



**POLITECHNIKA
GDAŃSKA**

WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI I INFORMATYKI



Imię i nazwisko studenta: Wojciech Zielonka

Nr albumu: 152682

Imię i nazwisko studenta: Filip Bąkowski

Nr albumu: 119272

Studia pierwszego stopnia

Forma studiów: niestacjonarne

Kierunek studiów: Informatyka

PRACA DYPLOMOWA INŻYNIERSKA

Tytuł pracy w języku polskim:

System mobilny i sieciowy do wspomagania rozgrywki w grach zespołowych typu ASG oraz Paintball

Tytuł pracy w języku angielskim:

Mobile and Web based system supporting team game like ASG and Paintball

Potwierdzenie przyjęcia pracy	
Opiekun pracy	Kierownik Katedry/Zakładu
<i>podpis</i>	<i>podpis</i>
dr inż. Krzysztof Bruniecki	prof. dr hab. inż. Bogdan Wiszniewski

Data oddania pracy do dziekanatu:



**POLITECHNIKA
GDAŃSKA**

WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI I INFORMATYKI



OŚWIADCZENIE

Ja, niżej podpisany(a), wyrażam zgodę na korzystanie z mojej pracy dyplomowej zatytułowanej:
System mobilny i sieciowy do wspomagania rozgrywki w grach zespołowych typu ASG oraz
Paintball do celów naukowych lub dydaktycznych.¹

Gdańsk, dnia

.....
podpis studenta

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r., nr 90, poz. 631) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),² a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza(y) praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia

.....
podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczaniem jej autorstwa.

Gdańsk, dnia

.....
podpis studenta

¹ Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.

² Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym: Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego. Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.



**POLITECHNIKA
GDAŃSKA**

WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI I INFORMATYKI



OŚWIADCZENIE

Ja, niżej podpisany(a), wyrażam zgodę na korzystanie z mojej pracy dyplomowej zatytułowanej:
System mobilny i sieciowy do wspomagania rozgrywki w grach zespołowych typu ASG oraz
Paintball do celów naukowych lub dydaktycznych.¹

Gdańsk, dnia

.....
podpis studenta

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r., nr 90, poz. 631) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),² a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza(y) praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia

.....
podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczaniem jej autorstwa.

Gdańsk, dnia

.....
podpis studenta

¹ Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.

² Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym: Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego. Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.

Spis treści

Wstęp.....	4
1. Programowanie równoległe oraz asynchroniczne w języku C# [Zielonka].....	5
1.1. Klasa Task jako podstawowa jednostka programowania wielowątkowego.....	5
1.2. Programowanie asynchroniczne.....	6
1.3. Programowanie równoległe	11
1.3.1. Techniki podziału danych na podzbiory	14
2. Charakterystyka wybranych wzorców projektowych [Zielonka]	16
2.1. Wzorzec Pipeline.....	16
2.2. Wzorzec Dataflow	17
2.3. Wzorzec WaitAllOneByOne.....	18
2.4. Wzorzec Producer-Consumer	19
2.5. Wzorzec MapReduce	20
3. Interoperacyjność na styku języków Java i C# w kontekście serializacji [Bąkowski].....	22
3.1. Serializacje do postaci strumienia bajtów w języku Java i serializacja binarna w C#.	22
3.2. Serializacja XML	22
3.2.1. Serializacja w C# z użyciem klasy XmlSerializer	22
3.2.2. Serializacja w Java	24
3.2.3. Biblioteki służące do serializacji XML dostępne zarówno w językach JAVA i C#	29
4. Realizacja projektu.....	30
4.1. Schemat i opis głównych komponentów projektu	31
4.2. Biblioteka serwerowa	31
4.3. Aplikacja serwerowa.....	34
4.4. Aplikacja kliencka do testowania	35
4.5. Mobilna aplikacja kliencka	36
Zakończenie	41
Załączniki	42
Kody źródłowe	42
Wykaz literatury	44
Wykaz rysunków.....	45
Wykaz tabel.....	46
Wykaz listingów	46

Wstęp

Celem niniejszej pracy było stworzenie systemu, który wspomaga graczy w różnego rodzaju wieloosobowych grach terenowych, takich jak np. *Paintball*, w łatwiejszym lokalizowaniu członków drużyny w terenie podczas rozgrywki. Pomysł projektu jest wzorowany na amerykańskim wojskowym systemie *Blue Force Tracking*, którego zadaniem jest wspomaganie walczących poprzez wyświetlanie w czasie rzeczywistym aktualnych pozycji wszystkich sprzymierzonych jednostek na polu walki, co pomaga w znaczącym stopniu zwiększyć możliwość koordynacji oraz kooperacji wojsk. Do realizacji celu projektu zostały wykorzystane technologie takie jak: GPS (ang. *Global Positioning System*), system Android oraz język programowania C#. Projekt składa się z 2 głównych części, aplikacji serwerowej oraz aplikacji mobilnej. Powodem dla którego zostało wybrane tego typu zagadnienie była chęć stworzenia aplikacji bazującej na własnym protokole komunikacyjnym pomiędzy różnymi systemami. Interoperacyjność została oparta na warstwie komunikacyjnej wykorzystującej XML (ang. *Extensible Markup Language*) oraz na protokole komunikacyjnym TCP (ang. *Transmission Control Protocol*). Dzięki opracowaniu własnych zapytań rozumianych przez serwer jesteśmy w stanie komunikować się pomiędzy różnymi systemami, w naszym przypadku jest to system Windows i Android. Kolejnym ważnym zagadnieniem, któremu zostały poświęcone 2 rozdziały, są elementy dotyczące programowania równoległego i asynchronicznego oraz rozdział opisujący wybrane wzorce projektowe. Wielowątkowość jest szczególnie ważna z powodu potencjalnie wielu użytkowników jednocześnie używających aplikacji, co przekłada się na obsługę wielu równoczesnych zapytań przez serwer. W dzisiejszych czasach możliwość przetwarzania dużych ilości danych równocześnie jest szczególnie ważna dla zadowolenia użytkowników oraz szybkości działania aplikacji, dlatego w pracy szczególny nacisk został położony na zagadnienia bezpośrednio związane z równoległym przetwarzaniem informacji.

1. Programowanie równoległe oraz asynchroniczne w języku C# [Zielonka]

Programowanie asynchroniczne oraz równoległe jest jednym z najważniejszych aspektów współczesnego sposobu projektowania aplikacji desktopowych, a co najbardziej istotne aplikacji serwerowych. Obecnie, każdy procesor składa się w rzeczywistości z wielu procesorów zwanych rdzeniami co zapewnia w teorii szybsze działanie aplikacji. Jednakże, aby uzyskać szybsze działanie należy wykorzystać moc obliczeniową procesora poprzez użycie bibliotek pozwalających na poprawne zarządzanie wątkami procesorów. Jest to szczególnie ważny problem w aplikacjach serwerowych, gdzie skalowalność aplikacji jest rzeczą niezbędną do komercyjnego zastosowania dla wielu użytkowników. W niniejszym rozdziale zostaną przedstawione aspekty programowania równoległego oraz asynchronicznego w języku C#.

1.1. Klasa `Task` jako podstawowa jednostka programowania wielowątkowego.

Klasa `Task`, która znajduje się w przestrzeni nazw `System.Threading.Tasks.Task` jest podstawową jednostką wątku w środowisku .NET. Pierwotnie do harmonogramowania wątku służyła metoda `Task.Factory.StartNew()` jednakże od standardu 4.5 została wprowadzona metoda `Task.Run()` która jest pewnego rodzaju skrótem¹. Obydwie metody przyjmują jako parameter `delegate`², dzięki czemu możemy przekazać argument jako funkcję anonimową (ang. *lambda expression*), która zostanie wykonana w obrębie nowego wątku.

Listing 1.1. Tworzenie nowego wątku za pomocą metody statycznej `Factory`.

```
Task task = Task.Factory.StartNew(() => {  
    Console.WriteLine("New thread");  
});
```

W powyższym przypadku nowy wątek zostanie najpierw stworzony, a następnie wprowadzony do zbioru wątków oczekujących na wybranie przez dyspozytora wątków. Inną możliwością jest stworzenie wątku samodzielnie, to znaczy bez mechanizmu fabryki.

Listing 1.2. Tworzenie nowego wątku za pomocą konstruktora klasy `Task`.

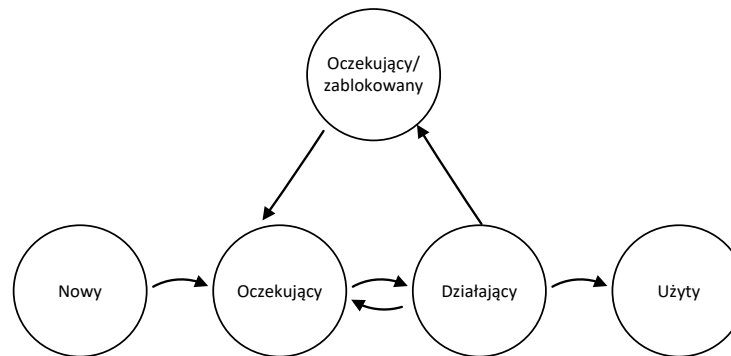
```
Task task = new Task(() => {  
    Console.WriteLine("New thread");  
});  
task.Start();
```

¹ Źródło internetowe: <https://blogs.msdn.microsoft.com/pfxteam/2011/10/24/task-run-vs-task-factory-startnew/>

² Delegata jest wskaźnikiem na funkcję. Możemy za jej pomocą przekazywać wyrażenia lambda lub zwyczajne metody zadeklarowane w ciele klasy.

Jednakże w tym przypadku należy wystartować nowy wątek *explicite*, w innym razie nie będzie on dostępny dla dyspozytora. Poniższy diagram przedstawia etapy życia wątków.

Rys. 1.1. Diagram stanów wątków.



Źródło: Opracowanie własne na podstawie: SCJP Sun Certified Programmer for Java 6 Study Guide.

Stan nowy oznacza, że wątek został stworzony jako nowa instancja, bez wywołania metody `Start()`. Na tym etapie wątek jest traktowany jako nieżywy (ang. *not alive*). Stan oczekujący jest stanem po wywołaniu metody `Start()`, w tym momencie wątek jest dostępny dla dyspozytora jako żywy (ang. *alive*). Wszystkie instancje klasy `Task`, na których została wywołana metoda `Start()` znajdują się w zbiorze wątków oczekujących (ang. *thread pool*). W momencie kiedy dyspozytor wybierze ze zbioru wątek oczekujący, zmienia on status na działający (ang. *running*). Na tym etapie wątek może przejść w dwa stany, - oczekujący lub użyty (ang. *dead*). Stan oczekujący/zablokowany jest w momencie kiedy wątek natrafi na blok synchronizujący, który np. zabezpiecza dane przed dostępem kilku wątków równocześnie. W języku C# taki blok oznacza się poprzez użycie słowa kluczowego `lock(){ }`³. Tego typu rozwiązania nazywane są blokowymi, ponieważ blokujemy wątki przed dalszą egzekucją w oczekiwaniu na dostęp do bloku synchronizującego. Wątek zablokowany, wraca do zbioru wątków oczekujących na wybranie przez dyspozytora. Stan użyty jest wówczas kiedy wątek skończy swoje zadanie. Wątek, który raz został użyty nie może zostać ponownie wystartowany, ponieważ zostanie rzucony wyjątek `System.InvalidOperationException`.

1.2. Programowanie asynchroniczne

Głównym podejściem w programowaniu asynchronicznym jest pisanie tych funkcji, które potencjalnie będą się długo wykonywały, w sposób asynchroniczny. Poprzednią koncepcją było tworzenie osobnych wątków, w których były wykonywane takie funkcje np. pobieranie dużej ilości danych z sieci. Podejście asynchroniczne zwiększa skalowalność aplikacji oraz jej wydajność. Przykładem jest aplikacja desktopowa (ang. *rich-client application*). Mając graf metod wywoływanych

³ Albahari J., Albahari B.: *C# 6.0 in a Nutshell*, O'Reilly Media, Sebastopol, 2015.

jedna po drugiej (ang. *call graph*) w tradycyjnym programowaniu synchronicznym, jeżeli jakaś operacja będzie wykonywała się przez długi czas, musimy wywołać cały graf na osobnym wątku, ponieważ w innym przypadku główny wątek aplikacji, który obsługuje cały interfejs użytkownika, przestanie odpowiadać na czas wykonywania metod z grafu. Dodatkowo każda z tych metod musi być bezpieczna w kwestii wielowątkowości. W przypadku asynchronicznego wywołania grafu metod, nie musimy tworzyć nowego wątku dopóki nie jest on potrzebny, zazwyczaj tylko dla metody wykonującej operacje odczytu/zapisu. Pozostałe metody mogą być wywołane w wątku głównym. Jest to ważne chociażby z tego względu, że tylko metody które są wykonywane w głównym wątku mają dostęp do obiektów interfejsu graficznego w tym różnego rodzaju kontrolek itd.⁴

Klasa `Task` idealnie nadaje się do programowania asynchronicznego, ponieważ wspiera kontynuację⁵ co jest esencją asynchroniczności. Poniżej znajduje się przykład programu asynchronicznego⁶:

Konsola 1.1. Wynik działania programu asynchronicznego.

```
Aplikacja konsolowa 1.1.

Wątek główny : 1
Początek metody nieblokującej Start()
Metoda DisplayPrimeCountsFrom()
Metoda GetPrimesCountAsync()
Metoda OtherOperations(). ID wątku 1
!----> Koniec metody Main() <----!
ID wątku 3. Rezultat: 78498

Metoda DisplayPrimeCountsFrom()
Metoda GetPrimesCountAsync()
ID wątku 3. Rezultat: 76122

Metoda DisplayPrimeCountsFrom()
Metoda GetPrimesCountAsync()
ID wątku 3. Rezultat: 74954

Metoda DisplayPrimeCountsFrom()
Metoda GetPrimesCountAsync()
ID wątku 7. Rezultat: 74024

Koniec metody Start()
```

Źródło: Opracowanie własne. Kod źródłowy w załączniku.

Metoda `GetPrimesCountAsync(int s, int x)` jest symulacją zajęcia, które pochłania większą ilość czasu. Zwraca liczbę liczb pierwszych w danym przedziale, dodatkowo struktura `int` jest opakowana w typ generyczny `Task<int>`, dlatego po wykonaniu tego zadania należy wywołać metodę

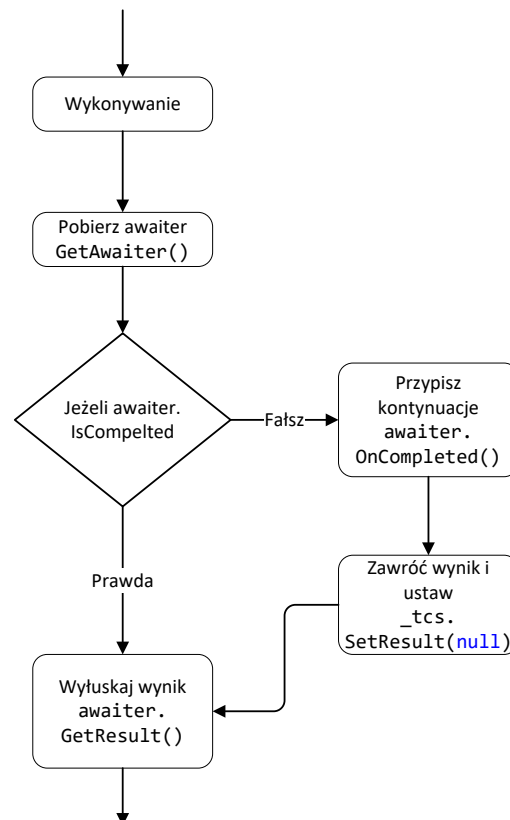
⁴ Albahari J., Albahari B.: *C# 6.0 in a Nutshell*, O'Reilly Media, Sebastopol, 2015, s. 590-591.

⁵ Chodzi tutaj o metodę w klasie `Task ContinueWith()`, która kontynuuje wątek po tym jak zakończyła się egzekucja instancji, na której została wywołana ta metoda.

⁶ Op. cit, s. 592.

GetResult() w celu wyłuskania wyniku operacji. Obiekt `Task<int>` zwracany przez metodę jest uchwytym do zadania wykonywanego w innym wątku. Metoda `DisplayPrimeCountsFrom(int i)` wywołuje się rekurencyjnie do momentu osiągnięcia warunku stopu. Następnie protercja `Task` zostaje zaktualizowana `_tcs.SetResult(null)` tak, aby obiekt zwracany w metodzie `Task DisplayPrimeCountsAsync()` mógł się wykonać asynchronicznie. Klasa `TaskCompletionSource` jest wykorzystywana do przekazania wyniku operacji oraz do informacji na temat wątku wykonywanego asynchronicznie dla kontynuacji. Dodatkowo jeżeli tworzymy aplikację okienkową i korzystamy z kontrolek chcąc wyświetlić wynik operacji, należy pamiętać o wskazaniu na wątek główny jako wątek, na którym chcemy kontynuować. W tym celu należy przekazać jako parametr w metodzie kontynuacyjnej metodę `TaskScheduler.FromCurrentSynchronizationContext()`.

Rys. 1.2. Przepływ wzorca projektowego oczekujący.



Źródło: Opracowanie własne na podstawie: Skeet J.: *C# in depth*, Manning, Shelter Island NY, 2014.

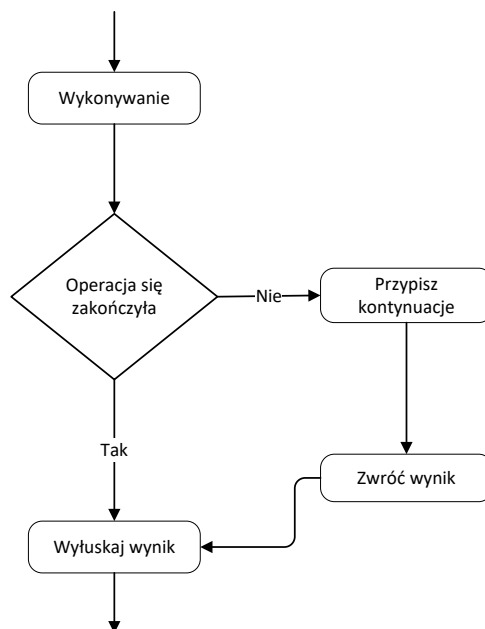
Przykład bez wsparcia od strony środowiska jest skomplikowany i nieczytelny, co sprzyja powstawaniu błędów. Dlatego inżynierowie języka C# wprowadzili mechanizm, który automatycznie zarządza kodem asynchronicznym. Mechanizm ten opiera się na słowach kluczowych `async` i `await`. Jeżeli przerobimy powyższy kod, używając tego mechanizmu, otrzymamy tylko jedną metodę, której postać będzie wyglądała w następujący sposób:

Listing 1.3. Przykład metody asynchronicznej.

```
async Task DisplayPrimeCountsAsync() {  
    for (int i = 0; i < 3; i++) {  
        int x = await GetPrimesCountAsync(i * 100000 + 2, 1000000);  
    }  
}
```

Po modyfikacjach kod znacznie się uprościł. Słowo kluczowe `await` w sposób automatyczny wyluskuje obiekt lub strukturę zwracaną w uchwycie do wątku. Ważnym aspektem jest fakt, iż `await` nie blokuje przepływu. Kompilator automatycznie tworzy kontynuację metody na wzór `OnCompleted()`. Natomiast słowo `async` w sposób explicite oznacza, że metoda jest asynchroniczna i zwraca `Task`, `Task<TResult>` lub jest typu `void`. Jest również informacją dla kompilatora, żeby traktował `await` jako słowo kluczowe, a nie jako zmienną⁷. Kompilator w sposób automatyczny zwraca obiekt `Task` z metody oznaczonej słowem `async`, pomimo że nie został jawnie zadeklarowany blok `return` tak jak było to w przykładzie bez użycia wsparcia języka C#. Dodatkowo kompilator samoczynnie przechwytuje wątek synchronizacyjny `SynchronizationContext.Current`, na którym będzie kontynuował asynchroniczne wywołanie.

Rys. 1.3. Przepływu metody asynchronicznej przy użyciu `async` i `await`.



Źródło: Opracowanie własne na podstawie: Skeet J.: *C# in depth*, Manning, Shelter Island NY, 2014

Na powyższym przykładzie, używając gotowych mechanizmów można w łatwy sposób tworzyć kod asynchroniczny. Słowo kluczowe `await` jest zamieniane przez kompilator na blok:

⁷ Skeet J.: *C# in depth*, Manning, Shelter Island NY, 2014.

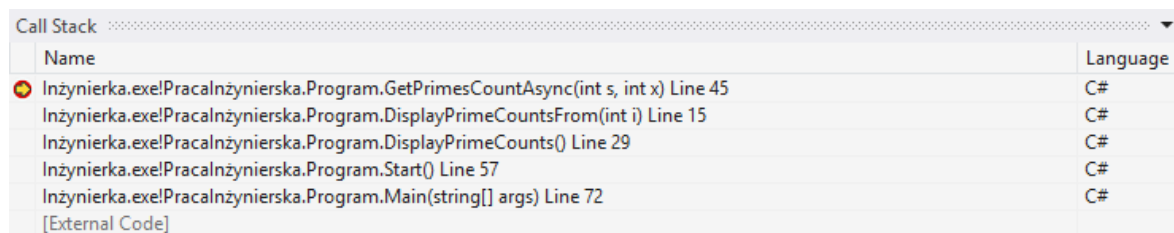
Listing 1.4. Przykład kodu bez wsparcia słów kluczowych `async` / `await`.

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted(() =>
{
    var result = awaiter.GetResult();
    statement(s);
});
```

Dzięki czemu programista nie musi samodzielnie pisać maszyny stanów, tak jak w przykładzie kodu asynchronicznego oraz nie musi deklarować wątku, który będzie wykonywał się zaraz po wątku pobocznym. Przypisując jawną kontynuację łączymy wątek, który wykonywał pewne zadanie (ang. *worker thread*) z wątkiem głównym np. zarządzającym interfejsem użytkownika (ang. *UI thread*) w ten sposób wracając do wątku głównego. Od wersji 4.5 środowiska .NET kompilator automatycznie zarządza zarówno kontynuacją wątków w prawidłowym porządku jak i poprawną obsługą wyjątków rzucanych w wątkach roboczych.

Wracając do przykładu aplikacji konsolowej (*Konsola 1.1.*), w metodzie głównej `Main()` na stos funkcji wkładamy na samym początku metodę asynchroniczną `async void Start()`. Następnie, w ciele tej metody wywoływane są kolejne metody, zgodnie z poniższym stosem:

Rys. 1.4. Stos funkcji po wywołaniu metody asynchronicznej `Start()`.

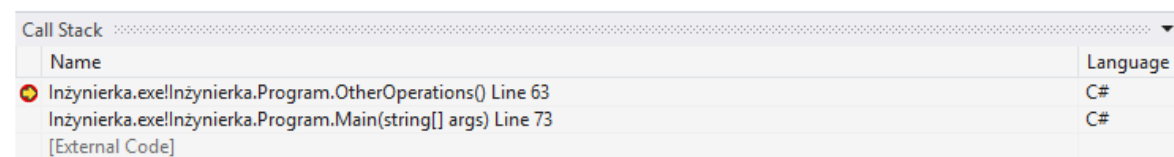


Name	Language
Inżynierka.exe!PracalInzynierska.Program.GetPrimesCountAsync(int s, int x) Line 45	C#
Inżynierka.exe!PracalInzynierska.Program.DisplayPrimeCountsFrom(int i) Line 15	C#
Inżynierka.exe!PracalInzynierska.Program.DisplayPrimeCounts() Line 29	C#
Inżynierka.exe!PracalInzynierska.Program.Start() Line 57	C#
Inżynierka.exe!PracalInzynierska.Program.Main(string[] args) Line 72	C#
[External Code]	

Źródło: Opracowanie własne.

Zgodnie z powyższym obrazkiem oraz z wynikiem działania aplikacji konsolowej, tworzony jest nowy wątek kontynuacyjny w metodzie `DisplayPrimeCountsFrom(int i)`, który wyświetla rezultat operacji wykonanych w wątku stworzonym w metodzie `GetPrimesCountAsync(int s, int x)`. Jednakże najważniejszym aspektem w programowaniu asynchronicznym jest fakt o nieblokowaniu przepływu programu. W metodzie `Main()` po wywołaniu `Start()` na stos funkcji jest wkładana kolejna metoda `OtherOperations()`, która jest przykładem nieblokującego przepływu.

Rys. 1.5. Stos funkcji po wywołaniu metody `OtherOperations()`.



Name	Language
Inżynierka.exe!Inzynierka.Program.OtherOperations() Line 63	C#
Inżynierka.exe!Inzynierka.Program.Main(string[] args) Line 73	C#
[External Code]	

Źródło: Opracowanie własne.

Jak widzimy w głównym wątku jest wykonywana kolejna metoda, która nie jest zablokowana przez metodę asynchroniczną `Start()`. Po wykonaniu nowej metody, stos funkcji wygląda następująco:

Rys. 1.6. Stos funkcji po wykonaniu metody `OtherOperations()`.

Name	Language
Inżynierka.exe\Inżynierka.Program.GetPrimesCountAsync(int s, int x) Line 46	C#
Inżynierka.exe\Inżynierka.Program.DisplayPrimeCountsAsync() Line 39	C#
[Resuming Async Method]	
[External Code]	
[Async Call]	
Inżynierka.exe\Inżynierka.Program.Start() Line 57	C#

Źródło: Opracowanie własne.

Na powyższym obrazie, program powrócił do metody `Start()`, aby kontynuować asynchroniczny przepływ programu. W podejściu programowania asynchronicznego, nowe wątki są tworzone dopiero w momencie wywołania metod asynchronicznych, nie ma potrzeby wywoływania metody `Start()` od samego początku w nowym wątku. Dodatkowo należy pamiętać, że metoda `DisplayPrimeCountsFrom(int i)`, która jest wywoływana rekurencyjnie, będzie zawsze znajdowała się w nowym wątku. Metoda `OnCompleted()` tworzy za każdym razem nowy wątek, a w jego kontekście jest wywoływana rekurencja.

1.3. Programowanie równoległe

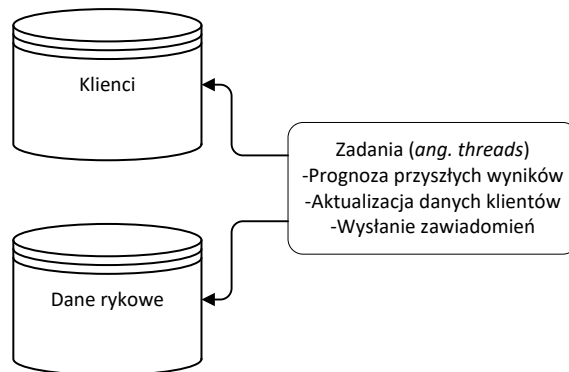
Programowanie, które wykorzystuje procesory wielordzeniowe, jest nazywane programowaniem równoległym. Jest to jedna z wielu dziedzin zajmujących się wielowątkowością.⁸ Wyróżniamy dwie główne koncepcje w programowaniu równoległym:

- Równoległość zadań (ang. *task parallelism*),
- Równoległość danych (ang. *data parallelism*).

Równoległość danych polega na podzieleniu zbioru danych na poszczególne wątki. Innymi słowy na wykonaniu np. operacji pierwiastkowania na każdej komórce tablicy dwuwymiarowej. Natomiast równoległość zadań polega na podzieleniu zadań i wykonaniu ich na zbiorze danych. Zadanie należy utożsamiać z wątkiem, który musi wykonać operację na pewnych danych. Poniższy diagram pokazuje przykład takich zadań.

⁸ Albahari J., *Threading in C#*, O'Reilly Media, 2006-2010 (www.albahari.com/threading).

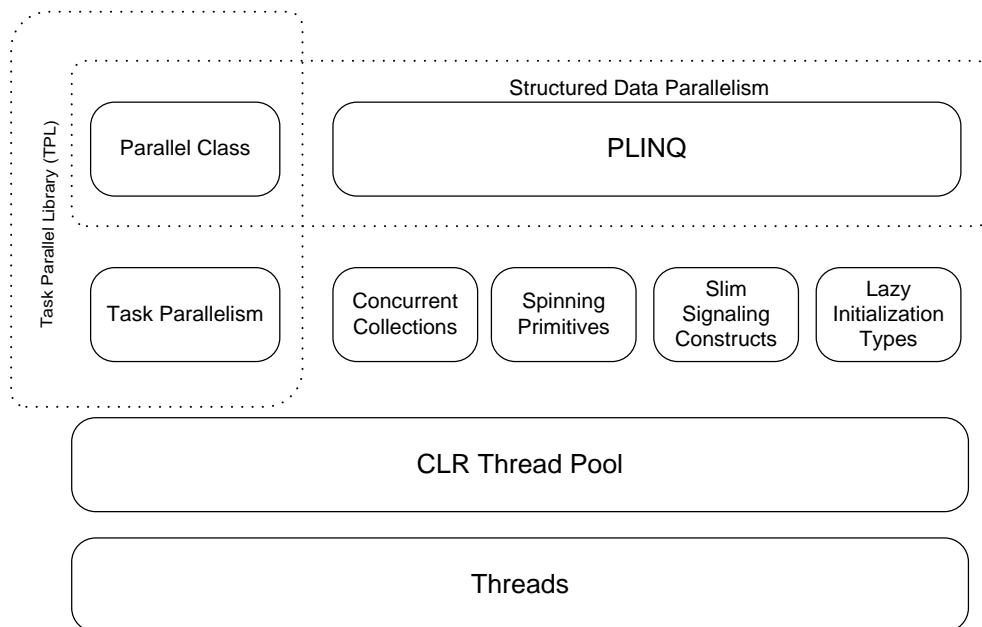
Rys. 1.7. Równoległość zadań w przypadku dwóch zbiorów danych oraz trzech zadań.



Źródło: Opracowanie własne.

W większości przypadków równoległość danych jest prostsza oraz łatwiej skalowalna ponieważ nie występuje tutaj wyścig dostępu do danych (ang. *race condition*). Fakt, że ilość danych jest zawsze większa od ilości zadań również przemawia na korzyść programowania równoległego. Równoległe przetwarzanie danych jest wielokrotnie szybsze od przetwarzania ich na pojedynczym wątku.

Rys. 1.8. Komponenty bibliotek do programowania równoległego.



Źródło: Albahari J., *Threading in C#*, O'Reilly Media, 2006-2010 (www.albahari.com/threading).

PLINQ (ang. *Parallel Language Integrated Query*) jest najbardziej zaawansowaną biblioteką do programowania równoległego w języku C#, równocześnie stanowiącą uzupełnienie biblioteki, która zajmuje się przetwarzaniem zbiorów (*LINQ*), o aspekt wielowątkowy. Metody biblioteki *LINQ* są metodami rozszerzającymi (ang. *extension methods*) wywołane w kontekście `this IEnumerable<TSource>`, dzięki czemu można je stosować na wszystkich kolekcjach

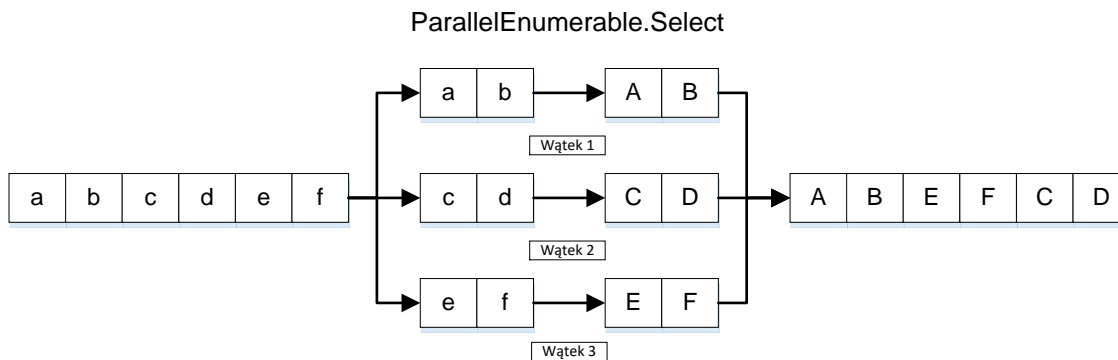
implementujących ten interfejs. *PLINQ* automatycznie wykonuje dzielenie zadania na podzadania, które później po wykonaniu, składa w końcowy rezultat. Poniżej znajduje się przykład użycia biblioteki *PLINQ* mający następującą postać:

Listing 1.5. Przykład użycia zapytania Select.

```
"abcdef".AsParallel().Select(c => char.ToUpper(c)).ToArray();
```

Na ciąg znaków "abcdef" wywołujemy metodę `AsParallel()`, jest ona rozszerzeniem interfejsu `IEnumerable`, na której następnie wywołujemy metodę `Select()`, służącą do wybrania odpowiednich danych ze zbioru (ang. *projection*). W powyższym przypadku na każdym elemencie tablicy znaków wywołujemy statyczną metodę `char.ToUpper()`, która zamienia znaki na wielką literę. Poniżej znajduje się diagram jak mógłby wyglądać podział pracy, na wykonanie tych operacji równoległe:

Rys. 1.9. Przykład podziału równoległego wykonywania pracy.



Źródło: Opracowanie własne na podstawie: Albahari J., *Threading in C#*, O'Reilly Media, 2006-2010 (www.albahari.com/threading).

W powyższym przykładzie należy zwrócić szczególną uwagę na zmianę kolejności elementów kolekcji wyjściowej w stosunku do kolekcji wejściowej. Można temu zapobiec stosując metodę `AsOrdered()`, jednakże przetwarzanie współbieżne, które musi utrzymywać porządek w kolekcji, jest znacznie mniej efektywne. Z reguły dzięki zastosowaniu równoległego przetwarzania zbioru danych uzyskujemy znaczący wzrost wydajności w przypadku czasu wykonywania zadań.

1.3.1. Techniki podziału danych na podzbiory.

Podział danych zależy od kilku czynników. Najważniejszym z nich jest mechanizm, który polega na podkradaniu przez wątki zadań z lokalnych kolejek. Każdy wątek posiada własną lokalną kolejkę zadań, do której są dołączane zadania stworzone przez zadanie aktualnie wykonywane w bieżącym wątku. Jeżeli w kolejce głównej nie będzie żadnych zadań oraz jeden z wątków roboczych (ang. *worker thread*) również nie będzie posiadał zadań we własnej lokalnej kolejce, wtedy zacznie wykonywać zadania znajdujące się w lokalnej kolejce innego wątku roboczego. W kolejności odwrotnej niż były wkładane do kolejki, ponieważ te zadania mają najmniejszy wpływ na pamięć danego wątku.⁹ Kolejnym ważnym czynnikiem są domyślne założenia dyspozytora wątków:

- Zadanie nie trwa dłużej niż 1-2 sekund,
- Kolejność wykonywania zadaniem jest losowa.

Biblioteka *PLINQ* automatycznie dzieli dane na zbilansowane podzbiory na których wykonywane są operacje. Możemy wyróżnić kilka sposobów podziału zbioru danych. Pierwszym z nich jest dynamiczny podział, w którym podzbiory na początku są małe, jednakże z czasem stają się coraz większe. Przyjmijmy że dany zbiór został rozdzielony na 4 wątki robocze *a, b, c, d*.

Rys. 1.10. Dynamiczny podział zbioru danych.

a	b	c	d	a	a	b	b	c	c	d	d	a	a	a	b	b	b	c	c	c	d	d	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Źródło: Opracowanie własne na podstawie: Hummel J., *Async and Parallel Programming*, Chicago .NET User Group, 2013.

Kolejną techniką jest podział statyczny na równe podzbiory, w tym przypadku tablice o 24 elementach dzielimy pomiędzy 4 rdzenie, tak aby każdy przetworzył równe 6 elementów.

Rys. 1.11. Przekrojowy statyczny podział zbioru danych.

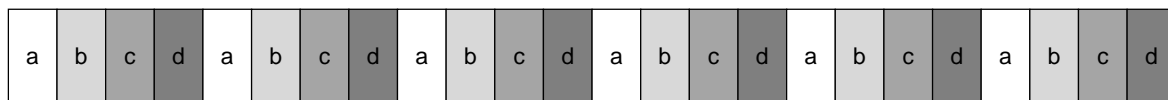
a	a	a	a	a	a	b	b	b	b	b	b	c	c	c	c	c	c	d	d	d	d	d	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Źródło: Opracowanie własne na podstawie: Hummel J., *Async and Parallel Programming*, Chicago .NET User Group, 2013.

Progresywny podział danych również należy do metod statycznych, takie podejście zwiększa prawdopodobieństwo zbilansowanego rozłożenia pracy pomiędzy dostępne wątki.

⁹ Hummel J., *Async and Parallel Programming*, Chicago .NET User Group, 2013.

Rys. 1.12. Progresywny statyczny podział zbioru danych.



Źródło: Opracowanie własne na podstawie: Hummel J., *Async and Parallel Programming*, Chicago .NET User Group, 2013.

Ostatnim rodzajem jest rozłożenie oparte na algorytmie haszującym, które jest dynamicznym sposobem rozłożenia pracy. W praktyce otrzymujemy losowo rozłożone zadania.

Rys. 1.13. Haszujący podział zbioru danych.



Źródło: Opracowanie własne na podstawie: Hummel J., *Async and Parallel Programming*, Chicago .NET User Group, 2013.

Biblioteka *PLINQ* dla metod (*GroupBy()*, *Join()*, *GroupJoin()*, *Intersect()*, *Except()*, *Union()*, *Distinct()*) używa domyślnie podziału wykorzystującego algorytm haszujący. Dla pozostałych metod w przypadku, kiedy kolekcje można indeksować np. lista, tablica, domyślnie wybierany jest przekrojowy podział (Rys. 1.11.). Dla wszystkich pozostałych kolekcji podział odbywa się dynamicznie (Rys. 1.10.). W szczególności podział przekrojowy jest szybszy dla długich sekwencji, w przypadku kiedy wykonanie operacji dla każdego elementu, zajmuje podobną ilość czasu.¹⁰

¹⁰ Albahari J., *Threading in C#*, O'Reilly Media, 2006-2010 (www.albahari.com/threading).

2. Charakterystyka wybranych wzorców projektowych [Zielonka]

Wzorzec projektowy jest gotowym rozwiązaniem problemu zaistniałego w danym środowisku. Każdy wzorzec składa się z czterech elementów¹¹:

1. *Nazwa wzorca* – jest to krótki opis problemu, rozwiązania oraz konsekwencji. Zazwyczaj składa się z dwóch słów,
2. *Problem* – opis problematycznej sytuacji, w której powinno się zastosować dany wzorzec np. jaką powinien mieć reprezentacje algorytm jako obiekt,
3. *Rozwiązanie* – opis poszczególnych elementów wzorca, ich relacji oraz wzajemnej współpracy. Rozwiązanie nie opisuje konkretnej implementacji. Zamiast tego jest swoistego rodzaju opisem postępowania w danej sytuacji,
4. *Konsekwencje* – zalety oraz wady implementacji danego wzorca. W kontekście systemu operacyjnego mogą opisywać koszty pamięci oraz czasu, a w kontekście wybranego języka jego szybkość.

Jedną z najważniejszych zalet wzorców projektowych jest ich uniwersalność. Tworzenie oprogramowania opierając się na standardach wyznaczonych przez wzorce ułatwia współpracę programistów nad danym zagadnieniem, jak i również późniejsze utrzymanie oprogramowania.

W niniejszym rozdziale zostaną opisane wybrane wzorce wykorzystywane w programowaniu wielowątkowym. Ze względu na fakt, iż programowanie wielowątkowe jest podatne na błędy projektowe, szczególnie ważne jest wykorzystywanie w tym aspekcie wzorców które w znaczący sposób ułatwiają późniejsze utrzymywanie kodu jak i znajdowanie błędów architektonicznych popełnianych przez programistów.

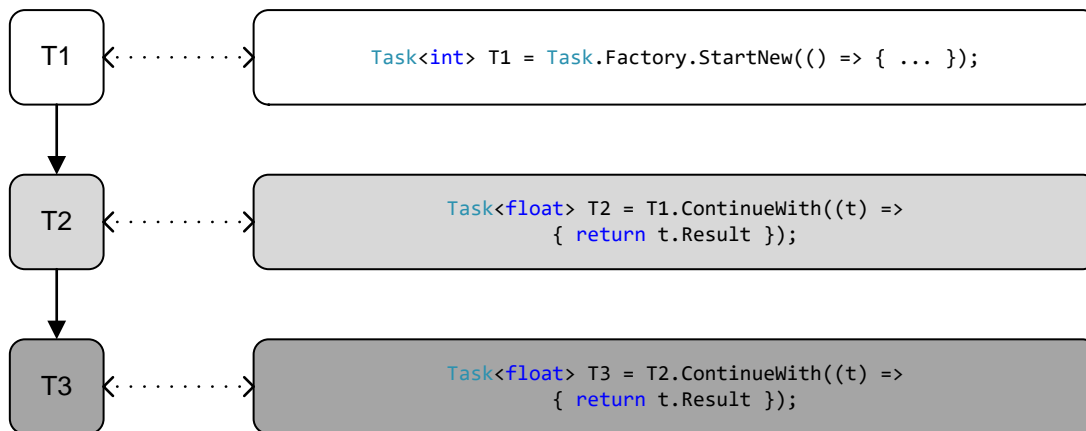
2.1. Wzorzec Pipeline

Pipeline jest wzorcem w którym, wątki są od siebie zależne, co oznacza że wykonywane są sekwencyjnie. Stosuje się go w zagadnieniach przepływu, przetwarzania obrazu oraz obsługi interfejsu użytkownika w programach okienkowych, kiedy wszystkie kontrolki z definicji muszą być obsługiwane z wątku głównego aplikacji (ang. *main thread*). Poniżej znajduje się diagram wzorca *Pipeline*¹²:

¹¹ Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

¹² Daryl A. Swade and James F. Rose., *A flexible pipeline data-processing environment*, 1998

Rys. 2.1. Wzorzec projektowy Pipeline.



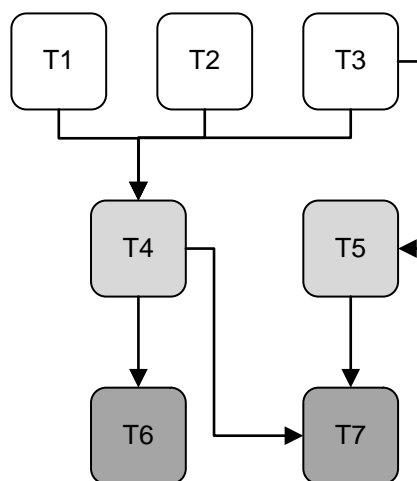
Źródło: Hummel J., *Async and Parallel Programming: Application Desing*, Pluralsight, 2012.

Idea wzorca opiera się na wykorzystaniu kontynuacji `ContinueWith()`, w tym miejscu można użyć równoznacznego mechanizmu `async` i `await`. Wzorzec *Pipeline* jest zazwyczaj używany na zbiorach danych, np. kolekcji obrazów, które przetwarzamy w pętli. W ten sposób uzyskujemy przetwarzanie wielowątkowe liniowe.

2.2. Wzorzec Dataflow

Wzorzec projektowy *Dataflow* jest rozszerzeniem wzorca *Pipeline*. Charakteryzuje się relacjami typu: wiele do jednego (ang. *many to one*) oraz jeden do wielu (ang. *one to many*). Przykładowe drzewo zależności znajduje się poniżej:

Rys. 2.2. Drzewo zależności wątków we wzorcu projektowym Dataflow.



Źródło: Hummel J., *Async and Parallel Programming: Application Desing*, Pluralsight, 2012.

Rozważmy sytuację w jakiej znajduje się wątek *T4*. Jest on połączony relacją wiele do jednego z wątkami *T1*, *T2*, *T3*. Oznacza to, że jest on zależny od wyników operacji trzech różnych wątków i jest wstrzymany do czasu pojawienia się rezultatów tychże wątków. W takich sytuacjach środowisko .NET oferuje kilka możliwych scenariuszy. Do wyboru są dwie statyczne metody `Task.Factory.ContinueWhenAll()` oraz `Task.Factory.ContinueWhenAny()`. Różnica pomiędzy tymi metodami jest taka, że pierwsza z nich czeka, aż wszystkie wątki zostaną wykonane, a druga czeka na dowolny pierwszy. Poniżej znajduje się przykład użycia zgodny z Rys. 2.2¹³.

Listing 2.1. Przykład zastosowania wzorca projektowego Dataflow.

```
Task<int> T1 = ...;
Task<int> T2 = ...;
Task<int> T3 = ...;

Task<double> T4 = Task.Factory.
    ContinueWhenAll<int, double>(
        new[] { T1, T2, T3 },(
            Task<int>[] tasks) => {
                double result = 0.0;
                foreach (var task in tasks)
                    result += task.Result;
                return result;
            });
```

Prototyp powyższej funkcji wygląda następująco:

Listing 2.2. Prototyp funkcji ContinueWhenAll.

```
Task<TResult> ContinueWhenAll<TAntecedentResult, TResult>(
    Task<TAntecedentResult>[] tasks,
    Func<Task<TAntecedentResult>[], TResult> continuationFunction);
```

Jako pierwszy argument przekazywana jest tablica uchwytów do wątków, na które metoda będzie czekała, aż do ich zakończenia. Drugim argumentem jest funkcja kontynuacyjna, w naszym przypadku jest to funkcja anonimowa, która sumuje wynik. Metoda `ContinueWhenAll()` pozwala na zmianę typu generycznego, w powyższym przypadku wątki *T1*, *T2*, *T3* zwracały strukturę `int`, jednakże uchwyt do wątku *T4* zwraca typ `double`.

2.3. Wzorzec *WaitAllOneByOne*

Wzorzec *WaitAllOneByOne* jest prostą koncepcją obsługi kolekcji wątków. Zamiast czekać, aż wszystkie wątki zakończą swoją pracę i na koniec zebrać wyniki możemy selektywnie wybierać aktualnie zakończony wątek z kolekcji i pobrać wynik jego operacji. Tego typu rozwiązanie

¹³ Meunier R., The Pipes and Filters Architecture, volume 1 of "Pattern Languages of Program Design", Addison-Wesley, 1995

idealnie nadaje się do sytuacji, w której niektóre wątki nie wykonają się poprawnie lub w sytuacji w której chcemy na bieżąco aktualizować wyniki operacji realizowanych współbieżnie. Poniżej znajduje się kod przedstawiający koncepcję działania wzorca *WaitAllOneByOne*:

Listing 2.3. Przykład zastosowania wzorca projektowego WaitAllOneByOne.

```
List<Task<int>> tasks = new List<Task<int>>();

for (int i = 0; i < N; i++)
    tasks.Add(Task.Factory.StartNew(() => ...));

while (tasks.Count > 0) {
    int i = Task.WaitAny(tasks.ToArray());
    // wykonaj operacje
    // t = tasks[i] - uchwyt do wątku zakończonego
    // t.Result
    tasks.RemoveAt(i);
}
```

W pętli `while` przetwarzane są aktualnie zakończone wątki. Metoda `int WaitAny(params Task[] tasks)` zwraca indeks w tablicy aktualnie zakończonego zadania. W ten sposób możemy przetworzyć wynik operacji, zaraz po jej zakończeniu bez konieczności czekania na pozostałe wyniki działań z kolekcji wątków. W momencie kiedy wszystkie zadania zostały obsłużone kolekcja będzie pusta co daje warunek kończący działania pętli `while`.

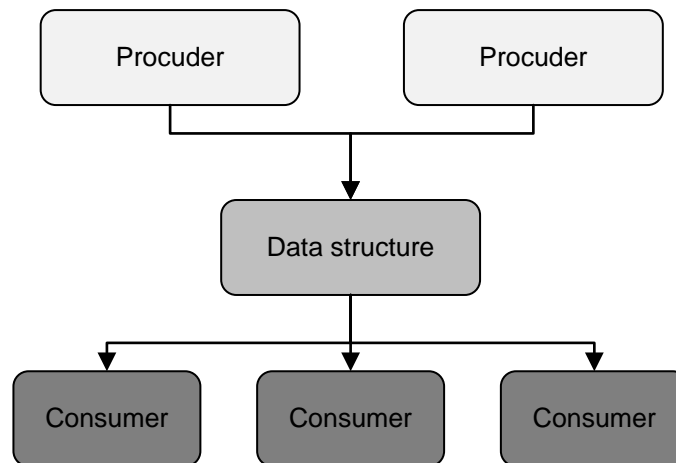
2.4. Wzorzec Producer-Consumer

Wzorzec *Producer-Consumer* ma zastosowanie w sytuacjach, kiedy prędkość generowania danych (ang. *data generation*) jest zupełnie inna, niż prędkość ich przetwarzania (ang. *data consumption*). Najważniejszym elementem tego rozwiązania jest struktura danych (ang. *data structure*). Pełni ona rolę stabilizującą przepływ danych pomiędzy wątkami tworzącymi, a wątkami, które te dane konsumują. W środowisku .NET taka kolekcja nazywa się `BlockingCollection<T>` należąca do przestrzeni nazw `System.Collections.Concurrent`, która pełni rolę buforu na dane.¹⁴

Klasa `BlockingCollection` pełni rolę bufora (kolekcji), który zapewnia właściwą synchronizację w środowisku wielowątkowym (ang. *thread-safe*). Kolekcja ta, ma określony stały rozmiar maksymalny (ang. *fixed-sized collection*) i kiedy jest zapełniona lub pusta, odpowiednio wątki produkujące lub konsumenckie są wstrzymywane. Klasa zapewnia kilka metod pomocniczych. Metoda `TryAdd()` dodaje elementy do kolekcji jeżeli bufor nie jest zapełniony, zwracając odpowiednią flagę `bool` jako wynik operacji. Analogiczną metodą do pobierania elementów z kolekcji jest `TryTake()`, która również zwraca flagę `bool`, natomiast element pobrany jest zwracany przez parametr oznaczony jako `out`.

¹⁴ Arpaci-Dusseau, Remzi H., *Operation Systems: Three Easy Pieces*, Arpaci-Dusseau Books, 2014

Rys. 2.3. Wzorec projektowy *Producer-Consumer*.



Źródło: Hummel J., *Async and Parallel Programming: Application Desing*, Pluralsight, 2012.

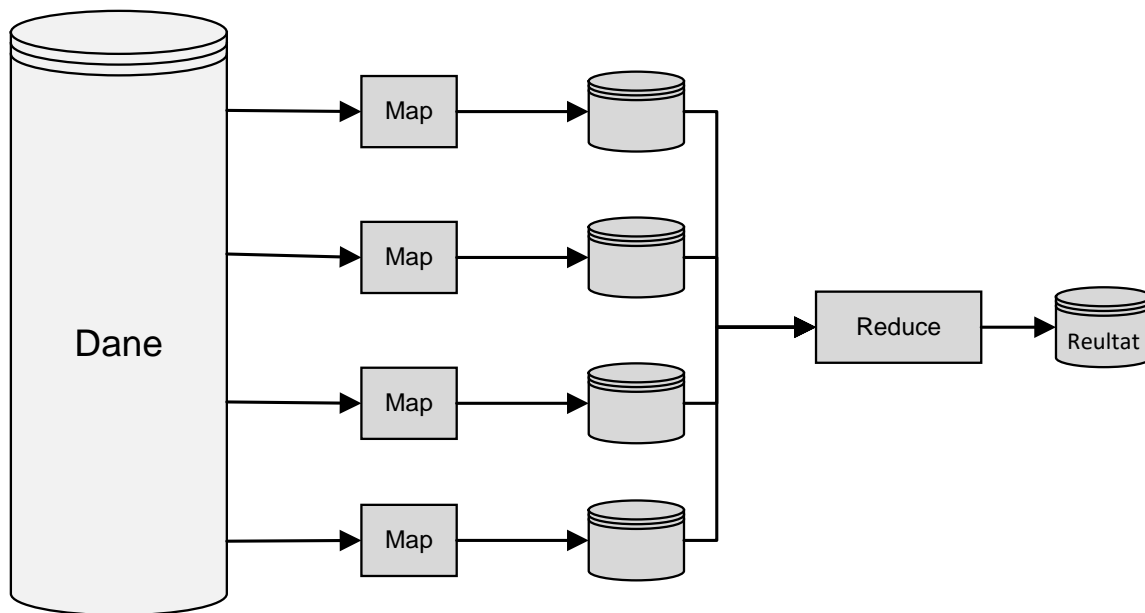
Wątki muszą być tworzone ze specjalną opcją `TaskCreationOptions.LongRunning`, która informuje dyspozytora, aby wstrzyknął (ang. *inject*) do zbioru wątków nowe wątki specjalnie przeznaczone dla danych zadań. W tym momencie środowisko .NET stworzy specjalny dedykowany zbiór nowych wątków (ang. *non-worker pool thread*). Wątki konsumenckie działają w pętli, dopóki nie otrzymają sygnału o zakończeniu pracy po przez metodę `CompleteAdding()`. W tym momencie kończy się proces przetwarzania.

2.5. Wzorec *MapReduce*

Wzorec *MapReduce* znajduje swoje zastosowanie w problemach typu data-mining lub wyszukiwaniu danych w dużych zbiorach. Składa się on z dwóch faz. W pierwszej odbywa się podział danej kolekcji na podzbiory (ang. *map*) oraz przetworzenie mniejszych zbiorów w osobnych wątkach. W drugiej fazie odbywa się redukcja (ang. *reduce*), połączenie wszystkich lokalnych rezultatów w jeden końcowy wynik. Wzorec *MapReduce* mocno się rozpowszechnił kiedy firma Google ujawniła, że jest to rozwiązanie, które zostało zaimplementowane w ich wyszukiwarce. W dzisiejszych czasach jest to jedno z najpopularniejszych podejść do rozwiązania problemów z bardzo dużą ilością danych, ponieważ dzięki podzieleniu zbioru na mniejsze części, zastosowanie przetwarzania wielowątkowego jest znacznie prostsze. Problem dużej współdzielonej kolekcji pomiędzy wiele wątków przestaje mieć jakiegokolwiek znaczenie, ponieważ każdy wątek posiad podzbiór na własny użytek.¹⁵

¹⁵ Lin J., Dyer C., *Data-Intensive Text Processing with MapReduce*, University of Maryland, 2010.

Rys. 2.4. Przepływ danych we wzorcu MapReduce.



Źródło: Hummel J., *Async and Parallel Programming: Application Desing*, Pluralsight, 2012.

Istnieje kilka strategii implementacyjnych dla przetwarzania oraz podziału danych. Jedną z nich jest użycie wzorca *WaitAllOneByOne* w fazie redukcji. W ten sposób podczas, gdy wątki kończą swoją pracę w różnym tempie, na bieżąco uzupełniany jest końcowy rezultat. Poniżej znajduje się przykład:

Listing 2.4. Pseudokod użycia wzorca projektowego MapReduce.

```
for (int i = 0; i < N; i++)
    tasks.Add(Task.Factory.StartNew<T>(
        (data) => { return map(data); });

while (tasks.Count > 0){
    int i = Task.WaitAny(tasks.ToArray());
    recude(tasks[i].Result);
    tasks.RemoveAt(i);
}
```

Innym podejściem jest użycie gotowych mechanizmów do podziału zbiorów z biblioteki *PLINQ* opisanych w rozdziale 1.3.1. Do tego zadania należy wykorzystać statyczną funkcję `Parallel.ForEach()`, która automatycznie podzieli zbiór pomiędzy różne wątki. Ważnym aspektem w tego rodzaju podejściu jest lokalna pamięć (ang. *Task Local Storage*). Jest to klasa dzielona pomiędzy wszystkie iteracje, przekazywana automatycznie do kolejnych kroków pętli. Na koniec jest zwracana jako obiekt wynikowy operacji wszystkich wątków.

3. Interoperacyjność na styku języków Java i C# w kontekście serializacji [Bąkowski]

Języki C# i Java oferują szereg możliwych sposobów serializacji, tzn. przekształcania obiektów do postaci ciągu bajtów. Rozdział ten jest poświęcony podstawowym metodom serializacji oferowanych przez oba środowiska programistyczne.

3.1. Serializacje do postaci strumienia bajtów w języku Java i serializacja binarna w C#.

Środowisko Java posiada metodę serializacji do postaci bajtów za pomocą interfejsu `java.io.Serializable`, który pozwala na serializację obiektu z niego korzystającego. Język C# posiada z kolei możliwość serializacji obiektów do postaci binarnej za pomocą klasy `BinaryFormatter`, która działa podobnie. Obie te metody są specyficzne dla swojego języka i nie mają swoich odpowiedników, przez co uniemożliwiają uzyskanie interoperacyjności na styku języka C# i Java przy ich użyciu.

3.2. Serializacja XML

Oba języki oferują narzędzia pozwalające na serializację danych do postaci XML. Język C# posiada klasę `XmlSerializer`, która umożliwia serializację obiektów. Z kolei język Java posiada wiele stosowanych bibliotek służących do serializacji danych. Najpowszechniej stosowanymi są biblioteka `XmlEncoder`, która współpracuje z `XmlDecoder` oraz `Xstream` i `JAXB`. Choć wszystkie te metody pozwalają na serializację do obiektu XML, posiadają znaczące różnice w procesie budowy kodu XML na podstawie obiektu, co może czynić kod otrzymany w wyniku serializacji nieczytelny w procesie deserializacji przez inną bibliotekę.

3.2.1. Serializacja w C# z użyciem klasy `XmlSerializer`

Najpowszechniejszym sposobem serializacji w języku C# jest serializacja z użyciem klasy `XmlSerializer`.

Tworzy ona przejrzysty kod, gdzie z obiektu jest tworzony węzeł nadrzędny o nazwie identycznej z nazwą klasy serializowanego obiektu, a poszczególne pola są reprezentowane przez węzły, których węzłem nadrzędnym jest ten reprezentujący sam obiekt. Przykładowo prezentuje obiekt klasy `Adres` na rysunku 3.1, który ma kilka pól definiujących adres.

Rys. 3.1. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą klasy `XmlSerializer`.

```
1 <Adres>
2   <ulica>Grunwaldzka</ulica>
3   <numer>10</numer>
4   <kodPocztowy>80-123</kodPocztowy>
5   <miasto>Gdańsk</miasto>
6 </Adres>
```

Źródło: Opracowanie własne

Język C# oferuje również atrybuty serializacji, które pozwalają na sterowanie procesem serializacji oraz wyglądem otrzymanego kodu XML. Atrybuty umieszcza się w klasie danych, bezpośrednio przed daną właściwością. Serializacja w C# pozwala na użycie między innymi następujących atrybutów ^{16 17}:

- Atrybut [XmlElement("NazwaWezłaWXml")] pozwala na określenie nowej nazwy węzła dla wybranej właściwości, która powinna zostać umieszczona między nawiasami w apostrofach, jako argument atrybutu.
- Atrybut [XmlAttribute("NazwaAtrybutuXml")] powoduje, że pole klasy staje się atrybutem węzła nadrzędnego, a nazwę atrybutu definiuje się poprzez umieszczenie jej między nawiasami w apostrofach, jako argument atrybutu.
- Atrybut [XmlArray("NazwaTablicy")] pozwala na zmianę nazwy węzła przechowującego tablicę obiektów, poprzez jej umieszczenie między nawiasami atrybutu w apostrofach, jako argument atrybutu.
- Atrybut [XmlIgnore] powoduje, że pomija wybraną właściwość w wynikowym kodzie XML.

Te i inne atrybuty pozwalają na sterowanie procesem serializacji i umożliwiają w pewnym stopniu zmianę wynikowego kodu XML.

Jedną z ważnych cech procesu serializacji jest fakt, że tworzony XML jest podatny na utratę części danych w procesie deserializacji w przypadku, gdy więcej niż jeden obiekt spośród serializowanych przechowuje referencje jakiegoś obiektu. W każdej części otrzymanego kodu XML w procesie serializacji, gdzie występuje referencja do danego obiektu, jest umieszczona jego kopia. Jednocześnie nie jest przechowywana informacja o umieszczeniu tego samego obiektu w innych częściach otrzymanego kodu XML. Powoduje to otrzymanie kilku kopii tego samego obiektu po procesie deserializacji, tracąc przy tym część danych. Przykładowo, gdy dwa obiekty klasy Student, posiadają referencję do tego samego obiektu klasy Adres, to w procesie deserializacji, każdy z nich otrzyma własną kopię. Widoczne to jest na załączonym rysunku 3.2.

¹⁶ Źródło: [https://msdn.microsoft.com/pl-pl/library/2baksw0z\(v=vs.120\).aspx](https://msdn.microsoft.com/pl-pl/library/2baksw0z(v=vs.120).aspx) (dostęp w dniu 20.10.2016)

¹⁷ Źródło: [https://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlattribute\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlattribute(v=vs.110).aspx) (dostęp w dniu 20.10.2016)

Rys. 3.2. Przykładowy kod XML obrazujący brak przechowywania informacji o referencjach przez XmlSerializer.

```
1 <Student Id="1">
2   <imie>Jan</imie>
3   <nazwisko>Kowalski</nazwisko>
4   <miejsceZamieszkania>
5     <ulica>Grunwaldzka</ulica>
6     <numer>10</numer>
7     <kodPocztowy>80-123</kodPocztowy>
8     <miasto>Gdańsk</miasto>
9   </miejsceZamieszkania>
10 </Student>
11 <Student Id="2">
12   <imie>Kaziemierz</imie>
13   <nazwisko>Nowak</nazwisko>
14   <miejsceZamieszkania>
15     <ulica>Grunwaldzka</ulica>
16     <numer>10</numer>
17     <kodPocztowy>80-123</kodPocztowy>
18     <miasto>Gdańsk</miasto>
19   </miejsceZamieszkania>
20 </Student>
```

Źródło: Opracowanie własne

Klasa XmlSerializer w procesie deserializacji z powodu braku informacji o referencjach, nie jest w stanie identycznie odtworzyć serializowanych danych, tracąc przy tym ich część.

3.2.2. Serializacja w Java

Język Java oferuje wiele bibliotek umożliwiających serializację obiektów do postaci XML. Najpopularniejszymi z nich są XmlEncoder, która współpracuje z XmlDecoder oraz Xstream i JAXB.

Klasa XmlEncoder jest pierwszą analizowaną z spośród bibliotek umożliwiającą serializację obiektów. Biblioteka ta jest dostępna w JDK (Java Developer Kit), które jest środowiskiem programistycznym służącym do tworzenia appletów i aplikacji w języku JAVA. Zawiera ono środowisko uruchomieniowe oraz narzędzia niezbędne to implementacji, kompilacji oraz debugowania napisanych programów. Do deserializacji obiektów zserializowanych przez XmlEncoder, używana jest klasa XmlDecoder

XmlEncoder prezentuje zdecydowanie inne podejście do procesu serializacji danych¹⁸. Serializacja obiektu klasy Adres, analogicznego do tego zaprezentowanego w podrozdziale poświęconemu klasie XmlSerializer, generuje kod XML, który jest widoczny na rysunku 3.3.

¹⁸ Źródło: <https://docs.oracle.com/javase/7/docs/api/java/beans/XMLEncoder.html> (dostęp w dniu 20.10.2016)

Rys. 3.3. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą klasy *XmlEncoder*.

```
1 <java version="1.8.0_101" class="java.beans.XMLDecoder">
2   <object class="xmlseriliazacjabase.Adres" id="Adres0">
3     <void class="xmlseriliazacjabase.Adres" method="getField">
4       <string>numer</string>
5       <void method="set">
6         <object idref="Adres0"/>
7         <int>10</int>
8       </void>
9     </void>
10    <void property="kodPocztowy">
11      <string>80-123</string>
12    </void>
13    <void property="miasto">
14      <string>Gdańsk</string>
15    </void>
16    <void property="ulica">
17      <string>Grunwaldzka</string>
18    </void>
19  </object>
20 </java>
```

Źródło: Opracowanie własne

Stworzone drzewo DOM w procesie serializacji XML przechowuje nie tylko dane serializowanych obiektów, ale również informacje o budowie danych obiektów. Dzięki temu przechowywane jest wiele informacji, które nie są potrzebne, a dodatkowo otrzymany kod XML nie jest czytelny. W przeciwieństwie do biblioteki *XmlSerializer* serializowane dane przechowują informację o referencjach do danego obiektu, jeśli występuje więcej niż jedna. Umożliwia to uniknięcie kopiowania obiektów w procesie deserializacji. Tak sformatowane dane nie pozwalają na poprawną deserializację przez bibliotekę *XmlSerializer*, o której była mowa w podrozdziale poświęconym serializacji do postaci XML w języku C#.

Biblioteka *Xstream* jest popularna wśród programistów, mimo że nie należy do JDK (Java Developer Kit). Wynik serializacji jest zbliżony do uzyskiwanego przez klasę *XmlSerializer*. Przykład wyniku, dla obiektu klasy *Adres*, został przedstawiony na rysunku 3.4.

Rys. 3.4. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą biblioteki *Xstream*.

```
1 <xmlserializationxstream.Adres>
2   <ulica>Grunwaldzka</ulica>
3   <numer>10</numer>
4   <kodPocztowy>80-123</kodPocztowy>
5   <miasto>Gdańsk</miasto>
6 </xmlserializationxstream.Adres>
```

Źródło: Opracowanie własne

Biblioteka *Xstream* serializuje w podobny sposób do *XmlSerializer*, gdzie obiekty są prezentowane przez pojedynczy węzeł i tak samo jest w przypadku pól obiektu. Jednak taki kod XML, nie jest czytelny dla klasy *XmlSerializer*, ponieważ nazwy węzłów reprezentujących sam obiekt są

inne. Biblioteka Xstream pozwala jednak na sterowanie procesem serializacji danych podobnie jak to było w przypadku XmlSerializer, co pozwala na uzyskanie kompatybilności obu procesów. Sterowanie procesem serializacji jest możliwe na dwa sposoby. Pierwszym jest wywoływanie odpowiednich metod na obiekcie klasy Xstream, który służy do serializacji i deserializacji. Drugim jest umieszczenie adnotacji poprzedzających poszczególne klasy i ich pola oraz metody. Biblioteka Xstream oferuje między innymi następujące sposoby na sterowanie procesem serializacji ^{19 20}:

- Metoda „alias("nazwa", Klasa.class)” lub adnotacja „@XStreamAlias("nazwa)” umieszczona przed deklaracją obiektu, pozwalają na zmianę nazwy węzła reprezentującego dany obiekt na nową. Przy użyciu metody należy podać dwa argumenty dla adnotacji w nawiasie. Pierwszym jest ciąg znaków reprezentujący nową nazwę, a drugi to typ klasy. W przypadku użycia adnotacji jako argument należy podać jedynie nową nazwę.
- Metoda „aliasField("nazwa", Klasa.class, "pole”)” lub wspomniana wcześniej adnotacja „@XStreamAlias("nazwa)” pozwalają na zmianę nazwy węzła reprezentującego dane pole. W tym przypadku adnotacja powinna zostać umieszczona przed odpowiednim polem klasy. Podobnie jak w przypadku metody trzeba podać nową nazwę oraz typ, a w przypadku adnotacji potrzebna jest tylko nazwa.
- Metoda „useAttributeFor(Klasa.class, "atrybut”)” lub umieszczenie adnotacji „@XStreamAsAttribute” przed odpowiednim polem spowoduje to, że wybrane pole stanie się atrybutem węzła nadrzędnego, reprezentującego obiekt. W przypadku metody trzeba podać typ klasy, a następnie nazwę atrybutu jaki ma reprezentować dane pole klasy. Adnotacja nie wymaga żadnego atrybutu, a nazwę argumentu można zmienić poprzez zastosowanie jej razem z adnotacją „@XStreamAlias("nazwa”)”.

Należy zwrócić uwagę, że podobnie jak w przypadku biblioteki XmlEncoder, serializowane dane przechowują informacje o referencjach, jeżeli więcej niż jeden obiekt posiada referencję do tego samego obiektu. Pole klasy przechowujące referencję do obiektu, do którego już inny obiekt przechowywał referencje i został wcześniej serializowany, zostaje zapisane w następujący sposób:

```
<adres reference="../../../Student/adres"/>
```

Węzeł nie przechowuje całego obiektu z jego wszystkimi polami. Przechowuje tylko i wyłącznie informację w postaci argumentu węzła o nazwie „reference”, gdzie w drzewie DOM umieszczony jest cały obiekt. Deserializacja takich danych przez XmlSerializer z języka C# nie będzie możliwa, ponieważ „XmlSerializer” nie obsługuje referencji. W tym celu trzeba użyć metody „setMode(XStream.NO_REFERENCES)” na obiekcie klasy „XStream”, który jest używany do

¹⁹ Źródło: <http://x-stream.github.io/alias-tutorial.html> (dostęp w dniu 20.10.2016)

²⁰ Źródło: <http://x-stream.github.io/annotations-tutorial.html> (dostęp w dniu 20.10.2016)

serializacji i deserializacji danych. Spowoduje to, że referencje do jednego nie będą przechowywane, a każde wystąpienie referencji spowoduje zduplikowanie danego obiektu.

Biblioteki XmlSerializer oraz XStream z reguły nie są kompatybilne, a proces serializowania i deserializowania posiada znaczące różnice, ale poprzez świadome sterowanie procesem serializacji i deserializacji danych jest możliwe uzyskanie kodu XML, który będzie poprawnie odczytywany przez obie.

Biblioteka **Jaxb** jest kolejną powszechnie stosowaną biblioteką przez programistów, służącą do serializacji i deserializacji obiektów. Wchodzi ona w skład JDK (Java Developer Kit). Warto zwrócić uwagę, że serializowany obiekt, który będzie stanowił korzeń drzewa XML, musi posiadać adnotację `@XmlRootElement`. W wyniku serializacji obiektu klasy `Adres`, analogicznego do wcześniej prezentowanych, otrzymuje się wynik zaprezentowany na rysunku 3.5.

Rys. 3.5. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą biblioteki Jaxb.

```
1 <adres>
2   <ulica>Grunwaldzka</ulica>
3   <numer>10</numer>
4   <kodPocztowy>80-123</kodPocztowy>
5   <miasto>Gdańsk</miasto>
6 </adres>
```

Źródło: Opracowanie własne

Otrzymany efekt przypomina rezultat serializacji za pomocą klasy `XmlSerializer` z języka C#. Jednak utworzone drzewo DOM posiada znaczące różnice, które uniemożliwią deserializację przez klasę `XmlSerializer`. Pierwszą jest fakt, że nazwy węzłów reprezentujących sam obiekt, zaczynają się od małej litery. Ważne jest również, że sposób serializacji tablic jest inny.

Rys. 3.6. Przykład serializacji obiektu przechowującego tablicę z użyciem biblioteki Jaxb.

```
1 <tablica>
2   <liczby>1</liczby>
3   <liczby>2</liczby>
4   <liczby>3</liczby>
5 </tablica>
```

Źródło: Opracowanie własne

Przykładowy obiekt klasy `Tablica` zaprezentowany na rysunku 3.6, przechowuje tablicę o nazwie `liczby`, która zawiera zmienne typu `int`. Zamiast utworzenia w procesie serializacji węzła reprezentującego tablicę, opinającego wszystkie elementy wewnątrz niej, proces serializacji tworzy osobne węzły dla każdego elementu tablicy. Jego węzłem nadrzędnym jest bezpośrednio węzeł reprezentujący obiekt do którego należała tablica, a nie tablica przechowująca ten element. Warto zwrócić uwagę, że proces serializacji więcej niż jednej referencji do jednego obiektu powoduje jego

zduplikowanie w procesie deserializacji, tak jak miało to miejsce w przypadku „XmlSerializer”, co może powodować utratę części danych w procesie deserializacji.

W celu uzyskania interoperacyjności z biblioteką „XmlSerializer” konieczne jest odpowiednie sterowanie procesem serializacji danych, które umożliwia nam biblioteka „Jaxb”. Procesem serializacji i deserializacji danych zarządza się za pomocą adnotacji przy deklaracji klasy oraz jej poszczególnych pól. W przypadku elementów prywatnych niezbędne jest utworzenie publicznych getterów i setterów, a adnotacje dotyczące serializacji powinny być umieszczone przed getterami. Poniżej znajdują się przykładowe adnotacje służące do sterowania procesem serializacji^{21 22 23 24}, które pozwalają na uzyskanie kompatybilności z procesem serializacji i deserializacji przez „XmlSerializer”:

- Adnotacja „@XmlRootElement” może być używana w połączeniu z elementami, które są klasą najwyższego poziomu stanowiącą korzeń drzewa DOM lub elementy typu enum. Dodatkowo umieszczenie za adnotacją „(name=„nazwaObiektu”)” pozwala na zmianę nazwy węzła reprezentującego ten element, poprzez umieszczenie nowej nazwy między apostrofami.
- Adnotacja „@XmlElement” powinna być przypisana do wszystkich getterów, które powinny być załączone w otrzymanym drzewie DOM w procesie serializacji XML. Bez niego zostaną pominięte w procesie serializacji. Dodatkowo można umieścić bezpośrednio za nim argument w postaci „(name=„nazwaPola”)”, który pozwala na zmianę nazwy węzła reprezentującego ten element. Adnotacja ta może być użyta również przy polach publicznych, które nie wymagają użycia getterów i setterów, lub wewnątrz adnotacji „@XmlElement”.
- Adnotacja „@XmlElement” służy jako kontener dla adnotacji „@XmlElement”, ponieważ nie jest możliwe użycie wielokrotnie adnotacji „@XmlElement” w stosunku do jednego pola. Jest to pożyteczne, jeśli kontener zawiera obiekty różnego typu, a nam zależy na rozróżnieniu nazw elementów poszczególnych typów.
- Adnotacja „@XmlElementWrapper” pozwala na opięcie jednym elementem reprezentacji XML. Jest to głównie używane w celu opięcia jednym elementem kolekcji elementów.
- Adnotacja „@XmlAttribute” pozwala na umieszczenie danego pola w drzewie XML w postaci atrybutu węzła nadrzędnego reprezentującego dany obiekt.

²¹ Źródło: [https://jaxb.java.net/tutorial/section_6_2_1-A-Survey-Of-JAXB-Annotations.html#Top-level Elements: XmlRootElement](https://jaxb.java.net/tutorial/section_6_2_1-A-Survey-Of-JAXB-Annotations.html#Top-level%20Elements:XmlElement) (dostęp w dniu 20.10.2016)

²² Źródło: [https://jaxb.java.net/tutorial/section_6_2_7_1-Annotations-for-Fields.html#The Annotation XmlElement](https://jaxb.java.net/tutorial/section_6_2_7_1-Annotations-for-Fields.html#The%20AnnotationXmlElement) (dostęp w dniu 20.10.2016)

²³ Źródło: [https://jaxb.java.net/tutorial/section_6_2_7_7-Wrapping-Repeated-Elements- XmlElementWrapper.html#Wrapping Repeated Elements: XmlElementWrapper](https://jaxb.java.net/tutorial/section_6_2_7_7-Wrapping-Repeated-Elements-XmlElementWrapper.html#Wrapping%20Repeated%20Elements:XmlElementWrapper) (dostęp w dniu 20.10.2016)

²⁴ Źródło: [https://jaxb.java.net/tutorial/section_6_2_7_3-Class-Fields-as-Attributes-XmlAttribute.html#Class Fields as Attributes: XmlAttribute](https://jaxb.java.net/tutorial/section_6_2_7_3-Class-Fields-as-Attributes-XmlAttribute.html#ClassFields%20as%20Attributes:XmlAttribute) (dostęp w dniu 20.10.2016)

Biblioteka Jaxb domyślnie nie jest kompatybilna z najpowszechniej stosowaną biblioteką do serializacji danych po stronie C#, czyli XmlSerializer. Jednak jest możliwe uzyskanie kompatybilności obu procesów poprzez sterowanie procesem serializacji za pomocą adnotacji.

3.2.3. Biblioteki służące do serializacji XML dostępne zarówno w językach JAVA i C#

Należy też zwrócić uwagę, że są dostępne biblioteki służące do serializacji i deserializacji obiektów do postaci XML, które posiadają swoją implementację zarówno w języku C# i JAVA. Taką biblioteką jest WOX²⁵. Jedną z głównych jej cech jest generowanie ustandaryzowanego kodu XML, który jest niezależny od języka programowania. Jest to istotne w kontekście interoperacyjności na styku dwóch języków, w tym przypadku C# i JAVA. WOX pozwala na łatwą serializację i deserializację danych niezależnie od używanego języka i bez potrzeby sterowania procesem serializacji.

Biblioteka Xstream, choć powstała pierwotnie jako biblioteka służąca do serializacji danych do postaci XML w języku JAVA, to posiada również swoją implementację w języku C#. Pozwala to na interoperacyjność na styku tych dwóch języków.

²⁵ Źródło: <http://woxserializer.sourceforge.net/> (dostęp w dniu 20.10.2016)

4. Realizacja projektu.

Zrealizowanym projektem jest systemem do wspomagania gier zespołowych opartym na systemach GPS i AGPS. Jego podstawową funkcjonalnością jest zapewnienie wsparcia graczom dzięki dostarczaniu aktualnych informacji o pozycji jednostek sprzymierzonych. Pomysł wzorowanych jest na amerykańskim wojskowym systemie *Blue Force Tracking*.

Metodyką przyjętą przy wytwarzaniu oprogramowania do projektu było programowanie zwinne (ang. *Agile software development*), gdzie jedna iteracja trwała 3 tygodnie. Podział prac występował następująco:

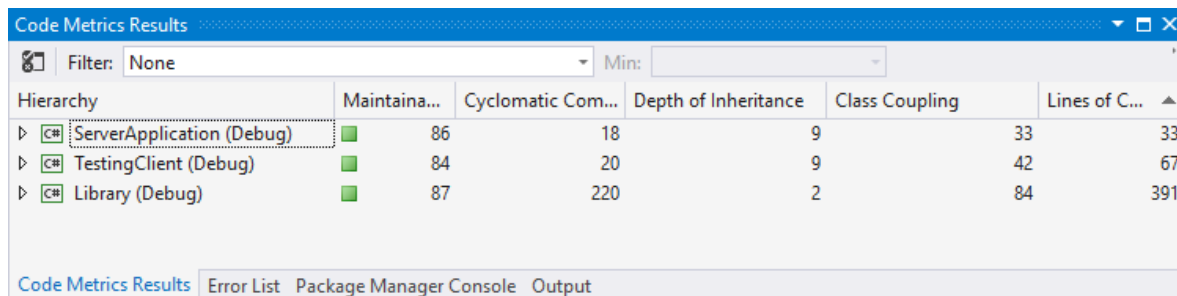
Tabela 4.1. Podział prac w projekcie.

Członkowie projektu	Zrealizowane zadania
Wojciech Zielonka	<ul style="list-style-type: none">• Biblioteka serwerowa (C#)• Aplikacja serwerowa (C#)• Aplikacja kliencka do testowania (C#)
Filip Bąkowski	<ul style="list-style-type: none">• Aplikacja kliencka (Java, Android)

W celu usprawnienia prac został zastosowany system kontroli wersji *Git* do zachowania lepszej kontroli na kodem źródłowym w projekcie²⁶. Prace nad projektem trwały około 7 miesięcy.

Poniżej znajduje się metryka kodu dotycząca zagadnień związanych z językiem C#.

Tabela 4.2 Metryki kodu dotyczące języka C#.



Hierarchy	Maintain...	Cyclomatic Com...	Depth of Inheritance	Class Coupling	Lines of C...
ServerApplication (Debug)	86	18	9	33	33
TestingClient (Debug)	84	20	9	42	67
Library (Debug)	87	220	2	84	391

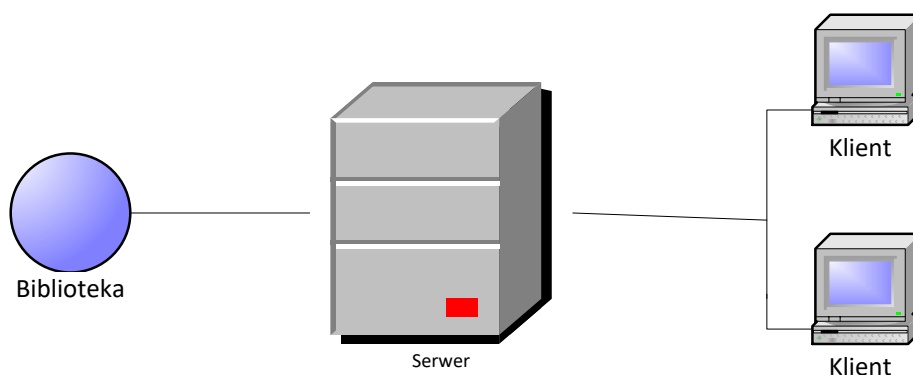
Źródło: Opracowanie własne.

²⁶ Źródło: <https://github.com/Zielon/GreenForceTracking>

4.1. Schemat i opis głównych komponentów projektu.

System składa się z trzech głównych części. Jest to biblioteka serwerowa, serwer oraz klienci. Biblioteka serwerowa została napisana w języku C#, do aplikacji serwerowej została wykorzystana technologia WPF (ang. *Windows Presentation Foundation*) natomiast aplikacja kliencka jest zrealizowana w języku Java w technologii Android. Poniżej znajduje się schemat głównych komponentów projektu.

Rys. 4.1. Schemat projektu.



Źródło: Opracowanie własne.

Aplikacja serwerowa korzysta z biblioteki stworzonej specjalnie na rzecz projektu. Za pomocą narzędzia Costura²⁷ biblioteka jest połączona z plikiem wykonywalnym .exe. Stworzona w ten sposób aplikacja może zostać uruchomiona na dowolnym komputerze, który posłuży za serwer systemu.

4.2. Biblioteka serwerowa

Biblioteka jest napisana wielowątkowo. Na każdego unikatowego klienta jest tworzony nowy wątek, który następnie jest wstrzykiwany do zbioru wątków roboczych. Serwer nasłuchuje za pomocą klasy `TcpListener` wszystkich połączeń nadchodzących na porcie 52400. Następnie wiadomość każdego klienta jest przetwarzana przez serwer.

Komunikacja pomiędzy komponentami została oparta na ustandaryzowanych wiadomościach XML. Każda wiadomość zwana również ramką jest strukturą XML. Przetwarzanie wiadomości polega na ich deserializacji²⁸, a następnie obsłudze gotowego obiektu. Poniżej znajduje się przykład dwóch wiadomości obsługiwanych przez serwer.

²⁷ Źródło: <https://github.com/Fody/Costura>

²⁸ Proces polegający na odczytaniu strumienia danych i przetworzenia ich na obiekt w rozumieniu programowania obiektowego.

Rys. 4.2. Ramka wysyłana podczas logowania do systemu.

```
1 <Frame>
2   <SystemUser>
3     <FrameType>Login</FrameType>
4     <Password>123</Password>
5     <LoggedIn>>false</LoggedIn>
6     <Login>root</Login>
7   </SystemUser>
8 </Frame>
```

Źródło: Opracowanie własne.

Ramka służąca do logowania jest obiektem *SystemUser* i podczas komunikacji jest serializowana oraz deserializowana naprzemiennie w zależności od określonej sytuacji. Kolejną ramką obsługiwaną przez serwer jest ramka służąca do wymiany danych o aktualnej pozycji każdego gracza oraz innych informacji dotyczących gry.

Rys. 4.3. Ramka wysyłana do serwera po zalogowaniu do systemu.

```
1 <Frame>
2   <Client>
3     <Login>root</Login>
4     <Lat>2.33</Lat>
5     <Lon>542.356</Lon>
6     <Message>#Message</Message>
7     <FrameType>Player</FrameType>
8   </Client>
9 </Frame>
```

Źródło: Opracowanie własne.

Obiekt *Client* jest serializowany do postaci widocznej na Rys. 4.3. Zawiera ona dwie propecje dotyczące aktualnych współrzędnych oraz pole na wiadomość przesłaną od klienta. W ten sposób uzyskujemy interoperacyjność pomiędzy niezależnymi systemami. W naszym przypadku pomiędzy systemem Android, a Windows. Wiadomości do każdego użytkownika są wysyłane w momencie zmiany współrzędnych dowolnego gracza. W tym przypadku został zaimplementowany wzorzec projektowy obserwator (ang. *Observer*), który w przypadku zmiany propecji określającej współrzędne propaguje zdarzenie do wszystkich subskrybentów. W ten sposób w każdym wątku przetwarzającym równocześnie odbywa się wysyłanie zmian dotyczących danego klienta, do każdego obecnego gracza. Kolejną ramką jest polecenie dodania markera na mapie. Mechanizm działa identycznie jak w przypadku ramki *Client*. Markerem jest punkt o określonych współrzędnych, który pojawia się u wszystkich graczy na urządzeniach mobilnych. W momencie kiedy gracz opuszcza rozgrywkę wszystkie jego markery są usuwane z mapy.

Rys. 4.4. Ramka do dodawania markera na mapie.

```
1 <Frame>
2   <Marker>
3     <Login>root</Login>
4     <Id>12</Id>
5     <Text>New marker</Text>
6     <Lng>2.45</Lng>
7     <Lat>1.46</Lat>
8     <Add>true</Add>
9   </Marker>
10 </Frame>
```

Źródło: Opracowanie własne.

W momencie kiedy gracz nie jest podłączony do serwera, co oznacza że opuścił on rozgrywkę, do wszystkich użytkowników jest propagowana ramka informująca o usunięciu danego gracza z serwera.

Rys. 4.5. Ramka informująca o usunięciu gracza z pokoju.

```
1 <Frame>
2   <RemovingUser>
3     <Login>String</String>
4   </RemovingUser>
5 </Frame>
```

Źródło: Opracowanie własne.

Interfejsem biblioteki serwerowej są zdarzenia (ang. *events*), które są subskrybowane przez aplikacje. W ten sposób uzyskany został generyczny serwer, który może zostać uruchomiony w dowolnej okienkowej aplikacji w środowisku .NET. W tym celu został zastosowany wzorzec projektowy *Publish/Subscribe*²⁹.

²⁹ Źródło: <https://msdn.microsoft.com/en-us/library/ff649664.aspx> (dostęp w dniu 06.01.2017)

4.3. Aplikacja serwerowa.

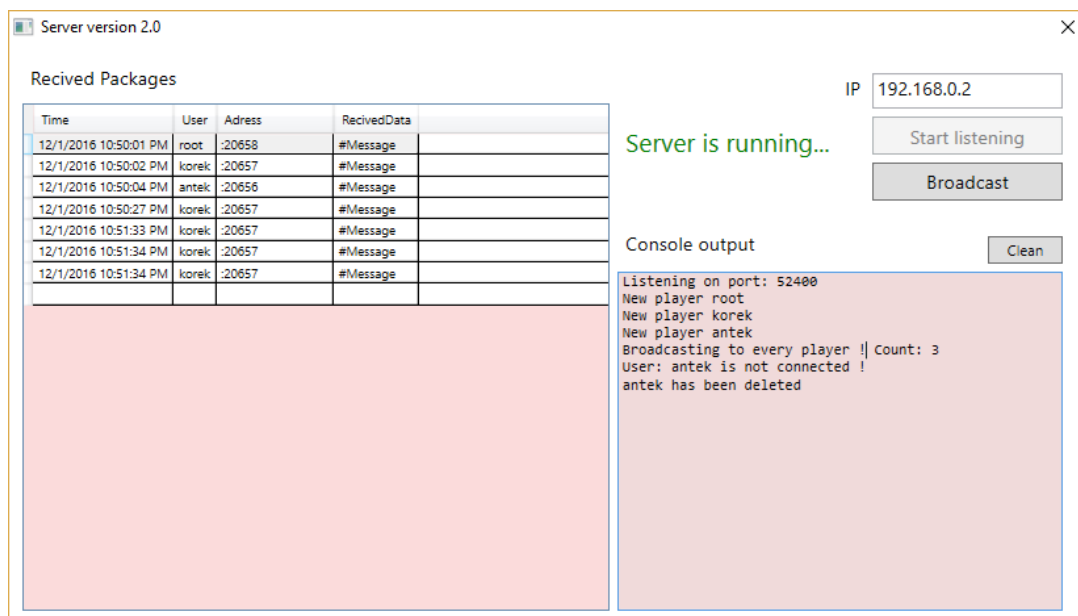
Aplikacja serwerowa wykorzystuje technologię *WPF*. Wszystkie zdarzenia, które są udostępniane w interfejsie serwerowym wykorzystywane są w interakcji z aplikacją. Poniżej znajduje się przykład wykorzystania biblioteki serwerowej w aplikacji właściwej.

Listing 4.1. Przykład inicjalizacji subskrypcji zdarzeń serwerowych.

```
var server = new Library.Server.Server(ip, 52400);
server.WindowEvent += (s, a) => ServerStatus.Content = a.Running;
```

Zdarzenie `WindowEvent` subskrybuje zmianę statusu wyświetlanego w interfejsie graficznym aplikacji. W ten sposób każde zdarzenie po stronie biblioteki ma swoje odzwierciedlenie widoczne dla użytkownika.

Rys. 4.6. Aplikacja serwerowa.



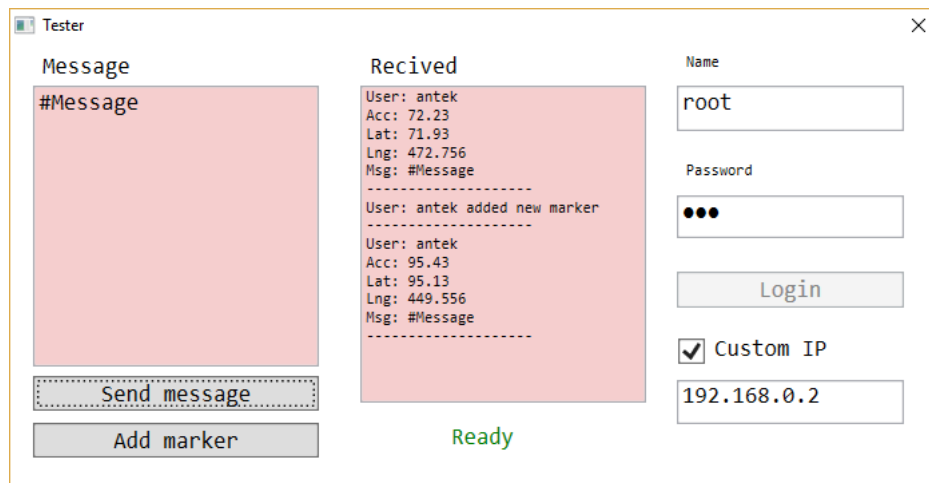
Źródło: Opracowanie własne.

Na Rys. 5.6 widzimy działającą aplikację serwerową wraz z przykładowymi interakcjami. Na lokalny adres IP: 192.168.0.2 są przekazywane wszystkie paczki TCP przychodzące na port 52400. Aplikacja wyświetla komunikaty oraz pokazuje każdą przychodzącą ramkę w tabeli *Recived Packages*.

4.4. Aplikacja kliencka do testowania

W celu symulowania wielu różnych użytkowników oraz sprawnego testowania komunikacji została napisana prosta aplikacja w języku C#, która tak samo jak w przypadku aplikacji serwerowej wykorzystuje tę samą bibliotekę.

Rys. 4.7. Aplikacja do testowania napisana w języku C#.



Źródło: Opracowanie własne.

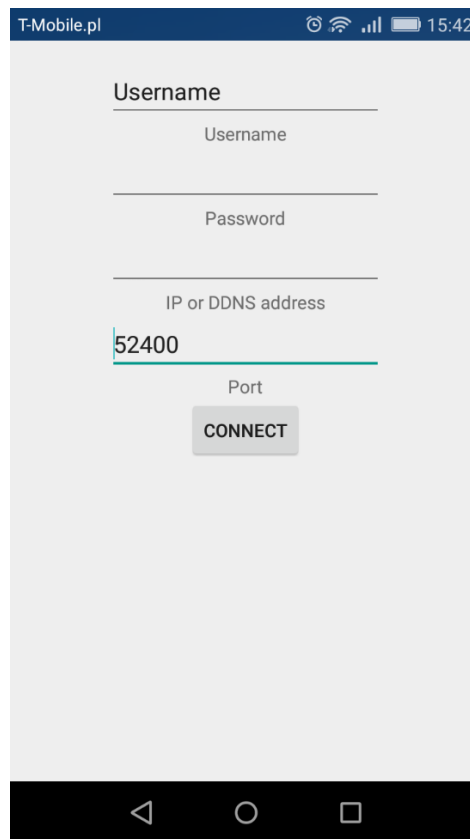
W oknie *Recived* pojawiają się wiadomości wysyłane przez serwer. Przed użyciem aplikacji użytkownik musi się zalogować. W aplikacji testowej współrzędne są inkrementowane co każde wysłanie w celu symulacji zmiany dla obserwatora wysyłającego pakiety do graczy.

4.5. Mobilna aplikacja kliencka

Aplikacja mobilna składa się z dwóch klas dziedziczących po klasie Activity, która jest jednym z podstawowych komponentów systemu Android. Służy ona głównie do interakcji z użytkownikiem i tworzenia okna aplikacji. Pierwszą klasą jest LoginActivity, która odpowiada za okno logowania do systemu. Drugą jest klasa RoomActivity, tworząca właściwe okno aplikacji, udostępniające takie funkcjonalności jak wyświetlanie mapy i czat.

LoginActivity obsługuje formularz logowania. Formularz zawiera takie pola jak login użytkownika, jego hasło, adres IP lub DDNS serwera oraz port. Po wypełnieniu formularza następuje próba połączenia z serwerem.

Rys. 4.8. Wygląd okna LoginActivity.



Źródło: Opracowanie własne.

Połączenie z serwerem odbywa się z użyciem klasy Socket. Zarządzaniem nawiązanym połączeniem zajmuje się klasa typu SocketCommunication, która pośrednio odpowiada za wysyłanie i odbieranie wiadomości. Dodatkowo udostępnia ona obsługę takich wydarzeń jak nawiązanie i utrata połączenia oraz problem z jego nawiązaniem. Wysyłanie i nasłuchiwanie wiadomości odbywa się w osobnych wątkach. Nadchodzące ramki od strony serwera są identyfikowane i skierowywane do

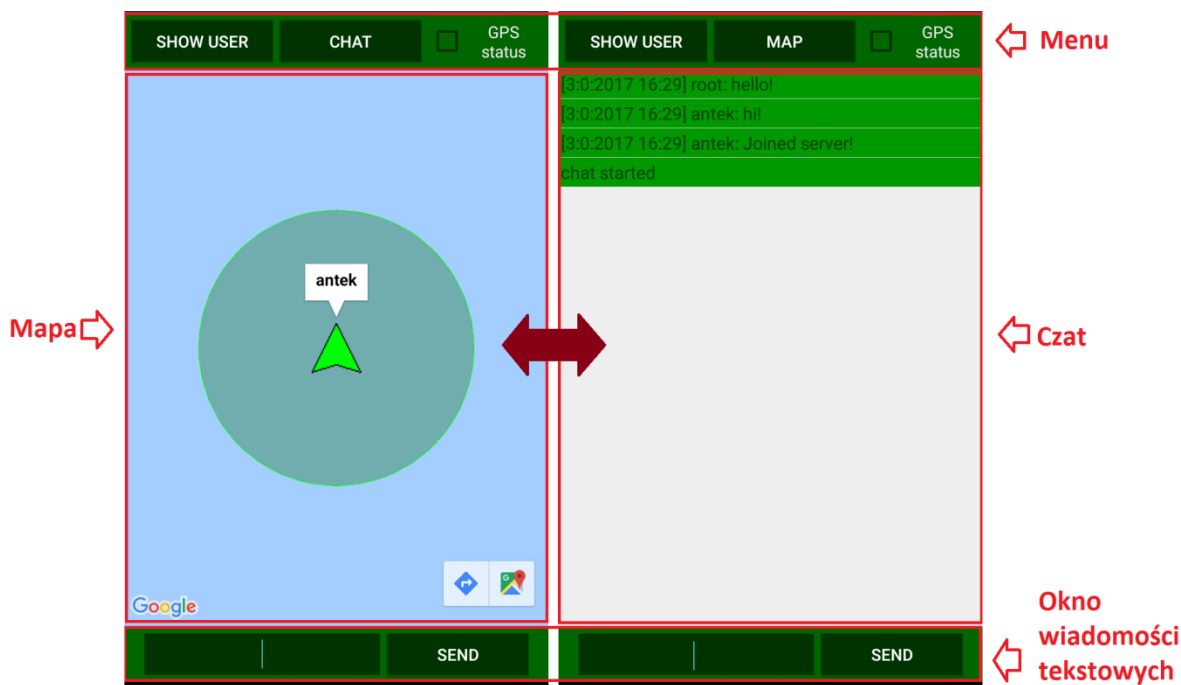
odpowiednich funkcji, które zajmują się ich przetwarzaniem. Aplikacja przetwarza między innymi ramki zawierające dane innych użytkowników, polecenia usuwania użytkowników offline, dane markerów itd. W przypadku niemożności uzyskania połączenia w czasie 10 sekund, aplikacja wyświetli okno dialogowe z informacją o problemie z jego nawiązaniem.

Po poprawnym nawiązaniu połączenia aplikacja kliencka tworzy obiekt typu `SystemUser` na podstawie danych z formularza, a następnie obiekt jest serializowany do postaci kodu XML takiego jak na rysunku 4.2. Otrzymany ciąg bajtów jest wysyłany do serwera, a aplikacja czeka na odpowiedź. Odpowiedź przychodzi w postaci takiej samej ramki, jak była wysłana, po czym jest deserializowana. Kluczowe jest pole `LoggedIn` definiujące czy logowanie powiodło się, czy też nie. W przypadku pomyślnego zalogowania inicjowany jest obiekt klasy `RoomActivity`. W przeciwnym wypadku wyświetla się okno dialogowe z informacją o niewłaściwych danych logowania, po czym można przystąpić do wprowadzenia nowych danych do formularza.

W pierwszej kolejności po utworzeniu okna `RoomActivity`, aplikacja inicjuje klasę `UserLocationListener`, która odpowiada za ustalanie pozycji użytkownika. Wykorzystuje ona do tego GPS i AGPS. GPS (Global Positioning System) jest to system geolokalizacji satelitarnej. Korzysta on z satelitów orbitujących dookoła Ziemi oraz stacji naziemnych kontrolujących i monitorujących. AGPS (Assisted GPS) jest systemem wyznaczającym pozycję na podstawie sieci telefonii komórkowej. Podstawową jego funkcją jest wsparcie systemu GPS, w postaci skrócenia czasu potrzebnego na połączenie z satelitami. Dodatkowo klasa ta korzysta z akcelerometru i czujnika magnetycznego, które pozwalają na wyznaczenie azymutu. Jeżeli użytkownik nie posiada odpowiednich ustawień lokalizacji, aplikacja otwiera okno dialogowe, które pozwala na przejście do odpowiednich ustawień urządzenia. Przy każdej aktualizacji położenia przez klasę `UserLocationListener`, wysyłana jest do serwera ramka `Client` taka jak na rysunku 4.3, otrzymana w wyniku serializacji danych użytkownika.

`RoomActivity` tworzy właściwe okno aplikacji, udostępniające szereg funkcjonalności oferowanych użytkownikowi przez system. `RoomActivity` składa się z czterech fragmentów, takich jak menu, pole wysyłania wiadomości tekstowych, czat oraz mapa. Mapa i czat są zarządzane przez klasę `FragmentManager` i w sposób dynamiczny są wyświetlane na przemian w zależności od potrzeb użytkownika. Układ poszczególnych fragmentów jest zaprezentowany na rysunku 4.9.

Rys. 4.9. Główny interfejs aplikacji (RoomActivity).



Źródło: Opracowanie własne.

Menu jest tworzone przy użyciu klasy FragmentMenu i służy do prostego zarządzania oknem, poprzez przycisk pozwalający na przełączanie nawzajem ze sobą fragmentów menu i czatu. Dodatkowo posiada przycisk, który służy do pokazania aktualnej pozycji użytkownika. Informuje również o statusie GPS, tzn. czy aplikacja odnalazła sygnał satelitów systemu GPS.

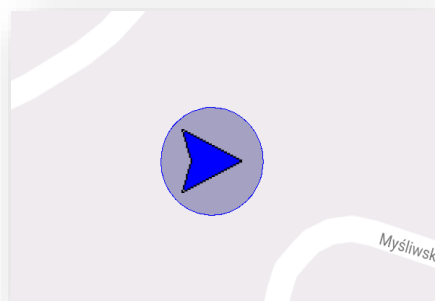
Pole wysyłania wiadomości jest tworzone przy użyciu klasy FragmentMessageBox. Zawiera ono formularz składający się z jednego pola i przycisku pozwalającego na wysłanie wiadomości tekstowej do serwera. Treść wiadomości jest umieszczana w polu Message i wysyłana wraz z danymi użytkownika za pomocą ramki Client takiej jak na rysunku 4.3, otrzymanej w wyniku serializacji danych użytkownika.

Czat jest tworzony przy użyciu klasy FragmentChat. Wyświetla on przy pomocy ListView wiadomości tekstowe wysłane przez użytkowników. Wiadomości zostają zapisane w obiekcie klasy ListArray wraz z loginem autora i datą wysłania wiadomości.

Mapa jest tworzona przy użyciu klasy FragmentMap i stanowi najbardziej rozbudowany element aplikacji. Wyświetlana jest za pomocą biblioteki Google Maps Android Api. Mapa oferuje szereg funkcjonalności użytkownikowi, takich jak określenie swojej pozycji, pozycji innych użytkowników, wyświetlanie znaczników naniesionych na mapę przez innych użytkowników oraz nanoszenie i usuwanie własnych znaczników. Pozycja użytkownika jest oznaczana znacznikiem, który jest reprezentowany przez niebieską strzałkę. Dodatkowo dookoła znacznika zostaje

naniesiony półprzezroczysty okrąg w kolorze niebieskim, który określa, z jaką dokładnością wyznaczono pozycja użytkownika. Znacznik jest zaprezentowany na rysunku 4.10.

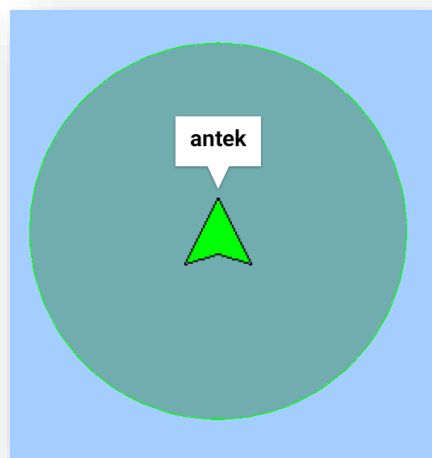
Rys. 4.10. Znacznik użytkownika.



Źródło: Opracowanie własne.

Pozycje innych użytkowników są reprezentowane podobnie jak pozycja użytkownika, z tą różnicą, że strzałka oraz okrąg są w kolorze zielonym. Dodatkowo po kliknięciu znacznika, wyświetla się nad nim login użytkownika, który jest widoczny przez krótki czas. Znacznik reprezentujący innych użytkowników jest zaprezentowany na rysunku 4.11.

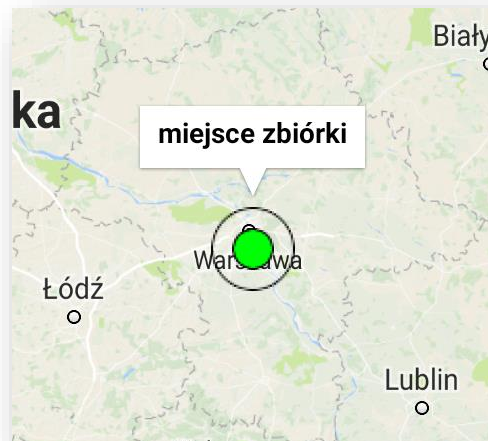
Rys. 4.11. Znacznik reprezentujący kolejnego użytkownika.



Źródło: Opracowanie własne.

Znaczniki oznaczające ważne miejsca są reprezentowane przez zieloną kropkę. Po kliknięciu tego znacznika wyświetla się tekst, jaki został przypisany do niego przez twórcę go użytkownika, który będzie widoczny przez niedługi czas. Znacznik ważnych miejsc jest zaprezentowany na rysunku 4.12.

Rys. 4.12. Znacznik reprezentujący ważne miejsce.



Źródło: Opracowanie własne.

Ponowne kliknięcie, zanim opis zniknie, powoduje otwarcie okna dialogowego, które wyświetla opis znacznika i oferuje możliwość usunięcia tego znacznika. Wybranie opcji usuwania znacznika otwiera kolejne okno dialogowe, upewniające się, że użytkownik na pewno chce usunąć wybrany znacznik, po czym aplikacja usuwa znacznik z mapy i wysyła ramkę do serwera z poleceniem usunięcia znacznika, taką jak na rysunku 5.4. Ramka jest otrzymana po modyfikacji odpowiednich danych marker i ich serializacji. W celu dodania nowego znacznika należy nacisnąć wybrany punkt na mapie i przytrzymać go, po czym otworzy się okno dialogowe pytające o chęć umieszczenia w wybranym punkcie markera. Po zatwierdzeniu zamyka się okno dialogowe i otwiera kolejne, w którym trzeba wprowadzić opis punktu, a następnie go zatwierdzić. Następnie aplikacja dodaje znacznik na mapie i wysyła ramkę do serwera z poleceniem dodania znacznika, taką jak na rysunku 5.4. Ramka powstaje w wyniku serializacji danych znacznika do postaci XML.

Zakończenie

Końcowym efektem projektu jest działający system, gotowy na obsługę wielu użytkowników. System umożliwia wymianę kluczowych informacji z punktu widzenia rozgrywki w grach terenowych typu Paintball. Znaczniki reprezentujące graczy na mapie pozwalają na łatwe ustalenie pozycji sojuszników. Użytkownicy dodatkowo posiadają do dyspozycji czat, za pomocą którego mogą wymieniać kluczowe informacje dotyczące bieżącej rozgrywki. Gracze mają możliwość umieszczania specjalnych znaczników reprezentujących kluczowe pozycje, które mogą reprezentować miejsca zbiórki, pozycje przeciwnika, itp. Po dodaniu, markery natychmiastowo wyświetlają się na mapie wszystkim sojusznikom. Taka funkcjonalność pozwala na łatwiejszą i szybszą wymianę informacji, co na równi umożliwia efektywną koordynację działań drużyny, jak i poszczególnych graczy.

Podczas tworzenia aplikacji serwerowej została napisana aplikacja testowa, dzięki której przeprowadzono symulacje jednoczesnej obsługi około 10 użytkowników. Próby okazały się pomyślnie i system działa poprawnie. Na każdego gracza przypada jeden wątek nasłuchujący przychodzące wiadomości oraz jeden wątek, który przetwarza oraz propaguje do pozostałych graczy informacje dotyczące rozgrywki. Takie rozwiązanie sprawia, że biblioteka serwerowa jest wysoce skalowalna, to znaczy w sposób automatyczny dostosowuje się do zasobów sprzętowych maszyny, na której uruchomiona została aplikacja serwerowa.

Zwinne metodyki wytwarzania oprogramowania pozwoliły na łatwiejszy podział zadań oraz sprawniejszą ich realizację. Jedna iteracja obejmowała trzytygodniową jednostkę czasu, po której następowało podsumowanie sprintu oraz przydział nowych zadań. Takie rozwiązanie pozwala na lepszą kontrolę nad zadaniami oraz prawidłową komunikację pomiędzy członkami zespołu oraz opiekunem pracy. Cały projekt wraz z przygotowaniem oraz realizacją zajął 7 miesięcy.

Załączniki

Kody źródłowe

Przykład kodu asynchronicznego pokazującego wsparcie języka C#.

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace PracaInzynierska
{
    class Program
    {
        TaskCompletionSource<object> _tcs = new TaskCompletionSource<object>();

        public void DisplayPrimeCountsFrom(int i)
        {
            Console.WriteLine("Metoda DisplayPrimeCountsFrom()");
            var awaiter = GetPrimesCountAsync(i * 100000 + 2, 1000000).GetAwaiter();
            awaiter.OnCompleted(() =>
            {
                Console.WriteLine("ID wątku {0}. Rezultat: {1}\n",
                    Thread.CurrentThread.ManagedThreadId,
                    awaiter.GetResult());

                if (i++ < 3) DisplayPrimeCountsFrom(i);
                else _tcs.SetResult(null);
            });
        }

        Task DisplayPrimeCounts()
        {
            DisplayPrimeCountsFrom(0);
            return _tcs.Task;
        }

        async Task DisplayPrimeCountsAsync()
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("Metoda DisplayPrimeCountsAsync()");
                int x = await GetPrimesCountAsync(i * 100000 + 2, 1000000);
                Console.WriteLine("ID wątku {0}. Rezultat: {1}\n",
                    Thread.CurrentThread.ManagedThreadId, x);
            }
        }

        Task<int> GetPrimesCountAsync(int s, int x)
        {
            Console.WriteLine("Metoda GetPrimesCountAsync()");
            return Task.Run(
                () =>
                    ParallelEnumerable.Range(s, x).Count(n => Enumerable.
                        Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0))
            );
        }

        public async void Start()
        {
            Console.WriteLine("Początek metody nieblokującej Start()");

            // ze wsparciem środowiska .NET
            await DisplayPrimeCountsAsync();
            Console.WriteLine("Koniec metody Start()");
        }
    }
}
```

```

        // bez wsparcia środowiska .NET
        DisplayPrimeCounts().ContinueWith(
            (t) => Console.WriteLine("Koniec metody Start()"));
    }

    public void OtherOperations() {

        Console.WriteLine("Metoda OtherOperations(). ID wątku {0}",
            Thread.CurrentThread.ManagedThreadId);
    }

    static void Main(string[] args)
    {
        var program = new Program();
        Console.WriteLine("Wątek główny : {0}", Thread.CurrentThread.ManagedThreadId);

        program.Start();
        program.OtherOperations();

        Console.WriteLine("!----> Koniec metody Main <----!");
        Console.ReadKey();
    }
}

```

Źródło: Opracowanie własne na podstawie: Albahari J., Albahari B.: *C# 6.0 in a Nutshell*, O'Reilly Media, Sebastopol, 2015, s. 590-591.

Wykaz literatury

1. Albahari J., Albahari B.: *C# 6.0 in a Nutshell*, O'Reilly Media, Sebastopol, 2015
2. Skeet J.: *C# in depth*, Manning, Shelter Island NY, 2014
3. Albahari J., *Threading in C#*, O'Reilly Media, 2006-2010
4. Hummel J., *Async and Parallel Programming*, Chicago .NET User Group, 2013
5. Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
6. Lin J., Dyer C., *Data-Intensive Text Processing with MapReduce*, University of Maryland, 2010.
7. Arpaci-Dusseau, Remzi H., *Operation Systems: Three Easy Pieces*, Arpaci-Dusseau Books, 2014
8. Meunier R., *The Pipes and Filters Architecture*, volume 1 of "Pattern Languages of Program Design", chapter 22. Addison-Wesley, 1995
9. Daryl A. Swade and James F. Rose. *A flexible pipeline data-processing environment. In Proceedings of the AIAA/USU Conference on Small Satellites*, 1998
10. JAXB Reference Implementation - Project Kenai, <https://jaxb.java.net/>, (dostęp 20.10.2016 r.).
11. XStream - About Xstream, <http://x-stream.github.io/>, (dostęp 20.10.2016 r.).
12. Nauka programowania z Microsoft Developer Network | MSDN, <https://msdn.microsoft.com/>, (data dostępu 20.10.2016 r.).
13. Oracle Help Center, <https://docs.oracle.com/>, (dostęp 20.10.2016 r.).
14. Web Objects in XML, <http://woxserializer.sourceforge.net/>, (dostęp 20.10.2016 r.).

Wykaz rysunków

Rys. 1.1. Diagram stanów wątków.....	6
Rys. 1.2. Przepływ wzorca projektowego oczekujący.....	8
Rys. 1.3. Przepływu metody asynchronicznej przy użyciu async i await.	9
Rys. 1.4. Stos funkcji po wywołaniu metody asynchronicznej Start().	10
Rys. 1.5. Stos funkcji po wywołaniu metody OtherOperations().	10
Rys. 1.6. Stos funkcji po wykonaniu metody OtherOperations().	11
Rys. 1.7. Równoległość zadań w przypadku dwóch zbiorów danych oraz trzech zadań.	12
Rys. 1.8. Komponenty bibliotek do programowania równoległego.	12
Rys. 1.9. Przykład podziału równoległego wykonywania pracy.	13
Rys. 1.10. Dynamiczny podział zbioru danych.	14
Rys. 1.11. Przekrojowy statyczny podział zbioru danych.	14
Rys. 1.12. Progresywny statyczny podział zbioru danych.	15
Rys. 1.13. Haszujący podział zbioru danych.	15
Rys. 2.1. Wzorzec projektowy Pipeline.....	17
Rys. 2.2. Drzewo zależności wątków we wzorcu projektowym Dataflow.	17
Rys. 2.3. Wzorzec projektowy Producer-Consumer.	20
Rys. 2.4. Przepływ danych we wzorcu MapReduce.	21
Rys. 3.1. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą klasy XmlSerializer.	22
Rys. 3.2. Przykładowy kod XML obrazujący brak przechowywania informacji o referencjach przez XmlSerializer.	24
Rys. 3.3. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą klasy XmlEncoder.....	25
Rys. 3.4. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą biblioteki Xstream.	25
Rys. 3.5. Przykładowy kod XML otrzymany w wyniku serializacji z pomocą biblioteki Jaxb.	27
Rys. 3.6. Przykład serializacji obiektu przechowującego tablicę z użyciem biblioteki Jaxb.	27
Rys. 4.1. Schemat projektu.....	31
Rys. 4.2. Ramka wysyłana podczas logowania do systemu.....	32
Rys. 4.3. Ramka wysyłana do serwera po zalogowaniu do systemu.....	32
Rys. 4.4. Ramka do dodawania markera na mapie.	33
Rys. 4.5. Ramka informująca o usunięciu gracza z pokoju.	33
Rys. 4.6. Aplikacja serwerowa.	34
Rys. 4.7. Aplikacja do testowania napisana w języku C#.	35
Rys. 4.8. Wygląd okna LoginActivity.....	36
Rys. 4.9. Główny interfejs aplikacji (RoomActivity).	38
Rys. 4.10. Znacznik użytkownika.....	39
Rys. 4.11. Znacznik reprezentujący kolejnego użytkownika.	39
Rys. 4.12. Znacznik reprezentujący ważne miejsce.	40

Wykaz tabel

Tabela 4.1. Podział prac w projekcie	30
Tabela 4.2 Metryki kodu dotyczące języka C#.....	30

Wykaz listingów

Listing 1.1. Tworzenie nowego wątku za pomocą metody statycznej Factory.....	5
Listing 1.2. Tworzenie nowego wątku za pomocą konstruktora klasy Task.....	5
Listing 1.3. Przykład metody asynchronicznej.....	9
Listing 1.4. Przykład kodu bez wsparcia słów kluczowych async i await.....	10
Listing 1.5. Przykład użycia zapytania Select.....	13
Listing 2.1. Przykład zastosowania wzorca projektowego Dataflow.....	18
Listing 2.2. Prototyp funkcji ContinueWhenAll.....	18
Listing 2.3. Przykład zastosowania wzorca projektowego WaitAllOneByOne.....	19
Listing 2.4. Pseudokod użycia wzorca projektowego MapReduce.....	21
Listing 4.1. Przykład inicjalizacji subskrypcji zdarzeń serwerowych.....	34