Biuletyn WAT
Vol. LXV, Nr 1, 2016

Konstrukcja diagramu klas UML z zastosowaniem Model-Driven Development

TOMASZ GÓRSKI, MATEUSZ SOWA

Wojskowa Akademia Techniczna, Wydział Cybernetyki, Instytut Systemów Informatycznych,
00-908 Warszawa, ul. gen. S. Kaliskiego 2, gorski@wat.edu.pl, mateusz.sowa92@gmail.com

Streszczenie. Transformacje modeli odgrywają istotną rolę w projektach projektowania systemów informatycznych wykorzystujących Model-Driven Development (MDD). Pozwalają automatyzować powtarzalne i dobrze określone czynności, przez co skracają czas projektowania oraz umożliwiają zmniejszenie liczby błędów. W podejściu obiektowym kluczowymi elementami są przypadki użycia. Są one opisywane, modelowane, a później projektowane, aż uzyskiwany jest działający kod aplikacji. W artykule przedstawiono transformację *Communication-2-Class* umożliwiającą automatyzację konstrukcji diagramu klas języka Unified Modeling Language (UML) tworzonego w realizacji przypadku użycia w ramach modelu analitycznego/projektowego. Diagram klas UML tworzony jest na podstawie diagramu komunikacji UML dla realizacji przypadku użycia. Dzięki temu diagram klas przedstawia wszystkie klasy zaangażowane w realizację przypadku użycia i związki między nimi. Wtyczka realizująca transformację *Communication-2-Class* została zrealizowana w środowisku IBM Rational Software Architect. W artykule przedstawiono także wyniki testów opracowanej wtyczki realizującej transformację *Communication-2-Class* pokazujące możliwości skrócenia czasu projektowania realizacji przypadku użycia.
Słowa kluczowe: Model-Driven Development, transformacje, Unified Modeling Language, Model analityczny/projektowy, diagram klas UML, diagram komunikacji UML
DOI: 10.5604/12345865.1197989

1. Wprowadzenie

Model-Driven Development (MDD) to podejście do tworzenia oprogramowania, które podkreśla potrzebę tworzenia modeli jako podstawowego artefaktu. Modele te poddawane są procesowi doprecyzowania, przez zastosowanie transformacji i uszczegóławiania, aż do uzyskania działającego oprogramowania. MDD można

określić jako „pogląd, że możemy zbudować model systemu, który możemy następnie przekształcić się w prawdziwą rzecz" [1]. Jest to uogólnienie inicjatywy OMG Model-Driven Architecture (MDA) [2, 3], która proponuje kompleksowe podejście do wytwarzania oprogramowania.

Model-Driven Achitecture opiera się na trzech zasadach:

— bezpośrednia reprezentacja problemu — proces tworzenia oprogramowania powinien koncentrować się na problemie, który jest do rozwiązania, a nie na konkretnej technologii,
— automatyzacja — zautomatyzowanie tych aspektów tworzenia oprogramowania, które nie mają wiele wspólnego z ludzką kreatywnością,
— otwarte standardy — ponowne użycie, budowa właściwej infrastruktury, wykorzystanie wolnego oprogramowania.

MDD przechodzi przez różne poziomy abstrakcji, od poziomu biznesowego, Computation Independent Model (CIM), do kodu wykonywalnego. Pomiędzy tymi poziomami tworzone są Platform Independent Model (PIM) oraz Platform Specific Model (PSM) [2]. CIM opisuje kontekst oraz wymagania na system. PIM używa zdolności operacyjnych aplikacji w sposób neutralny w stosunku do platformy technologicznej. PSM dodaje dane odnoszące się do zastosowania specyficznej platformy technologicznej. Przez definiowanie odpowiednich transformacji możliwe jest uzyskanie modelu docelowego z modelu źródłowego. Podczas projektowania zstępującego (ang. *top-down*) proces rozpoczyna się od CIM, jest przekształcany do PIM, a następnie w PSM. Wreszcie kod aplikacji może być także generowany z PSM przy zastosowaniu transformacji. Klasyfikacja i zastosowanie transformacji są opisywane w literaturze [4].

Dzięki zastosowaniu transformacji możliwe jest znaczne skrócenie czasu tworzenia opisu architektonicznego oraz oprogramowania. Analizy dostępne dla systemów informatycznych projektowanych w sektorze medycznym pokazują, że w projektach gdzie stosowana jest inżynieria sterowana modelami, notuje się trzykrotne skrócenie czasu wytwarzania oprogramowania [5, 6]. Przy projektowaniu rozwiązań informatycznych dla zastosowań w sieci Internet oszczędności z zastosowania inżynierii sterowanej modelami są ponad dwukrotne (skrócenie czasu wytwarzania oprogramowania o 59%) [7]. Szczególnie ma to znaczenie przy projektowaniu złożonych rozwiązań integracyjnych wielu systemów informatycznych. Z analizy literatury [8] wynika, że dla złożonych systemów istotne staje się automatyzowanie implementacji przepływów integracyjnych. Pojawia się możliwość 40-krotnego skrócenia czasu implementacji przepływów integracyjnych (przy 100 przepływach do implementacji). Ponadto MDA umożliwia zaoszczędzenie czasu spędzonego nad poszczególnymi etapami cyklu życia projektu informatycznego [10].

Badania pokazują, że dzięki zastosowaniu transformacji notuje się 10-krotny spadek liczby zgłaszanych błędów w oprogramowaniu [5, 6]. W tym kontekście automatyczne generowanie kodu aplikacji oferuje wiele korzyści, takich jak szybki

rozwój wysokiej jakości kodu, zmniejszenie liczby przypadkowych błędów programistycznych, zwiększona spójność projektu i kodu [9].

Ponadto, dzięki zastosowaniu transformacji, zapewnione jest utrzymanie kompletności opisu architektonicznego oraz spójności elementów między modelami. W kontekście podejścia obiektowego automatyzowane są powtarzalne czynności utworzenia struktur, diagramów UML oraz zawartości diagramów UML. Zastosowanie standardowych struktur i konstrukcji ma wpływ na czytelność projektu i kodu. Wiąże się to z szybkością znajdowania odpowiednich treści w projekcie.

Nie wolno jednak zapominać o nakładzie ponoszonym na projektowanie samych transformacji. Stosowanie MDA jest opłacalne przy projektowaniu złożonych systemów informatycznych, szczególnie ze zmienną platformą technologiczną. Zastosowanie MDA pozwala na szybkie dostosowanie się do zmian w technologiach dzięki zastosowaniu modeli PIM oraz PSM.

Celem tego artykułu jest przedstawienie transformacji typu model w model, *Communication-2-Class*, automatyzującej tworzenie fragmentu modelu analitycznego przy projektowaniu przypadków użycia. Transformacji podlega diagram komunikacji języka UML do postaci diagramu klas języka UML. Transformowane modele są na tym samym poziomie abstrakcji. Z punktu widzenia notacji, prezentowanej transformacji podlegają modele wyrażone w tej samej notacji, języku UML [11]. W związku z powyższym prezentowana transformacja jest endogenna, horyzontalna oraz jednokierunkowa [4].

Pozostała część artykułu została ułożona w następujący sposób. W sekcji 2 przedstawiono przegląd prac dotyczących podobnych zagadnień. W sekcji 3 opisano zasadę działania transformacji *Communication-2-Class*. Sekcja 4 stanowi opis implementacji transformacji *Communication-2-Class* dla automatyzacji tworzenia diagramu klas w projekcie przypadku użycia. Sekcja 5 przedstawia wyniki testów opracowanej transformacji. W sekcji 6 zamieszczono podsumowanie i korzyści płynące z zastosowania transformacji *Communication-2-Class* w projektowaniu systemu informatycznego w kontekście czasu realizacji projektu. W sekcji tej przedstawiono także kierunki dalszych prac.

2. Prace powiązane tematycznie

W literaturze przedmiotu [4, 12] przyjmuje się następującą definicję transformacji typu model w model [13]: „Transformacja jest automatyczną generacją modelu docelowego z modelu źródłowego zgodnie z definicją transformacji. Definicja transformacji jest zbiorem reguł transformacji, które razem opisują, jak model w języku źródłowym może być transformowany w model w języku docelowym. Reguła transformacji jak jedna lub wiele konstrukcji z języka źródłowego może być transformowana w jedną lub wiele konstrukcji w języku docelowym”.

Po to aby można było transformować modele, muszą być one wyrażone w jakimś języku modelowania. Każdy model musi być zgodny z metamodelem, który określa składnię i semantykę określonego typu modeli. Metamodele są zwykle definiowane przy użyciu diagramu klas języka UML. Natomiast istnieją inne języki stosowane do tego celu [12], np. MetaObject Facility, Ecore metametamodel definiowany dla środowiska Eclipse Modeling Framework i Kernel MetaMetaModel. Ponadto, stosowane są różne podejścia dla definiowania i uruchamiania transformacji. Używa się podejść bazujących na bezpośredniej manipulacji (ang. *direct-manipulation*), gdzie transformacje tworzone są w języku programowania operującym na modelach w pamięci operacyjnej komputera (np. Java Metadata Interface), deklaratywnych, gdzie definiowane są reguły transformacji jako reguły matematyczne (np. Query/View/Transformation [14]), czy wykorzystujących transformacje grafów, gdzie modele traktowane są jako grafy (np. Attribute Graph Grammar). Ze względu na język modelowania można wyróżnić transformacje endogenne i egzogenne. Transformacje endogenne dotyczą modeli wyrażonych w tym samym języku modelowania. Natomiast transformacje egzogenne dotyczą modeli wyrażonych w różnych językach modelowania. Transformacje można podzielić także na horyzontalne i wertykalne. Z transformacją horyzontalną mamy do czynienia wtedy, gdy zarówno model źródłowy, jak i docelowy są na tym samym poziomie abstrakcji. Natomiast transformację uznaje się za wertykalną, jeśli modele źródłowy i docelowy są na różnych poziomach abstrakcji. Ponadto, transformacje modeli mogą być jednokierunkowe albo dwukierunkowe. Jednokierunkowe umożliwiają wygenerowanie modelu docelowego z modelu źródłowego, a dwukierunkowe pozwalają na utrzymanie spójności między modelami źródłowym i docelowym niezależnie od tego, w którym z nich zostaną wprowadzone zmiany. W transformacji dwukierunkowej niezbędne jest zastosowanie związku śledzenia (ang. *traceability*) [15]. Literatura przedmiotu jest bogata w zagadnienia transformacji modeli. Szczegółowa taksonomia transformacji modeli przedstawiona została w [4]. W literaturze opisywane są różne języki tworzenia transformacji: Query/View/Transformation (QVT) [14], ATL (Atlas Transformation Language) [16, 17]. Istotnymi zagadnieniami są weryfikacja i walidacja transformacji. Weryfikacja i walidacja transformacji modeli z zastosowaniem niezmienników (ang. *invariants*) przedstawiona została w [18]. Ponadto, przedstawiane są propozycje zastosowania języka QVT do budowy języka deklaratywnego do specyfikacji kontraktów wizualnych (ang. *visual contracts*), umożliwiającego weryfikację transformacji zdefiniowanych w dowolnym języku transformacji [19]. Aktualny, szeroki przegląd zagadnienia weryfikacji transformacji modeli znajduje się w [12]. Zagadnienie transformacji modeli wyrażonych w BPMN jest aktualne w literaturze. W [20] przedstawiono zagadnienie transformacji dwukierunkowej między modelami BPMN i BPEL. Dostępne są także prace pokazujące transformacje modeli przekształcające modele BPMN w modele UML. Dotyczą one transformacji BPMN w diagramy aktywności języka UML [21] oraz transformacji

BPMN w diagramy przypadków użycia języka UML [22]. MDA znajduje także swoje zastosowanie w programowaniu aspektowym [23] oraz w projektowaniu rozwiązań w architekturze usługowej [24]. Do implementacji transformacji zastosowana została rozszerzona platforma programistyczna Eclipse [25], która jest oprogramowaniem otwartym. Standardowe oprogramowanie otwarte jest także stosowane w administracji publicznej [26].

Przedstawiona w artykule transformacja *Communication-2-Class* dotyczy transformowania modeli UML. Należy podkreślić, że jest to rozwiązanie kompletne, wpisujące się w proces projektowania przypadku użycia i podnoszące efektywność tego procesu. Transformacja została zaimplementowana i przetestowana na różnej wielkości realizacjach przypadków użycia. Dzięki tej transformacji dostępna jest w narzędziu IBM RSA pełna ścieżka projektowania przypadku użycia. Zagadnienie dokumentowania decyzji architektonicznych jest istotne zarówno w trakcie projektowania systemu informatycznego [27], jak i na etapie jego eksploatacji [28].

3. Transformacja *Communication-2-Class*

Przy budowie systemu informatycznego zakres jego funkcjonalności, w podejściu obiektowym, określa diagram przypadków użycia języka UML. Przebieg zdarzeń w ramach przypadku użycia przedstawiany jest na diagramie aktywności. Dla każdego z przypadków użycia tworzona jest, w modelu analitycznym, realizacja przypadku użycia. W celu zaprojektowania przypadku użycia, w ramach realizacji przypadku użycia, tworzone są diagramy: sekwencji, komunikacji oraz klas. Budowa diagramu sekwencji realizowana jest przez projektanta — dystrybuowane są działania z diagramu aktywności do operacji poszczególnych obiektów klas identyfikowanych na diagramie sekwencji. Po utworzeniu diagramu sekwencji z pomocą przychodzi transformacja dostępna w środowisku IBM Rational Software Architect (RSA), która generuje diagram komunikacji na podstawie diagramu sekwencji dla realizacji przypadku użycia. Natomiast diagram klas dla realizacji przypadku użycia należy utworzyć ręcznie.

Środowisko RSA zawiera mechanizm pozwalający na generowanie diagramu klas [29]. Umożliwia on zbudowanie diagramu klas dla całego projektu, z elementów, które są już zdefiniowane. Jednak wraz ze wzrostem złożoności projektowanego systemu spada czytelność diagramów. Ponadto możliwa jest wizualizacja zaznaczonego zbioru klas na diagramie klas UML. Wywoływane jest to z menu kontekstowego RSA *Visualize* > *Add to New Diagram File* > *Class Diagram*.

Natomiast zaproponowana transformacja *Communication-2-Class* generuje diagram klas dla wybranego przypadku użycia, na podstawie informacji zawartych na diagramie komunikacji UML uprzednio utworzonego dla tego przypadku użycia. Diagram klas UML tworzony jest w kontekście określonego przypadku użycia. Istotne, że znajdą się na nim tylko klasy biorące udział w realizacji przypadku użycia.

Ponadto, nowym elementem transformacji *Communication-2-Class* jest tworzenie związków asocjacji (ang. *association*) oraz zależności (ang. *dependency*) między klasami biorącymi udział w realizacji przypadku użycia. Transformacja *Communication-2-Class*, na podstawie wywoływanych operacji oraz ich parametrów, potrafi określić typ związków łączących klasy.

Założenia przyjęte dla transformacji *Communication-2-Class*:

- Utworzenie diagramu klas z diagramu komunikacji.
- Obiekty na diagramie komunikacji odpowiadają klasom na diagramie klas.
- Aktorzy na diagramie komunikacji są pomijani w transformacji na diagram klas.
- Asocjacje z diagramu komunikacji odpowiadają asocjacjom na diagramie klas. Zwrot związku asocjacji na diagramie klas określony jest w kierunku klasy, której operacja jest wywoływana.
- Jeżeli na diagramie komunikacji parametrem wywołanej operacji jest typ obiektowy, to na diagramie klas dodawana jest klasa tego typu obiektowego i tworzony jest do niej związek zależności od klasy wywołującej operację.

W tabeli (tab. 1) przedstawiono mapowanie elementów diagramu komunikacji na elementy diagramu klas.

Tabela 1

Mapowanie elementów diagramu komunikacji na elementy diagramu klas

Diagram komunikacji	Diagram klas
Aktor	–
Obiekt	Klasa
Połączenie między obiektami	Związek asocjacji
Parametr z obiektowym typem danych w operacji wywoływanej na połączeniu między obiektami	Klasa
	Związek zależności

Transformacja *Communication-2-Class* została zaimplementowana w środowisku RSA. Wcześniej określono przypadek użycia „Generuj diagram klas" definiujący przebieg zdarzeń przy korzystaniu z opracowanej transformacji (tab. 2).

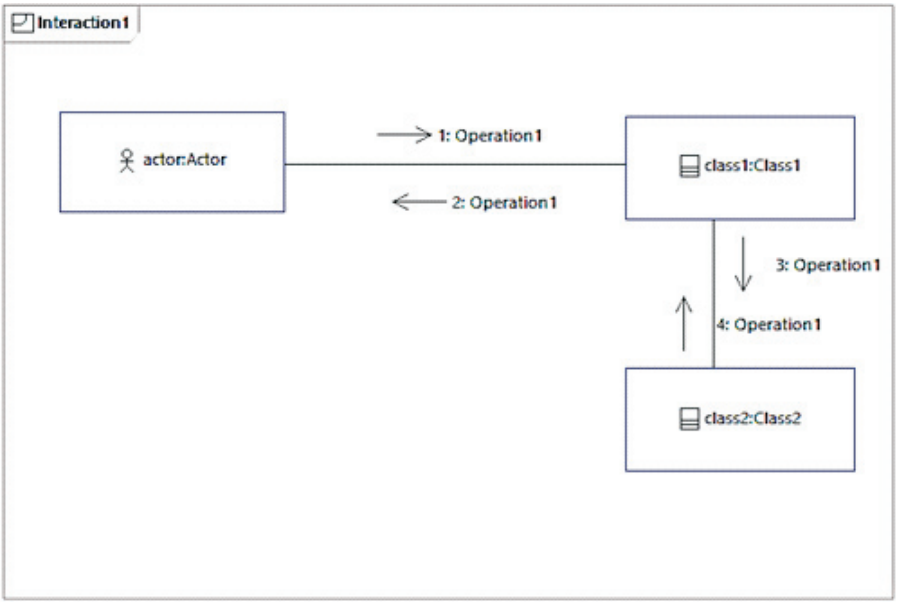
Na rysunkach (rys. 1, rys. 2) został przedstawiony przykład generacji diagramu klas z diagramu komunikacji. Na rysunku 1 widoczna jest asocjacja między instancjami klas *Class1* oraz *Class2*. Ten sam związek został przeniesiony na diagram klas. Związek zależności (ang. *dependency*) widoczny na rysunku 2 powstał na podstawie operacji *Operation1*, należącej do klasy *Class2*, gdzie jej parametr jest typu *Class3*.

Utworzony w ten sposób diagram klas będzie zawierał wszystkie klasy biorące udział w realizacji przypadku użycia oraz związki między klasami. Dopracowanie diagramu klas będzie wymagało już tylko zweryfikowania własności związków, np. liczności związku asocjacji.

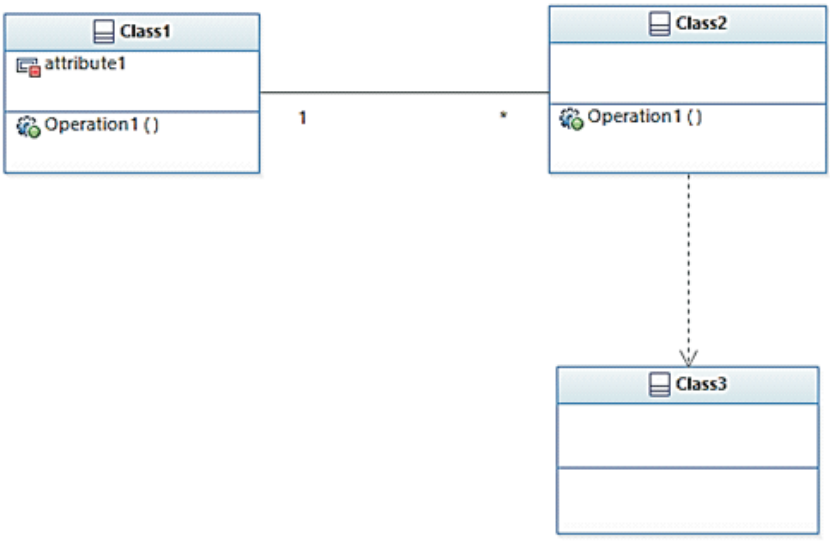
Tabela 2

Przebieg zdarzeń generowania diagramu klas z diagramu komunikacji

Warunki wstępne	Poprawnie utworzony diagram komunikacji
Aktorzy	Użytkownik
Przebieg podstawowy	1. Użytkownik klika prawym przyciskiem myszy na wybrany diagram komunikacji. 2. Z menu kontekstowego zostaje wybrana akcja *Create Class Diagram*. 3. Przeprowadzona zostaje weryfikacja, czy został wybrany poprawny diagram, wynik pozytywny. 4. Umożliwienie edytowania modelu. 5. Pobranie informacji o elementach w modelu. 6. Pobranie informacji o wybranym diagramie. 7. Utworzenie związków i klas na podstawie diagramu komunikacji. 8. Utworzenie nowego diagramu klas. 9. Dodanie elementów do diagramu klas.
Przebieg alternatywny	3.1. Przeprowadzona zostaje weryfikacja czy został wybrany poprawny diagram, wynik negatywny. 3.2. Wyświetlenie komunikatu o błędnym typie diagramu.



Rys. 1. Przykład diagramu komunikacji



Rys. 2. Diagram klas dla przykładu diagramu komunikacji

4. Implementacja transformacji *Communication-2-Class*

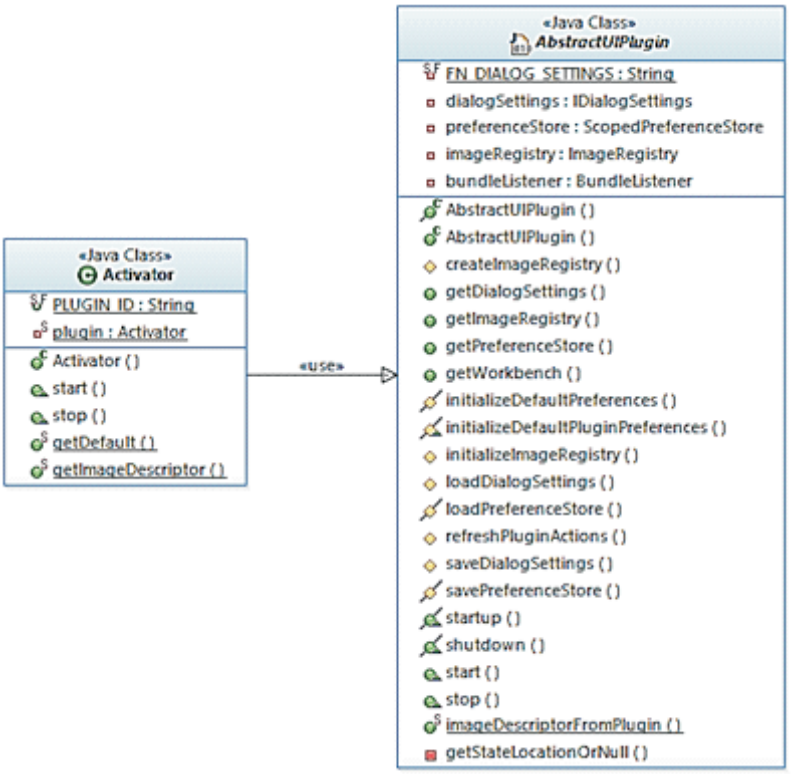
Transformacja *Communication-2-Class* została zaimplementowana w środowisku RSA w postaci wtyczki (ang. *plugin*). Środowisko RSA jest zbiorem rozszerzeń Eclipse. Wybranym punktem rozszerzeń jest *org.eclipse.ui.popupMenus*, dzięki któremu możliwe jest dodawanie nowych elementów do menu kontekstowego. Zaimplementowana wtyczka na wejściu przyjmuje diagram komunikacji. Wybrano dostęp do niego z widoku eksploratora projektów (ang. *Project Explorer*). Zadecydowało to o umieszczeniu akcji w menu kontekstowym widoków, a dokładnie w *org.eclipse.ui.navigator.ProjectExplorer#PopupMenu*. W pliku odpowiedzialnym za konfigurację wtyczki (*plugin.xml*) został dodany również warunek ograniczający możliwość uruchomienia rozszerzenia tylko na obiektach będących diagramem.

Wtyczka została podzielona na trzy pakiety:

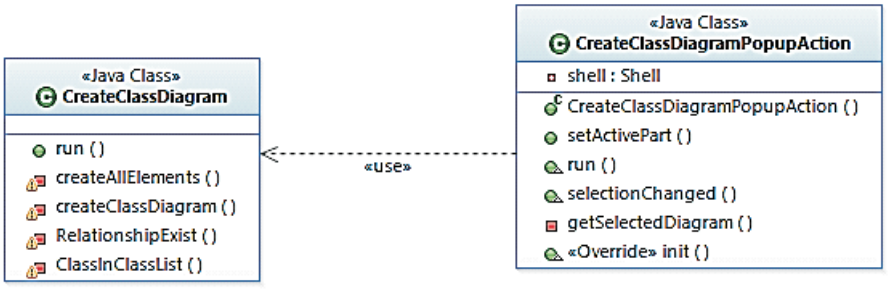
— Activator — kontroluje cykl życia wtyczki,
— Actions — zawiera w sobie klasy odpowiedzialne za wszystkie akcje wykonywane w ramach wtyczki,
— Model — zawiera klasy reprezentujące elementy diagramu, zarówno klas, jak i komunikacji.

Pakiet *Activator* zawiera klasę *Activator* rozszerzającą *AbstractUIPlugin* (rys. 3). Klasa ta kontroluje cykl życia wtyczki. Znajduje się w niej m.in. identyfikator wtyczki, metody odpowiedzialne za uruchomienie i zatrzymanie wtyczki.

Pakiet *Actions* zawiera w sobie klasy odpowiedzialne za wszystkie akcje wykonywane w ramach wtyczki (rys. 4).



Rys. 3. Elementy pakietu *Activator*



Rys. 4. Klasy w pakiecie *Actions*

Wtyczka rozpoczyna swoje działanie w momencie wybrania pozycji *Create Class Diagram* z menu kontekstowego. Pierwszą wykonaną przez nią akcją jest sprawdzenie, czy wybrano odpowiedni typ diagramu. Diagram musi być typu *Communication*, w przeciwnym wypadku zostanie wyświetlony komunikat o błędzie i wtyczka zakończy swoje działanie. Jeżeli wybrany diagram przejdzie tę weryfikację pozytywnie, jego

referencja zostanie przekazana do funkcji *run(Diagram diagram)*, należącej do nowo utworzonej instancji klasy *CreateDiagramClass*. W tej klasie odbywa się utworzenie diagramu klas. Następnie utworzona zostaje klasa anonimowa, wywołująca metodę *createAllElements(diagram)*, i tworzona jest lista klas, których instancje użyte są w diagramie komunikacji. Dalej wtyczka zajmuje się komunikatami. Komunikaty na diagramie komunikacji przesyłane są za pośrednictwem wiadomości. Każda wiadomość ma złącze (ang. *connector*) posiadające dwa końce (ang. *ends*). Reprezentują one elementy przesyłające komunikat. Mogą to być zarówno aktorzy, jak i klasy. We wtyczce pod uwagę brane są jedynie klasy.

Wtyczka, oprócz klas, uwzględnia utworzenie związków asocjacji oraz zależności. Zgodnie z przyjętym założeniem, asocjacje występują pomiędzy klasami, których instancje na diagramie komunikacji przesyłają między sobą komunikaty. W przypadku wystąpienia takich zależności należy utworzyć nową asocjację lub dodać do diagramu już istniejącą. W kodzie wtyczki sprawdzana jest każda asocjacja przeglądanej klasy. Jeżeli klasa nie posiada takiego związku z klasą, na której wywołuje operacje, jest ona tworzona i dodawana do listy asocjacji. W przeciwnym wypadku jest ona tylko dodawana do listy asocjacji. Związek zależności należy utworzyć lub

```
Parameter p = iter.next();

if (p.getType() instanceof Class) {
    Class p1 = (Class) p.getType();
    List<Dependency> deplist = c.getClientDependencies();
    for (Iterator<Dependency> iterd = deplist.iterator();
                                              iterd.hasNext();;) {
        Dependency dep = iterd.next();
    Class cl = (Class) dep.getRelatedElements().get(1);
        if (cl.equals(p1)
                && !RelationshipExist(c, p1, dependencyList)
                && ClassInClassList(p1, classesInModel)) {

            dependencyList.add(new ClsDependency(c, p1, dep));
        }
    }
    if (!RelationshipExist(c, p1, dependencyList)
            && ClassInClassList(p1, classesInModel)) {

    dependencyList.add(
                      new ClsDependency(c, (Class) p.getType(),
                      (Dependency)c.createDependency(p.get-
                      Type())));
    }
}
```

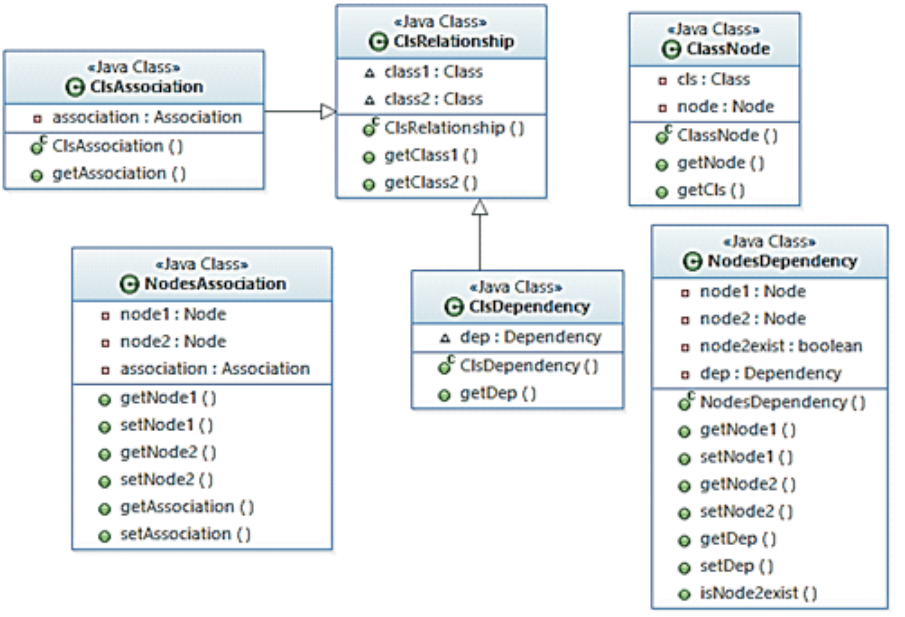
Rys. 5. Kod realizujący dodawanie związków zależności do diagramu klas

dodać do diagramu klas w przypadku, gdy parametrem operacji klasy jest inna klasa. Sprawdzane jest, czy operacja posiada parametry, które są instancją klasy *Class* z pakietu *org.eclipse.uml2.uml.Class*. Kolejną czynnością jest zweryfikowanie, czy pomiędzy klasami istnieje już zależność. W przypadku pozytywnej weryfikacji związek zależności zostaje dodany do listy związków. W przeciwnym wypadku związek zależności jest również tworzony (rys. 5).

Utworzenie diagramu klas realizowane jest przez wywołanie metody *createClassDiagram*. Opisane powyżej działania mają na celu dostarczenie tej metodzie danych. Metoda ta ma następującą postać: *createClassDiagram(Namespace m, List classesList, List dependencyList, List associationsList)*.

Działanie metody *createClassDiagram* ma następujący przebieg:

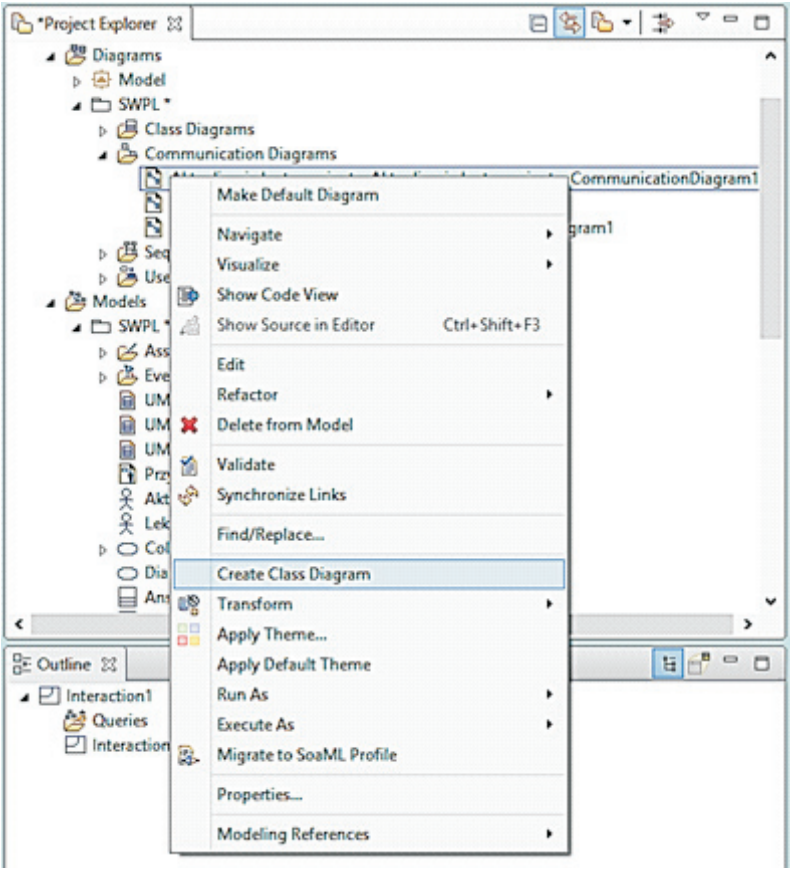
— Utworzenie diagramu klas.
— Dodanie do nowo utworzonego diagramu klas reprezentacji graficznej klas oraz utworzenie listy obiektów *ClassNode* (powiązań klasy z jej reprezentacją graficzną).
— Utworzenie i dodanie związków zależności do diagramu klas.
— Utworzenie listy wiążącej klasy na diagramie z asocjacjami.
— Dodanie związków asocjacji do diagramu klas.
— Otworzenie edytora nowo utworzonego diagramu klas.
— Wyświetlenie elementów na diagramie klas.



Rys. 6. Klasy w pakiecie Model

Pakiet *Model* zawiera klasy reprezentujące elementy diagramu, zarówno klas, jak i komunikacji (rys. 6). Zostały one utworzone w celu łatwiejszego zarządzania tymi elementami. Zawierają atrybuty, które można łatwo zdefiniować po nazwach klas:

— *ClassNode* — powiązanie między klasą a jej graficznym odpowiednikiem na diagramie.
— *ClsRelationship* — definiuje związek pomiędzy klasami.
— *ClsAssociation* — rozszerza *ClsRelationship*, konkretyzując związek asocjacji pomiędzy klasami.
— *ClsDependency* — rozszerza *ClsRelationship*, konkretyzując związek zależności pomiędzy klasami.
— *NodesAssociation* — powiązanie między graficznym odpowiednikiem klas a asocjacją między nimi.
— *NodesDependency* — powiązanie między graficznym odpowiednikiem klas a związkami zależności między nimi.



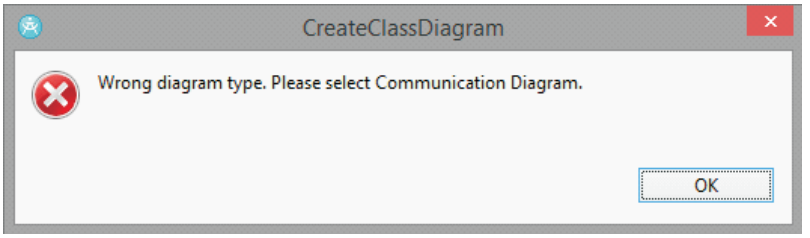
Rys. 7. Umiejscowienie opcji *Create Class Diagram* w menu kontekstowym

Każda z wyżej wypisanych klas oprócz atrybutów zawiera wyłącznie metody pozwalające na pobieranie i ustawianie ich wartości.

Najprostszym sposobem instalacji nowej wtyczki jest umieszczenie pliku wtyczki z rozszerzeniem **.jar* w katalogu *plugins* (*IBM\SDP\plugins*).

Po instalacji wtyczki i uruchomieniu środowiska pojawi się nowa opcja w menu kontekstowym eksploratora projektów *Create Class Diagram*. Będzie ona widoczna jedynie, gdy menu kontekstowe zostanie wywołane na elemencie typu *Diagram* (rys. 7).

Wtyczka uruchomi się tylko wywołana na diagramie komunikacji. Próba uruchomienia generacji diagramu klas na innym typie diagramu zakończy się wyświetleniem komunikatu o błędnym typie diagramu (rys. 8).



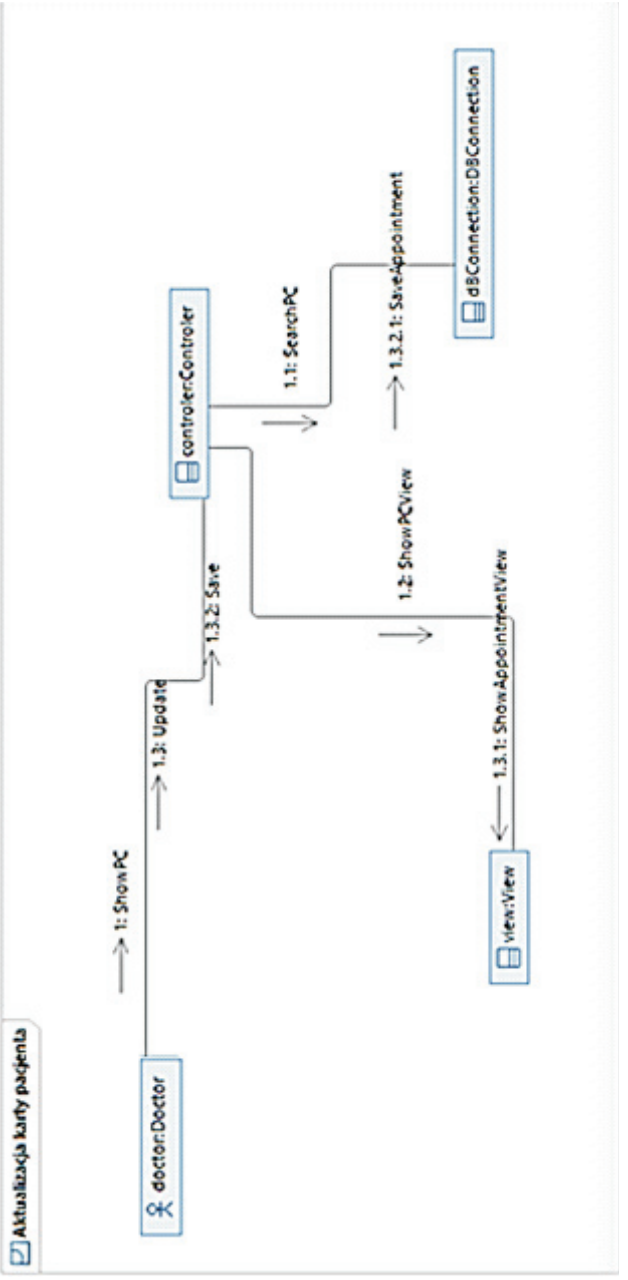
Rys. 8. Komunikat po wybraniu innego typu diagramu niż diagram komunikacji

5. Testy działania opracowanej transformacji

Zweryfikowano działanie opracowanego mechanizmu automatyzacji tworzenia diagramu klas na przykładach modelu analitycznego/projektowego fragmentu systemu wspomagającego pracę lekarza w przychodni zdrowia oraz modelu analitycznego/projektowego fragmentu systemu sprzedaży. W ramach modelu analitycznego/projektowego fragmentu systemu wspomagającego pracę lekarza w przychodni zdrowia modelowaniu poddany został przypadek użycia *Aktualizacja karty pacjenta*. Dla tego przypadku użycia opisano przebieg zdarzeń, a następnie utworzono realizację przypadku użycia i utworzono diagram sekwencji, dystrybuując działania do klas. Na podstawie diagramu sekwencji wygenerowano diagram komunikacji, używając mechanizmu dostępnego w środowisku RSA. Diagram komunikacji dla przypadku użycia *Aktualizacja karty pacjenta* przedstawiono na rysunku (rys. 9).

Testy przeprowadzono na komputerze o następujących parametrach wydajnościowych:

— procesor Intel Core i5-2410M,
— karta graficzna Geforce GT 540M/ 1 GB DDR3,
— RAM DDRIII 4GB.



Rys. 9. Diagram komunikacji dla realizacji przypadku użycia *Aktualizacja karty pacjenta*

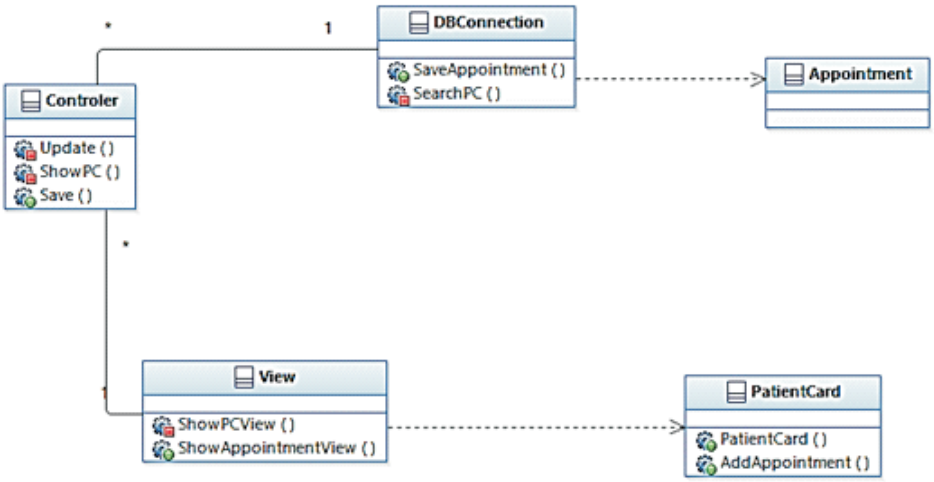
W testach jako miary wydajności działania transformacji *Communication-2-Class* przyjęto:

— czas generacji diagramu klas z otwarciem edytora graficznego — T_{ko},
— czas generacji diagramu klas — T_k,
— czas generacji/modelownia pojedynczej klasy — T_{pk}.

W celu pomiaru miar wydajności działania transformacji *Communication-2-Class* do kodu wtyczki dodano atrybuty zapisujące czas bezpośrednio przed uruchomieniem kodu odpowiedzialnego za generację diagramu klas oraz czas bezpośrednio po zakończeniu generacji diagramu klas. Nie dokonano żadnych zmian w kodzie generującym diagram klas. Ten sposób rejestracji czasu generacji nie wprowadza żadnych dodatkowych narzutów czasowych na generację diagramu klas. Dokonano tego w kodzie metody *run* w klasie *CreateClassDiagramPopupAction*.

Następnie przeprowadzono dwa testy funkcjonowania opracowanej transformacji *Communication-2-Class*.

Pierwszy test miał na celu walidację poprawności transformacji *Communication-2-Class* oraz uzyskanie wartości miar wydajności dla przypadku użycia, w którym występuje kilka klas. W pierwszym teście uruchomiono transformację *Communication-2-Class* na diagramie komunikacji dla realizacji przypadku użycia *Aktualizacja karty pacjenta*. Uzyskano poprawny diagram klas ze wszystkimi pięcioma wymaganymi klasami, łącznie z klasami będącymi w związkach zależności (rys. 10). Walidacja poprawności diagramu klas została dokonana przez analityka. Uzyskano także wartości miar wydajności T_{ko} i T_k dla realizacji przypadku użycia *Aktualizacja karty pacjenta*.



Rys. 10. Diagram klas dla realizacji przypadku użycia *Aktualizacja karty pacjenta*

Celem drugiego testu było uzyskanie wartości miar wydajności dla generacji diagramu klas o rząd wielkości większego od tego w teście pierwszym. W tym celu wykonano fragment modelu analitycznego/projektowego dla systemu sprzedaży. Dokonano modelowania przypadku użycia *Złożenie zamówienia*. W wyniku uruchomienia transformacji *Communication-2-Class* uzyskano diagram klas z 50 klasami biorącymi udział w realizacji tego przypadku użycia. Wyniki testów przedstawiono w tabeli (tab. 3).

Tabela 3

Podsumowanie testów

Liczba klas	T_{ko} [s]	T_k [s]	T_{pk} [s]
PU *Aktualizacja karty pacjenta* — 5 klas	0,567	0,281	0,0562
PU *Złożenie zamówienia* — 50 klas	2,984	1,476	0,0295

Ponadto zamodelowano diagramy klas z testów pierwszego i drugiego ręcznie bez użycia opracowanej transformacji, mierząc czas realizacji tej czynności. W tym przypadku uzyskano czas T_k oraz T_{pk}, gdyż analityk najpierw otwierał pusty diagram klas i dodawał klasy do diagramu. Wyniki testów przedstawiono w tabeli (tab. 4).

Tabela 4

Podsumowanie testów

Liczba klas	T_k [s]	T_{pk} [s]
PU *Aktualizacja karty pacjenta* — 5 klas	110	22
PU *Złożenie zamówienia* — 50 klas	920	18,4

Z analizy wartości w tabelach 3 oraz 4 wynika, że czasy generacji diagramu klas są 400 (5 klas)-600 (50 klas) razy krótsze niż czasy ręcznego modelowania tego diagramu.

Ponadto, w przypadku generacji diagramu klas przy zastosowaniu transformacji *Communication-2-Class*, zaobserwowano, że przy 10-krotnym wzroście rozmiaru diagramu klas czas generacji diagramu klas wzrósł tylko 5-krotnie. Wynika z tego, że wraz ze wzrostem rozmiaru diagramu klas (w sensie liczby klas) skraca się czas potrzebny na generację pojedynczej klasy T_{pk}. Czas generacji/modelowania pojedynczej klasy T_{pk} dla diagramu klas z 50 klasami stanowi 53% czasu potrzebnego na generację pojedynczej klasy dla diagramu klas z 5 klasami. Najwyraźniej kosztowne czasowo są operacje tworzenia struktur danych wtyczki, a operowanie nimi jest mniej czasochłonne. Możliwe, że wpływ może mieć także rozmiar poszczególnych klas oraz liczba klas istniejących w projekcie ze standardowych bibliotek — atrybuty operacji. Za tym drugim argumentem przemawiać może fakt, że w przypadku ręcznego modelowania diagramu klas także zaobserwowano zjawisko skrócenia czasu

generacji/modelowania pojedynczej klasy. Czas generacji/modelowania pojedynczej klasy T_{pk} dla diagramu klas z 50 klasami stanowi 83% czasu potrzebnego na modelowanie pojedynczej klasy dla diagramu klas z 5 klasami.

Ponadto porównano wartości T_{ko} i T_k dla transformacji *Communication-2-Class*. Czas potrzebny na otwarcie edytora graficznego i umieszczenie na diagramie wygenerowanych klas stanowi 50,4% czasu T_{ko} dla diagramu klas z 5 klasami oraz 50,5% czasu T_{ko} dla diagramu klas z 50 klasami. Wynika z tego, że czas potrzebny na otwarcie edytora graficznego i umieszczenie na diagramie wygenerowanych klas jest wprost proporcjonalny do liczby klas podlegających generacji, niezależnie od rozmiaru diagramu klas.

6. Podsumowanie

W artykule przedstawiono transformację *Communication-2-Class* umożliwiającą automatyzację konstrukcji diagramu klas języka Unified Modeling Language (UML) tworzonego w realizacji przypadku użycia w ramach modelu analitycznego/projektowego. Diagram klas UML tworzony jest na podstawie diagramu komunikacji UML. Dzięki temu diagram klas przedstawia wszystkie klasy zaangażowane w realizację przypadku użycia i związki między nimi. Wtyczka realizująca transformację *Communication-2-Class* została zrealizowana w środowisku RSA. Należy podkreślić, że jest to rozwiązanie kompletne, wpisujące się w proces projektowania przypadku użycia i podnoszące jego efektywność. Platforma Eclipse, będąca podstawą RSA, okazała się bardzo przydatna w budowie wtyczki realizującej transformację *Communication-2-Class*. Transformacja została przetestowana na różnej wielkości realizacjach przypadków użycia. W artykule przedstawiono także wyniki testów opracowanej wtyczki realizującej transformację *Communication-2-Class* pokazujące możliwości skrócenia czasu projektowania realizacji przypadku użycia. Czasy generacji diagramu klas są 400 (5 klas)-600 (50 klas) razy krótsze niż czasy ręcznego modelowania tego diagramu. Ponadto, w przypadku generacji diagramu klas przy zastosowaniu transformacji *Communication-2-Class* zaobserwowano, że przy 10-krotnym wzroście rozmiaru diagramu klas czas generacji diagramu klas wzrósł tylko 5-krotnie. Wynika z tego, że wraz ze wzrostem rozmiaru diagramu klas (w sensie liczby klas) skraca się czas potrzebny na generację pojedynczej klasy. W obecnym rozwiązaniu walidacja wygenerowanego diagramu klas realizowana jest przez analityka. W toku dalszych prac planowane jest zaprojektowanie i implementacja oprogramowania walidującego wygenerowany diagram klas. Rozważane jest także opracowanie transformacji dwukierunkowej z diagramu klas do kodu aplikacji z uwzględnieniem stereotypów klas. Będzie to miało konsekwencje w postaci generacji do kodu aplikacji realizującej logikę biznesową, a także struktur bazy danych i interfejsu użytkownika. Tego typu transformacja mogłaby potencjalnie znacznie poprawić spójność projektu z kodem aplikacji.

Źródło finansowania pracy — działalność statutowa uczelni.

Artykuł wpłynął do redakcji 29.09.2015 r. Zweryfikowaną wersję po recenzjach otrzymano 13.01.2016 r.

LITERATURA

[1] Mellor S., Clark A., Futagami T., *Model-driven development — guest editor's introduction*, Softw., IEEE, 20 (5), 2003, 14-18.

[2] OMG, MDA Guide rev. 2.0 (18 June 2014), http://www.omg.org/cgi-bin/doc?ormsc/14-06-01, dostęp 23 września 2015.

[3] Lahman H.S., *Model-Based Development: Applications*, Addison-Wesley, Westford, 2011.

[4] Mens T., van Gorp P., *A taxonomy of model transformation*, Electronic Notes In Theoretical Computer Science, 152, 2006, 125-142.

[5] Groote J.F., Osaiweran A.A.H., Wesselius J.H., *Analyzing the effects of formal methods on the development of industrial control software*, IEEE ICSM, 2011, 467-472.

[6] Van den Brand M.G.J., Groote J.F., *Advances in Model Driven Software Engineering*, ERCIM News, 91, 2012, 23-24.

[7] Koch N., Knapp A., Kozuruba S., *Assessment of Effort Reduction due to Model-to-Model Transformations in the Web Domain*, Lecture Notes in Computer Science, 7387, 2012, 215-222.

[8] Górski T., *Automatyzacja projektowania przepływów mediacyjnych*, [w:] *Projektowanie systemów informatycznych: modele i metody*, Wojskowa Akademia Techniczna, 2014, 19-34.

[9] Karakostas B., Zorgios Y., *Engineering Service Oriented Systems: A Model Driven Approach*, IGI Global, Londyn, 2008.

[10] Sewall S.J., *Executive Justification for Adopting Model Driven Architecture (MDA)*, http://www.omg.org/mda/mda_files/11-03_Sewall_MDA_paper.pdf, dostęp 23 września 2015.

[11] Unified Modeling Language Specification Version 2.4.1, OMG 2011, http://www.omg.org/spec/UML/2.4.1/, dostęp 23 września 2015.

[12] Calegari D., Szasz N., *Verification of model transformations a survey of the State-of-the-art.*, Electronic Notes In Theoretical Computer Science, 292, 2013, 5-25.

[13] Kleppe A.J., Warmer J., Bast W., *MDA Explained, The Model Driven Architecture: Practice and Promise*, Addison-Wesley, Boston, 2003.

[14] Rensink A., Nederpel R., *Graph Transformation Semantics for a QVT Language*, Electronic Notes in Theoretical Computer Science, 211, 2008, 51-62.

[15] Santiago I., Jiménez Á., Vara J.M., De Castro V., Bollati V.A., Marcos E., *Model-Driven Engineering as a new landscape for traceability management: A systematic literature review*, Information and Software Technology, 54, 2012, 1340-1356.

[16] Bézivin J., Jouault F., *Using ATL for checking models, Electronic Notes In Theoretical Computer Science*, 152, 2006, 69-81.

[17] Jouault F., Allilaire F., Bézivin J., Kurtev I., *ATL: A model transformation tool*, Science of Computer Programming, 72, 2008, 31-39.

[18] Cabot J., Clarisó R., Guerra E., de Lara J., *Verification and validation of declarative model-to-model transformations through invariants*, The Journal of Systems and Software, 83, 2010, 283-302.

[19] Guerra E., de Lara J., Wimmer M., Kappel G., Kusel A., Retschitzegger W., Schönböck J., Schwinger W., *Automated verification of model transformations based on visual contracts*, Automated Software Engineering, 20, 2013, 5-46.

[20] Mazanek S., Hanus M., *Constructing a bidirectional transformation between BPMN and BPEL with a functional logic programming language*, Journal of Visual Languages and Computing, 22, 2011, 66-89.

[21] Cibrán M.A., *Translating BPMN Models into UML Activities*, Lecture Notes in Business Information Processing, 17, 2009, 236-247.

[22] Górski T., *Model-to-model transformations of architecture descriptions of an integration platform*, Journal of Theoretical and Applied Computer Science, 8 (2), 2014, 48-62.

[23] Mehmood A., Jawawi D.N.A., *Aspect-oriented model-driven code generation: A systematic mapping study*, Information and Software Technology, 55, 2013, 395-411.

[24] Ameller D., Burgués X., Collell O., Costal D., Franch X., Papazoglou M.P., *Development of service-oriented architectures using model-driven development: A mapping study*, Information and Software Technology, 62, 2015, 42-66.

[25] Shavor S., D'Anjou J., Fairbrother S., Kehn D., Kellerman J., McCarthy P., *Eclipse Podręcznik Programisty*, Wydawnictwo Helion, Gliwice, 2005.

[26] Wieczorkowski J., *Zastosowanie oprogramowania standardowego w administracji publicznej*, Roczniki Kolegium Analiz Ekonomicznych, 29, 2013, 343-352.

[27] Connor R.V., Lepmets M., *Exploring the Use of the Cynefin Framework to Inform Software Development Approach Decisions*, Proceedings of the 2015 International Conference on Software and System Process, ICSSP 2015, ACM, 2015, 97-101.

[28] Czarnecki A., Klich L., Orłowski C., *Simulation of the IT Service and Project Management Environment*, Biuletyn WAT, 62, 1, 2013, 161-180.

[29] Xu F., Wood A., *Reverse engineering UML class and sequence diagrams from Java code with IBM Rational Software Architect. Three techniques to overcome limitations*, 2008, http://www.ibm.com/developerworks/rational/library/08/0610_xu-wood/, dostęp 23 września 2015.

T. GÓRSKI, M. SOWA

Construction of UML class diagram with Model-Driven Development

Abstract. Model transformations play a key role in software development projects based on Model-Driven Development (MDD) principles. Transformations allow for automation of repetitive and well-defined steps, thus shortening design time and reducing a number of errors. In the object-oriented approach, the key elements are use cases. They are described, modelled and later designed until executable application code is obtained. The aim of the paper is to present transformation of a model-to-model type, Communication-2-Class, which automates construction of Unified Modelling Language (UML) class diagram in the context of the analysis/design model. An UML class diagram is created based on UML communication diagram within use case realization. As a result, a class diagram shows all of the classes involved in the use case realization and the relationships among them. The plug-in which implements Communication-2-Class transformation was implemented in the IBM Rational Software Architect. The article presents the tests results of developed plug-in, which realizes Communication-2-Class transformation, showing capabilities of shortening use case realization's design time.

Keywords: Model-Driven Development, transformations, Unified Modelling Language, analysis/design model, UML class diagram, UML communication diagram
DOI: 10.5604/12345865.1197989