Spis treści

[1. Wstęp 4](#_Toc120360955)

[2. Istniejące rozwiązania 5](#_Toc120360956)

[3. Wzorce projektowe 6](#_Toc120360957)

[3.1. Singleton 6](#_Toc120360958)

[3.2. Fabryka 7](#_Toc120360959)

[3.3. Fabryka abstrakcyjna 7](#_Toc120360960)

[3.4. Prototyp 9](#_Toc120360961)

[3.5. Fasada 10](#_Toc120360962)

[3.6. Pusty Obiekt 11](#_Toc120360963)

[3.7. Kontener IoC 12](#_Toc120360964)

[3.7.1. Inversion of Control 12](#_Toc120360965)

[3.7.2. Dependency Injection 12](#_Toc120360966)

[3.7.3. Kontener IoC 12](#_Toc120360967)

[4. Kompilator Roslyn 13](#_Toc120360968)

[4.1. Model zestawu SDK .NET Compiler Platform 13](#_Toc120360969)

[4.2. Drzewo składni 13](#_Toc120360970)

[4.2.1. Węzły składni 14](#_Toc120360971)

[4.2.2. Tokeny składniowe 14](#_Toc120360972)

[4.2.3. Trivia składniowa 14](#_Toc120360973)

[4.3. Semantyka 14](#_Toc120360974)

[4.3.1. Kompilacja 15](#_Toc120360975)

[4.3.2. Symbole 15](#_Toc120360976)

[4.4. Generator źródeł 15](#_Toc120360977)

[5. Projekt 17](#_Toc120360978)

[6. Implementacja 18](#_Toc120360979)

[6.1. Użyte biblioteki 18](#_Toc120360980)

[6.1.1. Microsoft.CodeAnalysis 18](#_Toc120360981)

[6.1.2. System.Collections 19](#_Toc120360982)

[6.1.3. System.Linq 19](#_Toc120360983)

[6.2. Generatory 19](#_Toc120360984)

[6.2.1. Singleton 21](#_Toc120360985)

[6.2.2. Factory 22](#_Toc120360986)

[6.2.3. AbstractFactory 23](#_Toc120360987)

[6.2.4. Prototype 23](#_Toc120360988)

[6.2.5. Facade 23](#_Toc120360989)

[6.2.6. NullObject 23](#_Toc120360990)

[6.2.7. ContainerIoC 23](#_Toc120360991)

[6.3. Testy 23](#_Toc120360992)

[7. Porównanie z innymi rozwiązaniami 24](#_Toc120360993)

[8. Wyzwania 25](#_Toc120360994)

[9. Wnioski 26](#_Toc120360995)

[10. Bibliografia 27](#_Toc120360996)

[11. Spis rysunków 29](#_Toc120360997)

1. Wstęp

Celem tej pracy jest użycie kompilatora Roslyn do stworzenia rozszerzenia generującego kod typowych wzorców projektowych występujących w oprogramowaniu biznesowym. Wzorce projektowe to typowe rozwiązania problemów często napotykanych podczas projektowania oprogramowania. Każdy z nich stanowi plan, który po odpowiednim dostosowaniu pomaga poradzić sobie z konkretnym problemem w projekcie kodu. Wzorce projektowe różnią się poziomem skomplikowania, szczegółowością oraz skalą w jakiej można je zastosować.[[1]](#footnote-1) Wzorce, które zostały uwzględnione w tej pracy, zostały omówione w rozdziale 2.

Użyty w tej pracy kompilator Roslyn, zapewnia dostęp do szczegółowego modelu kodu aplikacji, który został utworzony przez kompilatory. Taki model, kompilatory tworzą, poprzez weryfikację składni i semantyki kodu aplikacji, oraz używają tego modelu do tworzenia wykonywalnych danych wyjściowych z kodu źródłowego.

Przez ostatnie lata, aby zwiększyć produktywność, coraz częściej programiści używają zintegrowanych funkcji środowiska projektowego (IDE), takich jak: IntelliSense, refaktoryzacja, inteligentna zmiana nazwy, czy wyszukiwania referencji. Korzystają oni z narzędzi do analizy kodu, aby poprawić jakość oraz z generatorów kodu, aby ułatwić tworzenie aplikacji. Z czasem, narzędzia te, stają się inteligentniejsze. Potrzebują dostępu do coraz większej liczby modeli tworzonych przez kompilatory podczas przetwarzania kodu aplikacji. Jest to podstawowa funkcja kompilatora Roslyn, czyli udostępnianie użytkownikom końcowym informacji o kodzie aplikacji.[[2]](#footnote-2) Pojęcia związane z Roslyn zostały omówione w rozdziale 3.

Generatory źródeł, w ramach Roslyn APIs, umożliwiają programistom C# sprawdzanie kodu podczas kompilowania[[3]](#footnote-3).

Więcej informacji na temat generatorów źródeł zostało zamieszczone w rozdziale 3.3.

1. Istniejące rozwiązania

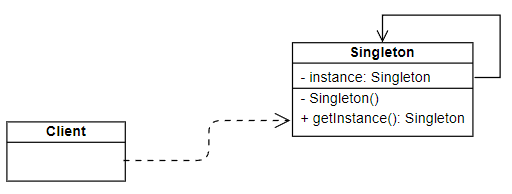
W tym rozdziale zostaną przedstawione istniejące rozwiązania, które wspomagają programistę w implementacji wzorców projektowych. Te istniejące rozwiązania będą stanowić podstawę dla analizy generatorów.

1. Wzorce projektowe

W tym rozdziale zostaną przedstawione wzorce projektowe, do których zostały zaimplementowane generatory kodów.

* 1. Singleton

Singleton to kreacyjny wzorzec projektowy, którego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji. Ponadto singleton zapewni globalny dostęp do stworzonego obiektu.



Il. 1. Singleton - diagram UML

Wzorca Singleton należy używać w następujących warunkach:

1. jeśli musi istnieć dokładnie jeden egzemplarz klasy dostępny klientom w znanym miejscu,
2. kiedy potrzebna jest możliwość rozszerzenia jedynego egzemplarza przez tworzenie podklas, a klienci powinni móc korzystać ze wzbogaconego egzemplarza bez konieczności wprowadzania zmian w ich kodzie.[[4]](#footnote-4)

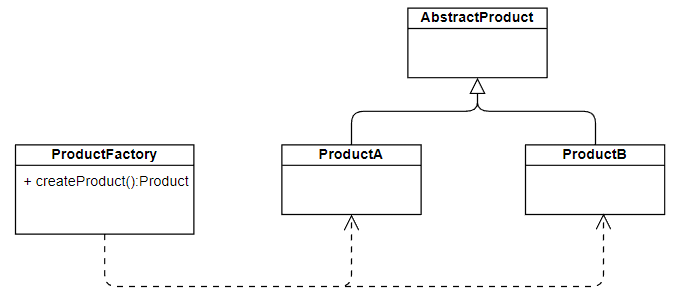
Wzorzec ten zapewnia kilka korzyści:

1. zapewnia kontrolę dostępu do jedynego egzemplarza,
2. pozwala zmniejszyć przestrzeń nazw,
3. umożliwia dopracowywanie operacji i reprezentacji,
4. umożliwia określenie dowolnego limitu liczby egzemplarzy,
5. jest bardziej elastyczny od operacji statycznych.[[5]](#footnote-5)

Singleton to najbardziej znienawidzony wzorzec projektowy w historii stosowania wzorców.[[6]](#footnote-6)

* 1. Fabryka

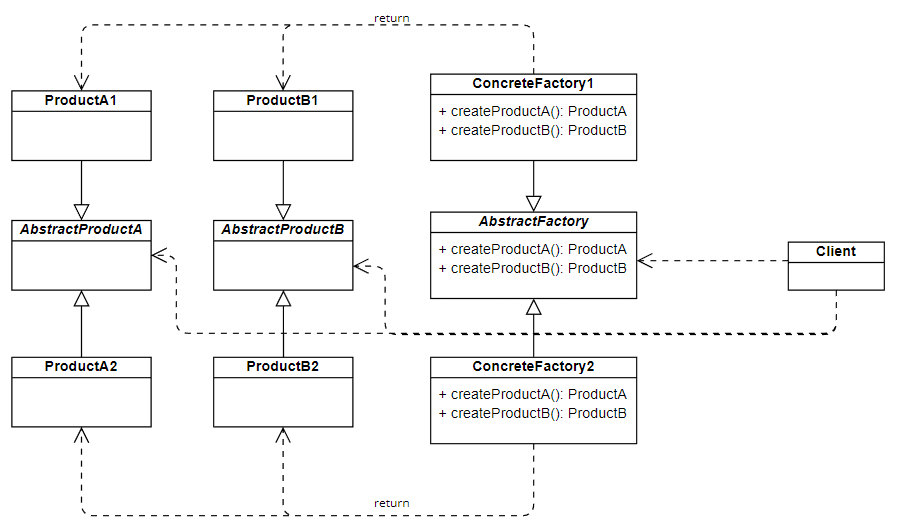
Fabryka to kreacyjny wzorzec projektowy, który udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej. Jednocześnie pozwala podklasom zmieniać typ tworzonych obiektów. Jest to niejednoznaczny termin mogący oznaczać metodę lub klasę która ma za zadanie coś wytworzyć. W tej pracy została użyta Fabryka prosta[[7]](#footnote-7), która przedstawia klasę posiadającą jedną metodę kreacyjną z obszerną instrukcją warunkową, która na podstawie parametrów metody decyduje jakiej klasy produktu instancje stworzyć i zwrócić.



Il. 2. Fabryka - diagram UML

* 1. Fabryka abstrakcyjna

Fabryka abstrakcyjna to kreacyjny wzorzec projektowy, którego celem jest dostarczenie interfejsu do tworzenia różnych obiektów jednego typu dla tej samej rodziny klas bez specyfikowania ich konkretnych klas.



Il. 3. Fabryka Abstrakcyjna - diagram UML

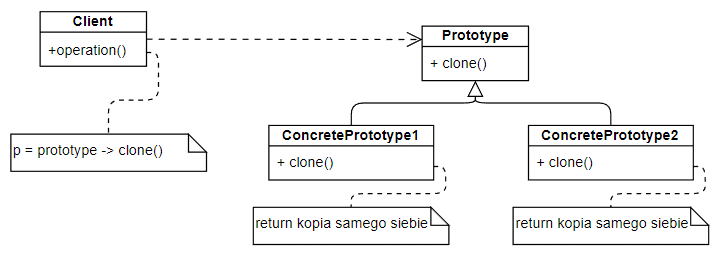
Ten wzorzec należy stosować w następujących warunkach:

1. kiedy system powinien być niezależny od sposobu tworzenia, składania i reprezentowania jego produktów,
2. jeśli system należy skonfigurować za pomocą jednej z wielu rodzin produktów,
3. jeżeli powiązane obiekty-produkty z jednej rodziny są zaprojektowane do wspólnego użytku i trzeba wymusić jednoczesne korzystanie z tych obiektów,
4. kiedy programista chce udostępnić klasę biblioteczną produktów i ujawnić jedynie ich interfejsy, a nie implementacje.

Wzorzec Fabryka abstrakcyjna ma następujące zalety i wady:

1. izoluje klasy konkretne,
2. ułatwia zastępowanie rodzin produktów,
3. ułatwia zachowanie spójności między produktami,
4. utrudnia dodawanie obsługi produktów nowego rodzaju. [[8]](#footnote-8)
   1. Prototyp

Prototyp to kreacyjny wzorzec projektowy, który umożliwia kopiowanie istniejących już obiektów bez tworzenia zależności pomiędzy kodem programu, a klasami obiektu.



Il. 4. Prototyp - diagram UML

Ten wzorzec należy stosować w następujących warunkach:

1. klasy tworzonych egzemplarzy są określane w czasie wykonywania programu,
2. programista chce uniknąć tworzenia hierarchii klas fabryk odpowiadającej hierarchii klas produktów,
3. egzemplarze klasy mogą przyjmować jeden z niewielu stanów.

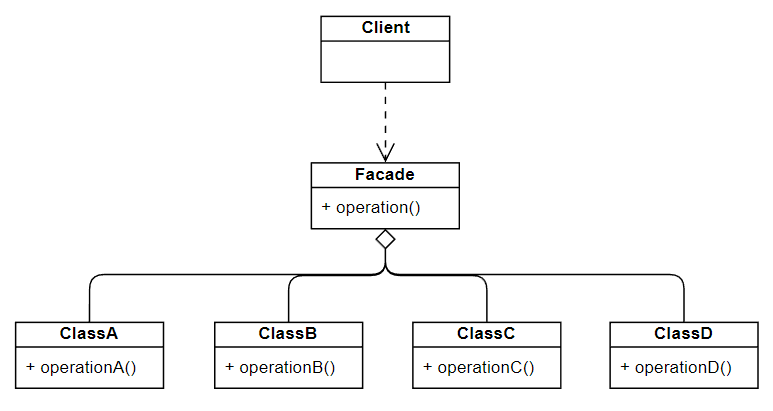
Korzyści które płynące z zastosowania wzorca Prototyp to:

1. możliwość dodawania i usuwania produktów w czasie wykonywania programu,
2. możliwość określania nowych obiektów przez zmianę wartości,
3. możliwość określania nowych obiektów przez modyfikowanie struktur,
4. zmniejszanie liczby podklas,
5. możliwość dynamicznego konfigurowania aplikacji za pomocą klas.

Główną wadą wzorca Prototyp jest to, że w każdej podklasie trzeba zaimplementować operację Clone(). Może to sprawić problemy. Dodanie tej operacji jest trudne, jeśli dane klasy już istnieją. Zaimplementowanie operacji Clone() może okazać się skomplikowane także wtedy, jeżeli w tych klasach używane są obiekty nieobsługujące kopiowania lub mają referencje cykliczne.[[9]](#footnote-9)

* 1. Fasada

Wzorzec fasada to strukturalny wzorzec projektowy, którego celem jest stworzenie prostego interfejsu dla zestawu klas. Fasada ułatwia wykonywanie złożonych funkcjonalności, korzystając z różnych obiektów i przy tym ukrywa szczegóły implementacji.



Il. 5. Fasada - diagram UML

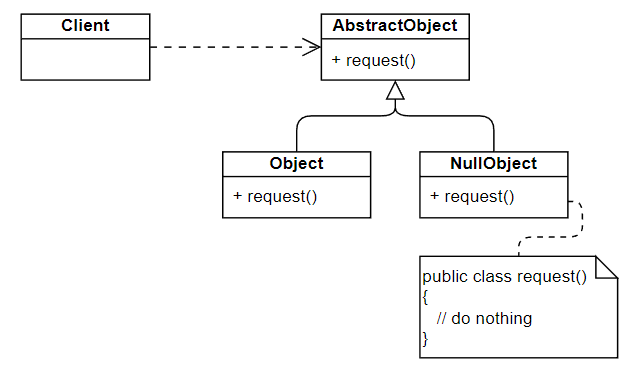
Wzorca Fasada należy stosować w następujących warunkach:

1. gdy programista chce udostępnić prosty interfejs do złożonego podsystemu,
2. gdy występuje wiele zależności między klientami i klasami z implementacją abstrakcji,
3. gdy programista chce podzielić podsystem na warstwy.

Stosowanie wzorca Fasada daje następujące korzyści:

1. oddziela klienta od komponentów podsystemu, przez co zmniejsza liczbę obiektów, których używa klient i ułatwia korzystanie z podsystemu,
2. pomaga zachować luźne powiązania między podsystemem i klientami,
3. nie umożliwia aplikacjom korzystania z klas podsystemów, jeśli jest to konieczne.
   1. Pusty Obiekt

Wzorzec Pusty Obiekt to behawioralny wzorzec projektowy, który ma na celu realizację braku obiektu poprzez dostarczenie materialnej alternatywy, która oferuje domyślnie działanie puste, czyli niewykonujące żadnych operacji.



Il. 6. Pusty Obiekt - diagram UML

Zalety tego wzorca to:

1. eliminacja zbędnych i powtarzających się instrukcji warunkowych,
2. umożliwia wielokrotne wykorzystanie pustego zachowania.[[10]](#footnote-10)
   1. Kontener IoC
      1. Inversion of Control

Inversion of Control (IoC) czyli Odwrócenie Sterowania to paradygmat programowania polegający na zmianie odpowiedzialności pewnych części aplikacji. Zastosowanie IoC powoduje, że zewnętrzny framework wywołuje kod programisty w odpowiednich momentach.

* + 1. Dependency Injection

Dependency Injection, czyli Wstrzykiwanie Zależności, jest wzorcem projektowym oryginalnie pochodzącym ze środowiska programistów Java. Jest to implementacja paradygmatu Inversion of Control. Jest to implementacja na tyle popularna, że powszechnie jest używanie tych dwóch pojęć zamiennie. Podstawową zasadą działania Dependency Injection jest posiadanie serwisu, który zajmuje się uzupełnianiem potrzebnych zależności.

* + 1. Kontener IoC

Kontener IoC automatyzuje proces tworzenia obiektów i zapewnia ich zależności. Posiada on globalny rejestr serwisów istniejących w aplikacji, oraz zajmuje się tworzeniem obiektów zgodnie z wymaganiami. Zarządza też czasem życia obiektów które powołuje. Programista musi tylko określić od czego dana klasa zależy. Platforma .NET udostępnia wbudowany kontener – IServiceProvider.[[11]](#footnote-11)

1. **Kompilator** Roslyn

Roslyn to implementacja typu open-source kompilatorów C#, jak i Visual Basic, który umożliwia tworzenie analizatorów i poprawek kodu. Analizatory interpretują składnię i semantykę w celu wykrywania błędów, które należy poprawić. Poprawki kodu natomiast zawierają co najmniej jedną sugerowaną poprawkę do rozwiązywania błędów kodowania znalezionych przez analizatory lub diagnostykę kompilatora.[[12]](#footnote-12)

W tym rozdziale zostaną omówione najważniejsze zagadnienia związane z kompilatorem Roslyn.

* 1. Model zestawu SDK .NET Compiler Platform

Kompilatory przetwarzają kod, zgodnie z regułami ustrukturyzowanymi, które często różnią się od sposobu odczytywania i zrozumienia kodu przez ludzi. Podstawowa wiedza na temat modelu używanego przez kompilatory jest niezbędna do zrozumienia interfejsów API używanych podczas kompilowania narzędzi opartych na platformie Roslyn.

Zestaw SDK kompilatora platformy .NET składa się z kilku warstw interfejsów API:

1. interfejsów API kompilatora,
2. interfejsów API diagnostycznych,
3. interfejsów API skryptorów,
4. interfejsów API obszarów roboczych.[[13]](#footnote-13)
   1. Drzewo składni

Drzewo składni jest podstawową niezmienną strukturą danych uwidacznianą przez interfejsy API kompilatora. Reprezentuje ono leksykalną i składniową strukturę kodu źródłowego. Każde drzewo składni składa się z węzłów, tokenów i triwii.[[14]](#footnote-14)

* + 1. Węzły składni

Węzły składni są jednym z podstawowych elementów drzew składniowych. Reprezentują one konstrukcje składniowe, takie jak deklaracje, instrukcje, klauzule i wyrażenia.[[15]](#footnote-15)

* + 1. Tokeny składniowe

Tokeny składniowe reprezentują najmniejsze fragmenty składni kodu. Składają się one ze słów kluczowych, identyfikatorów, literałów i interpunkcji. Nigdy nie są one rodzicami innych węzłów lub tokenów.[[16]](#footnote-16)

* + 1. Trivia składniowa

Trivia składniowa reprezentuje część tekstu źródłowego, które w dużej mierze są nieistotne dla działania kodu, takie jak białe znaki, komentarze czy dyrektywy preprocesora. Trivia, podobnie jak tokeny, to typy wartości.[[17]](#footnote-17)

* 1. Semantyka

Model semantyczny hermetyzuje reguły języka, zapewniając łatwy sposób poprawnego dopasowania identyfikatorów do prawidłowego elementu programu. Reprezentuje on wszystkie informacje semantyczne dla pojedynczego pliku źródłowego.[[18]](#footnote-18)

* + 1. Kompilacja

Kompilacja jest reprezentacją wszystkiego, co jest potrzebne do skompilowania programu w języku C# lub Visual Basic, który zawiera wszystkie odwołania do zestawu, opcje kompilatora oraz pliki źródłowe. Ponieważ wszystkie te informacje znajdują się w jednym miejscu, elementy zawarte w kodzie źródłowym można opisać bardziej szczegółowo. Każdy zadeklarowany typ, składowa lub zmienna jest reprezentowana jako symbol. Kompilacja udostępnia różne metody, które ułatwiają znajdowanie symboli.[[19]](#footnote-19)

* + 1. Symbole

Symbol reprezentuje odrębny element zadeklarowany przez kod źródłowy lub zaimportowany z zestawu jako metadane. Każda przestrzeń, nazwa, typ, metoda, właściwość, pole, zdarzenie, parametr lub zmienna lokalna jest reprezentowana przez symbol.[[20]](#footnote-20)

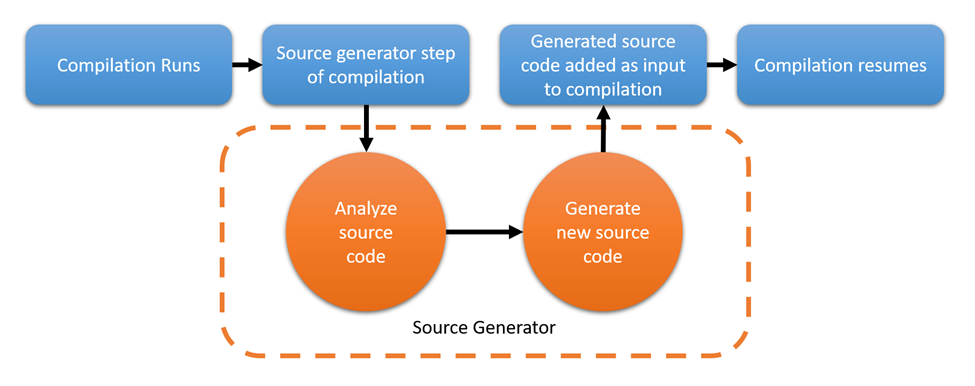
* 1. Generator źródeł

Generatory źródeł są udostępniane w ramach zestawu SDK .NET Compiler Platform. Mogą one tworzyć nowe pliki źródłowe języka C#, które są na bieżąco dodawane do kompilacji użytkownika. Generatory umożliwiają wykonanie dwóch głównych czynności:

1. pobieranie obiektu kompilacji, który reprezentuje cały kod użytkownika,
2. generowanie plików źródłowych języka C#, które można dodać do obiektu kompilacji podczas kompilacji.

Kod użytkownika jest sprawdzany przy użyciu wszystkich zaawansowanych metadanych kompilatora podczas kompilacji. Następnie generator emituje kod języka C# z powrotem do tej samej kompilacji. Generatory źródeł mogą być traktowane jako analizatory, które emitują kod źródłowy języka C#.

Generatory źródeł działają jako faza kompilacji, co zostało przedstawione na ilustracji 1. [[21]](#footnote-21)



Il. 7. Działanie generatora źródeł jako faza kompilacji

Do generatorów źródeł, obecnie mogą być tylko używane zestawy .NET Standard 2.0. [[22]](#footnote-22)

1. Projekt

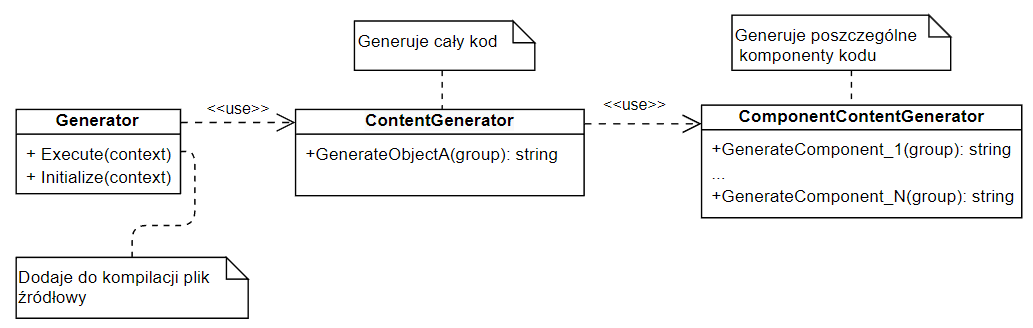
Projekt składa się z jednej biblioteki zawierające generatory odpowiadające każdemu uwzględnionemu w tej pracy wzorcowi projektowemu.

Każdy generator źródeł musi zaimplementować interfejs Microsoft.CodeAnalysis.ISourceGenerator oraz posiadać atrybut Microsoft.CodeAnalysis.GeneratorAttribute. Zaimplementowany interfejs posiada dwie metody: ISourceGenerator.Initialize oraz ISourceGenerator.Execute. Nie wszystkie generatory źródeł wymagają inicjowania i tak samo jest w tym projekcie, gdzie w każdym generatorze metoda ISourceGenerator.Initialize została pusta. Akcja generowania będzie wykonywana w metodzie ISourceGenerator.Execute.

Do metody ISourceGenerator.Execute jest przekazywany parametr typu GeneratorExecutionContext, który zawiera w sobie obecną kompilację, oraz umożliwia dodawanie nowych plików źródłowych. Kompilacja natomiast zawiera drzewo składni oraz model semantyczny, z których to zostały wzięte potrzebne dane.

Wybór generatora do odpowiedniego wzorca projektowego następuję poprzez oznaczenie klasy lub interfejsu atrybutem. Następnie generator, korzystając z modelu semantycznego, filtruje drzewo składni w celu odnalezienia tego atrybutu. Przefiltrowane dane zostają przesłane do odpowiednich klas, które zwracają kod źródłowy w postaci string. Na koniec, poprzez wywołanie metody GeneratorExecutionContext.AddSource, zostaje dodany nowy plik źródłowy do obecnej kompilacji.

Przybliżone działanie każdego generatora zostało przedstawione na Il. 8.



Il. 8. Uogólniony schemat UML generatora

Połączenie biblioteki generatorów z projektem, który ją wykorzystuje, następuje poprzez dodanie następującego polecenia do projektu:



Wygenerowane pliki źródłowe są dostępne w drzewie projektu pod następującą ścieżką:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Il. . Ścieżka Generatorów w projekcie

1. Implementacja

W tym rozdziale opisano użyte biblioteki, implementacje poszczególnych generatorów, oraz testy.

* 1. Użyte biblioteki

Do zaimplementowania generatora kodów wzorców projektowych zostały użyte następujące pakiety: Microsoft.CodeAnalysis, System.Collections, System.Linq.

* + 1. Microsoft.CodeAnalysis

Microsoft.CodeAnalysis jest to platforma kompilatora .NET („Roslyn”). Jest to pakiet all-in-one, czyli nadzbiór wszystkich pakietów.[[23]](#footnote-23) W tym projekcie zostały użyte następujące podpakiety:

1. Microsoft.CodeAnalysis.CSharp,
2. Microsoft.CodeAnalysis.Diagnostics,
3. Microsoft.CodeAnalysis.Text.
   * 1. System.Collections

System.Collections zawiera klasy, które definiują kolekcje ogólne, umożliwiające programistom tworzenie kolekcji o silnym typie. Takie rozwiązanie zapewnia większe bezpieczeństwo i wydajność typów niż niegeneryczne kolekcje. [[24]](#footnote-24)

* + 1. System.Linq

System.Linq udostępnia klasy i interfejsy, które pozwalają operować na kolekcjach obiektów, baz danych SQL, czy danych XML za pomocą zapytań, które używają Language-Integrated Query.

LINQ udostępnia dwa rodzaje zapisu zapytania:

1. notacja z kropką:



1. notacja standardowych zapytań:



W projekcie została użyta notacja z kropką.

* 1. Generatory

Do implementacji generatorów zostały utworzone bazowe statyczne klasy, które posłużyły do powtarzających się operacji we wszystkich generatorów. Jedną z tych klas jest DeclarationsSyntaxGenerator, która zawiera metody zwracające pogrupowane składnie klas lub interfejsów. Metody te, wykorzystując technologię LINQ, wyszukają w składni takie obiekty, które mają nałożony na siebie odpowiedni atrybut, a następnie zwracają pogrupowaną kolekcję.



Kolejną statyczną bazową klasą generatorów jest BaseNamesGenerator. Zawiera ona metody zwracające nazwy poszczególnych części generowanego kodu źródłowego, takie jak: nazwa interfejsu, nazwa klasy lub nazwy używanych bibliotek.



Ostatnią statyczną klasą bazową generatorów jest BaseCodeGenerator, która zawiera metody zwracające gotowe kawałki kodu źródłowego, które są wykorzystywane przez wszystkie generatory, takie jak: deklaracja klasy lub używane biblioteki wraz z przestrzenią nazw.



Do implementacji generatorów zostały także utworzone statyczne klasy pomocnicze, które odpowiednio manipulują kolekcją.



Pełne działanie generatorów zostanie przedstawione na przykładzie generatora wzorca Singleton.

* + 1. Singleton

Do generowania wzorca projektowego Singleton został utworzony atrybut SingletonAttribute. Dzięki zaimplementowanym wcześniej statycznym klasom bazowym, następuje pobranie z obecnej kompilacji pogrupowanej kolekcji składni klas, które zawierają atrybut SingletonAttribute.



Następnie, dla każdej składni zostaje wygenerowany kod źródłowy, zawierający klasę częściową Singleton, oraz dodany do obecnej kompilacji.



Generator generuje kod źródłowy w postaci string dzięki zaimplementowanej statycznej klasy SingletonContentGenerator, która wykorzystuje zaimplementowane metody w statycznej klasie SingletonContentComponentsGenerator.





Dla sprawdzenia poprawności działania generatora zostały utworzone testy jednostkowe, które zostały opisane dokładniej w podrozdziale 6.3., oraz testy manualne na przykładowej klasie Configuration.



Dla tej klasy zostanie automatycznie wygenerowany kod źródłowy wzorca projektowego Singleton.



* + 1. Factory

Do generowania wzorca projektowego Fabryka zostały utworzone dwa atrybuty: FactoryAttribute oraz FactoryProductAttribute. Pierwszym atrybutem programista oznacza interfejs, a drugim klasy, które implementują ten interfejs, oraz mają zostać uwzględnione w metodzie fabrycznej. Generator dla każdego interfejsu generuje trzy dodatkowe źródła: interfejs fabryki, klasę fabryki oraz typ wyliczeniowy, zawierający nazwy klas, które zostały oznaczone atrybutem FactoryProductAttribute.

* + 1. AbstractFactory

Do generowania wzorca projektowego Fabryka Abstrakcyjna zostały utworzone dwa atrybuty: AbstractFactoryAttribute i AbstractFactoryClassAttribute. Pierwszy atrybut jest nakładany na interfejs, oraz przyjmuje w konstruktorze nazwę wygenerowanego interfejsu fabryki abstrakcyjnej. Drugim atrybutem oznacza się klasy generowane przez tą fabrykę, oraz przyjmuje on w konstruktorze nazwę konkretnych fabryk, które implementują interfejs fabryki abstrakcyjnej.

Największym wyzwanie implementacyjnym dla tego generatora było wyselekcjonowanie odpowiednich klas pod odpowiednią fabrykę abstrakcyjną. Problem ten został rozwiązany za pomocą przeciążenia bazowej metody statycznej filtrującej pogrupowaną kolekcję o przyjmowanie w argumentach tej metody kolekcji nazw interfejsów.



* + 1. Prototype
    2. Facade
    3. NullObject
    4. ContainerIoC
  1. Testy

1. Porównanie z innymi rozwiązaniami
2. Wyzwania
3. Wnioski
4. Bibliografia

Freeman, E., Robson, E., Bates, B. i Sierra, K. (2004). *Head First Design Patterns. A Brain-Friendly Guide.* (P. Koronkiewicz i G. Kowalczyk, Tłumacze)

Gamma, E., Helm, R., Johnson, R. i Vlissdes, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* (T. Walczak, Tłum.) Addison-Wesley Professional.

Micorosoft Corporation. (2021, Wrzesień 15). *Documentation - The .NET Compiler Platform SDK*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

Micorosoft Corporation. (2022, Październik 7). *Documentation - ICloneable*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/api/system.icloneable.clone?view=net-6.0

Micorosoft Corporation. (2022, Czerwiec 06). *Documentation - Source Generators*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview

Microsoft Corporation. (2016, 11 15). *NuGet - System. Kolekcje*. Pobrano z lokalizacji https://www.nuget.org/packages/System.Collections

Microsoft Corporation. (2021, Wrzesień 15). *Documentation - Understand the .NET Compiler Platform SDK model*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model

Microsoft Corporation. (2021, Wrzesień 15). *Documentation - Work with semantics*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-semantics

Microsoft Corporation. (2021, Wrzesień 09). *Microsoft - Work with syntax*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-syntax

Microsoft Corporation. (2022, Czerwiec 11). *Documentation - Source Generators*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview

Microsoft Corporation. (2022, 11 14). *Nuget - Microsoft.CodeAnalysis*. Pobrano z lokalizacji https://www.nuget.org/packages/Microsoft.CodeAnalysis/

Microsoft Corporation. (brak daty). *Documentation - IServiceProvider Interface*. Pobrano z lokalizacji https://learn.microsoft.com/en-us/dotnet/api/system.iserviceprovider?view=net-7.0

Nesteruk, D. (2019). *Design Patterns in .NET: Reusable Approaches in C# and F# for Object-Oriented Software Design.* (R. Meryk, Tłum.) APress Media.

Wikipedia. (2022, Maj 20). *Pusty obiekt (wzorzec projektowy)*. Pobrano z lokalizacji https://pl.wikipedia.org/wiki/Pusty\_obiekt\_(wzorzec\_projektowy)

1. Spis rysunków

[Il. 1. Singleton - diagram UML 6](#_Toc120213503)

[Il. 2. Fabryka - diagram UML 7](#_Toc120213504)

[Il. 3. Fabryka Abstrakcyjna - diagram UML 8](#_Toc120213505)

[Il. 4. Prototyp - diagram UML 9](#_Toc120213506)

[Il. 5. Fasada - diagram UML 10](#_Toc120213507)

[Il. 6. Pusty Obiekt - diagram UML 11](#_Toc120213508)

[Il. 7. Działanie generatora źródeł jako faza kompilacji 16](https://d.docs.live.net/663aaec744012124/Dokumenty/Praca%20dyplomowa%20WSEIZ/Praca_v1.0.2.docx#_Toc120213509)

[Il. 8. Uogólniony schemat UML generatora 19](#_Toc120213510)

1. (Gamma, Helm, Johnson i Vlissdes, 1995, str. 16) [↑](#footnote-ref-1)
2. (Micorosoft Corporation, 2021) [↑](#footnote-ref-2)
3. (Micorosoft Corporation, 2022) [↑](#footnote-ref-3)
4. (Gamma, Helm, Johnson i Vlissdes, 1995, str. 130) [↑](#footnote-ref-4)
5. (Gamma, Helm, Johnson i Vlissdes, 1995, str. 131) [↑](#footnote-ref-5)
6. (Nesteruk, 2019, str. 69) [↑](#footnote-ref-6)
7. (Freeman, Robson, Bates i Sierra, 2004) [↑](#footnote-ref-7)
8. (Gamma, Helm, Johnson i Vlissdes, 1995, strony 102-104) [↑](#footnote-ref-8)
9. (Gamma, Helm, Johnson i Vlissdes, 1995, strony 120-123) [↑](#footnote-ref-9)
10. (Wikipedia, 2022) [↑](#footnote-ref-10)
11. (Microsoft Corporation, brak daty) [↑](#footnote-ref-11)
12. (Micorosoft Corporation, 2021) [↑](#footnote-ref-12)
13. (Microsoft Corporation, 2021) [↑](#footnote-ref-13)
14. (Microsoft Corporation, 2021) [↑](#footnote-ref-14)
15. (Microsoft Corporation, 2021) [↑](#footnote-ref-15)
16. (Microsoft Corporation, 2021) [↑](#footnote-ref-16)
17. (Microsoft Corporation, 2021) [↑](#footnote-ref-17)
18. (Microsoft Corporation, 2021) [↑](#footnote-ref-18)
19. (Microsoft Corporation, 2021) [↑](#footnote-ref-19)
20. (Microsoft Corporation, 2021) [↑](#footnote-ref-20)
21. (Microsoft Corporation, 2022) [↑](#footnote-ref-21)
22. (Microsoft Corporation, 2022) [↑](#footnote-ref-22)
23. (Microsoft Corporation, 2022) [↑](#footnote-ref-23)
24. (Microsoft Corporation, 2016) [↑](#footnote-ref-24)