AKADEMIA GÓRNICZO-HUTNICZA

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Autoreferat rozprawy doktorskiej

Wykorzystanie transformacji grafowych do wykluczania konfliktów w czasie refaktoryzacji prowadzonej w środowisku rozproszonym

mgr Adrian Nowak

Promotor: dr hab. Leszek Kotulski, prof. AGH

Kraków, 2013

1 Wprowadzenie

Złożoność systemów informatycznych, konieczność ich ciągłego rozwoju i pielęgnacji, a przy tym kwestie zapewnienia jakości i odpowiedniej wydajności pracy zespołów programistów-projektantów, powodują iż zapanowanie nad projektami oraz prowadzenie prac rozwojowych jest zadaniem bardzo trudnym. Taki stan rzeczy skłania do poszukiwania odpowiednich metodyk tworzenia i pielęgnacji oprogramowania oraz do opracowywania coraz to efektywniejszych sposobów pracy i współpracy członków zespołów, a co za tym idzie poszukiwania nowych możliwości rozwoju środowisk programistycznych. Refaktoryzacja, która z definicji ma podnosić jakość projektu, będąc podstawowym elementem takich nowoczesnych metodyk jak programowanie ekstremalne czy Scrum zakładających iteracyjne tworzenie oprogramowania, jest techniką wspomagająca zadanie pierwsze. Pozwala przygotować oprogramowanie na przyszłe rozszerzenia, czyni go łatwiejszym do analizy, pozwala wyszukać błędy czy wreszcie usprawnić programowanie. W pracy uznając refaktoryzację za kluczową technikę zdefiniowaliśmy środowisko wspomagające współpracę członków zespołów projektowych.

Podstawową motywacją w prowadzonych badaniach była obserwacja, iż oprogramowanie jest tworzone i rozwijane przez zespoły ludzi, które są najczęściej rozproszone geograficznie. Projektanci i programiści pracują na własnej kopii oprogramowania w swoim lokalnym środowisku, często nie tylko bez wiedzy o modyfikacjach wprowadzanych przez innych ale także o samych

członkach zespołu. Zarządzanie wprowadzaniem równoległych zmian oraz wsparcie ludzi przeprowadzających te zmiany jest fundamentalną kwestią podczas budowy i rozbudowy złożonych systemów dużej skali. Tymczasem okazuje się, iż brak jest narzędzi wspomagających przeprowadzanie refaktoryzacji w środowisku rozproszonym przyczyniających się tym samym do redukcji kosztów jej wprowadzania. Dostępne systemy zarządzania konfiguracją (ang. Software Configuration Management - SCM) stawiając na uniwersalność, wykorzystują wyłącznie techniki łączenia tekstów przez co są niezależne od języka programowania, lecz nie są w stanie wykorzystać żadnych informacji o przekształcanej strukturze. Powoduje to, że już przy najprostszych przekształceniach refaktoryzujących wprowadzanych równolegle przez dwóch programistów (np. przesunięcie zmiennej składowej między klasami i jednoczesna zmiana jej nazwy przez inną osobę) po scaleniu techniką łączenia tekstów pojawiają się poważne problemy. W najlepszym wypadku użytkownik dostaje sygnał o licznych, niezrozumiałych dla środowiska błędach, gdyż fragmenty scalone jako linie tekstowe mogą prowadzić do błędów kompilacji lub do niepoprawnych, choć czasem kompilowalnych, programów. Prawidłowe łączenie zmian zależne jest bowiem od syntaktyki kodu (odpowiednio modelu) oraz od semantyki operacji.

Ryzyko nakładania się zmian rośnie szczególnie podczas wykonywania refaktoryzacji za sprawą ich nieograniczonego zasięgu (poza ramy pakietów i modułów). Sytuacja komplikuje się dodatkowo, kiedy wprowadzanych zmian jest setki. Co więcej błędy wykrywane są dość późno, bowiem dopiero kiedy ostatni członek zespołu udostępnia swoje modyfikacje. Niebezpieczeństwa takie zmuszają członków zespołu do wzmożonego zaangażowania podczas procesu łączenia zmian co powoduje, że w praktyce w sposób sztuczny ogranicza się możliwość przeprowadzania refaktoryzacji lub wręcz jej zabrania. Komfort pracy w takich warunkach jest wątpliwy; zachodzi również sprzeczności takiego podejścia z paradygmatem programowania zwinnego.

Zależności między refaktoryzacjami oraz problem łączenia zmian po refaktoryzacji i nasilającego się występowania konfliktów podczas tego procesu staje się przedmiotem coraz intensywniejszych badań. Dotychczasowe podejścia nie uwzględniają jednak interakcji w rzeczywistym, rozproszonym środowisku wieloużytkownikowym, w którym projektanci mogą wprowadzać całe sekwencje operacji w sposób dynamiczny. Brak także odpowiedniego modelu formalnego dla opisu oprogramowania pozwalającego na właściwy podgląd jego ewolucji oraz na możliwość wykorzystania wiedzy o semantyce refaktoryzacji.

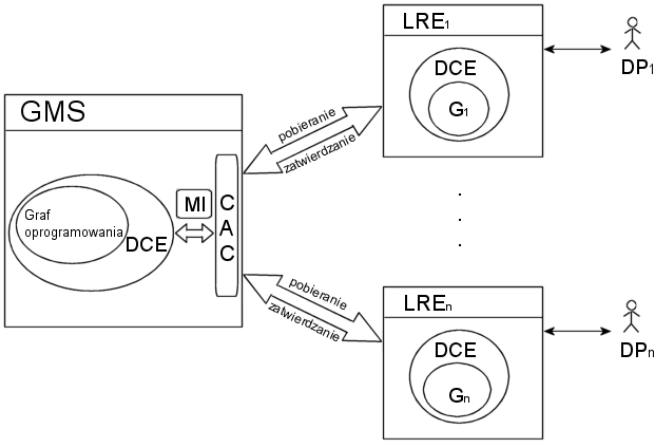
2 Teza i cel badań

W rozprawie sformułowano tezę, iż **możliwe jest opracowanie metod i systemu dla automatycznego wspomagania refaktoryzacji**, który:

- umożliwi równoległą pracę zespołów projektowych w środowisku rozproszonym (przy częściowej replikacji struktur danych) i wykryje błędy wynikające z równoległego zlecania refaktoryzacji;
- automatycznie wyeliminuje skutki konfliktów, w przypadku równoległego wykonywania elementarnych refaktoryzacji;
- umożliwi łączenie elementarnych operacji w sekwencje i wprowadzanie ich w sposób transakcyjny oraz zminimalizuje nakład pracy projektantów, w przypadku rozwiązywania konfliktów spowodowanych równoległym wykonaniem różnych sekwencji refaktoryzacji.

3 System kontroli refaktoryzacji z repozytorium grafowym

W celu wspomagania refaktoryzacji prowadzonej w środowisku rozproszonym przez zespoły projektantów-programistów zaprojektowany został system rozproszony DRE – Distributed Refactorization Environment. System może składać się z dowolnej liczby lokalnych środowisk LRE (odpowiadających środowiskom pracy poszczególnych projektantów-programistów) oraz węzła centralnego GMS (ang. Graph Management System) – rysunek 1. Koncepcja architektury zakłada zatem realizację rozproszonego i częściowo zreplikowanego repozytorium struktury oprogramowania podlegającego refaktoryzacji, przy założeniu, że refaktoryzacja tego oprogramowania odbywa się lokalnie w środowiskach LRE.



Rysunek 1. Model pracy w systemie DRE z rozproszonym repozytorium grafowym.

Nadrzędnym zadaniem tworzonego środowiska dla pracy zespołowej jest zarządzanie wprowadzaniem równoległych zmian refaktoryzujących. System musi zadbać o utrzymanie spójności oprogramowania, nawet gdy równolegle podjęte decyzje prowadzą do konfliktu. W drugiej kolejności, system musi starać się zminimalizować koszt (po stronie zespołów projektowych) rozwiązywania konfliktów, w sytuacjach gdy mechanizm synchronizacji nie będzie w stanie wykluczyć ich całkowicie. Koncepcja systemu zakłada również, że mogą występować duże odstępy czasowe pomiędzy momentem rozpoczęcia lokalnej operacji, a momentem decyzji o zatwierdzeniu wprowadzonych zmian z repozytorium (zatwierdzenia łańcuch refaktoryzacji). Formalnie od środowiska oczekujemy szeregu własności takich jak: otwartość, równoległość, rozproszenie, transparentność czasowa, niepodzielność, globalna spójność i trwałość.

Opierając się na doświadczeniach wykorzystania grafów oraz gramatyk grafowych m.in. do opisu i wnioskowania w systemach rozproszonych oraz agentowych za podstawę reprezentacji oprogramowania w środowisku DRE wybrana została reprezentacja grafowa. Jądro systemu DRE stanowi repozytorium grafowe, które utrzymuje reprezentację oprogramowania w postaci grafu. Repozytorium zapewnia mechanizmy dla kontroli generacji grafu, rejestracji przeprowadzonych operacji oraz możliwość przechowywania historii zmian. Gwarantuje ono również jednoznaczną identyfikację wierzchołkom grafu, która jest utrzymywana przez cały czas trwania projektu reprezentowanego oprogramowania. Repozytorium jest rozproszone – w każdym ze środowisk LRE utrzymywane jest repozytorium lokalne z odpowiednim fragmentem grafu oprogramowania natomiast za utrzymanie globalnego repozytorium oraz zapewnienie transparentnego dostępu (ze środowisk LRE) do obsługiwanego w nim grafu oprogramowania odpowiada GMS.

4 Reprezentacja oprogramowania i jego ewolucji

W celu zapewnienia wymaganych własności systemu oraz sprawnej synchronizacji prac członków zespołu model dowolnego programu reprezentujemy przez graf skierowany, atrybutywny i etykietowany zarówno wierzchołkowo jak i krawędziowo. Graf ten odzwierciedla aktualny stan modelowanego oprogramowania. Wierzchołki grafu są jednoznacznie indeksowane i odpowiadają poszczególnym komponentom oprogramowania, a więc definicjom klas (typów), zmiennych, metod oraz parametrów metod. Formalnie, w oparciu o podejście algebraiczne single pushout (SPO), graf oprogramowania zdefiniowano następująco:

Graf oprogramowania to atrybutowany wierzchołkowo i krawędziowo graf zorientowany o zaetykietowanych wierzchołkach i krawędziach GOP = (V, E, Σ, Γ, δ, γ, *s*, *t*, υ, λ, I), gdzie:

V – jest skończonym, niepustym zbiorem wierzchołków, którym zostały przypisane w sposób jednoznaczny indeksy przez funkcję indeksującą I,

E – jest zbiorem krawędzi; $E \subset V \times \Gamma \times V$,

Σ – jest zbiorem etykiet wierzchołkowych;

Σ = {N, Class, Variable, Operation, Method, Parameter},

Γ – jest zbiorem etykiet krawędziowych;

Γ = {ϕ, <<member>>, <<inheritance>>, <<type>>, <<lookup>>,

<<param>>, <<access>>, <<update>>, <<call>>, <<create>>},

δ: V $\rightarrow \Sigma$ – jest funkcją etykietującą wierzchołki,

γ: E $\rightarrow \Gamma$ – jest funkcją etykietującą krawędzie,

s: E $\rightarrow$ V – wskazuje wierzchołek będący punktem startowym krawędzi,

t: E $\rightarrow$ V – wskazuje wierzchołek będący punktem docelowy krawędzi,

υ: V $\rightarrow$ (*NodeAtrib* $\rightarrow$ *NodeAtribValue*) – jest funkcją atrybutującą wierzchołki, przypisuje wierzchołkowi zbiór par złożonych z nazwy atrybutu oraz wartość,

λ: E $\rightarrow$ (*EdgeAtrib* $\rightarrow$ *EdgeAtribValue*) – jest funkcją atrybutującą krawędzie, przypisuje krawędzi zbiór par złożonych z nazwy atrybutu oraz wartość,

I: V $\rightarrow$ N – jest funkcją różnowartościową indeksującą wierzchołki.

Aby uzyskać kompletny model reprezentacji oprogramowania konieczne było zrealizowanie w ramach repozytorium grafowego możliwości generacji (jako reakcji na działania projektanta) tylko i wyłącznie poprawnych grafów. W tym celu w pracy przyjęto rozwiązanie bazujące na doborze zbioru przekształceń (w tym refaktoryzacji o określonych warunkach stosowalności) określonych w oparciu o przypadki stosowania (ang. use cases) i odpowiadających wszystkim typowym operacjom stosowanym przez projektantów podczas pracy nad rozwojem oprogramowania (edycji, refaktoryzacji i usuwania), a ze złożenia przekształceń można uzyskać każdy możliwy (wyłącznie poprawny) graf oprogramowania. Zbiór w taki sposób dobrany może zostać wprost wykorzystany do opisu ewolucji oprogramowania.

Przekształceniom oprogramowania odpowiadają transformacje grafowe, które wraz z warunkami stosowalności zapobiegać będą również niepoprawnym konstrukcjom językowym (jak na przykład dodaniu dwóch metod o tej samej nazwie w jednej klasie). Zbiór transformacji stanowi

produkcje skonstruowanej w pracy gramatyki grafowej Gr_{GOP}, która wspomaga formalną specyfikację modelu w środowisku rozproszonym ze szczególnym uwzględnieniem opisu ewolucji oprogramowania zapewniając przy tym m.in. możliwość kontroli przeprowadzania refaktoryzacji. Aby mieć możliwość sterowania kolejnością wykonania produkcji, a także sprawnie ze sobą łączyć całe przekształcenia zastosowano specjalne diagramy sterujące – Derivation Control Diagram (DCD) oraz środowisko wykonania – Derivation Control Environment (DCE).

Do opisu gramatyki i transformacji grafu oprogramowania wykorzystana została notacja Single Pushout. Zapewnia ona bardzo dobrą czytelność, a do tego możliwości formalnego dowodzenia o strukturze grafu (dowodzenie takie jest w dużym stopniu niezależne od struktury opisywanych obiektów). Nie jest jednak efektywna dla implementacji budowanego systemu DRE wymagającego działania on-line. W celu zbudowania efektywnie funkcjonującego systemu wystarcza wprowadzić dodatkowe założenia odnośnie produkcji gramatyki, tak aby uzyskać gramatykę klasy ETPL(k). Zapisanie produkcji jako ETPL(k) jest tylko kwestią techniczną – zamiast pojedynczej produkcji przekształcenie reprezentować będzie ciąg produkcji i diagram sterujący. Gramatyka ETPL(k) pozwala rozwiązać problem generacji nowego grafu jak i wnioskowania o przynależności danego grafu do klasy grafów generowanej przez język (ang. membership problem) ze złożonością obliczeniową $O(n^2)$ co może być w przyszłości użyteczne dla rozszerzenia systemu o moduł identyfikacji fragmentów do przekształcenia i wsparcia procesów decyzyjnych.

5 Schemat eliminacji elementarnych konfliktów

W pracy przyjmujemy określenie konfliktu refaktoryzacji wprowadzanych w środowisku rozproszonym (opartym o tradycyjny system typu SCM) jako operacji, których równoległe zastosowanie i automatyczne łączenie prowadziłoby do niezgodnego z oczekiwaniami lub niepoprawnego programu.

Zasadniczą przyczyną większości z konfliktów refaktoryzacji jest zaburzenie warunków stosowalności operacji. Z globalnego punktu widzenia dochodzi do tego wobec jednej z refaktoryzacji po (równoległym) wprowadzeniu innej, w innym środowisku lokalnym. Warunki wstępne wymagane dla stosowania operacji są spełnione, jednak tylko lokalnie, gdyż zmiana warunków globalnych nie jest widoczna z perspektywy żadnego ze środowisk lokalnych. Zastosowanie produkcji nie jest już wtedy możliwe ze względu na niespełnione warunki stosowalności przekształcenia.

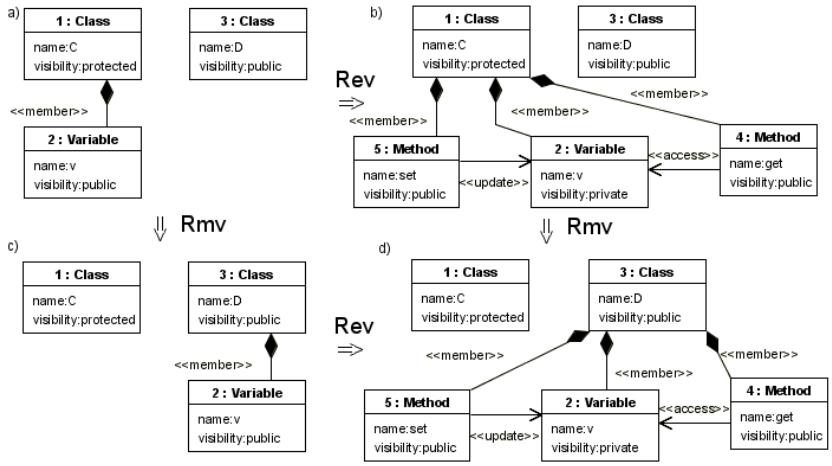
Powodem konfliktu może być zmiana kontekstu wykonania. Z taką sytuacją mamy do czynienia kiedy użytkownik zmieni nazwę komponentu, a inna refaktoryzacja odwołuje się do niego nadal poprzez poprzednią nazwę (co ma miejsce zawsze w przypadku tradycyjnych systemów SCM). W środowisku lokalnym lewa strona produkcji odpowiadającej refaktoryzacji jest dostępna, ale globalnie w grafie kontekst nie istnieje. Przyczyną błędów bywa także nieznajomość semantyki kodu oraz semantyki operacji, która przez wprowadzenie przypadkowych konstrukcji może prowadzić m.in. do niepoprawnego zachowania programu.

Wiele z możliwych przypadków konfliktów dla par refaktoryzacji ma podobny charakter. Jest tak z uwagi na podobną semantykę niezgodnych operacji. Potencjalne konflikty między podobnymi parami są zatem tej samej natury – powodowane tymi samymi lub podobnymi czynnikami. Wystarczy zająć się wybranymi problemami i rozwiążemy konflikty powodowane przez pary operacji reprezentatywnych.

W systemie DRE przechodzimy na wyższy poziom abstrakcji – nie mówimy o operacjach edycji tekstu, tylko o przekształceniach struktury kodu lub modelu oprogramowania, zgodnie z zaproponowanym zbiorem operacji. System znając semantykę operacji jest w stanie traktować je „inteligentnie”. Pozwoli to wyeliminować część konfliktów znane z tradycyjnego środowiska pracy opartego na plikach tekstowych.

Analizując przyczyny konfliktów doszliśmy do wniosku, iż dla sporej klasy problemów podstawowym powodem ich występowania jest brak unikalnej i niezmiennej identyfikacji komponentów. Z tego względu repozytorium grafowe, operując na komponentach opisywanego oprogramowania, od początku nadaje im unikatowe identyfikatory (ma to również znaczenie dla wydajność), które zachowują wartość na czas trwania projektu. Środowiska lokalne LRE potrafią zgrywać operacje, które są następnie, za pośrednictwem GMS, odtwarzane podczas synchronizacji z repozytorium globalnym. Te mechanizmy pozwolą na poprawne scalanie zmian obejmujące użycie tych samych komponentów w różnych środowiskach, a dzięki temu wyeliminowanie występowania konfliktów, z którymi nie radziły sobie środowiska typu SCM oparte na plikach tekstowych. Tym samym liczba konfliktów refaktoryzacji, które da się rozwiązać w sposób automatyczny w systemie DRE jest większa niż w środowiskach tradycyjnych. Jeżeli refaktoryzacje są zamienne lub zależne (mogą tworzyć łańcuch) w środowisku lokalnym to w budowanym, rozproszonym systemie DRE będą mogły być wprowadzane poprawnie w sposób automatyczny, bez ingerencji użytkownika.

Schemat na rysunku 2 pokazuje sposób synchronizacji z wykorzystaniem odtwarzania transformacji oraz identyfikacji komponentów zapewnianych przez repozytorium grafowe. Kolejność, w której projektanci w środowiskach lokalnych dokonują synchronizacji nie ma znaczenia dostaniemy dokładnie jeden poprawny graf wynikowy bez względu na szeregowanie odtwarzanych operacji – diagram jest przemienny. Kluczowym jest dalsze utrzymanie indeksacji po zastosowaniu obu produkcji na cały czas trwania projektu. Synchronizacji towarzyszy akcja semantyczna związana z żądaniem polegająca na przeprowadzeniu właściwej modyfikacji grafu oprogramowania (w oparciu o produkcje gramatyki) oraz zlecenie refaktoryzacji dla środowisk LRE operujących na współdzielonym fragmencie grafu. Na wyższym poziomie abstrakcji operacje wcześniej kolidujące okazują się operacjami zamiennymi.



Rysunek 2. Przykład procesu synchronizacji refaktoryzacji za pomocą transformacji na grafie oprogramowania i z wykorzystaniem identyfikacji komponentów.

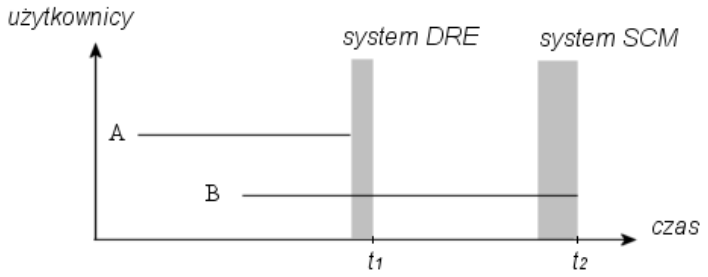
W systemie DRE wiele konfliktów pojawiających się podczas pracy w środowiskach tradycyjnych, opartych na plikach i systemie zarządzania wersjami, daje się wyeliminować w sposób automatyczny, w pozostałych przypadkach repozytorium grafowe umożliwi ich wykrycie oraz opóźnienie jednej z refaktoryzacji do czasu zakończenia poprzedniej. Wciąż jednak możliwe są sytuacje kiedy refaktoryzacja nie może być swobodnie przeprowadzona w środowisku DRE, gdyż wprowadzono wcześniej inną operację z nią niezgodną globalnie. W pełni automatyczne rozwiązanie bez stosowania specjalnych założeń nie jest możliwe, ponieważ prowadziłoby to do niepoprawnego programu lub nielogicznych i niezgodnych z zamierzeniami konstrukcji. W systemie DRE mówimy w takim przypadku o kolizji refaktoryzacji. Kolizje będą jednak ograniczone i będą dotyczyć wyłącznie sytuacji operacji niezgodnych (konflikt antysymetryczny). W takich przypadkach nie możemy w żaden

sposób zautomatyzować procesu, stad rola systemu ogranicza się do ostrzegania i wspomagania decyzji projektantów poprzez negocjacje lub dodatkowe heurystyki. W przypadkach kolizji, dla których nie da się wykorzystać prostych heurystyk, system wykluczy współbieżne wykonanie niepasujących operacji (jest to proste zadanie z punktu widzenia synchronizacji).

6 Zarządzanie łańcuchami refaktoryzacji

Z obserwacji praktycznych wynika, iż refaktoryzacje rzadko wprowadzane są pojedynczo. Kiedy programista zmienia strukturę kodu, ma najczęściej w zamyśle pewien cel (np. wydzielić cześć wspólna dwóch metod i przenieść ją do klasy nadrzędnej), a następnie wprowadzając serię przekształceń stara się go osiągnąć. Dlatego też w pracy posługujemy się koncepcję łańcuchów refaktoryzacji jako reprezentacji przygotowania złożonych operacji refaktoryzacji (będących sekwencjami operacji elementarnych) przez członków zespołów pracujących współbieżnie. Opracowano też dedykowane dla systemu DRE metody koordynacji przeprowadzania łańcuchów refaktoryzacji przez użytkowników z różnych środowisk lokalnych. Ponieważ przygotowanie kompletnego łańcucha refaktoryzacji jest zadaniem długotrwałym, mogą występować duże odstępy czasowe pomiędzy momentem rozpoczęcia lokalnej operacji, a momentem decyzji o zatwierdzeniu wprowadzonych zmian z repozytorium. System pozwala pracować projektantom w trybie opóźnionej synchronizacji tak by mogli swobodnie przeprowadzać łańcuchy refaktoryzacji oraz eksperymentować. Łańcuchy są traktowane przez system jako operacje niepodzielne i nierozerwalne dzięki specjalnemu wielopoziomowemu mechanizmowi kontroli opartemu na diagramach decyzyjnych DCD, który dostarcza możliwość wprowadzania łańcuchów oraz rejestracji operacji w każdym ze środowisk lokalnych.

Proces obsługi łańcuchów i eliminacji konfliktów w systemie DRE przebiega w sposób transparentny. Środowiska lokalne dostają informacje o zmianach dotyczących części grafu na którym pracuje programista, a które wprowadzono do repozytorium globalnego. Dzięki temu LRE są w stanie szybko weryfikować zgodność z serią zmian wprowadzanych lokalnie. Tak więc w sytuacji gdy dojdzie do konfliktu sprawdzenie potencjalnego wystąpienia konfliktów przygotowywanych lokalnie sekwencji refakotryzacji nastąpi natychmiast po modyfikacji środowiska globalnego (po zatwierdzeniu jednej z przygotowanych sekwencji). W systemie można liczyć na automatyczną eliminację konfliktów na zasadzie schematu opisanego wcześniej, a w pozostałych przypadkach, na propozycje zażegnania kolizji ze strony środowiska. Interakcja użytkowników jest konieczna tylko w przypadku poważnych kolizji. System DRE dopuszcza w takich przypadkach możliwość negocjacji – projektanci mogą wspólnie ustalić, które refaktoryzacje wybrać; protokół negocjacji nie jest jednak rozważany w pracy. Wybrane podejście gwarantuje również, iż podczas zatwierdzania lokalnych zmian do repozytorium centralnego nie będzie konfliktu.



Rysunek 3. Porównanie czasu rozpoznania potencjalnych konfliktów w modelu synchronizacji systemu DRE – podczas zatwierdzania, przez użytkownika A, w przeciwieństwie do typowego systemu SCM podczas synchronizacji użytkownika B.

Podstawowa zaleta wprowadzonego rozwiązania to sprawna detekcja niezgodnych zmian wykonanych przez innego członka zespołu na wspólnej części grafu oprogramowania. Kolizje wyłapywane są we wczesnym etapie – w momencie zatwierdzenia łańcucha refaktoryzacji przez pierwszego użytkownika (użytownik A na rysunku 3), a nie, jak w dotychczas funkcjonujących rozwiązaniach, gdy ostatni z współbieżnie pracujących członków zespołu (użytkownik B na rysunku 3) zakończy refaktoryzację. Tak więc zastosowanie mechanizmów zarządzania wprowadzaniem łańcuchów w systemie DRE minimalizuje koszt wprowadzania refaktoryzacji oraz zapewnia efektywność czasową i komfort pracy użytkownikom systemu.

7 Podsumowanie

W oparciu o repozytorium grafowe, bazujące na EDG grafach, modyfikowane pod kontrolą opracowanej gramatyki grafowych SPO i diagramów sterujących wywodem, jesteśmy w stanie zbudować wydajne środowisko pracy dla zespołów projektantów-programistów. Zaproponowane w pracy rozwiązanie będzie mogło zostać wykorzystane jako główny element nowoczesnego zintegrowanego środowiska rozproszonego do tworzenia oprogramowania.

Wprowadzony model opisu oprogramowania jest mocniejszy opisowo niż diagramy klas UML oraz posiada potencjał rozszerzenia dla dedykowanych zastosowań. Zyskujemy przede wszystkim możliwość analizy i wnioskowania, a także ujęcia dynamiki – dzięki reprezentacji operacji na wyższym poziomie abstrakcji oraz znajomości ich semantyki, można formalne opisać modyfikacje oraz ewolucję oprogramowania.

Zaprojektowane środowisko DRE jest w stanie wykluczyć konflikty na poziomie elementarnych refaktoryzacji i zapobiegać kolizjom operacji przy równoległym wprowadzaniu łańcuchów refaktoryzacji. Okazuje się całkowicie wystarczające do eliminacji konfliktów, których przyczyną była utrat identyfikacji komponentów – pozwala rozwiązywać takie problemy w sposób automatyczny.

Wprowadzone metody synchronizacji budowania lokalnych łańcuchów refaktoryzacji z jednej strony pozwalają zachować niezależność działań zespołu, a z drugiej **informują pozostałych projektantów o potencjalnych kolizjach natychmiast po zatwierdzeniu łańcucha refaktoryzacji przez jednego z nich**, a nie dopiero po zakończeniu pracy przez wszystkich uczestników współbieżnej refaktoryzacji. W sytuacjach spornych środowisko zapewniać będzie szybkie informowanie oraz możliwość prowadzenia negocjacji.

Publikacji doktoranta:

[1] Nowak A., Skrzypek M., Schaefer R.: Octopus – the platform for intelligent parallel computations, Artificial Intelligence Methods, 13-15 listopad 2002, Gliwice.

[2] Kotulski L., Nowak A.: Wykorzystanie gramatyk grafowych do kontroli migracji mobilnych agentów. Inżynieria wiedzy i systemy ekspertowe, Oficyna wydawnicza Politechniki Wrocławskiej, pp 319-325, 2003, Wrocław.

[3] Kotulski L., Nowak A.: Graph repository as a core of environment for distributed software restructuring and refactoring. Proceedings of the 24th IASTED International Multi-Conference Software Engineering, pp 356-360, 14-16 luty, 2006, Innsbruck, Austria.

[4] Kotulski L., Nowak A.: Formalizing Software Refactoring in the Distributed Environment by aedNLC Graph Grammar. Lecture Notes in Computer Science as SET06 IFIP Conference paper, 2006, Warszawa.

[5] Kotulski L., Nowak A.: Refactoring Merging Environment Supported by Graph Transformations. Proceedings of the IASTED International Conference Software Engineering, 12-14 luty, 2008, Innsbruck, Austria.