Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Łukasz Monkiewicz

Nr albumu 3044

Narzędzie wspomagające budowę diagramów klas UML

Praca magisterska

Napisana pod kierunkiem

dr inż. Mariusza Trzaski

Warszawa, październik, 2008

Streszczenie

Niniejsza praca skupia się na problemie wspomagania projektowania oprogramowania, a konkretnie na wspomaganiu tworzenia diagramów klas UML. Zostały przedstawione podstawy języka UML, jego zastosowanie oraz specyfikacja diagramów klas. Dodatkowo przedstawiono istniejące już narzędzia poświęcone tworzeniu diagramów UML, wraz z opisem ich możliwości, zaletami oraz wadami. Opisana została także funkcjonalność jaką powinno posiadać tego typu narzędzie. Zwrócono uwagę na rzeczy takie jak budowa interfejsu, jego sposób działania, a także sposób dostępu do poszczególnych funkcji. Opisano także wymagania dotyczące podstawowych mechanizmów, niezbędnych w edytorze dowolnego typu, takich jak cofanie zmian czy format zapisu oraz wymagania jakie musi spełnić narzędzie przeznaczone do budowy diagramów klas UML.

W dalszej części pracy, na przykładzie prototypu, opisany został sposób implementacji określonych wcześniej wymagań. Zostały przedstawione zalecane do wykorzystania wzorce projektowe wraz z ich opisem oraz sposobem ich implementacji w aplikacji przy użyciu środowiska Java. Zwrócona została uwaga na sposób przetwarzania poszczególnych konstrukcji z diagramów w procesie generowania kodu źródłowego. Dodatkowo opisane zostały zewnętrzne biblioteki wykorzystane do stworzenia prototypu. Użycie biblioteki Java Plugin Framework do budowy modułowej aplikacji z mechanizmem rozszerzeń pozwalających zwiększyć jej funkcjonalność czy też biblioteki FreeMarker zastosowanej do generowania plików zawierających kod źródłowy, wygenerowany na podstawie sporządzonych diagramów klas UML.

Spis treści

1.WSTĘP....5

1.1.Wspomaganie budowy diagramów klas UML....5
1.2.Cel pracy....5
1.3.Rozwiązanie przyjęte w pracy....5
1.4.Rezultaty pracy....6
1.5.Organizacja pracy....6

2.WSPOMAGANIE BUDOWY DIAGRAMÓW KLAS UML....7

2.1.Projektowanie systemów informatycznych....7
2.2.UML....8
2.2.1Historia oraz zastosowanie....8
2.2.2Specyfikacja diagramów klas....9
2.2.3Przykładowy diagram klas....9
2.3.Stan sztuki....10
2.3.1ArgoUML....10
2.3.2Microsoft Office Visio 2007....12
2.3.3Papyrus....13
2.4.Podsumowanie....14

3.OPIS NARZĘDZI ZASTOSOWANYCH W PRACY....15

3.1.Sun Java....15
3.2.Java Plugin Framework....16
3.3.FreeMarker....19

4.WYMAGANA FUNKCJONALNOŚĆ APLIKACJI....23

4.1.Podstawowa aplikacja....23
4.1.1Operacje na diagramie....23
4.1.2Mechanizm cofania zmian....25
4.1.3Zapisywanie diagramu do pliku....26
4.2.Rozszerzalność....26
4.2.1Struktura rozszerzeń....26
4.2.2Punkty rozszerzeń....27
4.3.Tworzenie diagramów klas....28
4.3.1Klasy oraz interfejsy....29
4.3.2Połączenia....30
4.3.3Komentarze....31
4.4.Generowanie kodu źródłowego....31
4.4.1Przygotowanie danych....31
4.4.2Budowa szablonów....32
4.4.3Rozszerzalność....32

5.UMLED - PROTOTYP....33

5.1.Budowa podstawowej części aplikacji....34
5.1.1Zarządzanie zasobami....34
5.1.2Mechanizm cofania zmian....35
5.1.3Punkt rozszerzeń „Project”....37
5.1.4Punkt rozszerzeń „Export”....37
5.1.5Punkt rozszerzeń „Tools”....38
5.1.6Format zapisu diagramów....38
5.2.Budowa diagramów....39
5.2.1Struktura diagramów....39
5.2.2Sposób wykreślania diagramów....42

5.2.3Elementy UML....43
5.2.4Struktura klas UML....45
5.2.5Struktura połączeń UML....46
5.2.6Struktura komentarzy UML....46
5.3.Interfejs aplikacji....47
5.4.Eksport diagramów....50
5.4.1Przygotowanie danych....50
5.4.2Generowanie kodu źródłowego....52
6.ZALETY, WADY ORAZ PLANY ROZWOJOWE....56
6.1.Zalety oraz wady przyjętych rozwiązań....56
6.2.Plany rozwojowe....57
6.3.Zastosowanie....59
7.PODSUMOWANIE....60
BIBLIOGRAFIA....61

1.Wstęp

Wspomaganie budowy diagramów klas UML poprzez dedykowane narzędzia jest niezbędne z powodu ciągle rosnącego poziomu skomplikowania systemów informatycznych. Aktualnie na rynku istnieje szereg narzędzi oferujących taką funkcjonalność, jednak narzędzia te posiadają wiele wad i bardzo ciężko znaleźć takie, które będzie zarówno wygodne w użyciu oraz będzie oferowało odpowiednią funkcjonalność.

1.1. Wspomaganie budowy diagramów klas UML

Tworzenie oprogramowania, a w szczególności tworzenie dużych systemów, dla zapewnienia odpowiedniego poziomu jakości w zakresie niezawodności, bezpieczeństwa czy też wydajności, wymaga szczegółowego planowania i projektowania. W procesie projektowania przeważnie uczestniczy wiele osób i dla zapewnienia jednolitego, zrozumiałego dla wszystkich sposobu opisu struktury czy tez zachowania systemu został stworzony język UML. Definiuje on szereg diagramów opisujących poszczególne aspekty działania aplikacji. Jednym z takich diagramów jest diagram klas opisujący strukturę klas oraz zależności pomiędzy nimi. Proces tworzenia tych diagramów można wspomóc poprzez wykorzystanie jednego z dostępnych narzędzi, jednak narzędzia te nie są doskonałe. Bardzo często mają skomplikowany i nie wygodny interfejs, przez co praca z nimi jest męcząca i irytująca. Natomiast narzędzia o wygodnym interfejsie przeważnie mają bardzo ograniczoną funkcjonalność, na przykład brak w nich funkcji generowania kodu źródłowego, czy też utrudnione jest dostosowanie ich do własnych potrzeb. Brakuje więc aplikacji, która łączy ze sobą wygodę oraz bogatą funkcjonalność.

1.2. Cel pracy

Celem pracy jest opracowanie aplikacji pozwalającej przede wszystkim na tworzenie diagramów klas UML. Oprócz tego powinna ona posiadać następujące funkcje:

- Prosty, przejrzysty, oraz wygodny w obsłudze interfejs użytkownika, którego obsługa jest maksymalnie intuicyjna.
- Możliwość wygenerowania kodu źródłowego Javy na podstawie stworzonego diagramu klas UML.
- Budowa aplikacji musi pozwalać na rozszerzenie jej możliwości, bez potrzeby ingerencji w kod źródłowy istniejących modułów.

1.3. Rozwiązanie przyjęte w pracy

Praca opiera się na wykorzystaniu środowiska Java Standard Edition w wersji 5. Opracowany prototyp, wykorzystuje standardowe biblioteki Javy takie jak Swing oraz Java2D do budowy interfejsu użytkownika oraz wykreślania diagramów. Natomiast struktura całej aplikacji została zbudowana przy wykorzystaniu darmowej biblioteki Java Plugin Framework, która oferuje możliwość modułowej budowy aplikacji, co z kolei umożliwia zaimplementowanie bardzo elastycznego mechanizmu rozszerzeń. Kolejną wykorzystaną zewnętrzną biblioteką jest FreeMarker. Jest to biblioteka, także

darmowa, oferująca funkcjonalność generowania plików tekstowych na podstawie wcześniej zdefiniowanych szablonów. Została ona użyta do stworzenia modułu zajmującego się generowaniem kodu źródłowego na podstawie diagramów klas.

1.4. Rezultaty pracy

Bezpośrednim rezultatem pracy jest opracowany prototyp. Posiada on wszystkie podstawowe funkcje niezbędne do pracy nad diagramami klas oraz możliwość generowania kodu źródłowego. Można powiedzieć, że definiuje on podstawowy zestaw funkcji jakie powinno posiadać każde narzędzie tego typu. Dodatkowo, dzięki swojej konstrukcji modułowej, stanowi on solidną podstawę do późniejszego rozwoju aplikacji wspomagających pozostałe aspekty projektowania oprogramowania, takie jak analiza przypadków użycia czy też analiza dynamiczna

1.5. Organizacja pracy

Praca rozpoczyna się od przedstawienia problemu związanego z projektowaniem oprogramowania. Opisuje przeszkody stojące na drodze do stworzenia dobrego projektu oraz przedstawia język UML, opisujący różnego rodzaju diagramy przedstawiające poszczególne części oraz zachowania projektowanej aplikacji. Zostały też opisane istniejące narzędzia wspierające budowę diagramów klas UML, wraz z wyszczególnieniem ich zalet oraz wad.

W rozdziale trzecim zostały z kolei opisane narzędzia oraz biblioteki wykorzystane przy implementacji prototypu. Zaś szczegóły samej jego implementacji zostały zawarte w rozdziale piątym. Zostały tam opisane najważniejsze mechanizmy, ich sposób działania, ich zalety oraz wady, a także wykorzystane struktury danych do realizacji budowy diagramów klas.

Wymagania stawiane przed aplikacją wspomagającą budowę diagramów klas UML zostały przedstawione w rozdziale czwartym. Wyszczególnione i opisane zostały wszystkie rodzaje elementów diagramu wraz z ich atrybutami oraz zostały przedstawione zostały ograniczenia w zakresie łączenia poszczególnych elementów ze sobą.

Pozostałe rozdziały mają charakter podsumowujący. Rozdział szósty zawiera głównie opis zalet oraz wad opracowanego prototypu. Posiada także szereg wytycznych przedstawiających możliwy przyszły rozwój aplikacji. Natomiast ostatni rozdział, rozdział siódmy, stanowi podsumowanie całej pracy.

2.Wspomaganie budowy diagramów klas UML

Proces projektowania jest jednym z najważniejszych, jeśli nie najważniejszym, elementem całego procesu tworzenia oprogramowania. Z tego właśnie powodu, niezbędne jest dostarczenie narzędzi, które będą mogły usprawnić ten proces.

2.1. Projektowanie systemów informatycznych

Tworzenie oprogramowania jest rzeczą prostą, jednak tworzenie dobrego oprogramowania wymaga dużego wysiłku oraz dokładnego zaprojektowania wszystkich zależności pomiędzy poszczególnymi elementami aplikacji. Należy zwrócić uwagę na takie rzeczy jak bezpieczeństwo, czyli sprawdzić czy poszczególne elementy systemu są dostępne tylko i wyłącznie dla osób do tego upoważnionych, a także należy tak go zaprojektować aby zapewnić integralność przechowywanych w nim danych. Podczas projektowania można przeanalizować różne sposoby realizacji warstwy odpowiedzialnej za bezpieczeństwo i wybrać najbardziej odpowiednią bez potrzeby testowania każdego sposobu na działającym aplikacji oraz bez modyfikowania istniejącego już kodu.

Inna ważną cechą systemu, mimo ogromnej i stale rosnącej mocy obliczeniowej dzisiejszych komputerów, jest wydajność, która nadal jest jedną z najważniejszych cech decydujących o sukcesie bądź porażce. Użytkownicy są wyczuleni na wszelkie, z ich punktu widzenia zbędne, przestoje w działaniu oprogramowania, dlatego oprogramowanie powinno w jak najbardziej optymalny sposób korzystać z dostępnej mocy komputera.

Kolejnym aspektem projektowania są konsultacje z klientem. Dzięki wielu konsultacjom z klientem w trakcje projektowania, mamy możliwość sprecyzowania wymagań klienta (klient nie zawsze dokładnie wie czego potrzebuje) oraz mamy możliwość wyeliminowania pewnych błędów (często wynikających z nieporozumień) na które byśmy napotkali dopiero w momencie przedstawienia prototypu, bądź gotowego już systemu klientowi. Naprawienie tych błędów w tak późnej fazie tworzenia oprogramowania może wiązać się z dużym nakładem pracy oraz z dużymi kosztami, może też spowodować niezadowolenie klienta, który z tego powodu, w przyszłości może zdecydować się na innego producenta oprogramowania.

Proces projektowania, poprzedzający proces implementacji, pozwala także na skupieniu się na problemie przedstawionym przez klienta bez potrzeby zagłębiania się w aspekty związane z technologią jaka zostanie użyta. Dopiero po sprecyzowaniu budowy systemu, mając pełny obraz wymagań, można zająć się wyborem technologii, która najlepiej będzie się nadawała do realizacji projektu. Poziom skomplikowania dzisiejszych aplikacji wymusza także zaplanowanie kolejności implementacji poszczególnych elementów tak, aby poszczególne zespoły programistów nie przeszkadzały sobie w pracy i mogły optymalnie pracować.

Wszystkie te aspekty projektowania systemów informatycznych pozwalają zaoszczędzić czas i pieniądze, pozwalają w jasny sposób przedstawić system nad którym pracujemy całemu zespołowi, dzięki czemu wyeliminujemy większość nieporozumień, które mogłyby się pojawić w fazie implementacji. Jednak aby projekt był zrozumiały dla wszystkich, musi on być opisany według określonego standardu, który w jednoznaczny sposób opisze cały system oraz wszystkie zależności w nim występujące. Takim standardem jest język UML, który wraz z narzędziami wspomagającymi budowę diagramów UML, pozwala w wygodny oraz wydajny sposób zaprojektować system.

2.2. UML

Język UML jest aktualnie standardem w dziedzinie projektowania oprogramowania, jest standardem wykorzystywanym do przedstawiania poszczególnych części projektowanego systemu w postaci różnego rodzaju diagramów.

2.2.1 Historia oraz zastosowanie

Przed powstaniem języka UML na rynku istniało kilkadziesiąt różnych języków i metod opisu budowy oraz działania systemów informatycznych, były one mniej lub bardziej podobne do siebie, jednak ich mnogość mogła powodować obniżenie wydajności pracy analityków oraz programistów, ponieważ mogło się zdarzyć, że nie wszyscy w zespole znali akurat te metody, które zostały wykorzystane przez inne osoby w zespole.

Pod koniec 1994 roku, pracownicy Rational Software, Grady Booch, autor metody OOAD (Object-Oriented Analysis and Design), oraz Jim Rumbaugh, autor metody OMT (Object Modeling Technique), rozpoczęli prace nad stworzeniem jednolitego języka opisującego. W 1995 roku dołączył do nich Ivar Jacobson wraz ze swoim OOSE (Object-Oriented Software Engineering). W trójkę, razem ze wsparciem ze strony OMG (Object Management Group), opracowali pierwszą specyfikacje języka UML, wersję 0.9 w czerwcu 1996 oraz wersję 0.91 w październiku 1996, która zawierała poprawki oraz sugestie otrzymane od szerszej grupy inżynierów zainteresowanych rozwojem języka. Do prac nad rozwojem języka włączyło się wiele firm, były to m.in. IBM, Oracle, Microsoft, HP. Efektem współpracy było stworzenie specyfikacji UML 1.0 w styczniu 1997 roku, oraz późniejsze zatwierdzenie języka poprzez OMG jesienią 1997 roku.[1]

Najnowszą wersją języka UML jest wersja 2.0, której specyfikacja (patrz [2]) definiuje 13 rodzajów diagramów, można je podzielić na dwie grupy: diagramy struktury opisujące to, co powinno się znaleźć w systemie oraz diagramy dynamiki opisujące to, co się w systemie dzieje.

Wyróżniamy następujące rodzaje diagramów struktury:

- Diagram klas – opisuje struktury klas i interfejsów oraz zależności pomiędzy nimi
- Diagram obiektów – pozwala dokładnie zobrazować skomplikowane relacje poszczególnych instancji obiektów, które mogą się pojawić na przykład w przypadku klas z rekurencyjnymi relacjami.
- Diagram pakietów – opisuje zależności pomiędzy poszczególnymi pakietami w systemie
- Diagram struktur złożonych – opisuje wewnętrzną strukturę obiektu, a także jego punkty interakcji z innymi obiektami systemu
- Diagram komponentów – przedstawia fizyczne elementy systemu oraz połączenia pomiędzy nimi
- Diagram wdrożeniowy – opisuje fizyczną konfigurację oprogramowania oraz sprzętu

Diagramy dynamiki dzielą się na następujące rodzaje:

- Diagram przypadków użycia – opisuje funkcje systemu
- Diagram aktywności – opisuje w jaki sposób poszczególne procesy współpracują ze sobą
- Diagram stanów – opisuje wszystkie możliwe stany obiektu oraz przejścia powodujące zmiany stanów
- Diagram sekwencji – szczegółowo opisuje w jaki sposób są wykonywane operacje oraz interakcje pomiędzy obiektami zaangażowanymi w wykonanie danej operacji

- Diagram komunikacji – opisuje wzajemne oddziaływanie na siebie obiektów oraz komunikaty jakie pomiędzy sobą wysyłają
- Diagram harmonogramowania – pozwala zobrazować zmiany stanów obiektów wraz z czasem w jakim będą one przebywać w poszczególnych stanach
- Diagram sterowania interakcją – obrazuje współpracę poszczególnych diagramów interakcji

2.2.2 Specyfikacja diagramów klas

Diagramy klas przedstawiają statyczną strykturę fragmentu systemu, opisują klasy danego systemu oraz ich zależności względem siebie.

Klasy oraz interfejsy, na diagramie, są przedstawione za pomocą prostokątów. Prostokąty te są podzielone na trzy części, w pierwszej z nich znajduję się nazwa klasy. Pozostałe dwie części są poświęcone atrybutom klasy, czyli polom oraz metodom. Pola znajdują się w części poniżej nazwy, natomiast metody w części najniższej. Zarówno pola jak i metody są reprezentowane za pomocą co najmniej nazwy, jednak mogą także być opisane dodatkowymi elementami określającymi typ pola, typ zwracany metody, argumenty metody, a także widoczność.

Relacje pomiędzy klasami oraz interfejsami są przedstawione za pomocą łączących je linii. Można wyróżnić trzy rodzaje relacji:

- Dziedziczenie
- Implementacja interfejsu
- Asocjacje

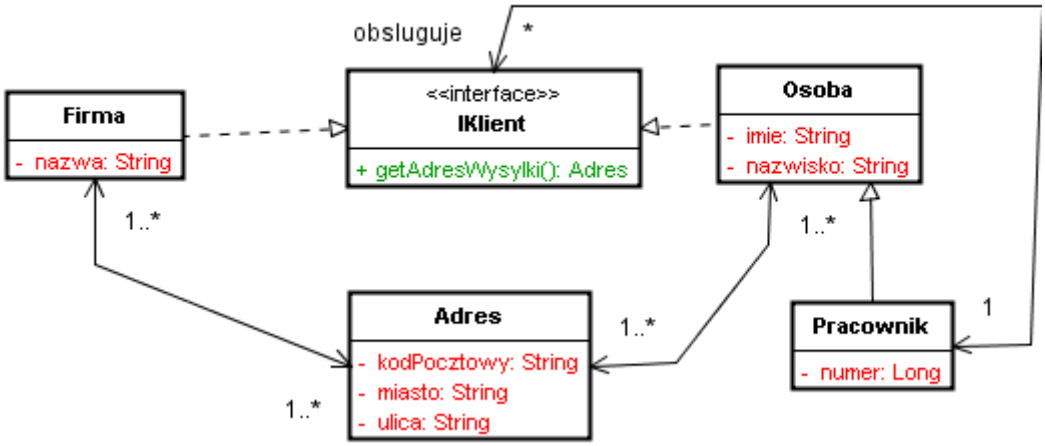
Relację dziedziczenia przedstawia się za pomocą linii zakończonej niezamalowanym trójkątem z jednej strony, ze strony klasy uogólniającej. Implementację interfejsu oznacza się podobnie, z tą różnicą, że linia jest przerywana, a jej zakończenie (ze strony interfejsu) jest strzałką.

Asocjacje opisują relacje pomiędzy co najmniej dwiema klasami, mogą one być skierowane (kierunek asocjacji jest oznaczany strzałką na jednym z końców) bądź dwukierunkowe. Każda asocjacja może posiadać nazwę, a także określać role poszczególnych klas danej relacji (nazwy ról umieszcza się przy końcach asocjacji). Dodatkowo każdy koniec asocjacji powinien posiadać liczność określająca minimalny i/lub maksymalny limit instancji danej klasy w relacji.

Agregacja jest też asocjacją, jednak przedstawia ona związek części z całością. Na diagramie oznacza się ją pustym rombem na jednym końcu asocjacji, po stronie całości. Szczególnym przypadkiem agregacji jest kompozycja, w jej przypadku część nie może istnieć bez całości, oznacza się ją podobnie jak agregację ale z zamalowanym rombem.

2.2.3 Przykładowy diagram klas

Przykładowy, prosty diagram klas przedstawiony na Rys.1 przedstawia fragment struktury systemu firmy świadczącej pewne usługi. Klientem tej firmy może być zarówno firma jak i osoba fizyczna, każdemu klientowi przyporządkowany jest jeden pracownik zajmujący się jego obsługą. Dodatkowo każdy klient posiada co najmniej jeden adres.



Rys. 1. Prosty diagram klas

2.3. Stan sztuki

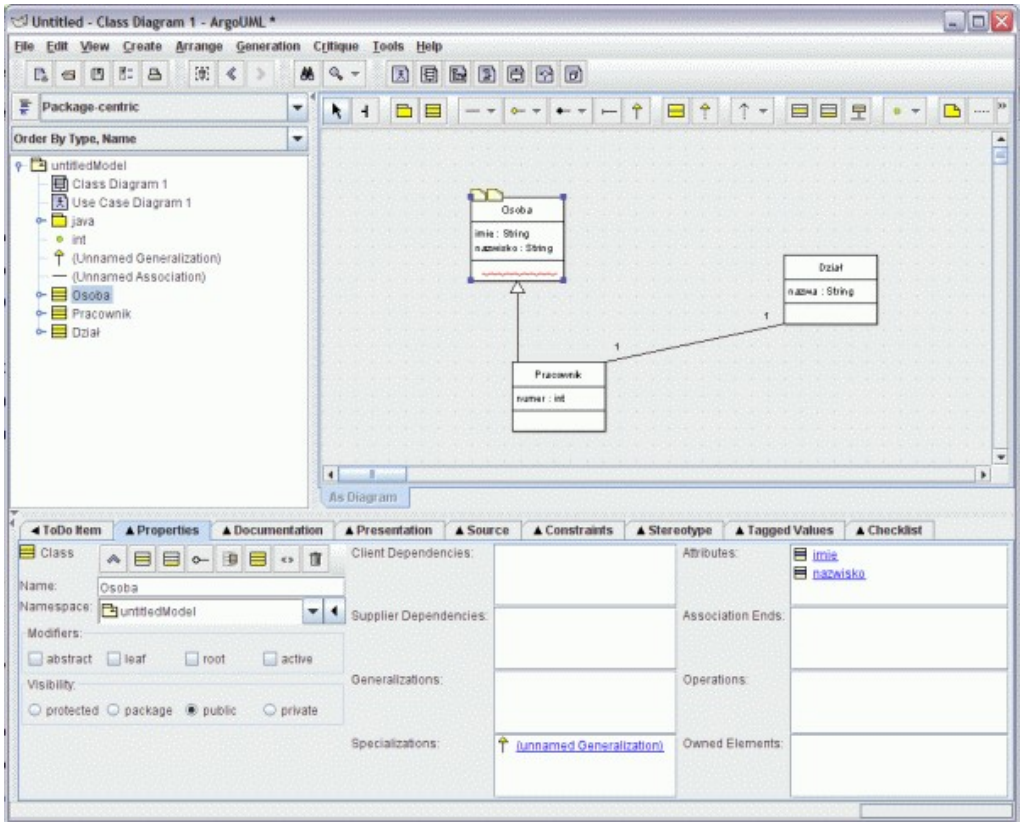
Aktualnie na rynku dostępne jest wiele różnych aplikacji wspierających tworzenie diagramów UML, różnią się one przede wszystkim funkcjonalnością, a także docelowym zastosowaniem. Część z tych aplikacji jest projektowana jako środowiska do tworzenia wszelkiego typu diagramów (np. Microsoft Visio), a część stricte jako aplikacje wspierające tworzenie oprogramowania (np. ArgoUML).

2.3.1 ArgoUML

ArgoUML jest darmowym środowiskiem do tworzenia diagramów UML. Posiada on wsparcie dla wszystkich 9 diagramów zdefiniowanych w wersji 1.4 języka UML. Do działania wymagana jest maszyna wirtualna Javy w wersji 1.4 lub nowszej. Dzięki zastosowaniu Javy, ArgoUML można uruchomić na dowolnej platformie systemowej ją obsługującą. Dodatkowo dostępna jest wersja w postaci pluginu, ArgoEclipse, przeznaczonego dla popularnego środowiska programistycznego Eclipse IDE, dzięki czemu korzystając tylko z jednego narzędzia możliwe jest przeprowadzenie wszystkich wymaganych czynności tworzenia oprogramowania, począwszy od analizy, poprzez implementację oraz testowanie, do wdrożenia. ArgoUML jest rozpowszechniany na licencji BSD, co pozwala na dowolną modyfikację jego kodu, a co za tym idzie, możliwość dostosowania aplikacji do naszych potrzeb. Modyfikacja taka jednak wymaga dogłębnego przeanalizowania i zrozumienia istniejącego kodu programu, co z kolei może być zadaniem drogim oraz czasochłonnym. Dużo lepszym rozwiązaniem byłoby na pewno zastosowanie mechanizmu rozszerzeń z prostym API. ArgoUML jest dostępny w 10 wersjach językowych.

Obszar roboczy ArgoUML (przedstawiony na Rys.2) jest podzielony na trzy części, diagram, drzewo diagramu oraz właściwości wybranego elementu. Drzewo diagramu wyświetla wszystkie klasy oraz interfejsy razem z ich wszystkimi atrybutami (pola i metody), a także wszystkie relacje występujące na diagramie oraz użyte typy danych. W przypadku dużych i skomplikowanych diagramów, takie drzewo może być bardzo przydatnym narzędziem pozwalającym oszczędzić czas.

Obszar właściwości zawiera wszelkie dane aktualnie wybranego elementu diagramu i pozwala w wygodny i stale dostępny sposób dane te modyfikować. Dodatkowo zawiera on podgląd na wygenerowany kod wybranego elementu diagramu. Obszar diagramu jest dosyć mały, co jest na pewno jego wadą, możliwe jest powiększenie go poprzez ukrycie drzewa diagramu i/lub właściwości, jednak w przypadku ukrycia obszaru właściwości ograniczona zostaje funkcjonalność poprzez brak dostępu do wszystkich opcji. W górnej części obszaru diagramu umieszczony jest szereg przycisków służących do tworzenia poszczególnych elementów diagramu za pomocą pojedynczego kliknięcia, asocjacje oraz inne typy relacji są tworzone za pomocą przeciągania odpowiedniego uchwytu z jednej klasy na drugą. Dodatkowo ArgoUML oferuje trzy różne notacje wyświetlania opisu diagramów (opis pól, metod itp.), są to notacja UML, notacja zgodna z Javą oraz notacja C++. Wszystkie te rzecz sprawiają, że tworzenie diagramów, pomimo małego obszaru roboczego, jest proste i wygodne. Jedynym odczuwalnym, a zarazem poważnym brakiem jest brak mechanizmu cofania zmian.



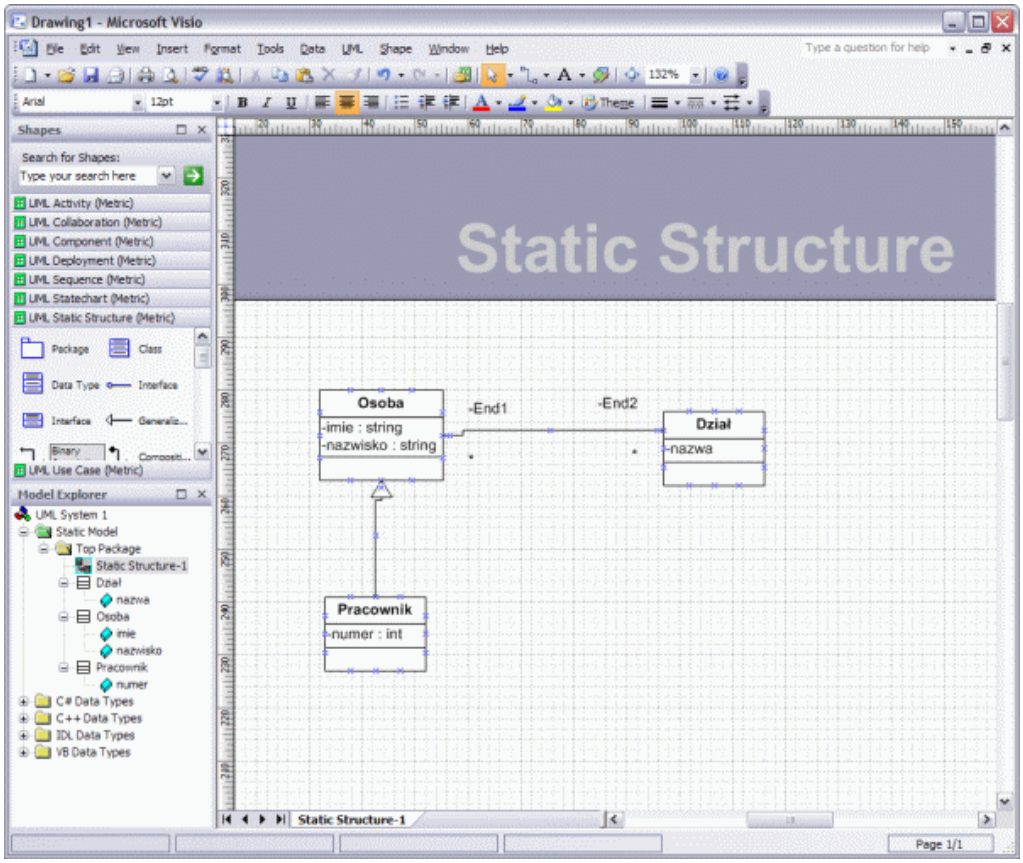
Rys.2. Obszar roboczy ArgoUML.

ArgoUML jako format zapisu diagramów używa XMI, który jest powszechnie wspierany przez inne narzędzia UML jako format wymiany dokumentów UML. Dodatkowo możliwe jest wygenerowanie na podstawie diagramów kodu źródłowego w formacie Javy, C++, C#, PHP4 oraz PHP5.

2.3.2 Microsoft Office Visio 2007

Microsoft Office Visio 2007 jest aplikacją przeznaczoną dla platformy Windows. Jest to narzędzie wspomagające tworzenie różnych rodzajów diagramów, nie konieczne związanych z projektowaniem systemów informatycznych. Dostępny jest w 26 wersjach językowych. Visio 2007 oferuje wsparcie dla 8 rodzajów diagramów UML: aktywności, współpracy, komponentów, wdrożeniowy, sekwencji, stanów, klas oraz przypadków użycia. Jest to aplikacja komercyjna o zamkniętym kodzie, jednak możliwe jest dodanie rozszerzeń przy pomocy dostępnego za darmo Visio 2007: Software Development Kit.

Obszar roboczy (przedstawiony na Rys.3) jest podzielony na dwie części, obszar diagramu oraz pasek narzędzi. Pasek narzędzi zawiera elementy wszystkich dostępnych diagramów UML podzielone na grupy według rodzajów diagramów oraz drzewo elementów zawierające wszystkie elementy znajdujące się na diagramie razem z ich atrybutami, co ułatwia nawigację w dużych projektach. Tworzenie poszczególnych elementów odbywa się na zasadzie „przeciągnij i upuść" z paska narzędzi na obszar diagramu. Dodawanie poszczególnych atrybutów elementów (takich jak na przykład pola lub metody klas) odbywa się za pomocą okna właściwości dostępnego po podwójnym kliknięciu na dany element. Niestety okno to posiada pewne wady. W przypadku dodawania lub edycji pewnych elementów, na przykład dodawania metod do klas, wymagane jest nawigowanie poprzez kilka zakładek oraz otwierania dodatkowych okien aby dotrzeć do na przykład argumentów metod. Kolejną wadą jest potrzeba zdefiniowania wszystkich typów danych przed ich użyciem. Lepszym rozwiązaniem byłoby automatyczne definiowanie nowych typów w momencie ich pierwszego użycia.

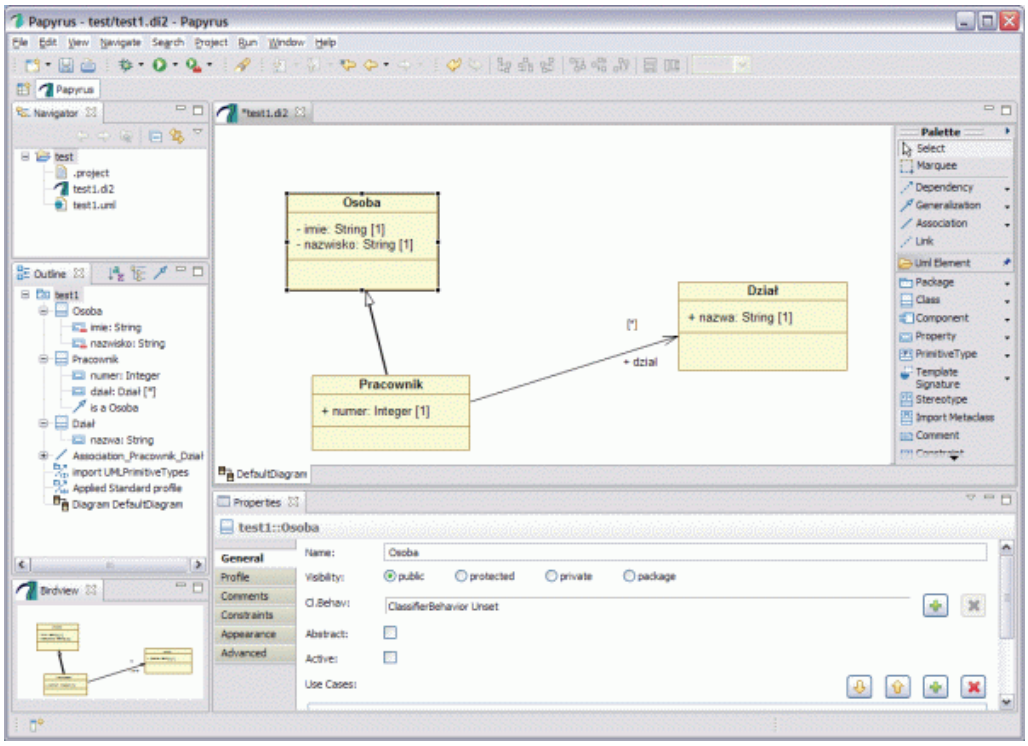


Rys. 3. Obszar roboczy Microsoft Office Visio 2007.

Niestety Visio nie oferuję żadnych opcji generowania kodu na podstawie diagramów ani zapisania diagramów w formacie, który byłby akceptowany przez inne narzędzia na rynku, jedyną opcją eksportu diagramu jest zapisanie go w postaci grafiki.

2.3.3 Papyrus

Papyrus jest darmową aplikacją, o otwartym kodzie, rozpowszechnianą na licencji EPL (Eclipse Public License). Jest dostępna jako samodzielna aplikacja oraz jako plugin do popularnego Eclipse IDE. Oferowana jest w wersjach na platformy Windows, Linux oraz Mac OS X. Otwartość kodu pozwala na dowolne modyfikacje, co z kolei umożliwia nam dostosowanie aplikacji do naszych potrzeb. Dodatkowo Papyrus oferuje mechanizm rozszerzeń, dzięki którym w łatwy sposób, korzystając z prostego API, możemy rozszerzyć jego możliwości o takie rzeczy jak dodatkowe generatory kodu, dodatkowe opcje eksportu diagramów oraz inne narzędzia wspomagające budowę diagramów. Standardowo Papyrus posiada możliwość tworzenia 6 rodzajów diagramów zgodnych ze specyfikacją UML w wersji 2.0. Są to diagramy klas, przypadków użycia, wdrożeniowych, komponentów, sekwencji oraz diagram struktur złożonych.



Rys. 4. Obszar roboczy Papyrus'a

Obszar roboczy programu (przedstawiony na Rys.4) możemy swobodnie zmieniać poprzez przenoszenie, ukrywanie lub dodawanie poszczególnych paneli. Panele te zawierają takie rzeczy jak pomniejszony widok całego diagramu służący do nawigacji po nim, drzewo diagramu zawierające wszystkie elementy znajdujące się na diagramie razem z ich atrybutami oraz panel właściwości zawierające wszelkie dane aktualnie wybranego elementu na diagramie. Możliwe jest takie skonfigurowanie środowiska, aby panel diagramu zajmował całą powierzchnię okna aplikacji, a wszystkie inne panele były dostępne w postaci wyskakujących okien po pojedynczym kliknięciu odpowiedniego przycisku, dzięki czemu otrzymujemy dużą powierzchnię do pracy nad diagramem

bez ograniczenia funkcjonalności aplikacji. Tworzenie diagramu jest bardzo intuicyjne, jednak nie obyło się bez kilku wad. Pierwszą rzeczą która rzuca się w oczy jest brak automatycznego dostosowywania rozmiaru klas do ich zawartości, przez co musimy ręcznie regulować ich rozmiar aby wszystkie pola i metody były widoczne. Kolejną rzeczą jest potrzeba definiowania wszystkich typów przed ich użyciem, dużo wygodniejszym rozwiązaniem byłoby automatycznie ich tworzenie podczas ich pierwszego użycia. Standardowo Papyrus posiada możliwość wygenerowania kodu Javy na podstawie diagramu, możliwe jest także dodanie wsparcia dla innych języków.

2.4. Podsumowanie

Ciągle rosnący poziom skomplikowania oprogramowania wymaga szczegółowego projektowania poszczególnych komponentów oraz ich interakcji pomiędzy sobą, aby zapewnić możliwie najwyższa wydajność i zapewnić możliwie wysoki poziom bezpieczeństwa. Z powodu rozmiarów dzisiejszych systemów projektowanie ich nie jest rzeczą prostą i wymaga wspomagania ze strony narzędzi wspierających język UML. Narzędzia te powinny oferować analitykom środowisko maksymalnie przyjazne oraz ergonomiczne. Powinno ograniczać liczbę zbędnych czynności podczas edycji diagramów (na przykład, zmiana typu parametru metody na diagramie klas nie powinna wymagać nawigowania po hierarchii wielu okien), oferować maksymalnie duży obszar roboczy oraz prosty i czytelny interfejs. Aplikacja taka powinna posiadać także możliwość rozszerzenia swojej funkcjonalności w możliwie najprostszy sposób, najlepiej w postaci wtyczek opartych o proste API, tak aby można było w prosty sposób dostosować ją do potrzeb danego projektu lub środowiska dla którego projektowany system będzie przeznaczony. Ponieważ narzędzie takie powinno oferować coś więcej niż tylko graficzną reprezentacje systemu, powinno ono posiadać także możliwość wygenerowania kodu na podstawie diagramu, w dowolnym języku oraz technologii. Format wygenerowanego kodu powinien dać się w prosty sposób zmienić, tak aby w łatwy sposób producent oprogramowania mógł go dostosować do standardów u niego panujących.

3.Opis narzędzi zastosowanych w pracy

Prototyp aplikacji został napisany przy pomocy języka Java 5 Standard Edition ze względu na charakterystykę tego języka (obiektowość oraz wieloplatformowość), a także ze względu na standardowo dostępne biblioteki do generowania grafiki (Java2D) oraz do tworzenia aplikacji desktopowych (Swing). Dla zapewnienia rozszerzalności aplikacji zastosowana została darmowa biblioteka Java Plugin Framework, a do generowania kodu źródłowego na podstawie diagramów biblioteka FreeMarker.

3.1. Sun Java

Java jest obiektowym językiem programowania opracowanym przez Sun Microsystems w 1995 roku. Większość składni bazuje na językach C oraz C++, jednak zdecydowano się na uproszczenie modelu obiektowego oraz usunięcie większości niskopoziomowych konstrukcji. Podczas projektowania języka szczególną uwagę poświęcono przede wszystkim obiektowości, wieloplatformowości, natywnemu wsparciu dla sieci komputerowych oraz na bezpiecznym uruchamianiu kodu ze zdalnych lokacji.

Aplikacje napisane w Javie są kompilowane do kodu bajtowego, który z kolei jest uruchamiany poprzez maszynę wirtualną Javy (JVM), dzięki czemu kod raz skompilowany może być uruchamiany na maszynach wirtualnych pod kontrolą różnych systemów operacyjnych. Dzięki wykorzystaniu maszyny wirtualnej do uruchomienia aplikacji, oprócz wieloplatformowości, dostępne jest jeszcze kilka udogodnień. Jednym z nich jest *Garbage collector*, jest to mechanizm zajmujący się automatycznym usuwaniem niepotrzebnych obiektów z pamięci. Takie rozwiązanie odciąża programistę od ręcznego dbania o usuwanie obiektów, a także zabezpiecza aplikacje przed wyciekami pamięci. Kolejną zaletą maszyny wirtualnej jest bezpieczeństwo. Ze względu na pośredniczenie maszyny wirtualnej w interakcji aplikacji z systemem, aplikacja taka posiada pewne ograniczenia w dostępie do zasobów systemu, dzięki czemu błędnie napisana aplikacja swym działaniem nie wpłynie na działanie, ani nie uszkodzi, systemu operacyjnego.

Dostępne są trzy rodzaje platform Javy:

- *Java Micro Edition* (*Java ME*) – przeznaczona głównie dla urządzeń przenośnych takich jak palmtopy, telefony komórkowe itp.
- *Java Standard Edition* (*Java SE*) – platforma ogólnego użytku, dla rozwiązań serwerowych oraz desktopowych.
- *Java Enterprise Edition* (*Java EE*) – przeznaczona dla rozwiązań serwerowych.

Java Standard Edition posiada bogaty zestaw bibliotek oferujących m. in. tworzenie aplikacji okienkowych (Swing), tworzenie grafiki 2D (Java2D), internacjonalizacje, komunikację sieciową. Wsparcie dla wszystkie tych bibliotek jest standardowo dostępne na każdej maszynie wirtualnej Javy.

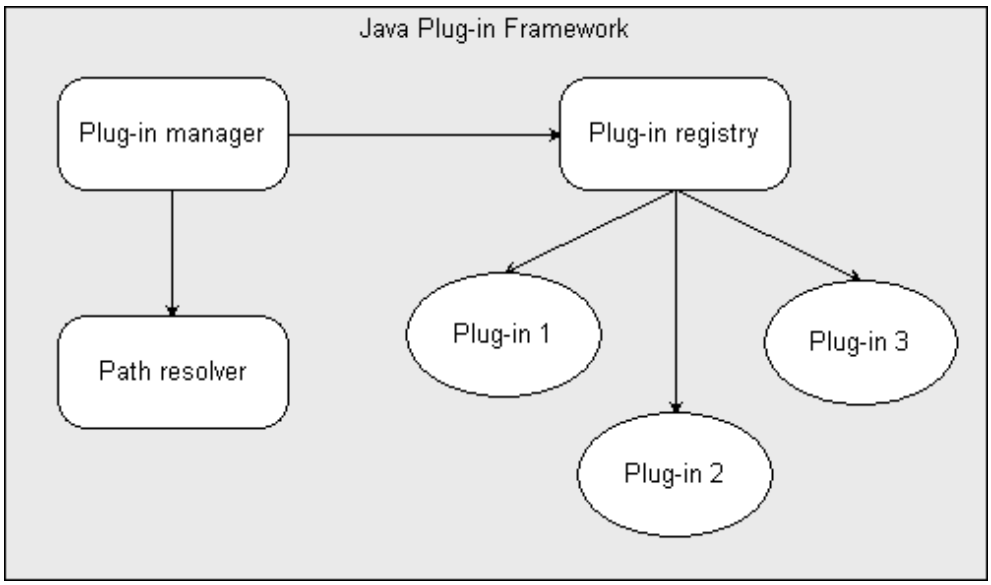
Swing, będący biblioteką Javy odpowiedzialną za interfejs użytkownika, zapewnia jednolity wygląd oraz zachowanie aplikacji na wszystkich platformach. Dzięki zastosowaniu wzorca MVC (*Model-View-Controler*), czyli odseparowania od siebie danych, widoku oraz logiki, jest on bardzo elastyczny i prosty w użyciu. Biblioteka zawiera szereg gotowych komponentów takich jak przyciski, tabele, suwaki, okna, z których szybko, w prosty sposób można zbudować nawet skomplikowany interfejs. Kolejnym atutem jest to, że każdy z dostępnych komponentów jest jednocześnie kontenerem,

czyli obiektem wewnątrz którego można umieścić inne komponenty (np. suwak wewnątrz przycisku). Dla większego uproszczenia tworzenia interfejsów dla rożnych rozdzielczości, w Swingu wprowadzono tak zwane menadżery rozkładu (*LayoutManager*). Jak sama nazwa wskazuje są one odpowiedzialne za rozmieszczenie poszczególnych komponentów wewnątrz kontenera. Standardowo biblioteka dostarcza wiele implementacji menadżerów rozkładu oferujących różne sposoby rozłożenia komponentów, jeśli jednak komuś to nie wystarcza, możliwe jest napisanie własnych. Interakcja użytkownika z interfejsem w Swingu jest oparta na modelu zdarzeniowym, czyli każde wciśnięcie przycisku, ruch myszy, kliknięcie generuje zdarzenie. Każde takie zdarzenie zawiera szczegółowe informacje na temat jego natury i pochodzenia. Do obsługi tych zdarzeń wykorzystywane są tzw. słuchacze (*Listener*), które należy przypisać do wszystkich komponentów, z których chcemy przechwytywać wywoływanie przez nie zdarzenia.

Do rysowania wszystkich komponentów *Swingu* wykorzystywana jest biblioteka *Java2D*. Zawiera ona szereg gotowych implementacji kształtów (linie, prostokąty, koła, wielokąty itp.) oraz różnych filtrów graficznych pozwalających w dowolny sposób modyfikować wygląd istniejących już komponentów. *Java2D* dzięki możliwości wykorzystania OpenGL do renderowania grafiki oferuje wysoką wydajność oraz wiele efektów wpływających na jakość generowanej grafiki (np. wygładzanie krawędzi).

3.2. Java Plugin Framework

Java Plugin Framework (JPF) jest darmową biblioteką Javy (rozpowszechnianą na licencji LGPL) mającą na celu uproszczenie budowy rozszerzalnych aplikacji. Framework ten oferuje mechanizmy zarządzające poszczególnymi rozszerzeniami, ich ładowaniem do pamięci, aktywacją, zależnościami między nimi, a także dostępem do ich zasobów.



Rys. 5. Zależności pomiędzy głównymi modułami JPF.

Działanie JPF opiera się na 3 głównych klasach przedstawionych na Rys.5, są to:

- `PluginRegistry` – jest odpowiedzialny za zbieranie oraz przechowywanie informacji o wszystkich znalezionych rozszerzeniach.
- `PathResolver` – odpowiada za mapowanie położenia lokalnych zasobów poszczególnych rozszerzeń na ich fizyczna lokalizacje.
- `PluginManager` – jest sercem aplikacji, odpowiada za wczytywanie, aktywacje oraz dezaktywacje poszczególnych rozszerzeń

Aplikacja oparta na JPF w zasadzie składa się tylko z rozszerzeń, a jedno z tych rozszerzeń jest oznaczone jako główne (*main plugin*). Po uruchomieniu aplikacji, `PluginManager` wczytuje oraz aktywuje główny plugin i później wczytuje oraz uruchamia pozostałe rozszerzenia dopiero w momencie kiedy są one potrzebne. Takie rozwiązanie pozwala na szybsze uruchamianie się aplikacji, a także znaczne zmniejszenie zapotrzebowania na zasoby, ponieważ każdy aktywowany plugin można dezaktywować w momencie kiedy go już nie potrzebujemy.

Każde rozszerzenie może, a główne musi, definiować punkty rozszerzeń (*extension points*). Punkty rozszerzeń, jak sama nazwa wskazuje, są miejscami do których mogą zostać podłączone rozszerzenia. Są one zdefiniowane za pomocą interfejsów, które z kolei są implementowane poprzez poszczególne rozszerzenia. Dostęp do każdego z punktów rozszerzeń w aplikacji jest prosty i umożliwia uzyskanie informacji o wszystkich dostępnych rozszerzeniach dla tego punktu, a także pozwala uruchomić wybrany plugin.

JPF oferuje także oddzielne, dla każdego z rozszerzeń, *ClassLoader'y*, czyli obiekty odpowiedzialne za lokalizacje oraz wczytywanie zasobów (klas, grafiki, plików językowych itp.). Dzięki temu rozwiązaniu, każde z rozszerzeń posiada własną przestrzeń nazw oraz zasoby, co umożliwia uniknięcie konfliktów oraz korzystanie z tych zasobów w prosty sposób.

Elementem opisującym wszystkie atrybuty rozszerzeń jest tzw. *Plugin Manifest*. Jest to plik XML, wczytywany i przechowywany poprzez `PluginRegistry`, który umożliwia w prosty sposób dostęp do wszystkich zawartych w nim danych, jest on podzielony na kilka sekcji:

- `requires` – specyfikuje jakich innych rozszerzeń dany plugin wymaga.
- `runtime` – zawiera informacje na temat mapowań zasobów oraz ich udostępnieniu dla innych pluginów.
- `extension` – zawiera informacje identyfikujące dany plugin oraz określa główną klasę rozszerzenia.
- `extension-point` – specyfikuje punkt rozszerzeń w danym pluginie.

Budowa rozszerzeń działających na platformie JPF jest bardzo prosta. Za przykład może posłużyć rozszerzenie oferujące możliwość eksportu diagramu do pliku graficznego.

Moduł aplikacji który udostępnia funkcjonalność pozwalającą na podłączenie tego typu rozszerzeń definiuje punkt rozszerzeń o identyfikatorze „`Eksport`”. Deklaracja tego punktu jest zawarta w pliku `plugin.xml` opisującego moduł aplikacji. Specyfikuje ona identyfikator danego punktu, a także listę parametrów jakie musi określić rozszerzenie go wykorzystujące. Deklaracja wygląda następująco:

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.4"
                  "http://jpf.sourceforge.net/plugin_0_4.dtd">
<plugin id="umled.core" version="0.0.1" class="umled.core.UMLed">
      ...
```

```
        <extension-point id="Export">
              <parameter-def id="class"/>
              <parameter-def id="name"/>
              <parameter-def id="description"/>
              <parameter-def id="icon" multiplicity="none-or-one"/>
              <parameter-def id="project-plugin-id"multiplicity="one"/>
        </extension-point>
</plugin>
```

Kolejnym elementem dostarczonym przez moduł aplikacji jest interfejs `ExportPlugin`. Interfejs ten specyfikuje metody, które musi implementować rozszerzenie i stanowi punkt wejścia służący do jego uruchomienia. Interfejs ten jest bardzo prosty:

```
public interface ExportPlugin {

            // Metoda wywoływana przez moduł aplikacji w celu
            // uruchomienia rozszerzenia
            public void init();
}
```

Po zaznajomieniu się z oboma wyżej wymienionymi elementami, można przystąpić do implementacji rozszerzenia. Pierwszym krokiem jest stworzenie własnej implementacji interfejsu `ExportPlugin`:

```
public class ImageExportPlugin implements ExportPlugin {

            public void init() {
                  // implementacja działania rozszerzenia

                  // otwarcie okna z wyborem pliku docelowego,
                  // wybór formatu graficznego
            }
}
```

Kolejną niezbędną czynnością jest stworzenie pliku `plugin.xml` opisującego właśnie stworzone rozszerzenie:

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.4"
                  "http://jpf.sourceforge.net/plugin_0_4.dtd">
<plugin id="umled.core.export.image" version="0.0.1">
            <requires>
                  <import plugin-id="umled.core"/>
            </requires>
            <runtime>
                  <library id="umled.core.export.image"
                        path="classes/" type="code" />
                  <library id="lang" path="lang/" type="resources" />
                  <library id="images" path="images/"
                        type="resources" />
            </runtime>
            <extension plugin-id="umled.core"
                        point-id="Export" id="ImageExportPlugin">
```

```
                    <parameter id="class"
                    value="umled.core.export.image.ImageExportPlugin"/>
                    <parameter id="name" value="Image file exporter"/>
                    <parameter id="description"
                          value="Exports diagram to image file."/>
                    <parameter id="icon" value="export_32.png"/>
                    <parameter id="project-plugin-id" value="*"/>
               </extension>
</plugin>
```

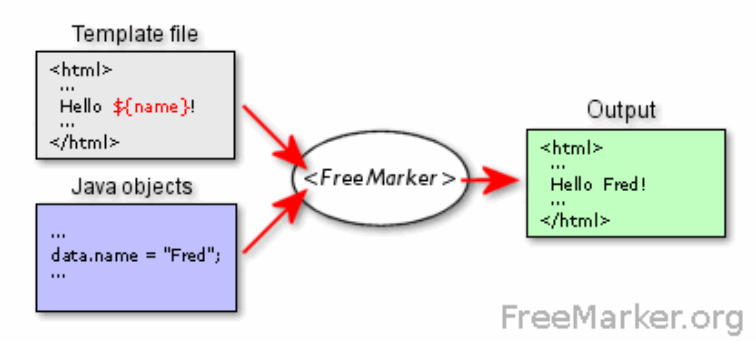
Plik ten specyfikuje kolejno następujące rzeczy:

- Identyfikator rozszerzenia – `umled.core.export.image`.
- Wymagane do działania rozszerzenie, w tym wypadku jest to główny moduł aplikacji o identyfikatorze `umled.core`.
- Określa lokalizacje zasobów rozszerzenia, są to kolejno: skompilowane klasy, pliki językowe oraz pliki graficzne.
- Punkt rozszerzeń dla którego dane rozszerzenie jest przeznaczone. Czyli punkt rozszerzeń `Export`, udostępniony przez moduł o identyfikatorze `umled.core`. Dodatkowo określa wartości wymaganych parametrów, przede wszystkim główną klasę rozszerzenia (wymagane jest podanie pełnej nazwy klasy) implementującą udostępniony przez główny moduł aplikacji interfejs, klasę `umled.core.export.image.ImageExportPlugin`. Kolejnym ważnym parametrem, jest parametr `project-plugin-id`. Parametr ten nie należy do standardowych parametrów wymaganych przez specyfikacje JPF, jednak w przypadku tej aplikacji jest wymagany i został on zdefiniowany na potrzeby tylko tej aplikacji. Określa on rodzaj rozszerzenia projektu aplikacji, z którym jest kompatybilne dane rozszerzenie. Wymaga on podania listy identyfikatorów kompatybilnych rozszerzeń lub znaku „*” w przypadku kompatybilności ze wszystkimi rodzajami.

Tak przygotowane rozszerzenie jest gotowe do działania. Jedyne co pozostaje do zrobienia to umieszczenie go w katalogu aplikacji przeznaczonym dla rozszerzeń. Może ono tam zostać umieszczone w postaci spakowanego pliku JAR lub jako oddzielny katalog zawierający wszystkie pliki. Zarówno w przypadku pliku JAR jak i katalogu, nazwa powinna odzwierciedlać identyfikator rozszerzenia.

3.3. FreeMarker

FreeMarker jest darmową biblioteką Javy oferującą mechanizmy generowania tekstu na podstawie szablonów. Został on zaprojektowany tak aby w prosty i praktyczny sposób mógł być wykorzystany do generowania stron internetowych, jednak bez problemu można go wykorzystać do generowania dowolnego rodzaju tekstu. W związku z podstawowym jego przeznaczeniem wykorzystano w nim wzorzec projektowy *Model-View-Controler* (MVC). Model ten zakłada oddzielenie od siebie danych, widoku oraz logiki, dzięki czemu kod wszystkich tych części jest przejrzysty i nad każdą z tych części mogą pracować różne osoby w tym samym czasie.



Rys. 6. Sposób działania FreeMarkera.

Do tworzenia szablonów według których generowany jest tekst został opracowany specjalny język o dość dużych możliwościach.

Oferuje on:

- podstawowe konstrukcje warunkowe oraz pętle
- tworzenie oraz modyfikacja zmiennych wewnątrz szablonu
- operacje na stringach
- operacje arytmetyczne oraz logiczne
- operacje na tablicach, w tym także na tablicach asocjacyjnych
- możliwość wywoływania metod z obiektów Javy

Dodatkowo FreeMarker posiada wsparcie dla znaczników JSP oraz posiada mechanizmy służące do poruszania się po dokumentach XML. Dostępna jest szeroka gama wtyczek dla różnych IDE oferujących zaawansowane funkcje edycji szablonów, kolorowanie składni, automatyczne podpowiedzi kodu oraz wskazywanie błędów.

Dzięki specjalnym obiektom, tzw. *Object wrappers*, zajmującymi się konwersją danych do formatu odpowiedniego dla wykorzystania w szablonach, możliwe jest użycie danych pochodzących z dowolnych źródeł bez potrzeby modyfikacji szablonów, mogą to być obiekty typu Java bean, dokumenty XML czy też wyniki zapytań SQL. Format danych przekazywany do szablonów umożliwia proste nawigowanie nawet po złożonych obiektach, praktycznie w taki sam sposób jak w kodzie Javy.

Sposób użycia FreeMarkera w aplikacji jest bardzo prosty. Został on poniżej przedstawiony na przykładzie generowania kodu źródłowego Javy z interfejsu UML. Jest to fragment rozszerzenia użytego w opracowanym prototypie.

Następujący fragment kodu przedstawia sposób wywołania biblioteki FreeMarker w aplikacji. Proces ten składa się z czterech kroków. Pierwszym z nich jest inicjalizacja biblioteki oraz jej skonfigurowanie. Odbywa się to poprzez utworzenie obiektu typu `Configuration`, a następnie dostosowanie go do własnych potrzeb. Na przykład poprzez wybór klasy zajmującej się konwersją danych obiektów (Object wrapper). W tym przypadku został wybrany wrapper traktujący wszystkie obiekty jako obiekty typu JavaBean (`ObjectWrapper.BEANS_WRAPPER`). Następnie, niezbędne jest przygotowanie danych. Może to być dowolny obiekt Javy, hierarchia obiektów, lub tablica asocjacyjna, zawierająca obiekty umieszczone pod określonymi kluczami. W przypadku tego przykładu została wykorzystana tablica asocjacyjna. Kolejne kroki to wczytanie szablonu do pamięci (`cfg.getTemplate(nazwa)`) oraz sparsowanie szablonu przy użyciu wcześniej przygotowanych danych (`template.process(dane, strumień)`).

```
Configuration cfg = new Configuration();
// Wybór typu konwertera danych
cfg.setObjectWrapper(ObjectWrapper.BEANS_WRAPPER);

Collection<DShape> interfaces = getInterfaces();
for (Map interfaceData: getInterfacesData(interfaces)){
      // Wczytanie szablonu
      Template template = cfg.getTemplate("interface.ftl");
      //...
      // Uzyskanie ścieżki zapisu pliku
      //...
      FileWriter out = new FileWriter(outputFile);
      // Parsowanie szablonu
      template.process(interfaceData, out);
      out.flush();
}
```

Kolejną częścią, niezbędną do działania biblioteki jest przygotowany szablon. Do jego zdefiniowania używa się specjalnego języka, opracowanego na potrzeby FreeMarkera. W swojej budowie przypomina on język JSP, tak jak on składa się ze znaczników oraz posiada wsparcie dla języka wyrażeń zapewniającego dostęp do danych przekazanych podczas wywołania parsowania szablonu z aplikacji. Sam format języka wyrażeń jest bardzo podobny do języka wyrażeń stosowanego w JSP, a jego użycie wygląda na przykład w następujący sposób `${pracownik.nazwisko}` – co oznacza pobranie wartości pola `nazwisko` z obiektu umieszczonego pod nazwą `pracownik` w przekazanej do przetwarzania tablicy asocjacyjnej z danymi.

```
<#assign bO="{">
<#assign bC="}">

public interface ${root.model.name}${extends} ${bO}
<#list root.model.methods as method>
      <#switch method.visibility>
        <#case 0>
            <#assign visibility="private ">
            <#break>
        <#case 1>
            <#assign visibility="protected ">
            <#break>
        <#case 2>
            <#assign visibility="public ">
            <#break>
        <#default>
            <#assign visibility="">
      </#switch>
      <#if method.isStatic() = true>
            <#assign static="static ">
      <#else>
            <#assign static="">
      </#if>

      <#if method.type == "">
            <#assign type = "void">
      <#else>
            <#assign type = method.type>
      </#if>
```

```
      <#assign params = "">
      <#list method.parameters as param>
            <#assign params = params + param[1] +" "+ param[0]>
            <#if param_has_next>
                  <#assign params = params+", ">
            </#if>
      </#list>

      ${visibility}${static}${type} ${method.name}(${params});
</#list>
${bC}
```

Sama składnia języka jest bardzo prosta, można nawet powiedzieć ze większość nazw poszczególnych znaczników w pełni wyjaśnia ich funkcje. W przykładowym szablonie, zamieszczonym poniżej, zostały wykorzystane następujące znaczniki:

- `<assign nazwa="wartosc">` - tworzy nową zmienną o nazwie „nazwa" i wartości „wartosc".
- `<#list kolekcja as zmienna>` - iteruje po danej kolekcji, odpowiednik `for(typ zmienna: kolekcja)` z Javy.
- `<#switch zmienna>` - instrukcja sterująca, w pełni odzwierciedla instrukcje `switch(zmienna)` z Javy.
- `<#if warunek>` - instrukcja warunkowa, odzwierciedla konstrukcję `if (warunek)`.

Tak przygotowany szablon jest gotowy do użytku, wystarczy go zapisać jako plik z rozszerzeniem FTL w miejscu dostępnym dla aplikacji i wskazanym jako miejsce przechowywania szablonów. Po uruchomieniu parsowania, przy wykorzystaniu wyżej opisanej szablonu, zostaje wygenerowany plik o następującej budowie (jego zawartość oczywiście zależy od przekazanych do parsera danych):

```
public interface Pracownik {

      public String getNIP();
      public Stanowisko getStanowisko();
}
```

4.Wymagana funkcjonalność aplikacji

Narzędzie wspomagające budowę diagramów klas przede wszystkim musi oferować przejrzysty interfejs, który umożliwi wygodną pracę nad diagramem. Interfejs powinien być pozbawiony zbędnych okien i posiadać możliwie największy obszar roboczy. Samo tworzenie diagramów powinno być intuicyjne, a modyfikowanie poszczególnych elementów diagramu nie powinno wymagać nawigacji po wielu oknach.

Dodatkowo na podstawie utworzonych diagramów aplikacja powinna móc generować kod źródłowy, a sam generator kodu powinien być konfigurowalny i w prosty sposób umożliwiać dodanie nowych języków lub modyfikacje istniejących.

Kolejną cechą dobrej aplikacji, jest możliwość rozszerzenia jej funkcjonalności. Powinna ona oferować możliwość dodania nowych typów diagramów, narzędzi czy też nowych formatów eksportu danych.

4.1. Podstawowa aplikacja

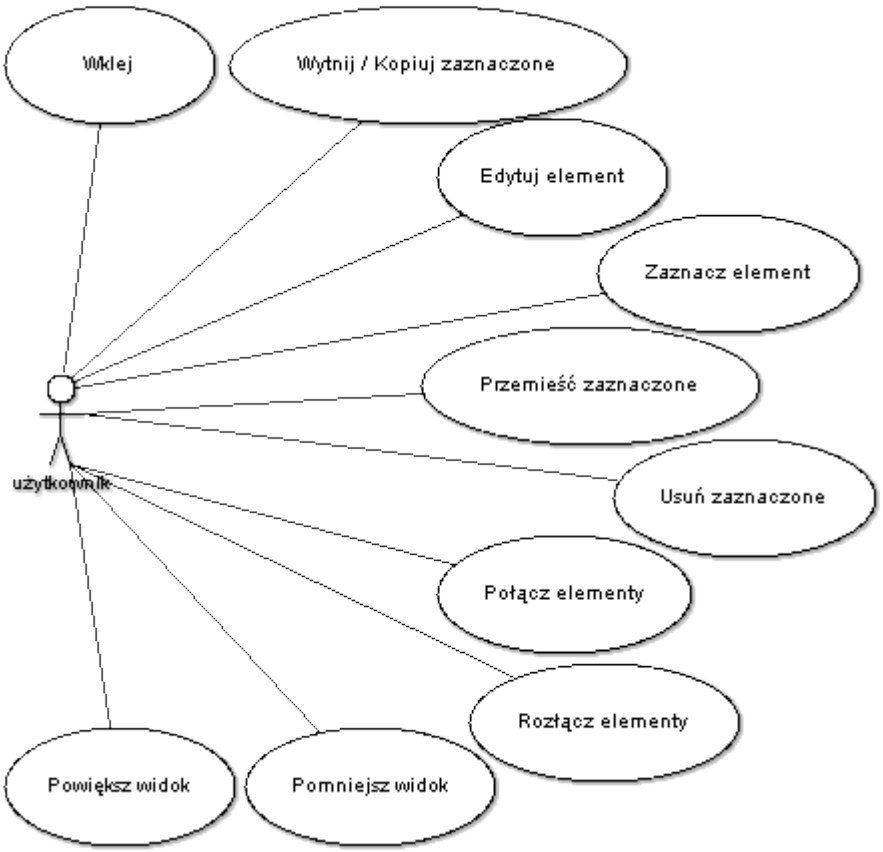
Dla zapewnienia łatwej rozszerzalności aplikacji, musi ona posiadać solidne fundamenty. Wszystkie podstawowe mechanizmy muszą się znajdować w jej głównej części, tak aby przy tworzeniu rozszerzeń nie trzeba było dla każdego z nich je oddzielnie implementować. Do tych mechanizmów można zaliczyć:

- operacje na diagramie
- cofanie zmian
- zapisywanie diagramu do pliku

4.1.1 Operacje na diagramie

Podstawowa aplikacja musi oferować szereg funkcji (przedstawionych na Rys. 7) pozwalających na wygodną i efektywną pracę nad diagramem. Są to operacje takie jak:

- Zaznaczanie pojedynczych elementów, a także zaznaczanie wielu elementów za pomocą wybrania obszaru diagramu przy użyciu kursora myszy. Dodatkowo funkcja ta powinna oferować możliwość swobodnego dodawania oraz usuwania kolejnych elementów do/z zaznaczenia (np. przy użyciu kliknięcia na element wraz z wciśniętym klawiszem CTRL).
- Wytnij, kopiuj, wklej – trzy podstawowe operacje pozwalające oszczędzić czas, powinny się znaleźć w każdym edytorze.
- Łączenie elementów – funkcja niezbędna na diagramie, który przedstawia wzajemne zależności pomiędzy wieloma różnymi elementami.
- Zmiana widoku – możliwość przybliżania oraz oddalania widoku jest niezbędna w pracy nad dużymi diagramami.
- Przemieszczanie elementów diagramu



Rys. 7. Dostępne operacje na diagramie w podstawowej aplikacji.

Dla zapewnienia jak największej dowolności kształtów występujących na diagramie, konstrukcja diagramu powinna opierać się na klasie `JPanel` z biblioteki Swing. Użycie tej klasy przede wszystkim gwarantuje wysoką jakość generowanej grafiki diagramu, jednocześnie zapewniając wysoką wydajność, dzięki wykorzystaniu bibliotek Java2D korzystających z możliwości sprzętowej akceleracji poprzez karty graficzne. Dodatkową zaletą tego rozwiązania jest dostęp do wszystkich mechanizmów biblioteki Swing oferujących m.in. obsługę myszy oraz klawiatury, co pozwala dowolnie kształtować interakcje użytkownika z diagramem. Możliwość wykorzystania Javy2D do wykreślania diagramu pozwala na tworzenie dowolnych kształtów o dowolnym stopniu skomplikowania, a także pozwala w prosty sposób skalować cały diagram – co pozwala na proste i wydajne stworzenie mechanizmu przybliżania oraz oddalania widoku diagramu.

Niestety takie zaimplementowanie diagramu ma też swoje wady. Ponieważ wszystkie elementy diagramu są oddzielnie wykreślane na jednym komponencie, niezbędne jest zaimplementowanie pewnych mechanizmów umożliwiających interakcję użytkownika z poszczególnymi elementami diagramu. Są to mechanizmy takie jak:

- wykrywanie elementów pod kursorem myszy oraz w zaznaczonym obszarze
- wykrywanie kliknięć na elementy diagramu
- przemieszczanie elementów diagramu

4.1.2 Mechanizm cofania zmian

Każdy edytor, niezależnie od tego czy służy do tworzenia diagramów, pisania dokumentów czy obróbki grafiki, powinien posiadać mechanizm pozwalający na cofnięcie wprowadzonych zmian. Funkcja ta przede wszystkim podnosi komfort pracy z narzędziem eliminując możliwość utraty danych poprzez nieprzemyślaną operacje użytkownika. Naprawa takiego błędu może pochłonąć wiele czasu, a zarazem często jest bardzo irytująca i zniechęcająca użytkownika do dalszej pracy.

Mechanizm ten można zrealizować na dwa sposoby:

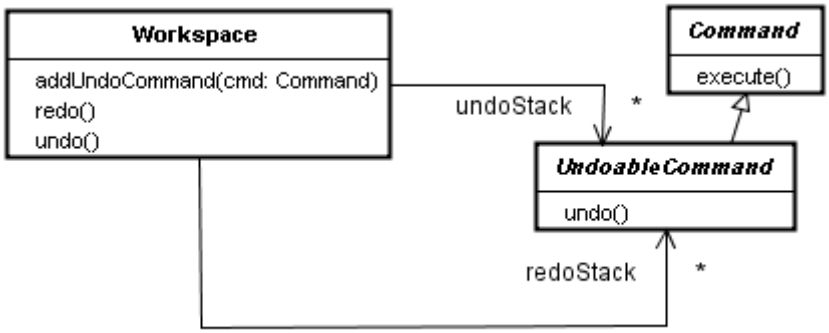
- zapisywanie stanu całego diagramu po każdej operacji i w razie potrzeby wczytanie wcześniejszego stanu
- zapisywanie poszczególnych operacji w postaci obiektów posiadających metody pozwalające cofnąć wykonaną operację (patrz [4] oraz [5])

Zapisywanie stanu całego diagramu jest rozwiązaniem prostym i może być stosowane w aplikacjach w których obszar roboczy jest stosunkowo mały i ma ograniczoną ilość obiektów. Niestety rozwiązanie to nie nadaje się zastosowania w edytorze diagramów, których rozmiar jest ograniczony praktycznie tylko poprzez wyobraźnię projektanta. Złożoność takich diagramów może dochodzić do dziesiątek obiektów i tworzenie kopii tych wszystkich obiektów w pamięci po każdej operacji byłoby rozwiązaniem nie efektywnym.

Idealnym rozwiązaniem w takiej sytuacji jest zastosowanie wzorca projektowego *Command Pattern*, polegającego na opakowaniu każdej operacji w obiekt. Wzorzec ten działa w następujący sposób. W momencie wykonania przez użytkownika dowolnej akcji tworzony jest obiekt jej odpowiadający. Obiekt taki posiada dwie metody:

- `execute` – metoda wykonująca daną operację
- `undo` – metoda cofająca wcześniej wykonaną operację

Po utworzeniu tego obiektu, może on zostać zapisany w dowolnym miejscu, a operacja przez niego wykonana może zostać w prosty sposób cofnięta.



Rys. 8. Budowa mechanizmu cofania zmian.

Dla zapewnienia jednolitości całej aplikacji, wszystkie jej funkcje powinny być realizowane za pomocą podklas klasy `Command`, natomiast funkcje które mają posiadać możliwość cofnięcia wykonanej operacji, powinny rozszerzać `UndoableCommand`, czyli jej podklasę. Alternatywnym rozwiązaniem jest rozszerzanie klasy `Command` przez wszystkie klasy operacji, niezależnie od tego

czy posiadają one możliwość cofnięcia zmian czy nie. Jedynym wyróżnikiem operacji pozwalających na cofnięcie zmian byłby implementowany przez nie interfejs `Undoable`.

Do przechowywania wszystkich wykonanych operacji na diagramie najlepiej nadają się stosy:

- `undoStack` – przechowuje wykonane operacje
- `redoStack` – przechowuje cofnięte operacje

Samo działanie mechanizmu (przedstawionego na Rys.8) jest bardzo proste. W momencie wykonania operacji na diagramie, czyli wywołania metody `execute` klasy `Command`, obiekt ją reprezentujący zostaje dodany na wierzch stosu `undoStack`. Natomiast w przypadku cofnięcia operacji, jej obiekt zostaje zdjęty z wierzchu stosu `undoStack`, zostaje umieszczony na wierzchu stosu `redoStack` i zostaje wywołana metoda `undo` operacji. Należy także pamiętać, że w momencie wykonania każdej nowej operacji, która zostaje dodana do `undoStack`, należy usunąć wszystkie operacje ze stosu `redoStack` aby uniemożliwić ich ponowne wykonanie.

4.1.3 Zapisywanie diagramu do pliku

Podstawowa aplikacja musi posiadać zaimplementowany jednolity format zapisu diagramów do pliku. Jest to niezbędne, ponieważ nie jest możliwe przewidzenie wszystkich możliwych rodzajów diagramów, o które zostanie rozszerzona aplikacja. Tak więc, format musi być tak opracowany, aby był on kompatybilny z każdym przyszłym rodzajem diagramów.

Rozwiązaniem może być połączenie kilku technologii dostępnych w środowisku Java, są to:

- Serializacja – zapisanie aktualnego stanu projektu w postaci strumienia bajtów.
- Właściwości (Properties) – mechanizm pozwalający w łatwy sposób zapisać (oraz odczytać) dane uporządkowane na podstawie par klucz-wartość i zapisane w pliku tekstowym.
- Kompresja – Java oferuje mechanizm kompresji w formacie ZIP, pozwala to na zapisanie szeregu plików w postaci jednego pliku przy jednoczesnym znacznym zmniejszeniu ich rozmiaru.

Wszystkie te technologie pozwolą na uzyskanie formatu zapisu który będzie się cechował zarówno kompatybilnością z dowolnym typem diagramów (serializacja) oraz z dowolnym typem rozszerzeń (plik `properties` musiałby zawierać informacje o wykorzystanych rozszerzeniach do utworzenia danego diagramu), a także zapewniłby stosunkowo mały rozmiar wynikowego pliku.

4.2. Rozszerzalność

Dla zapewnienia rozszerzalności aplikacji, dobrym rozwiązaniem będzie wykorzystanie biblioteki *Java Plugin Framework* (JPF). Biblioteka ta oferuje szereg mechanizmów w znacznym stopniu upraszczających budowę takiej aplikacji. Przede wszystkim, JPF zajmuje się znajdywaniem oraz uruchamianiem poszczególnych rozszerzeń, weryfikuje ich kompatybilność oraz zależności. Dodatkowo, każde rozszerzenie posiada własne, odseparowane od reszty aplikacji, zasoby, do których dostęp, poprzez mechanizmy JPF, jest bardzo prosty. Kolejnym atutem przemawiającym za użyciem tej biblioteki, jest możliwość spakowania całego rozszerzenia w pojedynczy plik JAR, co w znacznym stopniu upraszcza zarządzanie rozszerzeniami.

4.2.1 Struktura rozszerzeń

Każde z rozszerzeń powinno mieć określoną strukturę katalogów:

- classes – katalog wymagany przez JPF, przechowuje on implementacje danego rozszerzenia

- images – katalog przeznaczony dla wszelkiego rodzaju grafiki
- lang – katalog przeznaczony dla plików językowych danego rozszerzenia
- lib – katalog przeznaczony dla dodatkowych bibliotek wymaganych przez dane rozszerzenie

Taka struktura pozwoli na zaimplementowanie prostego API pozwalającego na korzystanie z zasobów poszczególnych rozszerzeń w jednakowy, nie skomplikowany sposób. Metody zarządzające tymi zasobami oraz rozszerzeniami powinny się znaleźć w głównej klasie aplikacji, będącej *Singletonem* (jest to klasa która umożliwia stworzenie tylko jednej jej instancji), dzięki czemu, dostęp do tych metod będzie bardzo prosty i wygodny z dowolnego miejsca aplikacji. Klasa ta powinna zawierać wszelkie informacje na temat dostępnych rozszerzeń oraz ich zasobów, a także może ona przechowywać dane konfiguracyjne aplikacji.

Metody umożliwiające dostęp do zasobów, dla maksymalnego uproszczenia, jako jeden z argumentów, powinny przyjmować główną klasę danego rozszerzenia, co jednoznacznie pozwala określić, w zasobach którego rozszerzenia aplikacja ma szukać plików językowych, grafiki czy też innych danych.

Dla zapewnienia maksymalnie dużych możliwości modyfikacji działania aplikacji przez rozszerzenia, należy pamiętać o takim implementowaniu klas, aby ich metody odpowiadały za jak najmniejszą funkcjonalność i należy unikać tworzenia metod które zajmowałyby się samodzielnie całą skomplikowaną funkcjonalnością. Dzięki takiej konstrukcji klas, możliwe będzie przesłonięcie wybranych metod, modyfikując tylko wybraną, małą część funkcjonalności aplikacji, bez potrzeby implementowania czegoś, co już jest zaimplementowane w metodzie, którą przesłaniamy. Dobrym przykładem takiej funkcjonalności jest metoda obsługująca kliknięcie myszką na diagramie. Reakcji na takie kliknięcie może być wiele i mogą się one różnić od siebie w zależności od miejsca w którym kliknięto, przycisku który został wciśnięty czy też wielu innych warunków. W takim przypadku, zamiast w tej jednej metodzie implementować rozpoznawanie sytuacji w jakiej kliknięto oraz wykonanie operacji wywołanej przez to kliknięcie, lepszym rozwiązaniem będzie zaimplementowanie w tej metodzie tylko samego rozpoznawania sytuacji, natomiast realizacje operacji zaimplementować w oddzielnych metodach. Takie rozwiązanie pozwala w prosty sposób zmodyfikować działanie wybranej operacji, bez potrzeby zbędnego, ponownego, implementowania pozostałej funkcjonalności.

4.2.2 Punkty rozszerzeń

Punkty rozszerzeń określają miejsca w aplikacji, do których mogą zostać podłączone dodatkowe rozszerzenia. Aplikacja powinna zawierać możliwie najwięcej takich punktów, co pozwoli na swobodne jej modyfikowanie. Najważniejszymi punktami rozszerzeń, będą:

- `project` – rozszerzenia oferujące różne typy projektów (rodzaje diagramów).
- `export` – rozszerzenia oferujące różne formaty eksportu danych.
- `menu` – rozszerzenia dodające nowe pozycje do menu aplikacji.

Najprostszym, oferującym najprostszą funkcjonalność rozszerzeń, punktem, jest punkt „menu”, który może zostać zrealizowany jako menu „Narzędzia” w aplikacji, które byłoby zapełniane dodatkowymi narzędziami, dostępnymi dla danego rodzaju projektu (na przykład narzędzie pozwalające na zarządzanie typami danych na diagramie klas).

Punkt „`export`”, podobnie jak punkt „`menu`”, może zostać zaimplementowany w postaci menu aplikacji, w którym pojawiałaby się lista dostępnych formatów eksportu danych, w zależności od rodzaju diagramu.

Z kolei punkt „`project`” odpowiada za wszystkie rozszerzenia oferujące różne rodzaje diagramów. Przy każdym tworzeniu nowego diagramu, powinna zostać wyświetlona lista dostępnych

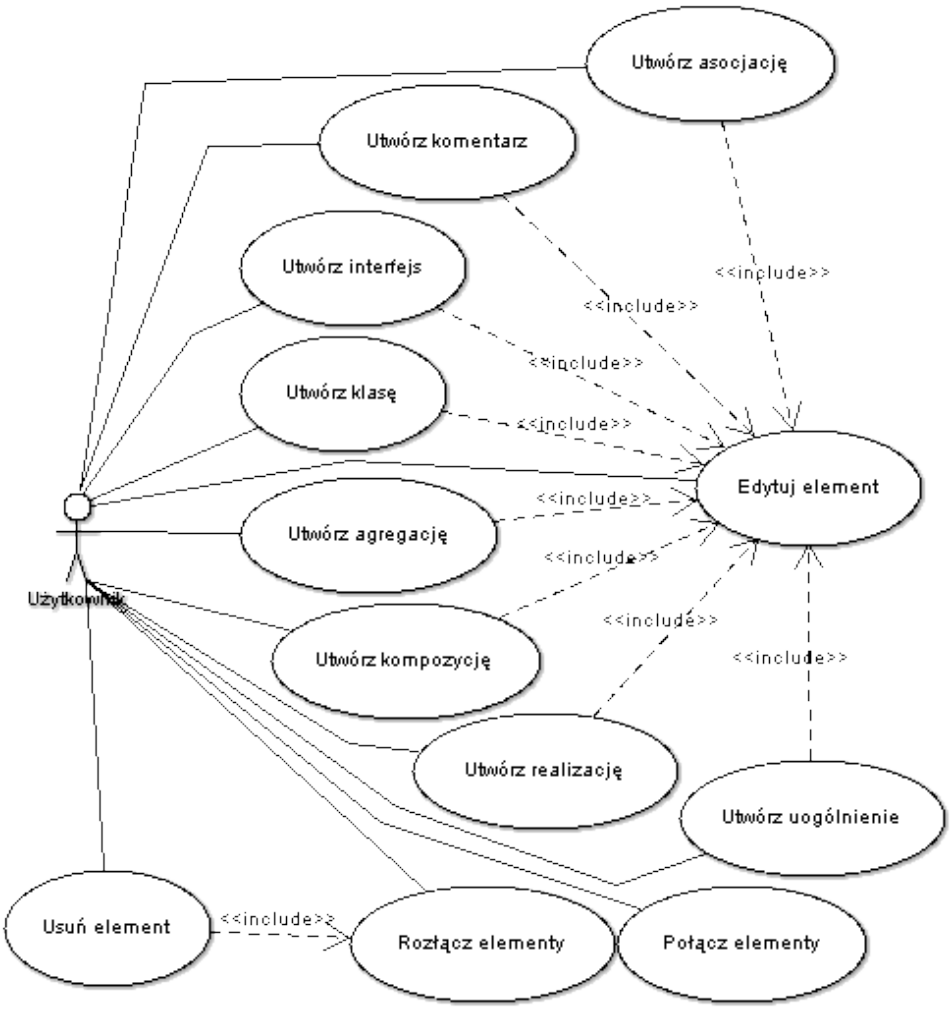
typów diagramów, dodatkowo, punkt ten powinien być wykorzystywany przy wczytywaniu zapisanych diagramów, dla sprawdzenia dostępności rozszerzenia, kompatybilnego z zapisanym diagramem. Implementacja tego punktu jest najbardziej skomplikowana z trzech wymienionych, ponieważ wpływa na działanie praktycznie całej aplikacji. Należy pamiętać, że rozszerzenie oferujące funkcjonalność diagramu, musi posiadać własną implementacje klasy `Project` (przechowuje podstawowe informacje na temat danego diagramu, a także jest główną klasą danego rozszerzenia) oraz klasy `ProjectPane` (klasa odpowiedzialna za wygląd diagramu). Dodatkowo w przypadku takiego rozszerzenia, niezbędne jest zaimplementowanie wszystkich rodzajów elementów występujących na diagramie, razem z zasadami ich wykreślania, a także zaimplementowanie własnej klasy odpowiedzialnej za obsługę myszy, na zdarzenia której w różny sposób mogą reagować poszczególne elementy.

4.3. Tworzenie diagramów klas

Tworzenie diagramów klas UML powinno być przede wszystkim proste. Osoba która nigdy wcześniej nie miała styczności z aplikacją, powinna bez problemu, w ciągu kilku minut, nauczyć się obsługi. Z tego powodu, aplikacja nie może być przeładowana różnego rodzaju oknami oraz paskami narzędzi z dziesiątkami przycisków. Najlepiej, jeśli okno zawierałoby tylko jeden pasek narzędzi – główny pasek narzędzi aplikacji, a całą funkcjonalność dotyczącą tworzenia diagramów można umieścić w kontekstowych menu, których zawartość (dostępne funkcje) będzie się różnić w zależności od aktualnie wybranego elementu. Zaletą takiego rozwiązania jest duży obszar roboczy, praktycznie obejmujący całe okno aplikacji, przy jednoczesnym zachowaniu pełnej funkcjonalności aplikacji (Rys. 9).

Edycja (oraz tworzenie) poszczególnych elementów diagramu, powinno się odbywać w odpowiednich oknach pojawiających się w momencie edycji (lub tworzenia), powinny one jasno opisywać wszystkie właściwości danego elementu oraz dostęp do żadnych z właściwości danego elementu, nie powinien wymagać otwierania kolejnego okna z wewnątrz aktualnego. Dzięki takiemu zbudowaniu okien, użytkownicy nie będą musieli błądzić po skomplikowanej hierarchii okien.

Sama implementacja diagramu, powinna korzysta klas dostarczonych przez główną aplikację, czyli `DShape` dla elementów takich jaki klasy i interfejsy, oraz `DPolylineShape` dla elementów będących relacjami oraz dla komentarzy. Dobrym rozwiązaniem będzie także zastosowanie wzorca *Model-View-Controler* (patrz [3]), czyli oddzielenia danych od algorytmów zajmujących się wykreślaniem elementów oraz od algorytmów zarządzających interakcją z użytkownikiem. Wszystkie podklasy klasy `DShape`, reprezentujące elementy diagramu, powinny posiadać tylko dane niezbędne do poprawnego ich wykreślenia, natomiast dane dotyczące treści diagramu UML, czyli na przykład nazwa klasy, powinny być umieszczone w oddzielnych obiektach, przechowujących wyłącznie dane UML.



Rys. 9. Operacje na diagramie dostępne dla użytkownika.

4.3.1 Klasy oraz interfejsy

Tworzenie klas oraz interfejsów UML powinno być proste oraz intuicyjne. Po wyborze odpowiedniej pozycji z menu kontekstowego, powinno pojawić się okno w którym użytkownik może określić właściwości danego elementu:

- nazwę
- pakiet
- określić czy jest to klasa abstrakcyjna (tylko w przypadku klas)
- określić wymagane przez tą klasę biblioteki

Po wykonaniu tych czynności, zostaje utworzona klasa, nie zawierająca żadnych metod ani pól, które można dodać do klasy dopiero po jej zaznaczeniu i wybraniu odpowiedniej pozycji z menu kontekstowego.

Dodawanie pól oraz metod do klasy jest analogiczne do tworzenia samej klasy. Po wybraniu z menu odpowiedniej pozycji, użytkownikowi wyświetlone zostaje okno zawierające wszystkie właściwości tworzonego pola lub metody:

- nazwa
- typ
- widoczność
- określenie czy dane pole lub metoda operuje na ekstensji klasy

Dodatkowo, metody umożliwiają zdefiniowanie następujących właściwości:

- lista parametrów metody
- określenie czy dana metoda jest metodą abstrakcyjną (tylko w przypadku klas abstrakcyjnych)
- ciało metody

Wybór typu powinien pozwalać na łatwe dodanie nowego typu, a także powinien umożliwiać wybór jednego z już użytych na diagramie typów, co pozwoli zaoszczędzić czas przy tworzeniu diagramu, a także pozwoli na wyeliminowanie części błędów pisowni w nazwie typu.

Lista parametrów metod, powinna pozwalać na zdefiniowanie dowolnej liczby parametrów o różnych typach. Wybór typu danego parametru powinien działać w taki sam sposób w jakim działa wybór typu metody lub pola. Dodatkowo, powinna być dostępna możliwość zmiany kolejności parametrów.

Dla uproszczenia obsługi, dostęp do właściwości istniejących już metod oraz pól nie powinien wymagać wyświetlenia najpierw właściwości klasy lub interfejsu, do którego należą. Dobrym i prostym rozwiązaniem jest zastosowanie dwukrotnego kliknięcia na wybraną metodę lub pole. Po wykonaniu tej czynności użytkownikowi przedstawione by zostało okno z właściwościami, takie samo jak podczas tworzenia metody, dodatkowo oferując możliwość usunięcia z klasy danej metody lub pola.

4.3.2 Połączenia

Tworzenie połączeń, podobnie jak tworzenie klas, powinno być maksymalnie uproszczone i odbywać się w analogiczny sposób. Jedynie w przypadku prostych połączeń (realizacja, uogólnienie), zostają one utworzone od razu po wybraniu przez użytkownika odpowiedniej pozycji z menu i nie jest potrzebne wyświetlenie okna właściwości, ponieważ połączenia te nie posiadają żadnych atrybutów. Wszystkie pozostałe połączenia, czyli asocjacje, agregacje oraz kompozycje, przed utworzeniem dają możliwość ustawienia ich atrybutów:

- oddzielne role dla każdego z końców asocjacji
- liczności dla każdego z końców
- kierunek asocjacji

Bardzo przydatną funkcjonalnością, byłoby wyświetlanie, w oknie właściwości asocjacji, informacji o klasach, do których aktualnie są podłączone jej końce. Dodatkowo, należałoby tak umieścić poszczególne atrybuty asocjacji (takie jak liczności i role), aby w prosty sposób można było zidentyfikować do której z klas dany atrybut się odnosi.

Samo podłączanie połączenia, powinno odbywać się poprzez przeciągniecie jednego z końców nad klasę lub interfejs, a następnie upuszczenie go na nią. Ta czynność powinna spowodować utworzenie połączenia. Ponieważ wiele połączeń może być podłączonych do jednej klasy, niezbędne jest zaimplementowanie algorytmu który, w przejrzysty sposób rozłoży podłączone końce połączeń na bokach klasy. Wybór boku może się odbywać na podstawie typu relacji (np. w przypadku uogólnienia, wykorzystywane byłyby tylko górny i dolny bok klas, w zależności od użytego końca połączenia) oraz na podstawie kąta pod jakim podłączone jest połączenie. Natomiast samo umiejscowienie końca na boku, jest uzależnione od dwóch czynników:

- odstępu pomiędzy poszczególnymi końcami, który może być zależny od ilości podłączonych połączeń
- kolejności końców, która może być uzależniona od kąta pod jakim są podłączone poszczególne połączenia.

Ważnym aspektem połączeń, jest też odpowiednie pozycjonowanie oraz wyświetlanie liczności oraz ról, tak aby w łatwy sposób można było zidentyfikować do którego połączenia dany opis należy, a także tak aby opis nie był przesłaniany poprzez inne elementy diagramu.

Dodatkowo, podczas operacji przeciągania końca połączenia, niezbędne jest sprawdzenie kompatybilności, tzn. czy utworzenie połączenia danego końca z określoną klasą lub interfejsem jest możliwe. Wyniki tego testu powinny być widoczne dla użytkownika przed próbą wykonania połączenia, czyli na przykład w momencie przeciągania końca połączenia nad klasą, klasa zostanie podświetlona jeśli połączenie jest możliwe.

Odłączanie połączeń powinno odbywać się w analogiczny sposób do ich tworzenia, użytkownik przeciąga wybrany, podłączony koniec w wolne miejsce na diagramie.

Użyteczną funkcjonalnością połączeń, jest też możliwość dowolnego modyfikowania ich kształtu na przykład poprzez dodawanie dodatkowych wierzchołków. Przemieszczanie tych wierzchołków umożliwi tworzenie dowolnych kształtów połączenia, co z kolei pozwoli na tworzenie bardziej przejrzystych diagramów.

4.3.3 Komentarze

Komentarze są bardzo prostymi elementami diagramu. Ich funkcjonalność ogranicza się tylko do podłączenia określonego tekstu do dowolnego elementu diagramu. Sam proces tworzenia komentarza przebiega tak samo jak w przypadku klas czy relacji, użytkownik wybiera z menu pozycję komentarza, a następnie w wyświetlonym oknie, wpisuje jego treść.

Sposób tworzenia połączeń komentarzy z innymi elementami jest taki sam jak w przypadku tworzenia połączeń relacji.

4.4. Generowanie kodu źródłowego

Generowanie kodu źródłowego na podstawie diagramu musi być zaimplementowane w taki sposób, aby łatwo i bez potrzeby pisania wielu klas, można było dodać nowe formaty kodu, lub wprowadzić modyfikacje w już istniejące. Z tego powodu, dobrym rozwiązaniem jest zastosowanie biblioteki *FreeMarker*, która oferuje przetwarzanie tekstu na podstawie szablonów. Składnia języka opisującego szablony umożliwia wykorzystanie instrukcji warunkowych, a także pętli, co pozwala na zbudowanie dowolnie skomplikowanych reguł określających wygląd generowanego tekstu. Same szablony nie wymagają kompilacji i są przechowywane w oddzielnych plikach, co umożliwia ich modyfikację bez potrzeby ponownej kompilacji całej aplikacji.

4.4.1 Przygotowanie danych

Ponieważ *FreeMarker* wymaga odpowiedniego formatu danych (tablicy asocjacyjnej, np. `Map` w Javie) do przetwarzania szablonów, niezbędne jest przekształcenie do tego formatu diagramu. Operację tą trzeba przeprowadzić dla każdej klasy oraz interfejsu osobno , tak aby zebrać wszystkie niezbędne informacje.

W przypadku interfejsów, wystarczy jeśli do szablonu zostaną przekazane tylko dane zawarte w klasie przechowującej dane UML danego interfejsu oraz zawartość połączonych z nim komentarzy. Natomiast w przypadku klas niezbędne jest zebranie większej ilości informacji:

- dane UML danej klasy
- dane UML klasy będącej uogólnieniem aktualnej klasy
- interfejsy które implementuje dana klasa
- asocjacje klasy wraz z danymi UML klas lub interfejsów będących częścią tych asocjacji oraz komentarzami ich dotyczącymi

4.4.2 Budowa szablonów

Sama budowa szablonu przede wszystkim powinna pozwalać zbudować prosty kod źródłowy klas oraz interfejsów Java, jednak dobrym rozwiązaniem byłoby zaimplementowanie bardziej zaawansowanych konstrukcji wynikających z budowy diagramu.

Jeśli klasa rozszerza klasę abstrakcyjną, to należałoby wygenerować w niej szkieletową implementację abstrakcyjnych metod z klasy rozszerzanej. Podobnie w przypadku interfejsów, można wygenerować szkieletową implementację ich metod.

W przypadku asocjacji, należy dodatkowo, w zależności od jej liczności, dodać metody odpowiedzialne za pilnowanie tego limitu. Czyli pozwalające dodać nowy obiekt do kolekcji (reprezentującej daną asocjację) tylko wtedy, kiedy aktualna liczba obiektów w kolekcji jest mniejsza niż maksymalna liczba określona przez jej liczność. Podobnie w przypadku usuwania obiektów z kolekcji. Kolejnym aspektem odpowiedniej implementacji asocjacji jest zagwarantowanie minimalnej ilości obiektów w kolekcji, w przypadku asocjacji, której liczność wymaga przynajmniej jednej instancji. Można to zagwarantować poprzez dodanie odpowiedniej implementacji do konstruktora. Wymagane by było przekazanie jako parametr do konstruktora instancji, wymaganej przez asocjacje klasy, a następnie dodanie tej instancji do kolekcji reprezentującej daną asocjację.

Dodatkowo, funkcja generowania kodu źródłowego, powinna umożliwiać, oprócz samego generowania plików klas i interfejsów, tworzenie struktury katalogów odpowiadającej zadeklarowanemu w klasach pakietowi,

4.4.3 Rozszerzalność

Dla zapewnienia możliwości łatwego dodawania nowych formatów generowanego kodu, można wykorzystać mechanizm rozszerzeń z biblioteki JPF. Takie rozwiązanie zapewni niezawodny i jednolity sposób działania wszelkiego rodzaju rozszerzeń w całej aplikacji.

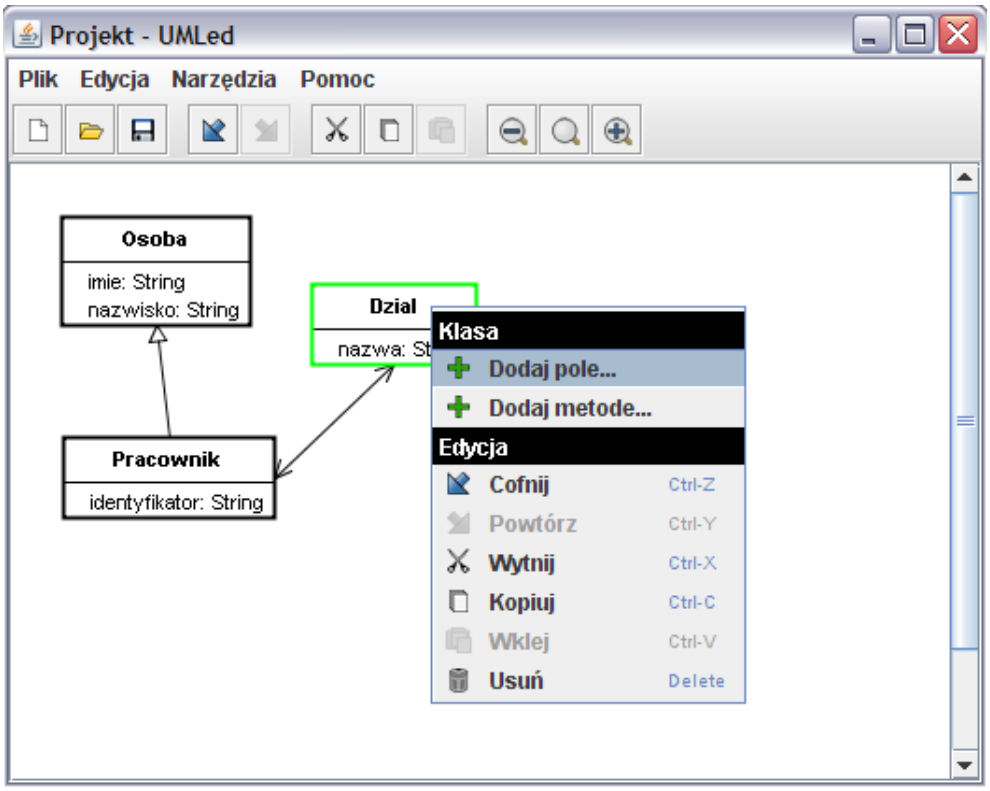
Rozszerzenie takie składałoby się dosłownie z kilku plików:

- `class.ftl` – zawiera szablon generowania klas
- `interface.ftl` – szablon generowania interfejsów
- `plugin.xml` – deskryptor rozszerzenia

Rozwiązanie to jest schludne oraz bardzo proste w użyciu dla każdego, kto by chciał w przyszłości dodać dowolnego typu format.

5.UMLed - prototyp

UMLed jest aplikacją napisaną z wykorzystaniem platformy Java 5 Standard Edition oraz bibliotek Java Plugin Framework (odpowiedzialna za zarządzanie rozszerzeniami) i FreeMarker (odpowiedzialna za przetwarzanie tekstu z użyciem zdefiniowanych szablonów). Wykorzystanie Javy, dzięki jej maszynie wirtualnej (Java Virtual Machine), umożliwia uruchomienie aplikacji pod kontrolą dowolnego systemu operacyjnego, a także zapewnia spójny i jednolity interfejs, niezależnie od środowiska w jakim jest uruchamiana.



Rys. 10. UMLed.

Prototyp oferuje pełne wsparcie dla tworzenia diagramów klas UML w wygodny i przejrzysty sposób. Edytor zawiera wszelkie niezbędne mechanizmy potrzebne do pracy nad diagramami, zawiera takie niezbędne funkcje jak mechanizm cofania zmian, czy też funkcje wytnij, kopiuj oraz wklej pozwalające zaoszczędzić czas wymagany na stworzenie diagramu. Interfejs aplikacji (przedstawiony na Rys.10) jest maksymalnie uproszczony i pozbawiony wszystkich zbędnych elementów, dzięki czemu obszar roboczy jest zajmuje praktycznie całą powierzchnie okna. Do jego obsługi niezbędna jest mysz, za pomocą której tworzy się oraz modyfikuje wszystkie elementy diagramu. Dostęp do poszczególnych funkcji dotyczących operacji na diagramie, jest zapewniony poprzez menu kontekstowe, które w zależności od wybranego elementu na diagramie wyświetlają możliwe do wykonania operacje.

Aplikacja oferuje też możliwość eksportu diagramu do pliku graficznego w formatach GIF, JPG, PNG oraz BMP, a także umożliwia wygenerowania kodu źródłowego Javy na podstawie diagramu. Sam mechanizm generowania oparty jest na szablonach, dzięki czemu w prosty sposób można dodawać nowe formaty kodu źródłowego czy też zupełnie nowe języki programowania.

UMLed, dzięki wykorzystaniu biblioteki Java Plugin Framework, jest zbudowany w oparciu o rozszerzenia, które są podłączane do głównej części aplikacji, oferującej tylko podstawowe funkcje w zakresie edycji diagramów dowolnego typu. Możliwe jest rozszerzenie jego funkcjonalności o dowolny nowy typ diagramów, nowe rodzaje eksportu danych czy też dodatkowe narzędzia przeznaczone do pracy nad diagramami. Dodatkowo każdy z modułów oferujących rozszerzenie funkcjonalności podstawowej aplikacji może posiadać dowolną ilość swoich rozszerzeń, które można zaimplementować korzystając z istniejących w aplikacji mechanizmów.

5.1. Budowa podstawowej części aplikacji

Podstawowy moduł aplikacji jest fundamentem dla wszystkich dodatkowych rozszerzeń. Oferuje on implementację podstawowych funkcji niezbędnych do pracy nad diagramem oraz odpowiada za zarządzanie poszczególnymi rozszerzeniami, a także ich zasobami.

Ze względu na wykorzystanie biblioteki Java Plugin Framework (JPF), niezbędne jest napisanie całej aplikacji w specyficzny sposób. JPF najbardziej efektywnie działa wtedy, kiedy ma do czynienia wyłącznie z modułami będącymi rozszerzeniami zgodnymi z jego specyfikacją. Czyli każda część aplikacji jest rozszerzeniem opisanym deskryptorem XML oraz jest umieszczona w jednym katalogu lub spakowana w archiwum JAR. Jedno z takich rozszerzeń może zostać oznaczone jako główne, tzw. *Application Plugin*. Rozszerzenie takie jest traktowane jako punkt wejścia do uruchomienia całej aplikacji, a samo uruchomienie odbywa się za pomocą specjalnej biblioteki do tego przeznaczonej (`jpf-boot`), której gotowa implementacja jest dostarczona razem z JPF. Biblioteka ta dodatkowo, oprócz zarządzania dostępnymi rozszerzeniami, umożliwia wyświetlenie ekranu powitalnego podczas wczytywania poszczególnych elementów aplikacji.

Dodatkowo podstawowy moduł, definiuje trzy punkty rozszerzeń pozwalające na podłączenie dodatkowych modułów rozszerzających lub modyfikujących funkcjonalność aplikacji. Są to punkty o nazwach:

- Project – pozwalający na podłączenie modułów dodających nowe rodzaje diagramów (projektów).
- Export – dla modułów oferujących nowe formaty eksportu diagramów.
- Tools – dla modułów udostępniających narzędzia wspomagające prace z diagramami.

5.1.1 Zarządzanie zasobami

Sercem całej aplikacji jest klasa `App`, która odpowiada za zarządzanie zasobami poszczególnych rozszerzeń. Klasa ta jest zaimplementowana jako *Singleton*, czyli klasa która umożliwia utworzenie tylko jednej jej instancji w danej maszynie wirtualnej. Sama implementacja takiej klasy jest bardzo prosta, a od zwykłych klas różni się tylko trzema detalami:

- Klasa posiada tylko jeden prywatny konstruktor, dzięki czemu niemożliwie jest utworzenie instancji tej klasy wewnątrz innej klasy.
- Posiada statyczne, prywatne pole zawierające referencję do instancji danej klasy.
- Posiada statyczną, publiczną metodę zwracającą referencję do instancji klasy. W przypadku kiedy instancja nie istnieje, metoda tworzy ją, poprzez wywołanie prywatnego konstruktora.

Taka implementacja tej klasy zapewnia łatwy dostęp do niej z dowolnego miejsca. Dodatkowo, klasa ta zawiera referencje do najważniejszych elementów aplikacji takich jak główne okno aplikacji (klasa `MainFrame`) czy też obszar roboczy (klasa `Workspace`).

Głównym zadaniem tej klasy jest ułatwienie pracy z rozszerzeniami oraz zarządzanie ich zasobami. Przechowuje ona wszelkie informacje konfiguracyjne uruchamianych rozszerzeń oraz oferuje metody opakowujące najważniejsze funkcje JPF, tak aby korzystanie z nich było wygodniejsze. Wszystkie te dane przechowywane są w tablicy asocjacyjnej, w której jako klucz wykorzystywane są obiekty reprezentujące klasę główną danego rozszerzenia. Dodatkowo, w oddzielnej tablicy przechowywane są referencje do zdefiniowanych w aplikacji punktów rozszerzeń, czyli miejsc, do których możliwe jest podłączenie zewnętrznych modułow.

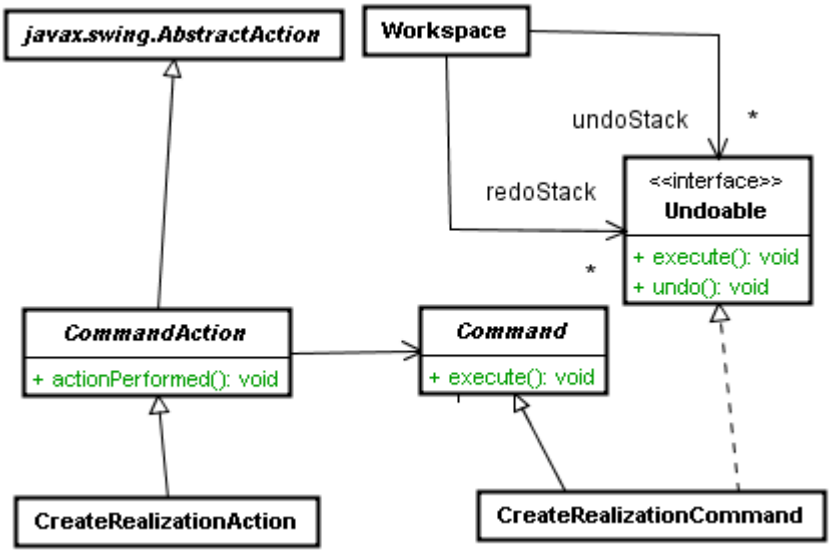
Dzięki przechowywaniu tych wszystkich informacji, możliwe było zaimplementowanie bardzo prostego w użyciu mechanizmu dostępu do zasobów. Na przykład, wczytanie grafiki znajdującej się wśród zasobów dowolnego rozszerzenia, wymaga jedynie wywołania metody klasy `App` z dwoma argumentami: nazwą pliku graficznego oraz klasą główna danego rozszerzenia.

W podobny sposób działa mechanizm internacjonalizacji, jednak z jedną różnicą. Wymaga on dodatkowego wczytania plików językowych do pamięci przed użyciem. Każde z rozszerzeń może posiadać dowolną liczbę plików językowych, których format musi być zgodny ze standardowymi plikami zasobów Javy (`ResourceBundle`). Są to zwykłe pliki tekstowe, zawierające w oddzielnych liniach kolejne wpisy, zawierające klucz identyfikujący daną wartość, oraz samą wartość. Aby plik został rozpoznany przez aplikacji musi on posiadać określoną nazwę: `lang.properties`, dla domyślnego języka, lub np. `lang_pl.properties`, dla języka polskiego. Po wczytaniu pliku do pamięci, samo korzystanie z jego zasobów jest praktycznie identyczne jak w przypadku dostępu do plików graficznych. Sprowadza się ono do wywołania metody z klasy `App` wraz z argumentami określającymi klucz interesującego nas wpisu oraz klasę główną rozszerzenia.

5.1.2 Mechanizm cofania zmian

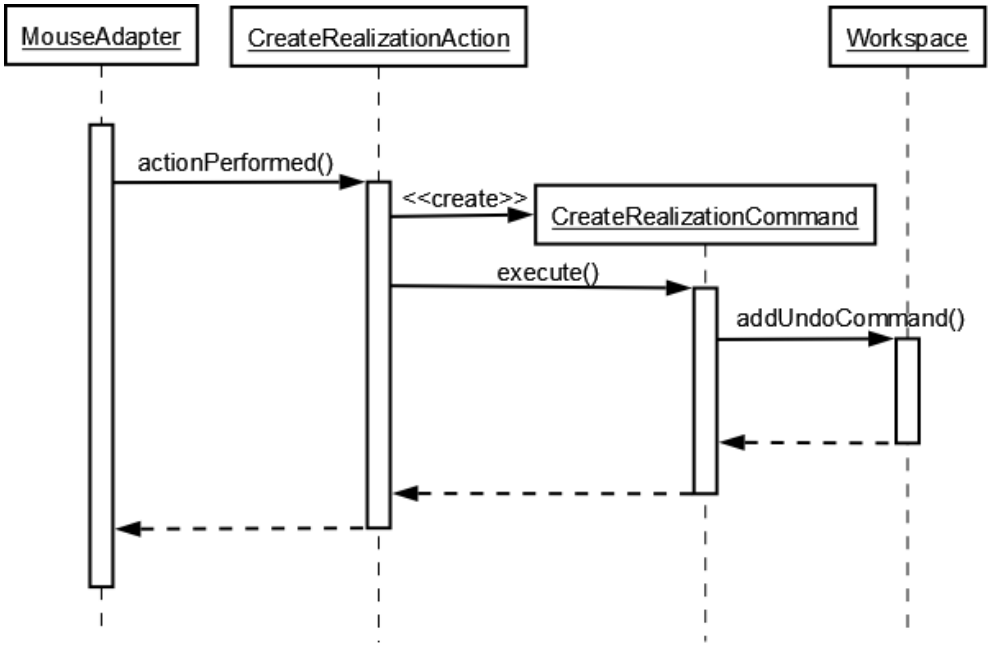
Zaimplementowany w aplikacji mechanizm cofania zmian (przedstawiony na Rys. 11) opiera się na wzorcu komendy (ang. *Command Pattern*). Wzorzec ten polega na opakowaniu określonych operacji w obiekty, które mogą być później przechowywane i wykorzystane, na przykład do cofnięcia wykonanej operacji.

Poszczególne operacje w aplikacji są reprezentowane poprzez klasy rozszerzające abstrakcyjną klasę `Command`. Każdy rodzaj operacji posiada oddzielną, reprezentującą ją klasę, co umożliwia wywołanie w prosty sposób dowolnej operacji z każdego miejsca aplikacji. Klasy te posiadają bezparametrową metodę `execute` zawierającą algorytm wykonujący daną operację. W przypadku operacji, które posiadają możliwość cofnięcia zmian, klasa komendy musi implementować interfejs `Undoable`. Interfejs ten, oprócz metody `execute` odpowiedzialnej za wykonanie operacji, specyfikuje także metodę `undo`, której implementacja powinna zawierać algorytm cofający wprowadzone zmiany. W przypadku komend operujących na diagramie, bardzo często niezbędne jest, dodatkowo wykorzystanie pola w klasie komendy, do przechowywania kopii całego, właśnie zmienianego, obiektu.



Rys. 11. Implementacja wzorca komendy w aplikacji.

Do przechowywania wszystkich wykonany operacji, służą dwa stosy w klasie `Workspace`, jeden z nich do przechowywania wykonanych operacji, a drugi do przechowywania cofniętych operacji. W momencie wykonania nowej operacji, zostaje ona dodana na wierzch stosu przechowującego operacje wykonane, czyli stosu `undoStack`. Jeśli użytkownik zdecyduje się ponownie wykonać cofniętą operację, to z wierzchołka stosu zostaję ona zdjęta i następuje jej ponowne wykonanie, a następnie umieszczenie jej na wierzchołku drugiego stosu, odpowiedzialnego za przechowywanie operacji cofniętych, czyli `redoStack`.



Rys. 12. Sposób wywołania wykonania operacji w aplikacji.

Powyżej, na rys. 12, został przedstawiony sposób działania mechanizmu wykonującego określoną operacje, w tym przypadku jest to operacja tworząca na diagramie element UML reprezentujący realizację interfejsu. Sam mechanizm jest bardzo prosty w działaniu. W momencie wybrania opcji utworzenia realizacji przez użytkownika, klasa zajmująca się obsługą myszy (`MouseAdapter`), wywołuje wybraną z menu akcję, reprezentującą określoną operację w interfejsie użytkownika (`CreateRealizationAction`). Akcja ta następnie tworzy nową instancję klasy reprezentującą żądaną operację, czyli klasę `CreateRealizationCommand`, po czym wywołuje metodę `execute`, która to powoduje wykonanie żądanej operacji. Ponieważ operacja tworzenia realizacji na diagramie, jest operacją którą można cofnąć, dodatkowo, metoda `execute`, poprzez wywołanie metody `addUndoCommand` z klasy reprezentującej obszar roboczy (`Workspace`), rejestruje obiekt operacji na stosie operacji wykonanych.

5.1.3 Punkt rozszerzeń „Project”

Punkt ten umożliwia dołączenie dodatkowych modułów rozszerzających funkcjonalność aplikacji o nowe typy diagramów (projektów). Rozszerzenia dla niego przeznaczone mogą praktycznie całkowicie zmienić funkcjonalność aplikacji. Dla osiągnięcia maksymalnej kontroli nad funkcjonalnością aplikacji powinny one posiadać własną implementację następujących klas:

- `ProjectPlugin` – jest to główna klasa rozszerzenia, reprezentuje ona aktualnie otwarty w aplikacji projekt.
- `Project` – klasa ta reprezentuje aktualnie otwarty w aplikacji projekt
- `ProjectPane` – klasa ta reprezentuje panel przedstawiający i wykreślający diagram w aplikacji.
- `UMLedMouseAdapter` – klasa odpowiedzialna za interpretację akcji wywołanych za pomocą myszy przez użytkownika, pozwala ona na swobodną modyfikacje zachowania aplikacji

Lista dostępnych rozszerzeń dla tego punktu jest wczytywana podczas tworzenia nowego projektu, po wyborze rodzaju projektu następuje inicjalizacja rozszerzenia poprzez odpowiadającą mu klasę `ProjectPlugin`, która to zajmuje się dalszym tworzeniem projektu.

Podczas wczytywania wcześniej utworzonego projektu z pliku, lista rozszerzeń jest także sprawdzana pod kątem dostępności rozszerzenia użytego do utworzenia tego pliku.

5.1.4 Punkt rozszerzeń „Export”

Ten punkt rozszerzeń jest przeznaczony dla modułów zajmujących się eksportem diagramów do innych formatów. Rozszerzenia dla niego przeznaczone mogą w dowolny sposób przekształcać diagram ponieważ cała logika procesu eksportu jest umieszczona właśnie w rozszerzeniach i praktycznie żadnych wymagań dla rozszerzeń tego typu nie ma. Jedyne co jest wymagane od rozszerzeń to aby posiadały własną implementację klasy `ExportPlugin`, która to zajmuje się inicjalizacją modułu.

Ponieważ rozszerzenia tego typu mogą być przeznaczone tylko dla pewnego rodzaju diagramów, np. możliwość eksportu diagramu do kodu źródłowego dla diagramów klas, konieczne było dodanie dodatkowego parametru do deskryptora rozszerzenia. Parametr ten, o nazwie `project-plugin-id`, powinien zawierać identyfikator rozszerzenia dla którego jest przeznaczony, lub znak „`*`” oznaczający, że dane rozszerzenie może być zastosowane do dowolnego rodzaju diagramu. Parametr ten jest sprawdzany przy wczytywaniu listy dostępnych rozszerzeń, po czym wszystkie niekompatybilne rozszerzenia zostają usunięte z listy wyświetlanej użytkownikowi.

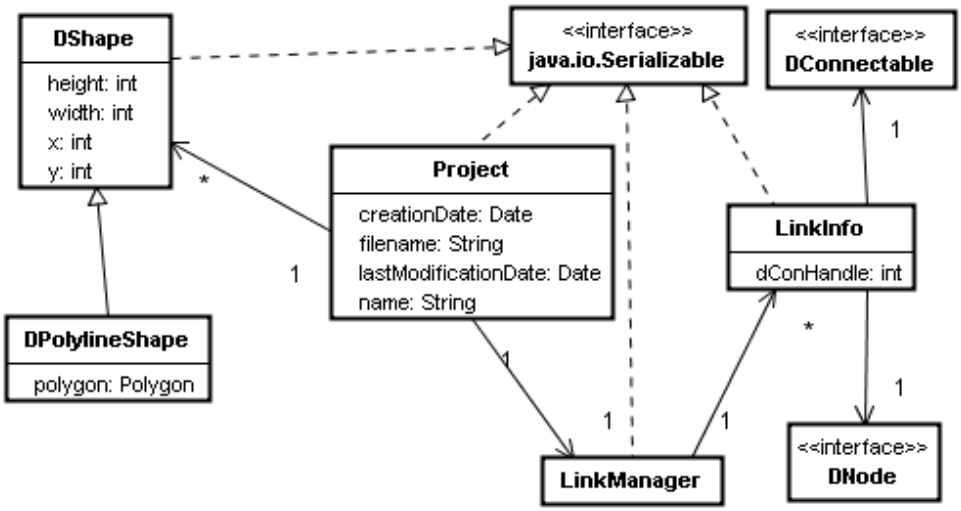
5.1.5 Punkt rozszerzeń „Tools”

Punkt rozszerzeń „Tools” jest przeznaczony dla modułów oferujących dodatkowe narzędzia. Rozszerzenia tego typu, podobnie jak w przypadku punktu „Export”, wymagają określenia rodzaju diagramu dla którego są przeznaczone. Dostęp do ich funkcjonalności wewnątrz aplikacji jest zapewniony dzięki menu „Narzędzia”, do którego w trakcie tworzenia nowego projektu, lub wczytywania istniejącego, dodawane są dostępne, kompatybilne z projektem narzędzia. Dla wygody implementacji, narzędzia są dodawane do menu w postaci obiektów typu `Action`, z biblioteki Swing Javy. Obiekty te pozwalają na zdefiniowanie wyświetlanej nazwy oraz ikony wyświetlanej w menu. Z tego powodu, rozszerzenia tego typu, muszą posiadać własną implementację klasy `Action`, która odpowiedzialna będzie za uruchomienie danego narzędzia. Dodatkowo, niezbędna jest implementacja klasy `ToolPlugin`, która to jest główną klasą rozszerzenia, zawierająca metodę zwracającą referencję do obiektu `Action` narzędzia.

5.1.6 Format zapisu diagramów

Do zapisywania diagramów do pliku został wykorzystany mechanizm serializacji, dostępny w środowisku Java. Sam proces serializacji, polega na automatycznym przekształceniu obiektu w strumień bajótw, który można później na przykład zapisać na dysku lub przesłać przez sieć. Wykorzystanie tego mechanizmu jest bardzo proste w użyciu. Aby obiekt mógł być poddany serializacji, jego klasa musi implementować interfejs `java.io.Serializable` (działa on na zasadzie flagi, wskazującej że dana klasa może zostać poddana serializacji, nie posiada on żadnych metod). Podczas procesu serializacji wszystkie wartości pól obiektu zostaną zapisane do strumienia, z wyjątkiem pól statycznych oraz pól oznaczonych znacznikiem `transient`. Ważną rzeczą, o której należy pamiętać podczas implementacji mechanizmów opartych na serializacji, jest nadanie poszczególnym klasom identyfikatorów oznaczających ich wersje (pole: `static final long serialVersionUID`). Jeśli tego się nie zrobi, identyfikator ten zostanie wygenerowany przez kompilator, co niesie ze sobą pewne niebezpieczeństwa. W przypadku gdy zapiszemy daną klasę do pliku, a następnie wprowadzimy w niej zmiany, na przykład dodając nowe pole, to niestety nie będziemy w stanie odczytać wcześniej zapisanej klasy i zostanie wywołany wyjątek `java.io.InvalidClassException` wskazujący na brak kompatybilności obu klas. Powodem takiego zachowania jest różny identyfikator każdej z klas, każdy z nich został wygenerowany podczas kompilacji i każdy z nich jest inny, a co za tym idzie, dla maszyny wirtualnej są to dwie różne klasy, pomimo tego że różnią się tylko jednym polem. Jeśli jednak ręcznie zapiszemy taki sam identyfikator w obu klasach, możliwe będzie odczytanie starszej wersji klasy, a wartość brakującego pola zostanie ustawiona na domyślną.

Należy pamiętać, że nie wszystkie rodzaje obiektów nadają się do serializacji, nawet jeśli implementują interfejs `java.io.Serializable`. Pierwszym rodzajem takich obiektów są obiekty reprezentujące takie rzeczy jak wątki, strumienie czy gniazda sieciowe. Są to obiekty ściśle związane z danym systemem i ich serializacja nie miałaby najmniejszego sensu. Kolejnym wyjątkiem są komponenty z biblioteki Swing. Ich serializacja jest jednak możliwa, ale w ograniczonym zakresie. Ponieważ są one związane z konkretną instancją maszyny wirtualnej, w której zostały utworzone, odtworzenie ich np. po ponownym uruchomieniu aplikacji będzie nie możliwe. Z tego właśnie powodu, przed dokonaniem serializacji diagramu, należy przekształcić jego strukturę danych tak, aby nie korzystała ona z żadnych komponentów Swing (czyli należy przenieść wszystkie elementy diagramu z klasy `ProjectPane` na przykład do klasy `Project`, tak jak przedstawiono na Rys.13).



Rys. 13. Struktura danych diagramu zapisywana do pliku.

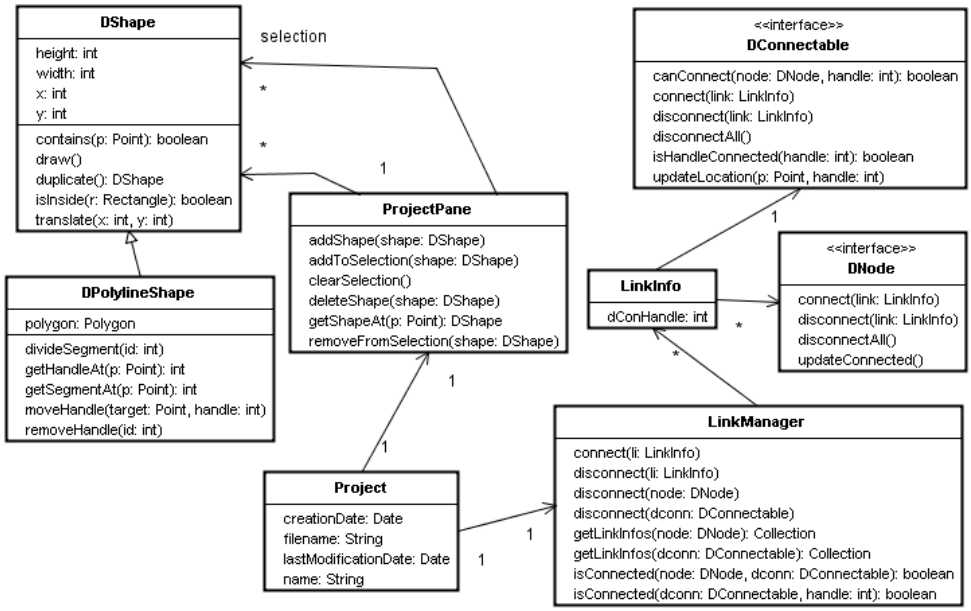
W przypadku aplikacji rozszerzalnej, w której możliwe jest utworzenie różnych rodzajów diagramów, pojawia się problem rozpoznawania typu diagramu, który został zapisany. Dodatkowo, ważna jest też wersja rozszerzenia wykorzystanego do stworzenia tego diagramu. Informacje te muszą się znaleźć w jednym pliku, razem ze wszystkimi obiektami poddanymi serializacji. Ponieważ nie jest możliwe zapisanie takich informacji podczas serializacji obiektów, zostaje utworzony oddzielny plik zawierający te dane. Do tego celu wykorzystany został mechanizm właściwości Javy (`Properties`), pozwalający na zapisywanie (oraz później odczyt) prostych danych tekstowych do pliku. Ponieważ takie zapisanie tych danych powoduje utworzenie dodatkowego pliku, należy w jakiś sposób scalić oba te pliki (plik z zapisanym diagramem oraz plik z opisem rodzaju diagramu) w jeden plik. Rozwiązaniem w tym przypadku jest kompresja obu plików w jedno archiwum (przy użyciu klas z pakietu `java.util.zip`), co dodatkowo owocuje znacznym zmniejszeniem rozmiarów takiego pliku.

5.2. Budowa diagramów

Do wyświetlania diagramów wykorzystana została klasa `ProjectPane`, będąca rozszerzeniem klasy `JPanel` z biblioteki Swing. Takie rozwiązanie umożliwia umieszczenie diagramu wewnątrz dowolnego komponentu interfejsu użytkownika oraz pozwala skorzystać z mechanizmów obsługi zdarzeń dostępnych dla wszystkich komponentów Swing, jednocześnie zapewniając swobodę w wykreślaniu diagramów dowolnego typu.

5.2.1 Struktura diagramów

Całą strukturę projektów wraz ze strukturą diagramów przedstawiono poniżej na Rys.14.



Rys.14. Struktura projektów.

Klasa `ProjectPane` jest główna klasą diagramu, będącą rozszerzeniem klasy `JPanel` z biblioteki Swing. Posiada ona własną metodę `paintComponent`, która przesłania odpowiadającą jej metodę z klasy `JPanel` i jest ona odpowiedzialna za cały algorytm wykreślania poszczególnych elementów diagramu. Zawiera ona wszelkie niezbędne informacje dotyczące diagramu, a także oferuje szereg metod umożliwiających jego modyfikacje. Są to następujące metody:

- `addShape`, `deleteShape` – metody dodające lub usuwające elementy diagramu.
- `addToSelection`, `clearSelection`, `removeFromSelection` – metody zarządzające zaznaczeniem elementów diagramu.
- `getShapeAt` – metoda pozwalająca znaleźć element diagramu znajdujący się w danym punkcie.

Dodatkowo klasa ta zawiera dwie kolekcje przechowujące referencje do wszystkich elementów diagramu oraz do aktualnie zaznaczonych elementów. Sama kolekcja przechowująca wszystkie elementy, posiada dodatkowo metody pozwalające na zmienianie kolejności jej elementów, dzięki czemu w prosty sposób wpływać na kolejność wykreślania poszczególnych elementów diagramu. Metody te oferują możliwość przemieszczania elementów o jedno miejsce w dowolną stronę, a także umożliwiają przeniesienie danego elementu na początek lub koniec kolekcji, co odpowiada przesunięciu elementu na diagramie odpowiednio na spód lub wierzch widoku.

Za reprezentacje poszczególnych elementów diagramu odpowiedzialne są dwie klasy, `DShape` oraz jej podklasa `DPolylineShape`. Klasy te posiadają informacje na temat wymiarów oraz położenia danego elementu na diagramie, a także metody ułatwiające operacje na tych elementach. Klasa `DShape` jest główna klasą reprezentującą elementy diagramu i jest ona nadklasą praktycznie wszystkich rodzajów elementów diagramu. Posiada ona następujące metody:

- `draw` – jest to metoda wywoływana poprzez diagram w celu wykreślenia danego elementu, opisuje ona dokładnie sposób w jaki ma zostać wykreślony dany element.

- `contains` – metoda pozwalająca stwierdzić czy dany punkt znajduje się wewnątrz danego elementu, funkcja bardzo użyteczna w przypadku skomplikowanych elementów w których różne ich obszary mogą inaczej reagować na kliknięcia.
- `isInside` – metoda pozwalająca stwierdzić czy dany element znajduje się w zaznaczonym obszarze, metoda niezbędna do poprawnej implementacji mechanizmów obszarowego zaznaczania wielu elementów.
- `translate` – metoda odpowiedzialna za zmianie położenia danego elementu na diagramie.
- `duplicate` – metoda oferująca funkcjonalność głębokiego kopiowania danego elementu.

Klasa `DPolylineShape` ma podobne zastosowanie do klasy `DShape`, z tą różnicą że jest przeznaczona do tworzenia elementów będących łamanymi liniami, przeważnie odpowiedzialnymi za wizualizacje różnego rodzaju połączeń pomiędzy innymi elementami diagramu. Klasa ta oferuje możliwość tworzenia krzywych łamanych z uchwytami w miejscach ich wierzchołków, co pozwala na dowolne modyfikowanie ich przebiegu przez użytkownika. Dodatkowo oferuje ona możliwość zdefiniowania różnego sposobu wykreślania dla każdego z końców krzywej. Najważniejszymi metodami tej klasy są:

- `moveHandle` – metoda pozwalająca na zmianę położenia danego wierzchołka krzywej.
- `divideSegment`, `removeHandle` – metody odpowiedzialne za dodawanie nowych wierzchołków oraz usuwanie istniejących.
- `getHandleAt`, `getSegmentAt` – metody znajdujące części elementu we wskazanym punkcie diagramu.

Ważnym mechanizmem diagramu jest zarządzanie połączeniami pomiędzy poszczególnymi elementami. Do jego realizacji wykorzystane są następujące klasy oraz interfejsy:

- `LinkManager` – klasa zarządzająca wszystkimi połączeniami.
- `LinkInfo` – klasa przechowująca informacje o danym połączeniu.
- `DNode` – interfejs identyfikujący dany element jako węzeł.
- `DConnectable` – interfejs oznaczający możliwość połączenia danego elementu z węzłem.

Interfejsy `DNode` oraz `DConnectable` specyfikują szereg metod niezbędnych do łączenia oraz rozłączania elementów, a także metody wpływające na zachowanie połączonych z danymi elementami innych elementów diagramu. Szczególnie użyteczną metodą interfejsu `DConnectable` jest metoda `canConnect`, stwierdzająca czy dane połączenie może być zrealizowane. Metoda ta przyjmuje jako argumenty węzeł do którego chcemy podłączyć dany element oraz identyfikator wierzchołka, który ma zostać podłączony do węzła. Dzięki temu możliwe jest w prosty sposób zrealizowanie reguł określających jakie elementy mogą zostać ze sobą łączone, a jakie nie.

Klasa `LinkManager`, będąca główna klasą mechanizmu połączeń, przede wszystkim zawiera kolekcje wszystkich istniejących połączeń w postaci tablicy obiektów `LinkInfo`, które to przechowują informacje na temat jednego połączenia. Obiekty te składają się z następujących elementów:

- identyfikatora wierzchołka
- referencji do węzła (`DNode`)
- referencji do elementu połączonego z węzłem (`DConnectable`)

Do zarządzania wszystkimi połączeniami, klasa `LinkManager`, oferuje zestaw metod pozwalających tworzenie nowych połączeń, zrywanie istniejących, sprawdzanie czy dany węzeł bądź

element jest już częścią jakiegoś połączenia oraz możliwość znalezienia wszystkich połączeń w których występuje dany węzeł lub element.

5.2.2 Sposób wykreślania diagramów

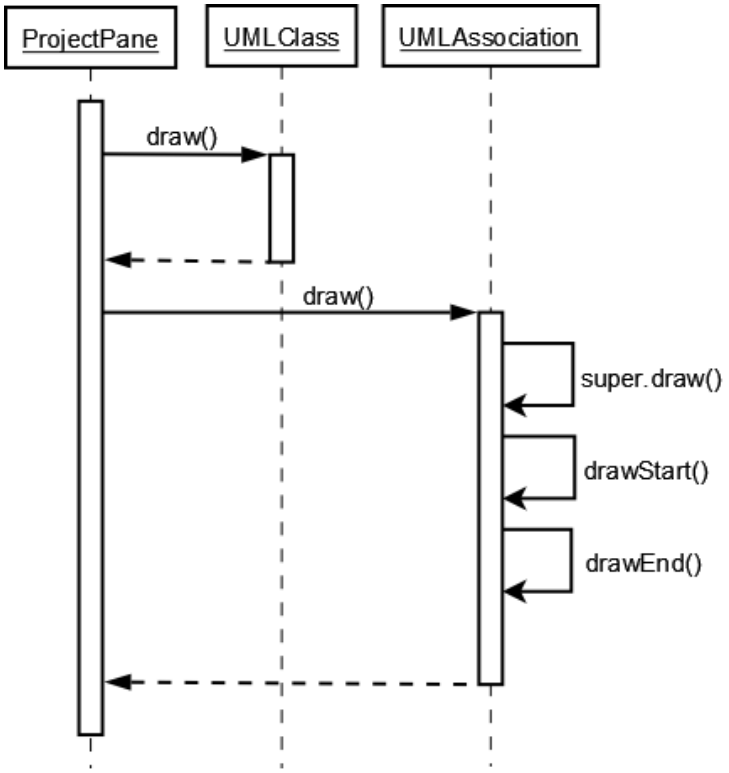
Elementy diagramów są podzielone na dwa rodzaje:

- Elementy proste, przeznaczone do reprezentowania prostych kształtów. Elementy te posiadają tylko jeden punkt identyfikujący ich położenie na diagramie. Są one reprezentowane przez klasę `DShape`, posiadającą metodę zawierającą szczegółowe instrukcje dotyczące wykreślania elementu.
- Elementy złożone, przeznaczone do reprezentowania kształtów będących liniami łamanymi. Elementy te posiadają co najmniej dwa punkty reprezentujące współrzędne punktów kontrolnych na diagramie, przez które to poprowadzone są linie je łączące. Elementy te są reprezentowane poprzez klasę `DPolylineShape`, będącą rozszerzeniem klasy `DShape`. Klasa ta, podobnie jak w przypadku elementów prostych, posiada metodę z instrukcjami wykreślania, oraz dodatkowo posiada parę metod, zawierających instrukcje wykreślania poszczególnych końców linii łamanej poprowadzonej przez punkty kontrolne.

Za cały proces wykreślania poszczególnych elementów odpowiada klasa `ProjectPane`. Klasa ta przesłania metodę `paintComponent`, z nadklasy, odpowiedzialną za wykreślanie danego komponentu na ekranie. Wywoływaniem tej metody zawsze zajmuje się kontener, w którym jest umieszczony dany komponent. Przez większość czasu, wywołania tej metody są automatyczne (np. w momencie kiedy dany obszar diagramu został przesłonięty oknem dialogowym i po zamknięciu okna wymaga ponownego wykreślenia), jednak w niektórych sytuacjach, w których kontener może nie wykryć potrzeby potrzeby ponownego wykreślenia, zostaje ono wymuszone poprzez wywołanie metody `repaint` komponentu.

Klasa `ProjectPane` zawiera uporządkowaną kolekcję elementów znajdujących się na diagramie. Kolejność elementów w kolekcji może być modyfikowana poprzez użytkownika (utworzenie nowego elementu lub zaznaczenie istniejącego automatycznie przesuwa dany element na koniec kolekcji). Sam proces wykreślania jest bardzo prosty. Metoda `paintComponent` iteruje kolejno po kolekcji elementów i wywołuje z każdego z nich metodę `draw`, zawierającą instrukcje wykreślania. Dzięki respektowaniu kolejności kolekcji podczas wykreślania, możliwe jest manipulowanie wzajemnym przesłanianiem się elementów diagramu.

Przebieg wykreślania poszczególnych elementów zależy od ich rodzaju. W przypadku prostych elementów (podklas `DShape`), metoda `draw` wykreśla cały element. Natomiast w przypadku elementów bardziej skomplikowanych (podklas `DPolylineShape`), metoda `draw` dodatkowo wywołuje metody odpowiedzialne za wykreślenie końców wykreślanej relacji. Na rys. 15 przedstawiono proces wykreślania elementów prostych oraz złożonych na przykładzie wykreślania elementów reprezentujących klasę oraz asocjację UML. W przypadku wykreślania klasy (`UMLClass`), wywołana zostaje tylko metoda `draw`, w której zawarta jest cała logika wykreślania tego elementu. Natomiast w przypadku wykreślania asocjacji, w pierwszej kolejności wywoływana jest metoda `draw` z nadklasy, która to wykreśli linie łączącą punkty kontrolne, a następnie wywoła metody `drawStart` oraz `drawEnd` wykreślające odpowiednio początek oraz koniec relacji. Klasy typu `DPolylineShape` posiadają dodatkowo też metodę `drawHandles`, która odpowiada za wykreślanie punktów kontrolnych na diagramie, jednak wywoływana jest tylko w przypadku, kiedy dany element jest zaznaczony przez użytkownika.



Rys. 15. Sposób wykreślania elementów diagramu.

5.2.3 Elementy UML

Cała funkcjonalność dotycząca tworzenia diagramów klas UML została zawarta w oddzielnym rozszerzeniu dla podstawowej aplikacji. Rozszerzenie to posiada implementację 8 elementów diagramów klas UML. Sposób wykreślania wszystkich tych elementów przez aplikację został przedstawiony na rys. 16

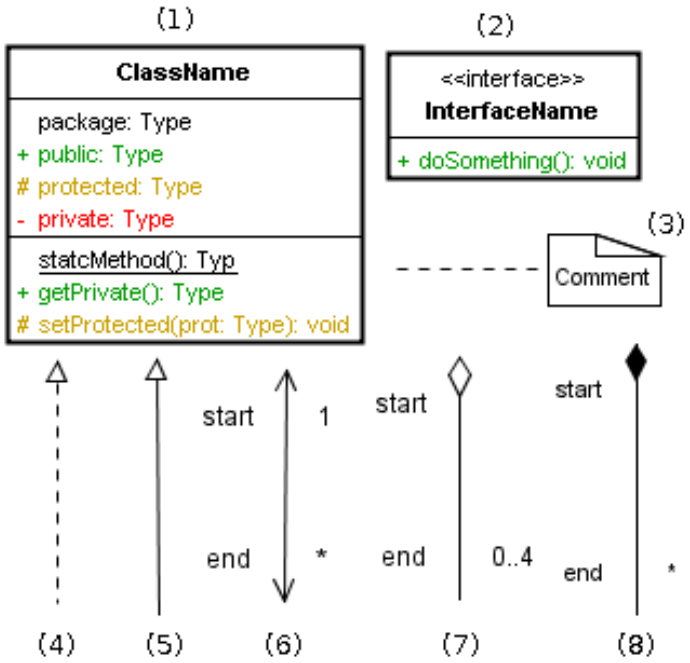
Wszystkie z tych elementów składają się z dwóch części, części reprezentującej dany element na diagramie, czyli klasy będącej rozszerzeniem klas `DShape` lub `DPolylineShape`, oraz z części reprezentującej dane UML danego elementu. Części te są ze sobą powiązane, każdy z elementów diagramu posiada referencję do obiektu posiadającego wszystkie jemu odpowiadające dane UML.

Klasy oraz interfejsy UML (oznaczone na rys.18 odpowiednio numerem 1 i 2) zostały zaimplementowane jako rozszerzenia klasy `DShape`, czyli jako elementy proste. Wszystkie pozostałe elementy są podklasami `DPolylineShape`, co pozwala na dowolne kształtowanie ich przebiegu poprzez punkty kontrolne, które można swobodnie dodawać i usuwać.

Elementy reprezentujące klasy oraz interfejsy UML pozwalają na zdefiniowanie pól (tylko w przypadku klas) oraz metod. Posiadają one, oprócz nazwy, także inne właściwości takie jak typ, widoczność, oznaczenie czy jest to pole lub metoda operująca na ekstensji klasy, czy też, w przypadku metod, pozwalają zdefiniować listę parametrów oraz ciało metody.

Poszczególne pola oraz metody klas i interfejsów zostały posortowane w celu zapewnienia większej wygody pracy z diagramami. Samo sortowanie odbywa się w dwóch krokach. W pierwszym z nich odbywa się sortowanie elementów ze względu na ich widoczność, a następnie, w kolejnym kroku, elementy zostają posortowane alfabetycznie wewnątrz poszczególnych bloków widoczności.

Dodatkowo widoczność poszczególnych elementów, oprócz standardowych oznaczeń UML (znaki „+”, „#” oraz „-”), została wyróżniona odpowiednimi kolorami tekstu.



Rys. 16. Elementy UML dostępne w rozszerzeniu.

Wszystkie zaimplementowane rodzaje relacji umożliwiają podłączenie poszczególnych końców do klas oraz interfejsów. Jednak zostały zaimplementowane pewne ograniczenia, które mają na celu zapewnienie poprawności budowanych diagramów. Ograniczenia te są sprawdzane w momencie kiedy użytkownik przeciąga jeden z końców nad klasę lub interfejs. Jeśli połączenie zostanie zweryfikowane jako poprawne, klasa zostanie podświetlona, umożliwiając utworzenie połączenia.

Zaimplementowano następujące ograniczenia:

- Koniec realizacji (oznaczona numerem 4 na rys. 16) oznaczony strzałką, umożliwia podłączenie tylko do interfejsu.
- Koniec realizacji bez strzałki umożliwia podłączenie tylko do klasy.
- Oba końce uogólnienia (nr 5) mogą być tylko podłączone do elementów tego samego typu, czyli jeśli np. jeden koniec jest podłączony do klasy, to i drugi koniec musi zostać podłączony do klasy. Utworzenie relacji uogólnienia pomiędzy klasą a interfejsem jest niemożliwe.
- Wszystkie rodzaje asocjacji (asocjacja – nr 6, agregacja – nr 7, kompozycja – nr 8) umożliwiają dowolne łączenie ze sobą poszczególnych klas oraz interfejsów.
- Komentarze (nr 3) nie posiadają żadnych ograniczeń. Możliwe jest podłączenie ich do dowolnego elementu diagramu.

Kolejnym aspektem wykreślania diagramu jest poprawne rozmieszczanie podłączonych do klas oraz interfejsów relacji tak, aby diagram nawet w przypadku dużej ich ilości pozostawał czytelny. Zostały zaimplementowane dwa rozwiązania tego problemu. Pierwszym z nich są reguły określające

do których krawędzi elementu może zostać podłączona dana relacja, reguły te prezentują się następująco:

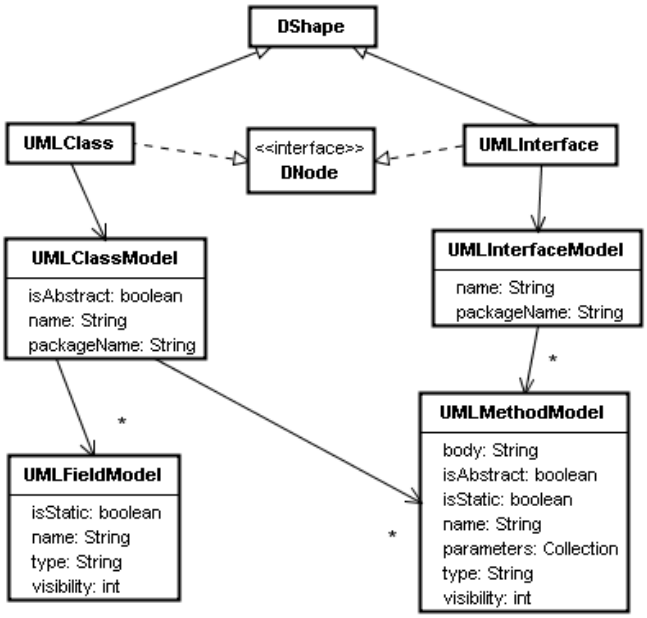
- Końce realizacji oraz uogólnienia oznaczone strzałkami mogą zostać podłączone tylko do dolnych krawędzi klas oraz interfejsów.
- Końce realizacji oraz uogólnienia nie posiadające strzałek mogą zostać podłączone tylko do górnych krawędzi klas oraz interfejsów
- Wszystkie pozostałe rodzaje połączeń nie posiadają ograniczeń w tym zakresie.

Wybór odpowiedniej krawędzi odbywa się automatycznie bez ingerencji użytkownika.

Drugim z rozwiązań jest odpowiednie rozmieszczanie poszczególnych relacji wzdłuż krawędzi. Rozwiązanie to opiera swoje działanie na następujących operacjach. Pierwsza z nich jest zmierzenie kąta pod jakim dana relacja jest podłączona do klasy lub interfejsu. Kąt ten jest mierzony na podstawie odcinka, łączącego środek klasy oraz kolejnego (po wierzchołku podłączonym do klasy) wierzchołka elementu relacji, oraz linii poziomej przeprowadzonej przez środek klasy. Na podstawie tego kąta jest następnie podejmowana decyzja o wyborze krawędzi do której ma zostać podłączona dana relacja, oczywiście w ramach wcześniej określonych reguł. Po czym, w celu zapobiegnięcia krzyżowania się poszczególnych relacji podłączonych do jednej krawędzi, zostaje określona ich kolejność w oparciu o ich kąt podłączenia. Kolejną operacja ma za zadanie równomierne rozmieszczenie relacji wzdłuż krawędzi. Zostaje zmierzona długość krawędzi i na podstawie ilości podłączonych do niej relacji zostaje wyliczona stała odległość jaką powinny zachować pomiędzy sobą poszczególne relacje.

5.2.4 Struktura klas UML

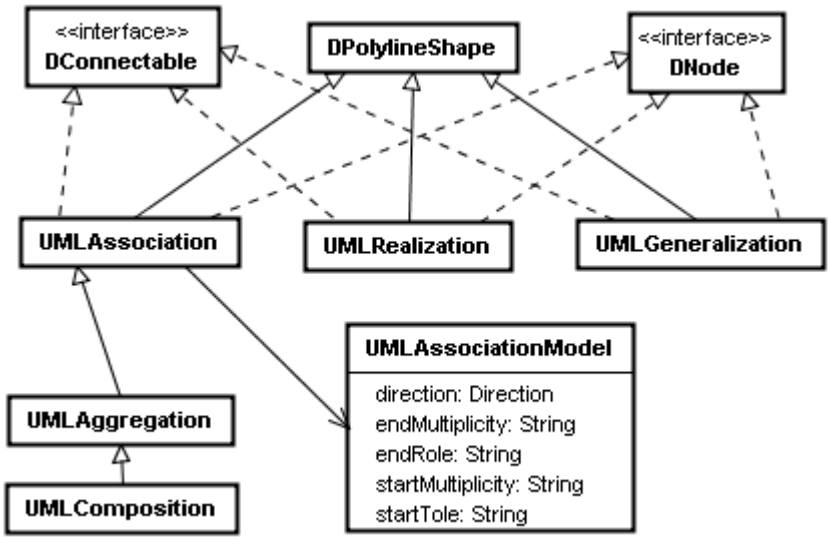
Sposób implementacji modelu danych przedstawiono na Rys.17. Klasy reprezentujące klasy oraz interfejsy UML rozszerzają klasę `DShape`, która reprezentuje proste elementy diagramu, natomiast wszystkie dane dotyczące diagramu UML, znajdują się w specjalnych, oddzielnych klasach `UMLClassModel` oraz `UMLInterfaceModel`. Dodatkowo, klasy `UMLClass` oraz `UMLInterface` implementują interfejs `DNode`, dzięki czemu możliwe jest podłączanie do nich różnego rodzaju elementów.



Rys.17. Struktura budowy klas oraz interfejsów.

5.2.5 Struktura połączeń UML

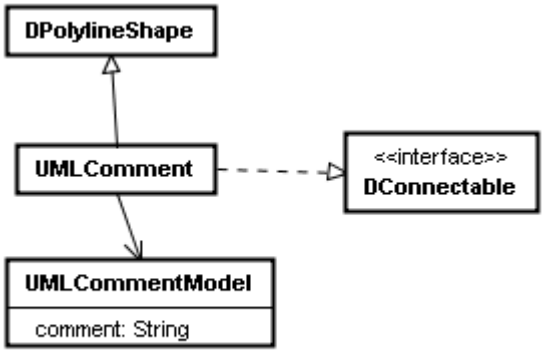
Sposób implementacji (przedstawiony na Rys.18) jest analogiczny do implementacji klas, z tą różnicą, że klasy połączeń rozszerzają klasę `DPolylineShape` i implementują interfejs `DConnectable`. Dodatkowo, implementują interfejs `DNode`, co czyni je także węzłami, dzięki czemu możliwe jest podłączenie do nich komentarzy.



Rys.18. Struktura budowy połączeń.

5.2.6 Struktura komentarzy UML

Sama implementacja klasy komentarza (przedstawiona na Rys.19), jest rozszerzeniem klasy `DPolylineShape`, w której jeden z końców został wykreślony jako pole tekstowe, w którym to jest wyświetlana treść komentarza (dodatkowo zostało tutaj wprowadzone zawijanie wierszy w celu zmniejszenia obszaru jaki zajmuje komentarz na diagramie). Ponieważ komentarz musi mieć możliwość podłączenia do dowolnego elementu diagramu, jego klasa implementuje interfejs `DConnectable`.



Rys.19. Struktura budowy komentarzy.

5.3. Interfejs aplikacji

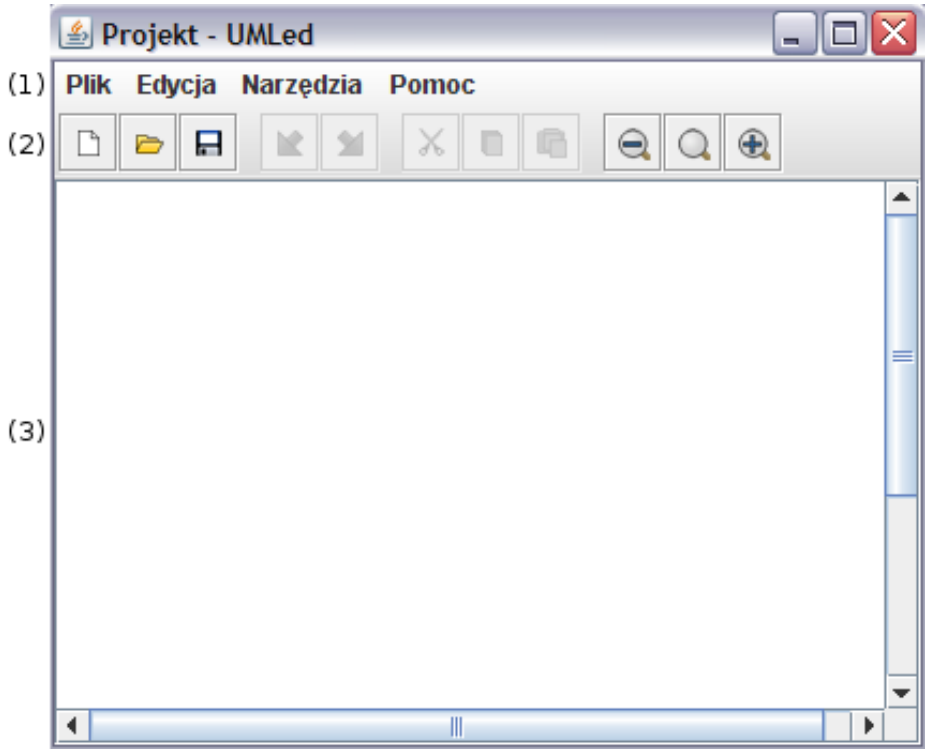
Interfejs aplikacji został stworzony przy użyciu standardowej biblioteki Javy, biblioteki Swing. Jego budowa oraz rozłożenie poszczególnych elementów zostało zaimplementowane w taki sposób aby odzwierciedlało wygląd oraz zachowanie większości dostępnych aktualnie na rynku aplikacji.

Okno aplikacje zostało podzielone na trzy części:

1. Menu aplikacji.
2. Pasek narzędzi.
3. Obszar roboczy.

Menu aplikacji składa się z następujących elementów:

- Menu „Plik”, udostępniające operacje na plikach.
- Menu „Edycja”, udostępniające operacje na diagramie.
- Menu „Narzędzia”, którego zawartość jest uzależniona od zainstalowanych rozszerzeń.
- Menu „Pomoc”, zawierające informacje o aplikacji.



Rys. 20. Interfejs aplikacji.

Większość pozycji menu może zostać wywołana poprzez skróty klawiszowe, w większości takie same jak w aktualnie dostępnych na rynku aplikacjach. Aby skróty te miały działanie globalne w aplikacji oraz dla zapewnienia możliwości zablokowania określonych pozycji menu, w zależności od

aktualnego stanu aplikacji lub diagramu, zostały one zaimplementowane przy użyciu dwóch tablic asocjacyjnych: `ActionMap` oraz `InputMap`. Tablice te są standardowo częścią okna aplikacji Swing. Pierwsza z nich przypisuje obiektom typu `Action`, czyli obiektom reprezentującym wywołanie operacji, identyfikator w formie ciągu znaków. Natomiast druga z tych tablic, przypisuje kombinacje klawiszy do identyfikatora akcji, co praktycznie powoduje przypisanie kombinacji klawiszy do konkretnych akcji. Tak skonfigurowane akcje są następnie użyte w głównym menu aplikacji, na pasku narzędzi czy też w menu kontekstowych, a ponieważ dla wszystkich tych miejsc została użyta ta sama instancja obiektu akcji, zablokowanie możliwości jej uruchomienia zostanie odzwierciedlone w całej aplikacji, niezależnie od umiejscowienia jej w interfejsie.

Pasek narzędzi odgrywa praktycznie taką samą role co menu aplikacji, z tą różnica, że zapewnia prosty dostęp do najczęściej używanych operacji, są to operacje takie jak:

- Tworzenie, zapisywanie oraz otwieranie plików diagramu.
- Cofanie oraz powtarzanie cofniętych zmian.
- Operacje na diagramie typu „Kopiuj”, „Wytnij” oraz „Wklej”.
- Kontrola stopnia powiększenia widoku diagramu.

Obszar roboczy jest w całości przeznaczony dla diagramu i jego funkcjonalność jest w całości uzależniona od rozszerzenia implementującego dany rodzaj diagramu.

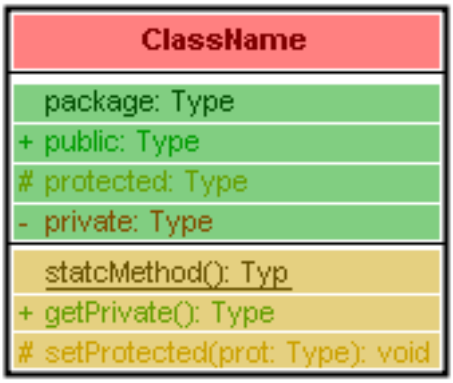
Cała interakcja użytkownika z diagramem odbywa się za pomocą myszy. Za obsługę jej działania odpowiedzialna jest klasa `UMLedMouseAdapter`. Klasa ta posiada szereg metod odpowiedzialnych za obsługę poszczególnych, elementarnych zdarzeń takich jak wciśnięcie przycisku czy też ruch kursora. Jednak dodatkowo posiada także metody odpowiedzialne za obsługę konkretnych zdarzeń takich jak podwójne kliknięcie czy przeciągnięcie elementu diagramu w inne miejsce. Dzięki takiej budowie, w bardzo prosty sposób można dostosować zachowanie aplikacji do własnych potrzeb. Z takiego rozwiązania korzysta rozszerzenie oferujące wsparcie dla diagramów klas. Posiada ona własną klasę odpowiedzialną za obsługę myszy, klasę `UMLCDMouseAdapter` (rozszerzenie klasy `UMLedMouseAdapter`). Klasa ta modyfikuje sposób działania aplikacji poprzez przesłonięcie między innymi metod odpowiedzialnych za obsługę podwójnego kliknięcia czy też za wyświetlenie menu kontekstowego.

Modyfikowanie położenia lub kształtu poszczególnych elementów diagramu wymaga ich wcześniejszego zaznaczenia (za pomocą kliknięcia na nie lub za pomocą zaznaczenia obszaru, w którym się dany element znajduje). Natomiast edycja odbywa się poprzez okna dialogowe, dostępne po dwukrotnym kliknięciu na interesujący nas element. Ponieważ elementy takie jak klasy lub interfejsy posiadają dużą ilość różnych atrybutów, nie jest możliwe umieszczenie ich wszystkich w jednym oknie zachowując jednocześnie prostotę oraz przejrzystość interfejsu. Zostało to rozwiązane poprzez wprowadzenie kilku rodzajów okien:

- okna właściwości klasy lub interfejsu
- okna właściwości pola
- okna właściwości metody

Każde z tych okien posiada tylko i wyłącznie właściwości danego elementu (klasy, konkretnej metody czy pola). Sam dostęp do tych okien odbywa się praktycznie tak samo jak dostęp do właściwości wszystkich innych elementów diagramu, czyli poprzez podwójne kliknięcie na interesujący nas element. Ponieważ w przypadku klas oraz interfejsów, elementem może być sama klasa, konkretna metoda czy pole, zostały w nich wprowadzone obszary (przedstawione na rys. 21) pozwalające zidentyfikować element który użytkownik wybrał. Każdy z tych obszarów odpowiada jednemu, konkretnemu elementowi i powoduje otwarcie innego typu okna (obszar oznaczony na

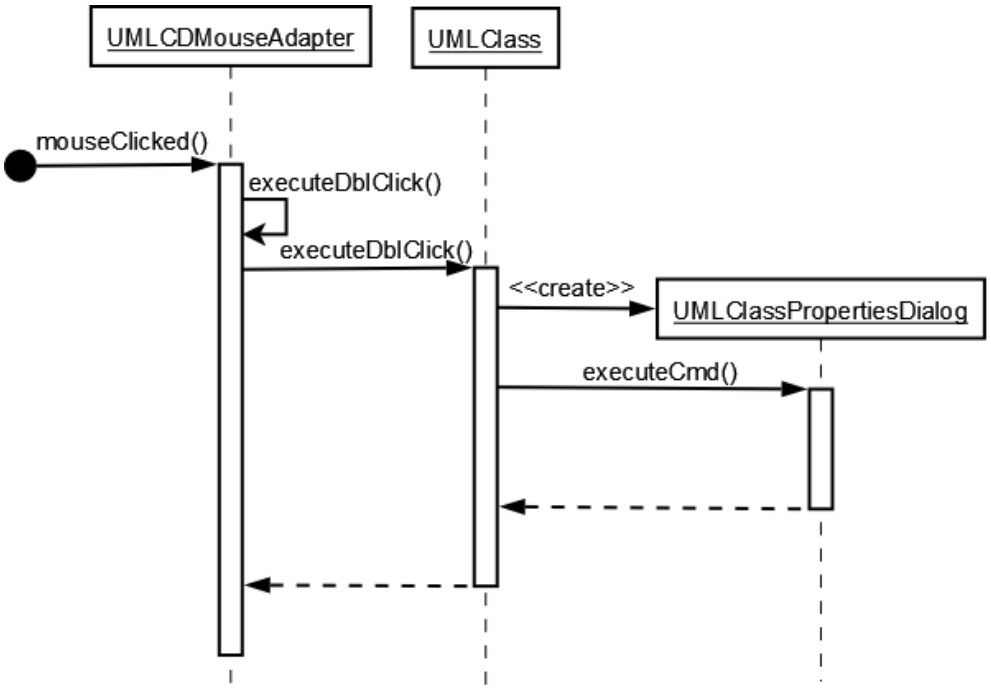
czerwono otwiera okno właściwości klasy, obszary zielone – właściwości metod, obszary żółte – właściwości pól).



Rys. 21. Obszary poszczególnych elementów klasy.

Przebieg procesu wyświetlania właściwości elementu po dwukrotnym kliknięciu na niego został przedstawiony na rys. 22. Proces ten rozpoczynany jest przez kliknięcie użytkownika na wybranym elemencie diagramu. W tym momencie, w klasie `UMLCDMouseAdapter` zostaje automatycznie wywołana metoda `mouseClicked`, odpowiedzialna za przetwarzanie kliknięć. Metoda ta analizuje czy to było pojedyncze czy podwójne kliknięcie, oraz który przycisk myszy został użyty. Po rozpoznaniu zdarzenia jako podwójnego kliknięcia, zostaje wywołana metoda je obsługująca, metoda `executeDblClick`. Odpowiada ona za znalezienie elementów diagramu, znajdujących się w miejscu kliknięcia. Gdy element zostanie znaleziony, w tym przypadku obiekt `UMLClass` reprezentujący klasę UML, zostaje wywołana kolejna metoda `executeDblClick`, należąca do wybranego elementu. Zadanie tej metody jest bardzo podobne, z tą różnicą że zamiast szukać elementu na diagramie, szuka ona elementu należącego do klasy i używa do tego celu zdefiniowanych obszarów. W momencie zidentyfikowania wybranego elementu, zostaje utworzone i wyświetlone jego okno właściwości, w tym przypadku okno właściwości klasy.

Takie zaimplementowanie dostępu do właściwości elementów pozwala na łatwy dostęp do edycji dowolnego elementu diagramu, bez potrzeby błądzenia po oknach dialogowych. Jednocześnie, sposób ten zapewnia dużą elastyczność w zachowaniu każdego z typów elementów diagramu, każdy z nich może inaczej reagować na kliknięcia, a także może definiować specjalne obszary, które mogą wprowadzać dodatkową, bardzo łatwo dostępną funkcjonalność.



Rys. 22. Sposób wyświetlania okna właściwości elementu.

5.4. Eksport diagramów

W aplikacji zaimplementowane zostały dwa rozszerzenia zajmujące się eksportem danych. Pierwszym z nich jest eksport do pliku graficznego. Rozszerzenie to korzystając z mechanizmów dostępnych w bibliotece Java2D, zapisuje wykreślony diagram do pliku, w jednym z czterech dostępnych formatów: BMP, JPG, PNG lub GIF.

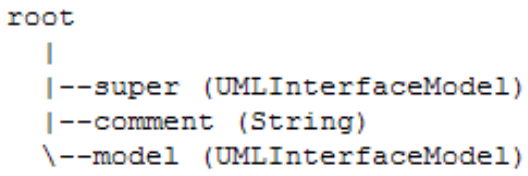
Drugie rozszerzenie oferuje możliwość generowanie kodu źródłowego (patrz [7]) na podstawie diagramów klas. Do generowania plików z kodem źródłowym wykorzystuje ono bibliotekę FreeMarker, która to przetwarza pliki tekstowe na podstawie zestawu szablonów. Rozszerzenie to dodatkowo udostępnia punkt rozszerzeń o nazwie „Templates", który pozwala na podłączenie dodatkowych zestawów szablonów, oferujących dowolny format generowanego kodu. Aktualnie zaimplementowany jest jedynie zestaw oferujący generowanie kodu źródłowego Javy.

5.4.1 Przygotowanie danych

Dane przeznaczone do przetwarzania przez mechanizmy FreeMarkera mogą być praktycznie w dowolnej postaci, radzi on sobie zarówno z nawigacją po skomplikowanych obiektach jak i z operacjami na tablicach. Jednak dla wygodnego dostępu do danych w szablonach, należało przekształcić dane diagramu, tak aby ich budowa była prosta i czytelna dla człowieka. Zostało to zrealizowane poprzez zastosowanie tablicy asocjacyjnej, w której pod określonymi nazwami zostały umieszczone poszczególne obiekty przechowujące dane poszczególnych elementów.

Strukturę danych przeznaczoną do przetwarzania interfejsów UML przedstawiono na rys. 23. Składają się na nią trzy elementy, najważniejszym z nich jest element oznaczony nazwą „model", jest to referencja do obiektu typu `UMLInterfaceModel`, reprezentującego dane UML interfejsu na diagramie. Ponieważ klasa `UMLInterfaceModel` posiada wszystkie informacje dotyczące

interfejsu, razem z dokładną specyfikacją metod w nim zdefiniowanych, przekazanie tylko tego obiektu do przetwarzania w zupełności wystarcza i nie są wymagane żadne dodatkowe klasy lub wpisy w tablicy dla przechowywania poszczególnych metod. W podobny sposób przekazywane są dane dotyczące ewentualnego interfejsu po którym dziedziczy interfejs przetwarzany, referencja do danych tego interfejsu została oznaczona nazwą „super". Dodatkowo, jako zwykły tekst dołączony jest ewentualny komentarz pod nazwą „comment". W przypadku kiedy interfejs posiada większą ilość komentarzy, zostają one scalone w jeden blok tekstu zawierający treść ich wszystkich, oddzielonych od siebie liniami odstępu.



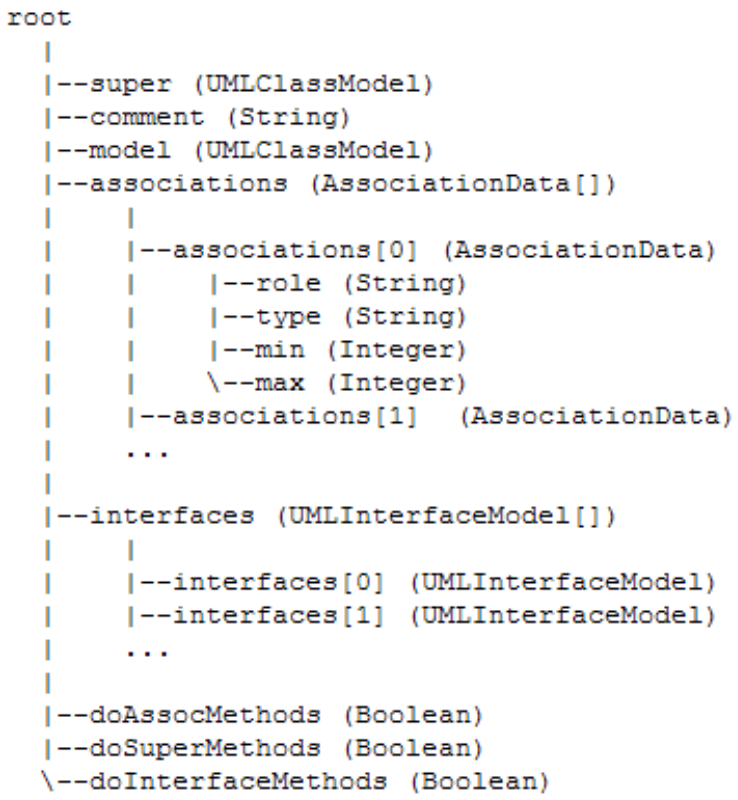
Rys. 23. Struktura danych dla szablonów interfejsów.

W przypadku przetwarzania klas UML struktura danych jest podobna, jednak posiada znacznie więcej danych. Dane dotyczące aktualnie przetwarzanej klasy, podobnie jak w przypadku interfejsów, są oznaczone nazwą „model" i są one przechowywane jako referencja do obiektu klasy `UMLClassModel`. Analogicznie do klasy `UMLInterfaceModel`, klasa ta przechowuje wszystkie dane klasy UML, włączając w to specyfikacje metod oraz pól.

Dane dotyczące interfejsów implementowanych przez klasę, są przekazywane w postaci tablicy obiektów typu `UMLInterfaceModel` i zostały umieszczone pod nazwą „interfaces". W podobny sposób przekazywane są dane dotyczące asocjacji. Jednak z tą różnicą, że na ich potrzeby została stworzona klasa opakowująca te dane, klasa `AssociationData`. Zawiera ona rolę asocjacji (w przypadku braku zdefiniowanej roli na diagramie, rola otrzymuje nazwę klasy do której prowadzi asocjacja), typ, a także liczność. Dane asocjacji są przekazywane jako tablica obiektów typu `AssociationData` i są oznaczone nazwa „associations".

Dodatkowo do przekazywanych danych są dołączane trzy flagi, pozwalające na wpływanie na kształt generowanego kodu. Użytkownik może ustawić te opcje w oknie dialogowym eksportu danych. Pozwalają one na włączenie lub wyłączenie następujących funkcji:

- `doAssocMethods` – odpowiada za generowanie metod realizujących ograniczenia wynikające z liczności asocjacji.
- `doSuperMethods` – odpowiada za generowanie implementacji abstrakcyjnych metod z nadklasy.
- `doInterfaceMethods` – odpowiada za generowanie implementacji metod specyfikowanych przez interfejsy implementowane przez klasę.



Rys. 24. Struktura danych dla szablonów klas.

5.4.2 Generowanie kodu źródłowego

Rozszerzenie odpowiedzialne za generowanie kodu źródłowego na podstawie diagramu klas, w praktyce zajmuje się tylko zebraniem i przygotowaniem danych z diagramu. Cała logika dotycząca samego przetwarzania danych i generowania na ich podstawie kodu jest zawarta w dodatkowych rozszerzeniach, podłączanych do punktu „Templates" podstawowego rozszerzenia. Każde z takich rozszerzeń musi zawierać dwa szablony dokumentów:

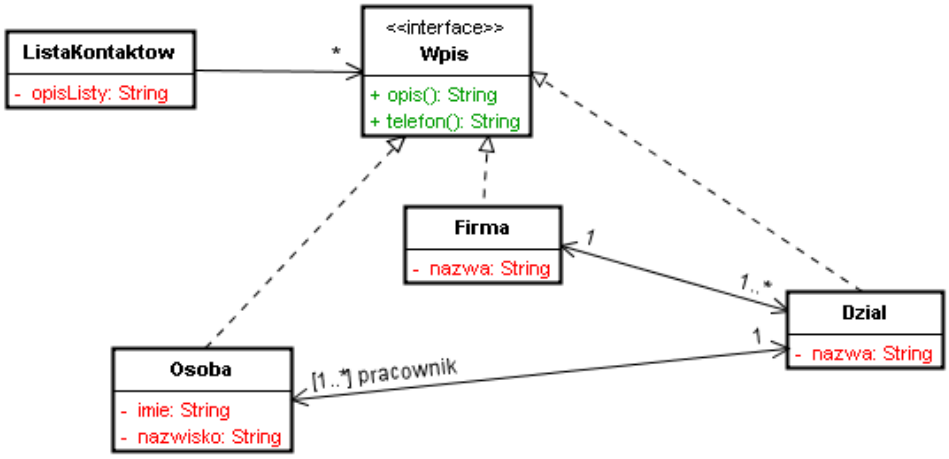
- `class.ftl` – szablon przeznaczony dla przetwarzania klas
- `interface.ftl` – szablon przeznaczony dla przetwarzania interfejsów

Dodatkowo musi posiadać własną implementacje interfejsu `TemplatePlugin`, udostępnionego przez główne rozszerzenie. Interfejs ten definiuje cztery metody, niezbędne do przetwarzania klas oraz interfejsów. Pierwsza para metod, czyli metody `getClassExtension` oraz `getInterfaceExtension`, zwracają rozszerzenie jakie powinny zawierać wygenerowane pliki klas oraz interfejsów. Natomiast pozostałe metody, `secondaryClassProcessing` oraz `secondaryInterfaceProcessing`, odpowiadają za dodatkowe przetwarzanie klas, przetwarzanie niezależne od szablonów. Użycie tych metod, pozwala np. na wygenerowanie dodatkowych plików, takich jak. deskryptory XML dla mechanizmów Java Persistence API czy też można je wykorzystać do logowania przebiegu całego procesu.

Format generowanego kodu oraz tworzone konstrukcje zostaną przedstawione na przykładzie prostej struktury danych (przedstawionej na rys. 25) reprezentujących listę kontaktów. Składa się ona z czterech klas, `ListaKontaktow` – reprezentująca listę jako całość oraz trzech klas

reprezentujących różne rodzaje wpisów, są to klasy `Osoba`, `Firma` oraz `Dzial`. Wszystkie te klasy dodatkowo implementują interfejs `Wpis`, zawierający metody pozwalające na zunifikowany dostęp do szczegółów kontaktów. Wszystkie te elementy są ze sobą połączone następującymi asocjacjami:

- `ListaKontaktow` zawiera dowolną ilość obiektów typu `Wpis`.
- `Firma` zawiera co najmniej jeden `Dzial`.
- `Dzial` jest przypisany dokładnie do jednej firmy i posiada co najmniej jednego pracownika (`Osoba`).
- `Osoba` jest przypisana tylko do jednego obiektu typu `Dzial`.



Rys. 25. Przykładowy diagram klas.

Format generowanych prostych elementów takie jak metody, pola czy interfejsy jest zgodny ze specyfikacją Javy 5.0 i nie wymaga szczegółowego opisu. Natomiast ciekawy jest sposób generowania konstrukcji takich jak asocjacje, uogólnienie czy też realizacja interfejsu.

W przypadku asocjacji, przede wszystkim zostaje wygenerowane pole przeznaczone do przechowywania referencji obiektów należących do niej. Posiada ono nazwę odpowiadającą zdefiniowanej roli asocjacji, lub jeśli rola nie została zdefiniowana, przyjmuje nazwę klasy lub interfejsu do którego dana asocjacja prowadzi. Dla asocjacji w których liczność ograniczona jest do jednego wystąpienia typ pola zostaje ustalony na klasę do której prowadzi asocjacja, jednak w przypadku kiedy liczność pozwala na większą ilość elementów, pole zostaje wygenerowane jako kolekcja obiektów o typie klasy. Jako przykład, może posłużyć fragment wygenerowanej klasy `Dzial`:

```
public class Dzial implements Wpis{
      private String nazwa;
      Firma firma;
      Collection<Osoba> pracownik;

      public Dzial(Firma firma_1, Osoba pracownik_1) {
            this.pracownik = new ArrayList<Osoba>();
```

```
            this.firma = firma_1;
            addPracownik(pracownik_1);
      }
      public void addPracownik(Osoba pracownik) {
            this.pracownik.add(pracownik);
      }
      public void removePracownik(Osoba pracownik) {
            if (this.pracownik.contains(pracownik)
                        && this.pracownik.size()-1 >= 1) {
                  this.pracownik.remove(pracownik);
            }
      }
}
```

Na przykładzie powyżej, zostały wygenerowane dwa pola asocjacji. Pierwsze z nich, dotyczące asocjacji łączącej `Dzial` z klasą `Firma`, której ograniczenie liczności pozwala tylko na jedno wystąpienie danej klasy, zostało wygenerowane jako zwykła, pojedyncza referencja. Dodatkowo został dodany wymagany parametr do konstruktora klasy, dzięki czemu spełnione zostaje wymaganie liczności dotyczące co najmniej jednego wystąpienia klasy `Firma` w asocjacji. Podobnie w przypadku drugiej asocjacji (z klasą `Osoba`), zostaje dodany do konstruktora parametr pozwalający na spełnienie minimalnego ograniczenia liczności (co najmniej jedno wystąpienie). Jednak ponieważ liczność pozwala na dowolną ilość wystąpień klasy `Osoba` w asocjacji, została ona wygenerowana jako kolekcja obiektów typu `Osoba`. Ten rodzaj asocjacji dodatkowo pociąga za sobą wygenerowanie pary metod odpowiedzialnych za dodawanie oraz usuwane obiektów z asocjacji. Są to metody `addXXX` oraz `removeXXX`, gdzie `XXX` odpowiada nazwie pola klasy, na którym metody operują. Dodatkową funkcjonalnością tych metod jest sprawdzanie liczby obiektów w kolekcji oraz zapewnienie realizacji ograniczeń liczności asocjacji.

Kolejną interesującą funkcją jest generowanie metod wymaganych przez implementowane interfejsy klasy. Jako przykład może tutaj posłużyć interfejs `Wpis`, który został wygenerowany następująco:

```
public interface Wpis {

      public String opis();
      public String telefon();
}
```

Oraz fragment klasy `Osoba`, która to implementuje ten interfejs:

```
public class Osoba implements Wpis{

      private String imie;
      private String nazwisko;
      Dzial dzial;

      public Osoba(Dzial dzial_1) {
            this.dzial = dzial_1;
      }
      // Methods from interface Wpis

      public String opis() {
            return null;
```

```
        }

        public String telefon() {
              return null;
        }
  }
```

Jak widać powyżej, generator tworzy pustą implementację wszystkich metod specyfikowanych przez implementowane interfejsy. Mechanizm ten działa zarówno w przypadku realizacji interfejsów, jak i w przypadku dziedziczenia po klasach abstrakcyjnych. W ich przypadku generator tworzy pustą implementację wszystkich abstrakcyjnych metod z klasy abstrakcyjnej.

Oczywiście działanie wszystkich wyżej wymienionych mechanizmów może być wyłączone przez użytkownika w momencie wyboru szablonu generowanego kodu. Natomiast sama implementacja reguł tych mechanizmów znajduje się w całości w plikach szablonów FreeMarkera, co pozwala na swobodną ich modyfikację bez potrzeby ingerencji w kod źródłowy rozszerzenia. Są one napisane przy użyciu specjalnego języka FTL (FreeMarker Template Language), który oferuje dosyć duże możliwości budowy różnych konstrukcji, z wykorzystaniem instrukcji warunkowych czy też pętli. Jednak dla generowania bardziej skomplikowanych struktur może być nie wystarczający i niezbędne może okazać się napisanie rozszerzenia, które w większym stopniu byłoby zaimplementowane w Javie.

6.Zalety, wady oraz plany rozwojowe

Prototyp, realizujący wymagania, określone w tej pracy, dla aplikacji wspomagającej tworzenie diagramów klas, jest daleki od uznania go za produkt skończony. Posiada on w większości wymaganą funkcjonalność, jednak aby mógł być stosowany w środowisku produkcyjnym, niezbędne jest rozszerzenie jego możliwości.

6.1. Zalety oraz wady przyjętych rozwiązań

Niewątpliwie do zalet prototypu można zaliczyć jego modułową budowę, która pozwala swobodne dostosowanie funkcjonalności do wymagań użytkownika. Dodatkowo, dzięki oparciu struktury aplikacji o bibliotekę Java Plugin Framework, efektywnie wykorzystuje ona zasoby systemowe między innymi poprzez wczytywanie do pamięci oraz uruchamianie kolejnych modułów dopiero w momencie, w którym użytkownik będzie wymagał ich funkcjonalności. Natomiast budowa samych rozszerzeń jest nie skomplikowana i w bardzo prosty sposób można przy ich pomocy dodać nową funkcjonalność, a udostępnione przez aplikację API oraz format deskryptorów rozszerzeń biblioteki JPF pozwalają na budowę dowolnie skomplikowanych modułów, jednocześnie zapewniając możliwość weryfikacji zależności pomiędzy poszczególnymi rozszerzeniami przed ich uruchomieniem.

Do udanych można także zaliczyć budowę diagramów. Ich struktura oraz udostępnione API pozwala na zaimplementowanie w prosty sposób dowolnego typu diagramu. Sama implementacja nowego typu diagramu praktycznie sprowadza się do przesłonięcia jednej metody, odpowiedzialnej za algorytm wykreślania, dla każdego z elementów diagramu. Natomiast cała funkcjonalność dotycząca interakcji z użytkownikiem jest w pełni funkcjonalna bez potrzeby wprowadzania jakichkolwiek zmian. Jednak w razie potrzeby, w bardzo prosty sposób możliwe jest także dostosowanie jej do własnych potrzeb. Sprowadza się to do przesłonięcia jednej z metod realizujących określoną funkcję, na przykład w przypadku chęci zmiany reakcji aplikacji na podwójne kliknięcie na dany element diagramu, wystarczy przesłonić metodę odpowiedzialną za obsługę tego zdarzenia w klasie danego elementu, lub, jeśli chcemy aby dotyczyło to dowolnych elementów diagramu, przesłonić metodę z klasy odpowiedzialnej za obsługę myszy.

Spornym elementem, pod względem zakwalifikowania do zalet lub wad, na pewno może być mechanizm zapisywania diagramu do pliku. Wykorzystuje on do tego celu mechanizm serializacji Javy, który polega na przekształceniu obiektów na strumień bajtów. Do zalet tego rozwiązania z pewnością można zaliczyć jego szybkość, prostotę, a także wielkość pliku wynikowego. Jako uzupełnienie tego mechanizmu w aplikacji dodatkowo zostało zaimplementowanie generowanie dodatkowego pliku, pełniącego rolę manifestu zawierającego szczegółowe dane dotyczące zapisanego diagramu. Dane te zawierają przede wszystkim informacje dotyczące rozszerzenia użytego do stworzenia danego diagramu oraz jego wersji. Jednak pomimo takich zabezpieczeń, nie można w pełni zagwarantować, że zapisany na jednym komputerze diagram, zostanie bez problemu odczytany na innym. Głównym powodem problemów może tu być na przykład starsza wersja maszyny wirtualnej Javy, niż ta której użyto do zapisania. Kolejnym problemem może być niezgodność wersji samych rozszerzeń diagramów. Są to problemy które nie powinny wystąpić w codziennym użytkowaniu, jednak ryzyko ich wystąpienia nadal istnieje.

Ocena interfejsu aplikacji i zakwalifikowanie go do zalet lub wad jest bardzo trudna i zależy od preferencji konkretnego użytkownika. Jednak zastosowany w aplikacji interfejs można chyba zaliczyć

do zalet z kilku powodów. Przede wszystkim oferuje on maksymalnie duży obszar roboczy, co zapewnia wygodę pracy. Dodatkowo interfejs nie posiada praktycznie żadnych pasków narzędziowych poza głównym, zawierającym podstawowe przyciski odpowiedzialne za operacje na plikach, cofanie zmian, operacje kopiuj, wytnij, wklej czy też kontrole stopnia powiększenia diagramu. Wszystkie inne funkcje, niezbędne do edycji diagramu, zostały umieszczone w menu kontekstowych, których zawartość zależy od aktualnie wybranego elementu diagramu. Kolejną zaletą interfejsu niewątpliwie jest fakt, że dostęp do edycji dowolnego z parametrów elementów diagramu, na przykład do parametru jakiejś metody z jakiejś klasy, jest możliwy przy użyciu maksymalnie trzech kliknięć. Co w znacznym stopniu upraszcza oraz przyspiesza prace nad diagramem, jednocześnie eliminując potrzebę nawigacji po skomplikowanych hierarchiach okien dialogowych. Wszystkie parametry elementów diagramu są po prostu dostępne bezpośrednio z obszaru roboczego.

Wszystkie te elementy tworzą razem solidne fundamenty aplikacji. Umożliwiają tworzenie dowolnych typów diagramów wraz z funkcjonalnością łączenia ze sobą poszczególnych elementów. Zawierają w pełni funkcjonalną obsługę interakcji z użytkownikiem, bardzo przydatny mechanizm cofania zmian wprowadzonych na diagramie czy też przejrzysty interfejs. Dodając do tego prosty w użyciu mechanizm rozszerzeń o dużych możliwościach, otrzymujemy w pełni działającą platformę na której możemy oprzeć, dostosowaną do naszych potrzeb, aplikację do budowy diagramów dowolnego typu.

W przypadku rozszerzenia oferującego możliwość generowania kodu źródłowego na podstawie diagramów, do generowania i formatowania kodu zostały wykorzystane tekstowe szablony. Zakwalifikowanie takiego rozwiązania do zalet lub wad może być sporne. Z jednej strony taki plik tekstowy bardzo łatwo się edytuje i bardzo szybko można zmodyfikować działanie istniejącego szablonu lub dodać nowy. Jednak pojawia się problem w przypadku generowania bardziej skomplikowanych konstrukcji. Mogą one w takim przypadku wymagać naprawdę skomplikowanego szablonu, który może stać się bardzo mało przejrzysty i trudny do edycji. W niektórych przypadkach może się okazać niezbędne zaimplementowanie generowania kodu w postaci normalnego, skompilowanego kodu Javy. Niestety takiej możliwości nie przewiduje dostępne rozszerzenie, które operuje wyłącznie na szablonach, a więc w przypadku skomplikowanych struktur generowanego kodu, niezbędne będzie napisanie całego rozszerzenia od podstaw.

6.2. Plany rozwojowe

Budowa aplikacji, bazująca na rozszerzeniach, pozwala w bardzo łatwy sposób dodawać nową funkcjonalność. Najprostszą, a zarazem najbardziej podatną na dostosowywanie do własnych potrzeb, dodatkową funkcjonalnością z pewnością jest opcja eksportu diagramów do kodu źródłowego. Bardzo często każdy z użytkowników, czy nawet firm, ma własne upodobania oraz wymagania dotyczące generowanego języka oraz formatowania kodu. Z tego powodu, niezbędne może okazać się dodanie języków takich jak C#, PHP, czy nawet wygenerowanie skryptu tworzącego strukturę bazy danych. Dodatkowo prawdopodobnie powinny powstać różne warianty generowania kodu dla poszczególnych języków, na przykład dla Javy generowanie kodu jako zwykłe klasy typu JavaBean, lub jako klasy przeznaczone dla Java Persistence API (czyli mechanizmu mapowania klas modelu obiektowego do modelu relacyjnego bazy danych), czy też jako Enterprise JavaBeans, czyli komponenty przeznaczone do uruchamiania w środowisku serwera aplikacji. Możliwości jest tutaj wiele i praktycznie można stworzyć oddzielny zestaw szablonów dla każdej używanej obecnie technologii.

Oprócz funkcji generowania kodu źródłowego, niezbędne może okazać się także przekształcenie diagramu klas do innego formatu. Realizowane jest to za pomocą rozszerzeń zajmujących się eksportem. Obecnie prototyp zawiera dwa takie rozszerzenia, pierwsze z nich zapisuje diagram jako plik graficzny, natomiast drugie generuje kod źródłowy na podstawie

szablonów. Kolejne mogłoby oferować możliwość eksportu diagramu do formatu XMI, czyli XML Metadata Interchange. Jest to międzynarodowy standard wymiany metadanych, oznaczony jako ISO/IEC 19503:2005. Został on opracowany przez konsorcjum OMG (Object Management Group) jako format przeznaczony do przenoszenia diagramów UML pomiędzy różnymi narzędziami, przy wykorzystaniu technologii XML. Niestety poszczególne narzędzia UML w różny sposób implementują ten standard, przez co ciężko zagwarantować pełną kompatybilność, jednak mimo to warto by było dodać obsługę tego standardu do aplikacji.

Następnym do zagospodarowania obszarem aplikacji jest z pewnością menu „Narzędzia", czyli punkt rozszerzeń „Tools". Rozszerzenia dla tego punktu mogą zawierać dodatkową funkcjonalność przeznaczoną dla poszczególnych typów diagramów. Ciekawym, a zarazem użytecznym rozszerzeniem może być na przykład narzędzie przeznaczone do zarządzania typami danych użytych w diagramie klas. Mogłoby ono pozwalać na definiowanie nowych typów, inspekcje zdefiniowanych wraz z informacjami o wykorzystujących je klasach, czy też oferować zmianę danego typu na inny w całym diagramie. Kolejne rozszerzenia mogą oferować podobne funkcje, ale zajmujące się innymi danymi, na przykład pozwalające zarządzać pakietami lub importami poszczególnych klas. Inne mogą zajmować się generowaniem i dodawaniem klas do diagramu, lub nawet całych struktur. Na przykład rozszerzenie może tworzyć strukturę klas oraz interfejsów realizującą wybrany przez użytkownika wzorzec projektowy. Szczególnym przypadkiem rozszerzeń generujących struktury klas na diagramie będą rozszerzenia operujące na zewnętrznych danych, czyli po prostu zajmujące się importem danych.

Sam import danych, jako dane wejściowe, powinien wspierać przynajmniej, wspomniany wcześniej, format XMI. Jednak przydatne byłoby także wspieranie własnych formatów innych narzędzi. Bardzo ciekawym, a zarazem użytecznym, rozszerzeniem importującym byłoby na pewno rozszerzenie realizujące tzw. „Reverse Engineering" diagramu, czyli generowanie struktury diagramu klas UML na podstawie istniejącego już kodu źródłowego. Podobnie jak w przypadku generowania kodu źródłowego, musi ono oferować użytkownikowi możliwość wyboru języka kodu źródłowego oraz pozwolić na dodawanie nowych języków w prosty sposób. Dodatkowo, mogłoby oferować możliwość wybiórczego generowania klas na diagramie, na przykład tylko z zakresu jednego pakietu czy też grupy pakietów. Rozszerzenie to, razem z rozszerzeniem generującym kod źródłowy utworzyłoby spójne narzędzie dla projektanta, który bez problemu mógłby przy jego pomocy zaprojektować od podstaw nowy system, a także w łatwy sposób rozbudować już istniejący.

Największe możliwości rozszerzenia funkcjonalności oferuje punkt rozszerzeń „Project", umożliwiający podłączenie rozszerzeń zawierających różne implementacje diagramów i praktycznie pozwala na zmianę całego obszaru roboczego. W przyszłości na pewno powinno zostać dodane wsparcie dla pozostałych diagramów UML, a w szczególności dla diagramów sekwencji oraz przypadków użycia. Zaimplementowanie tych diagramów otwiera zupełnie nowe możliwości, można na przykład w pewnym stopniu zintegrować diagramy klas i sekwencji, tak aby można było praktycznie tylko przy wykorzystaniu wizualnej edycji diagramów stworzyć działającą aplikację. Jednak budowa prototypu pozwala także na zaimplementowanie innych diagramów, praktycznie oferując pełną dowolność kształtów i zachowania. Można na przykład wprowadzić wsparcie dla diagramów przepływu, czy też diagramów obrazujących budowę sieci.

Podsumowując, modułowa budowa prototypu pozwala na praktycznie dowolną modyfikacje aplikacji przy stosunkowo nie dużych nakładzie środków. Możliwe jest dostosowanie aplikacji do potrzeb każdego, niezależnie od tego czy jest analitykiem, potrzebującym narzędzia pozwalającego mu w prosty sposób zaprojektować dany system, czy też jest programistą, wymagającym od aplikacji możliwości szybkiego stworzenia prototypu.

6.3. Zastosowanie

Założenia dotyczące funkcjonalności aplikacji, przyjęte na początku tej pracy, zakładają uzyskanie aplikacji umożliwiającej wspomóc proces projektowania oprogramowania, w zakresie tworzenia diagramów klas. Zostały one spełnione, dzięki czemu aplikacja może z sukcesem zostać zastosowana w tej dziedzinie. Szczególnie użyteczna na pewno będzie dla analityka. Prosty w użyciu oraz przejrzysty interfejs użytkownika, umożliwia wykorzystanie aplikacji zarówno podczas rozmowy z klientem, dotyczącej uzgodnienia wymagań, jak również w wewnętrznym procesie analizy przeprowadzonej przez firmę lub dany zespół pracujący nad projektem. Dodatkowo, możliwości aplikacji w zakresie generowania kodu źródłowego, w znacznym stopniu ułatwiają pracę programistą, którzy mogą zaoszczędzić sporo czasu traconego na implementowanie wszystkich podstawowych pól oraz metod poszczególnych klas oraz interfejsów zdefiniowanych na diagramie. Szczególnie w przypadku diagramów, które posiadają zdefiniowaną bardzo dużą ilość małych prostych klas lub interfejsów, których jedyną rolą może być identyfikowanie typu obiektu.

Oczywiście nie jest to cały zakres możliwych zastosowań. Wykorzystanie rozszerzeń w aplikacji, poszerza go dość znacząco. Może to być na przykład dodanie możliwości tworzenia diagramów przypadków użycia czy też możliwość generowania struktury bazy danych. Ogólnie, zastosowanie aplikacji, w pełni zależy od używającej jej osoby lub firmy oraz od użytych rozszerzeń, które to, mogą być stworzone właśnie według konkretnych wymagań użytkownika. Taka budowa aplikacji uniemożliwia dokładne określenie jej końcowych możliwości, właśnie ze względu na wykorzystującego ją użytkownika i zastosowane lub napisane przez niego rozszerzenia.

7.Podsumowanie

Narzędzia wspomagające budowę diagramów klas UML są niezbędne w procesie projektowania oprogramowania. Jednak aby takie narzędzie było funkcjonalne i nadające się do użytku musi ono posiadać pewną minimalną funkcjonalność, która została opisana w tej pracy. Składają się na nią między innymi podstawowe funkcje takie jak mechanizm kopiowania oraz wklejania poszczególnych elementów diagramu, mechanizm cofania wprowadzonych zmian, czy też wygodny interfejs. Dodatkowo interfejs aplikacji powinien cechować się prostotą, nie może być przytłaczający. Funkcje, przyciski oraz menu powinny być logicznie uporządkowane, tak aby użytkownik nie mający wcześniej styczności z aplikacją mógł bardzo szybko nauczyć się jej obsługi.

Wszystkie te funkcje zostały zaimplementowane w opracowanym prototypie. Realizuje on je w możliwie optymalny sposób, wykorzystując sprawdzone wzorce projektowe takie jak wzorzec polecenia (ang: Command Pattern) do realizacji mechanizmu cofania zmian, czy też wzorzec Model-View-Controller do budowy struktury diagramu klas w aplikacji. Dodatkowo, prototyp posiada modułową budowę, co z kolei umożliwia późniejsze rozszerzenie możliwości aplikacji o nową funkcjonalność bez potrzeby ponownej jej kompilacji. Sam mechanizm rozszerzeń jest na tyle elastyczny, że pozwala praktycznie całkowicie zmienić funkcjonalność aplikacji w bardzo prosty sposób. Umożliwia on dodanie rozszerzeń realizujących funkcjonalność nowych rodzajów diagramów, dodatkowych narzędzi wspomagających pracę, czy też rozszerzeń oferujących nowe formaty zapisu diagramów, czyli po prostu funkcjonalność eksportu danych. Jednym z takich formatów zaimplementowanych w prototypie jest generowanie kodu źródłowego Javy na podstawie diagramów. Sam proces eksportu, realizowany przez to rozszerzenie, bierze pod uwagę takie rzeczy jak liczności asocjacji, na podstawie których generuje stosowne metody odpowiedzialne za ich realizacje w aplikacji, czy też abstrakcyjne nadklasy oraz implementowane interfejsy, na podstawie których generowane są implementacje zdefiniowanych w nich metod.

Wszystkie te elementy sprawiają, że narzędzie to może być bez problemu wykorzystane w różnego rodzaju projektach. Dodatkowo, może być dla każdego z nich indywidualnie dostosowane za pomocą rozszerzeń, tak aby w możliwie najszerszym zakresie realizowało wymaganą funkcjonalność.

Bibliografia

[1]. *History of UML*. http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm

[2]. D. Pilone, N. Pitman. *UML 2.0 in a Nutshell*. chapter 1-2, O'Reilly, June 2005.

[3]. T. Sundsted. *MVC meets Swing*. http://www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html

[4]. T. Meshorer. *Add an undo/redo function to your Java apps with Swing.* http://www.javaworld.com/javaworld/jw-06-1998/jw-06-undoredo.html

[5]. B. Paranj. *Java Tip 68: Learn how to implement the Command pattern in Java.* http://www.javaworld.com/javaworld/javatips/jw-javatip68.html

[6]. C. Haase, R. Guy. *Filthy Rich Clients: Developing Animated and Graphical Effects for Desktop Java Applications*. chapter 1-5, Prentice Hall, August 2007.

[7]. E. Lervik, V. B. Havdal. *Java the UML Way*. June 2002.