natywne systemu operacyjnego.
- Shell – moduł aplikacji hosta z głównym szkieletem interfejsu WPF, definiujący regiony dostępne dla modułów.
- ThinHost – aplikacja, w ramach której działa *Nomad* z domeną jądra.
- XML (z ang. Extensible Markup Language) – język znaczników, przeznaczony do reprezentacji danych w sposób ustrukturalizowany.
`http://www.w3.org/XML/`
- IoC (z ang. Inversion of Conrol) – odwrócenie sterowania. Koncepcja w programowaniu, kiedy to kod niższego poziomu wywołuje kod wyższego poziomu.
- Demon – to proces, który pracuje w tle bez interakcji z użytkownikiem `http://pl.wikipedia.org/wiki/Demon_(informatyka)`
- Dispatcher – utrzymuje priorytetową kolejkę zadań (ang. work items), wykonując je sekwencyjnie w określonym wątku. W szczególności wykorzystywany jest *dispatcher* wątku gui
`http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher.aspx`
- ThreadPool – to wzorzec, który zapewnia wykonywanie zadań na zasadzie kolejki zadań – pierwszy wolny wątek pobiera zadanie z kolejki do wykonania [Gra02], ale także odpowiednia klasa implementująca ten wzorzec w platformie .NET. `http://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx`
- PHP (z ang. PHP: Hypertext Preprocessor) – jest to obiektowy, skryptowy język programowania zaprojektowany do generowania stron internetowych.
`http://www.php.net/`
- ZIP – format kompresji i archiwizacji danych.
`http://www.pkware.com/documents/casestudies/APPNOTE.TXT`
- WPF (z ang. Windows Presentation Foundation) – zestaw bibliotek jak i nazwa technologii firmy *Microsoft* opartych o środowisko *.NET* służących budowaniu bogatego interfejsu użytkownika.
`http://msdn.microsoft.com/en-us/library/ms754130.aspx`
- WinForms – nazwa zwyczajowa nadana podstawowemu środowisku tworzenia graficznych aplikacji użytkownika tworzonemu w środowisku *.NET*.

Dodatek C

Wykaz źródeł istniejących rozwiązań

Poniżej zaprezentowano adresy do projektów wspomnianych czy też opisanych w pracy:

- Al Platform – `http://www.agile-sys.com/productsA.php`
- CLR Addins – `http://clraddins.codeplex.com/` `http://msdn.microsoft.com/en-us/library/bb384200.aspx`
- Compact Plugs (*.NET*) – `http://compactplugs.codeplex.com/`
- MEF – `http://www.codeplex.com/MEF` `http://msdn.microsoft.com/en-us/library/dd460648.aspx`
- Mono.Addins – `http://monoaddins.codeplex.com/`
- .NET Based Add-in/Plug-in Framework with Dynamic Toolbars and Menus – `http://www.codeproject.com/KB/macros/Net_AddinProjFrmwork.aspx`
- Prism – `http://compositewpf.codeplex.com/`
- Plux.NET – `http://ase.jku.at/plux/index.html`
- OSGI (*Java*) – `www.osgi.org`

Dodatek D

Opis elementów zamieszczonych na płycie CD

Na dołączonej płycie CD znajdują się: źródła pracy dyplomowej, praca dyplomowa w wersji elektronicznej, zawartość repozytorium z serwera ciągłej integracji.

D.1 Elementy powiązane z pracą dyplomową

Źródła pracy znajdują się w */TexSource*. Elektroniczna wersja pracy – */Praca_dyplomowa.pdf*.

D.2 Elementy powiązane z badaniami

Projekt dotyczący badania nad wydajnością komunikacji – */Research/AppDomainTestSuite*
Projekt dotyczący badania nad ograniczeniami uprawnień dla modułów – */Research/NomadSandboxing*

D.3 Elementy powiązane z wydaniem Nomada

- źródła całości platformy, wraz z samouczkami oraz przykładową aplikacją – */Source/*
- dokumentacja platformy – */Documentation/*
- skompilowany kod platformy – */Release/*
- wszystkie wykorzystane biblioteki – */Libraries/*
- skrypt budujący platformę – */build.ps1*

Literatura

[Abi09] Vagif Abilov. Performance implications of cross-domain calls for unit tests. [on-line] `http://bloggingabout.net/blogs/vagif/archive/2009/03/20/performance-implications-of-cross-domain-calls-for-unit-tests.aspx`, 2009.

[Bec02] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[EFB04] Kathy Sierra Eric Freeman, Elisabeth Freeman, Bert Bates. *Head First Design Patterns*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2004.

[Fow02] Martin Fowler. Inversion of control containers and the dependency injection pattern. [on-line] `http://martinfowler.com/articles/injection.html`, 2002.

[Fow04] Martin Fowler. Event aggregator. [on-line] `http://martinfowler.com/eaaDev/EventAggregator.html`, 2004.

[Fow05] Martin Fowler. Inversion of control. [on-line] `http://martinfowler.com/bliki/InversionOfControl.html`, 2005.

[Gra02] Mark Grand. *Java Enterprise Design Patterns Vol. 3 : Patterns in Java*. John Wiley & Sons, Inc., 2002.

[JA10] Ben Albahari Joseph Albahari. *C# 4.0 in a Nutshell*. O'Reilly Media, Inc., 2010.

[KC08] Brad Adams Krzysztof Cwalina. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (2nd Edition)*. Addison-Wesley, Reading, MA, USA, 2008.

[MER09] Alex Ionescu Mark E. Russinovich, David A. Solomon. *Windows Internals, 5th Edition*. Microsoft Press, 2009.

[Mes07] Gerard Meszaros. *xUnit Test Patterns*. Addison Wesley Publishing Company, 2007.

[Mica] Microsoft. Msdn ca2116: Aptca methods should only call aptca methods. [on-line] `http://msdn.microsoft.com/en-us/library/ms182297.aspx`.

[Micb] Microsoft. Msdn how to: Run partially trusted code in a sandbox. [on-line] `http://msdn.microsoft.com/en-us/library/bb763046.aspx`.

[Micc] Microsoft. Msdn strong-named assemblies. [on-line] `http://msdn.microsoft.com/en-us/library/wd40t7ad.aspx`.

[Micd] Microsoft. Msdn using libraries from partially trusted code. [on-line] `http://msdn.microsoft.com/en-us/library/8skskf63.aspx`.

[Mic03a] Microsoft. Msdn. [on-line] `http://msdn.microsoft.com`, 2003.

[Mic03b] Microsoft. Msdn dictionary. [on-line] `http://msdn.microsoft.com/en-us/library/xfhwa508.aspx`, 2003.

[Mic09] Microsoft. Composite application guidance. [on-line] `http://compositewpf.codeplex.com/releases/view/14982#DownloadId=59928`, 2009.

[Net02] Oracle. Sun Developer Network. Core j2ee patterns - service locator. [on-line] `http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html`, 2002.

[NM10] Net Applications INC. Net MarketShare. Operating system market share. [on-line] `http://marketshare.hitslink.com/os-market-share.aspx?qprid=9`, 2010.

[RCM08] Micah Martin Robert C. Martin. *Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#*. Helion, 2008.

[Ric10] Jeffrey Richter. *CLR via C#, 3rd eidition*. Microsoft Press, 2010.

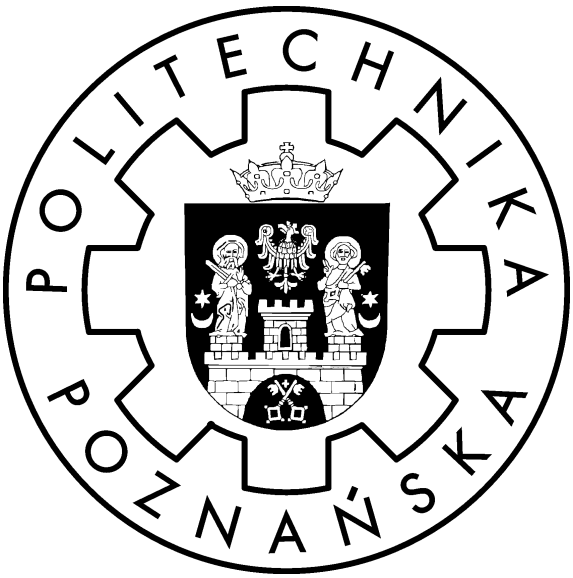
[SF09] Nat Pryce Steve Freeman. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, wydanie 1, 2009.

[Sup04] Microsoft Support. How to build an office 2000 com add-in in visual basic. [on-line] `http://support.microsoft.com/kb/238228/en-us/`, 2004.

[THC01] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. *Wprowadzenie do algorytmów, wydanie czwarte*. Wydawnictwo Naukowo-Techniczne, 2001.

[Tro07] Andrew Troelsen. Pro c# with .net 3.0, special edition, 2007.

[Wol10] Reinhard Wolfinger. *Dynamic Application Composition with Plux.NET*. Praca doktorska, Uniwersytet Johannesa Keplera, Linz Austria, 2010.


POLITECHNIKA
POZNAŃSKA