1. WSTĘP

1.1. Problematyka ponownego użycia

Podążając za innymi dziedzinami inżynierii, informatyka zmierza w stronę ponownego użycia [Szy2001]. Mimo, iż jest ona dziedziną stosunkowo młodą, zasoby programistyczne (w sensie napisanych programów, procedur i funkcji oraz klas) są olbrzymie. Tym bardziej więc atrakcyjna wydaje się idea ponownego użycia fragmentów kodu (ang. *code reusability*). Ponowne użycie nie tylko pozwala zaoszczędzić czas poświęcony na ponowne pisanie kodu. Pozwala też – a może nawet przede wszystkim – poprawić jakość wytwarzanego oprogramowania, gdyż przedmiotem powtórnego użycia są zwykle fragmenty dobrze przetestowane i sprawdzone w praktyce[Ave2001, Way1994]. Co więcej, mimo iż sam proces ponownego użycia pozornie nie wnosi wiele do projektu architektury (nie rozpatrujemy tu ponownego użycia fragmentów architektury czy projektu), to ukierunkowanie procesu wytwarzana oprogramowania na pozyskanie gotowych (zwykle funkcjonalnie niezależnych) modułów, prowadzi zwykle do zmniejszenia liczby powiązań międzymodułowych, a tym samym do łatwiejszego w utrzymaniu projektu. Za ponownym użyciem świadczą jednak przede wszystkim liczby: Toshiba na skutek wprowadzenia ponownego użycia zanotowała 20 – 30% spadek liczby błędów w tworzonym oprogramowaniu, Hewlett-Packard oszacował zmniejszenie liczby błędów nawet o 76%(!) przy jednoczesnym wzroście produkcyjności o około 50%.

Ponownemu użyciu mogą podlegać jednostki programowe różnych rozmiarów. Najmniejszą jednostką programową, która może zostać ponownie użyta jest funkcja[1, 2]. Zwykle jednak pojedyncza funkcja nie jest wystarczająco niezależna od pozostałych fragmentów kodu, aby sama mogła stanowić przedmiot ponownego użycia. Dlatego też, często najmniejszymi rozpatrywanymi w kontekście ponownego użycia jednostkami są moduły, klasy czy wreszcie komponenty. Komponenty są pod tym względem szczególnie interesujące, gdyż już

[1] Procedury, o ile nie zaznaczono inaczej, będą w tej pracy rozpatrywane jako szczególne rodzaje funkcji których rezultat jest stałą pewnego określonego typu (pustego) oznaczanego symbolicznie *void*.

[2] Oczywiście można łatwo wyobrazić sobie pozyskanie z innego programu fragmentu funkcji, jednak takie przypadki ze względu na swoją specyfikę traktuję jako rodzaj pozyskania wiedzy, stąd nie będą one w tej pracy rozpatrywane.

z definicji stanowią jednostki instalowane niezależnie[3], mogą więc być poddane ponownemu użyciu szczególnie łatwo[Pal2003, Sma1998, Szy2001].

Niezależnie od rodzajów jednostek programowych, nie ulega wątpliwości iż dla prawidłowego wdrożenia ponownego użycia w procesie wytwarzania oprogramowania niezbędny jest mechanizm wyszukiwania gotowych rozwiązań. Aby ułatwić identyfikację jednostek mających zastosowanie w rozwiązywanym problemie stosuje się więc repozytoria jednostek programowych. Repozytoria te pozwalają gromadzić jednostki wytwarzane w ramach organizacji. Posiadają one też różnego rodzaju funkcje wyszukiwania, umożliwiające projektantom i programistom przeszukiwanie repozytorium. Jednostki programowe w repozytorium często przechowywane są wraz z dodatkowym opisem, ułatwiającym wyszukiwanie zainteresowanym użytkownikom. Opis taki ma często postać słowną, może też być klasyfikacją opartą na taksonomiach czy nawet na sieciach semantycznych. Odpowiedni opis słowny i/lub klasyfikacja muszą zostać wprowadzone wraz z jednostką przez jej twórcę, lub osobę zajmującą się zarządzaniem repozytorium. W chwili obecnej dostępnych jest wiele rozwiązań repozytoriów umożliwiających katalogowanie wytwarzanych w ramach przedsiębiorstwa jednostek programowych w celu ułatwienia ich ponownego użycia [Ezr2002]. Dostępne są również globalne repozytoria w sieci Internet, zawierające szereg projektów gotowych do wykorzystania w całości lub w części, np. [Src2004]

Kluczowym mechanizmem repozytorium jest wyszukiwanie. Jego efektywność ma bowiem duży wpływ na nakłady poniesione na ponowne użycie oraz na nastawienie programistów do samej idei procesu ponownego użycia. Wyszukiwanie polega na utworzeniu zapytania przy użyciu *języka zapytań do repozytorium*, po czym wykonaniu tego zapytania w celu otrzymania zbioru jednostek programowych. Zbiór ten następnie musi zostać przeglądnięty ręcznie w celu znalezienia odpowiedniej dla danych zastosowań jednostki programowej.

[3] Jest to niewątpliwie wyróżnik komponentów, choć nie jedyny i sam w sobie nie stanowi podstawy do wprowadzenia pojęcia „programowanie komponentowe". Ze względu jednak na ukierunkowanie tej pracy na ponowne wykorzystanie kodu, często – szczególnie w odniesieniu do programowania obiektowego - komponent będzie w uproszczeniu traktowany jako instalowany niezależnie zbiór obiektów.

Język zapytań[4] określa ograniczenia, którym podlega sposób wyrażania użytkownika o wymaganiach dotyczących poszukiwanego komponentu. Niestety, istniejące obecnie języki zapytań do repozytorium charakteryzują się niewielką elastycznością. Oznacza to, że poszukiwanie musi odbywać się w sposób przewidziany przez opisującego jednostkę w repozytorium. Twórca taksonomii musi przewidzieć odpowiednie dla użytkowników repozytorium kategorie, konstruktor sieci semantycznej musi odpowiednio zaprojektować sieć i dostarczyć odpowiednich relacji[Sow2004]. Wreszcie nawet autor zwykłego opisu słownego zmuszony jest opisać jednostkę tak, aby charakteryzowała wszystkie możliwe aspekty jej funkcjonalności. Niestety, nie zawsze jest to możliwe. Dlatego, jeżeli w istniejących dziś repozytoriach spróbujemy znaleźć np. klasę reprezentującą stos, znajdziemy ją niemal natychmiast. Jeżeli poszukamy stosu z nieblokującym pobieraniem danych – znajdziemy go po chwili. Jeżeli jednak wolelibyśmy, żeby procedura pobrania zwracała wartość pustą zamiast rzucać wyjątek, jeżeli stos jest pusty – prawie na pewno jesteśmy skazani na ręczne przeglądanie zbioru odnalezionych stosów.

Kolejnym problemem jest kwestia nakładów ponoszonych na ponowne użycie. Przez nakłady rozumie się tu nie tylko inwestycję w utworzenie repozytorium, dostarczenie środków niezbędnych dla procesu ponownego użycia czy wreszcie szkoleń dla załogi. Duży (jeśli nie przeważający) udział w nakładach ma czas, jaki zmuszeni są poświęcić twórcy jednostek programowych na ich opisywanie, czy wreszcie czas, jaki poszukujący gotowych rozwiązań użytkownicy repozytorium tracą konstruując zapytania, które w zależności od zastosowanego języka mogą być mniej lub bardziej skomplikowane. Jeżeli oczekiwana jednostka programowa nie zostanie znaleziona, nakłady te są bezpowrotnie stracone.

Ciekawym paradoksem jest, że wytwarzając oprogramowanie metodą *test-driven development*, w momencie stwierdzenia, iż danej jednostki programowej nie uda znaleźć się w repozytorium, programista zmuszony jest wyspecyfikować zachowanie jednostki ponownie – tworząc testy jednostkowe. Specyfikacja taka jest przy tym dokładniejsza i bardziej naturalna dla programisty od specyfikacji np. za pomocą zbioru słów kluczowych. Naturalna zatem wydaje się próba połączenia tych podejść i oparcia języka zapytań do repozytorium na językach skryptów testowych takich jak jUnit.

[4] Język zapytań do repozytorium często utożsamiany jest tu z samą metodą przeszukiwania, gdyż to ona właśnie decyduje zwykle o konstrukcji samego języka.

1.2. Cel i zakres pracy

Celem niniejszej pracy jest opracowanie nowego języka zapytań do repozytoriów, a co za tym idzie nowej metody poszukiwania jednostek programowych w repozytoriach, która pozwoli uwolnić użytkownika od istniejących dziś ograniczeń. Ograniczenia te oczywiście nie mogą zostać całkowicie zredukowane, jednak metoda poszukiwań powinna w maksymalnym stopniu odcinać proces przeszukiwania od intencji twórcy jednostki programowej i repozytorium. Tworzony język zapytań do repozytorium powinien też umożliwić wyrażanie możliwie różnorodnych wymagań od poszukiwanych jednostek programowych. Ponieważ trudno, w typowych przypadkach, zaprzeczyć użyteczności istniejących metod dostępu do repozytorium, zakłada się ponadto, że tworzona metoda umożliwi połączenie wyszukiwania przy pomocy nowego języka zapytań oraz istniejących dotychczas.

Zakłada się, że tworzony język zapytań powinien umożliwić redukcję nakładów ponoszonych na ponowne użycie. Dotyczy to głownie nakładów ponoszonych na etapie konstrukcji zapytań do repozytorium oraz przeszukiwania zbioru wyników będącego rezultatem wyszukiwania. Częściowo cel ten może zostać zrealizowany przez wdrożenie wymienionych wcześniej postulatów. Dodatkowa redukcja kosztów może zostać osiągnięta poprzez integrację procesu z procesem tworzenia testów jednostkowych metodą *test-driven development.* Integracja taka - polegająca na oparciu języka zapytań do repozytoriów na tworzonych testach jednostkowych – pozwoli na uniknięcie zbędnych nakładów na konstrukcję zapytań w procesie wyszukiwania jednostek programowych.

Na początku niniejszej pracy dokonany jest przegląd i ocena efektywności istniejących rozwiązań pozwalających tworzyć a następnie przeszukiwać repozytoria jednostek programowych. Rozpatrywane są m.in. podejścia oparte na wyszukiwaniu po nazwie, opisach, słowach kluczowych. Omawiane są również bardziej zaawansowane mechanizmy pozwalające wyszukiwać jednostki programowe w oparciu o wieloaspektowe taksonomie (zbiór taksonomii opisujących ortogonalne aspekty klasyfikowanej jednostki) oraz podejścia bazujące na wykorzystaniu sieci semantycznych opisujących semantykę i kontekst wywołania jednostki. Omawiane są również sposoby i korzyści płynące z łączenia poszczególnych metod. Wszystkie metody oceniane są zarówno na podstawie danych dostępnych

w literaturze, rozważań teoretycznych, jak i eksperymentów przeprowadzonych w celu liczbowego uzasadnienia wniosków.

W dalszej części pracy proponowane są nowe metody wyszukiwania jednostek programowych stanowiące uzupełnienie dla omawianych wcześniej. Przedstawiona jest m.in. metoda oparta o kanoniczny opis jednostek programowych w wybranych punktach charakterystycznych oraz metoda niekanoniczna bazująca na opisie pożądanych cech wyszukiwanej jednostki w postaci scenariuszach użycia. Rozważania na temat proponowanych podejść prowadzone są dla różnych rodzajów jednostek programowych, rozpoczynając od typowo teoretycznego przykładu funkcji 1-argumentowych, poprzez pojedyncze funkcje z wieloma argumentami, a na złożonych zbiorach klas i komponentach skończywszy. W pracy omówiono również kwestię integracji przedstawianych metod wyszukiwania z procesem wytwarzania oprogramowania.

2. METODY PONOWNEGO UŻYCIA

Ponowne użycie oznacza „*systematyczną praktykę wytwarzania oprogramowania ze zbioru niezależnych elementów, taką, że podobieństwa w wymaganiach i/lub architekturze pomiędzy aplikacjami mogą być wykryte w celu osiągnięcia znaczących korzyści w produktywności, jakości i wydajności biznesowej"* [Ezr2002]. Oczywiście może się zdarzyć, że mamy odczynienia nie z systematyczną praktyką, a z nieustandaryzowanym procesem, w którym ponowne użycie realizowane jest w sposób chaotyczny, a odpowiedzialność za wybór metod czy przedmiotów ponownego użycia pozostawiona jest pojedynczym programistom. Trudno jednak spodziewać się w takim przypadku powtarzalnych rezultatów, czy to w formie końcowego produktu i jego atrybutów, czy w formie efektywności ponownego użycia, które w różnych projektach może zarówno do znaczących strat jak i pokaźnych korzyści. Dlatego ponowne użycie należy rozpatrywać wyłącznie w kategoriach poprawnie wdrożonego w organizacji procesu, na który składa się [Ezr2002]:

- Zrozumienie, w jaki sposób ponowne użycia oddziałuje na cele biznesowe organizacji;
- Zdefiniowanie strategii technicznych i strategii zarządzania do osiągnięcia maksymalnej korzyści z ponownego użycia;
- Zintegrowanie ponownego użycia z całym procesem wytwarzania oprogramowania i z procesem cyklicznej poprawy jakości;
- Upewnienie się, że cały personel ma niezbędne kompetencje i motywację;
- Ustalenie odpowiedniego wsparcia organizacyjnego, technicznego i pieniężnego;
- Użycie odpowiednich miar w celu kontroli wydajności ponownego użycia.

Wdrożenie takiego procesu nie jest ani łatwe, ani tanie. Należy jednak traktować to jako doskonałą inwestycję. Jednak aby osiągnąć znaczące korzyści z ponownego użycia należy identyfikować potencjalne elementy systemu mogące podlegać ponownemu wykorzystaniu w jak najwcześniejszych fazach cyklu życia oprogramowania. Oprócz oczywistych zalet związanych ze zorientowaniem całego projektu pod kątem pozyskiwanych istniejących komponentów, wczesna identyfikacja możliwości ponownego użycia np. na poziomie projektu zwykle oznacza, że również w dalszych fazach proces ponownego użycia przyniesie znaczące korzyści i uda się go przeprowadzić w miarę łatwo.

Przedmiotem ponownego użycia są tzw. artefakty (Ezran używa angielskiego określenia *assets*). Mogą one być nie tylko fragmentami kodu, ale także dowolnymi artefaktami powstałymi podczas wytwarzania oprogramowania takimi jak elementy projektu (tzw. wzorce projektowe [Gam1994]), architektury, czy wymaganiami [Som1997] . W tym rozdziale zajmiemy się artefaktami niezależnie od ich charakteru. W rozdziałach następnych wprowadzono pojęcie komponentu (przy zastrzeżeniach poczynionych we wstępie) oraz jednostki programowej, oznaczające podzespół będący wykonywalnym fragmentem kodu.

Niestety, z wdrożeniem procesu zorientowanego na ponowne użycie wiążą się także czynniki ryzyka. Aby ocenić efektywność wprowadzonego procesu muszą być one zidentyfikowane a następnie monitorowane. Najważniejsze z nich to [Ezr2002]:

- Artefakty nie są wytwarzane.
  Wytwarzanie artefaktów wymaga dodatkowego nakładu pracy. Przystosowanie artefaktów do ponownego użycia może napotykać na opór programistów a nawet kadry zarządzającej, szczególnie w obliczu nadchodzących terminów oddania produktu. Należy przewidzieć dodatkowy czas na czynności związane z produkcją nadających się do ponownego użycia artefaktów [Str1995, Ezr2002] i odróżnić czynności ich przystosowywania od zwykłych czynności wytwarzania oprogramowania. Dodatkowo pozwoli to na zachowanie lepszej kontroli nad procesem ponownego użycia.

- Artefakty są produkowane, ale nie są znajdowane.
  Nawet jeżeli w organizacji istnieje repozytorium artefaktów (patrz następny rozdział) i artefakty są odpowiednio klasyfikowane i zarządzane, istnieje możliwość, że właściwe artefakty wciąż nie będą znajdowane. Należy przewidzieć odpowiednie szkolenia instruujące personel jak korzystać z repozytorium. Nie można również używać metryk oceny produktywności programistów takich jak liczba linii kodu/ miesiąc, gdyż zniechęcają one programistów do koncepcji ponownego użycia (patrząc przez pryzmat takich metryk im więcej kodu potrafią oni odzyskać z innych projektów, tym mniejsza jest ich produktywność)

- Zbyt dużo czasu zawiera zrozumienie i ocena artefaktu.
  Wraz z podzespołami powinna być przechowywana nie tylko informacja dla mechanizmów katalogujących (taka jak np. klasyfikacja za pomocą odpowiedniej taksonomii), ale także szczegółowy opis cech funkcjonalnych i niefunkcjonalnych danego artefaktu. Zarezerwowanie odpowiedniej ilości czasu na czynności związane z dokumentacją, oraz wprowadzenie odpowiednich metryk (np. % skomentowanych metod) z pewnością podziała motywująco na wprowadzających artefakty do repozytorium.

- Artefakty nie nadają się do ponownego użycia, gdyż ich jakość jest zbyt niska.
  Jakość artefaktów musi być sprawdzana szczególnie starannie, gdyż błędy w nich mogą propagować się na wiele projektów. Należy również zapewnić odpowiedni mechanizm zgłaszania błędów ułatwiający sprzężenie zwrotne od użytkowników artefaktu do autora.

- Artefakty nie nadają się do ponownego użycia, gdyż nie spełniają wymagań funkcjonalnych.
  Jeżeli zbyt mało czasu poświęca się na wcześniejsza analizę wymagań, istnieje groźba, że artefakty będą tworzone w celu sprostania bieżącym potrzebom, a nie z myślą o przyszłym zastosowaniu. Sprzyja temu zwłaszcza stosowanie ponownego użycia a posteriori.

- Artefakty nie nadają się do ponownego użycia, gdyż nie spełniają wymagań technicznych.
  W ramach organizacji istotne jest stosowanie o ile tylko to możliwe wspólnych standardów, protokołów itp. Nie tylko ułatwi to poziome ponowne użycie (ponowne użycie pomiędzy różnymi dziedzinami aplikacji), ale także wymianę wiedzy między zespołami.

- Przedsiębiorstwo może nie być gotowe do wdrożenia ponownego użycia lub proces ten nie jest odpowiedni dla rodzaju prowadzonej działalności.
  W celu wdrożenia ponownego użycia należy przydzielić odpowiednią liczbę zasobów. Jak już wspomniano wcześniej, ponowne użycie jest swojego rodzaju inwestycją. Jeżeli jednak inwestycja ta prowadzona jest bez odpowiednich środków szybko może okazać się, że wdrożenie procesu na tym etapie przyniesie więcej szkody niż pożytku. Wdrożenie należy także poprzedzić szczegółową analizą celowości i zakresu ewentualnego wdrożenia procesu zorientowanego na ponowne użycie.

- Przedsiębiorstwo nie jest w stanie określić, czy ponowne użycie rozwija się we właściwym kierunku i czy przynosi korzyści.
  Jest to sygnał, że proces prowadzony jest niepoprawnie. Należy ustanowić procedury monitorowania efektywności procesu ponownego użycia i zidentyfikować ewentualne obszary jego poprawy.

- Ponowne użycie wymaga nowych procesów i zastosowania zaawansowanych technik i metod.
  Zespoły powinny zostać wyposażone w odpowiednie narzędzia wspomagające proces ponownego użycia. Należy przewidzieć czas i zasoby na odpowiednie szkolenia.

Rozróżniamy dwa rodzaje ponownego użycia podzespołów: pionowe i poziome. Przez *pionowe ponowne użycie* (ang. *vertical reuse*) rozumie się ponowne użycie podzespołów w ramach pojedynczej dziedziny aplikacji. Pionowe ponowne użycie zwykle dotyczy fragmentów aplikacji ściśle związanych z jej dziedziną i najczęściej wykorzystywane jest przez organizacje specjalizujące się w wytwarzaniu oprogramowania dla specyficznych

segmentów rynku. Przedmioty takiego ponownego użycia często stanowią cenną wiedzę danej organizacji i rzadko udostępniane są innym. Przez przeciwstawienie, *poziome ponowne użycie* (ang. *horizontal reuse*) oznacza ponowne użycie przekraczające granicę pojedynczej dziedziny aplikacji. Najczęściej (choć oczywiście nie zawsze) tego rodzaju ponowne użycie dotyczy artefaktów uniwersalnych, takich jak wzorce projektowe, struktury danych, wzorce komunikacji itp. Ze względu na swoją nieostrość, klasyfikację tą należy traktować wyłącznie jako poglądową.

Proces ponownego użycia może przebiegać w trzech różnych trybach. Najbardziej korzystną sytuacją jest ta, w której proces taki zintegrowany jest z procesem wytwarzania oprogramowania. Artefakty wytwarzane są na potrzeby tworzonych aplikacji, jednak z myślą o ich użyciu w następnych projektach. Podzespoły mogą być również tworzone z wyprzedzeniem, niezależnie od tworzonych aplikacji. Osobny zespół (zespoły) tworzą artefakty, które potencjalnie mogą zostać wykorzystane w przyszłych aplikacjach. Istnieje jednak niebezpieczeństwo, że tworzony artefakt rozminie się z zapotrzebowaniem i poniesione nakłady zostaną zmarnowane. Dlatego ten styl pracy nadaje się głownie dla przedsiębiorstw produkujących duże, specjalizowane komponenty dla innych odbiorców. Najmniej efektywnym, jednak wymagającym najmniejszych nakładów jest styl pracy, w którym potencjalnie nadające się do ponownego użycia artefakty identyfikowane są po stworzeniu aplikacji. Niestety efektywność takiego podejścia jest niewielka, gdyż artefakty, które nie były tworzone z myślą o ponownym użyciu zwykle są zbyt zorientowane na potrzeby danej aplikacji. Może się również okazać, ze związki pomiędzy kandydatami na reużywalne artefakty są zbyt silne, a zatem wydzielenie ich w celu powtórnego użycia może wymagać dużego nakładu pracy na refaktoryzację. Dlatego ten styl pracy można zalecić jedynie w krótkim okresie przejściowym.

Ponowne użycie nie zawsze jest możliwe w sposób bezpośredni. Ten idealny przypadek, w którym artefakt nie wymaga modyfikacji przed zastosowaniem w nowym projekcie nazywany jest ponownym użyciem typu czarnej skrzynki (ang. *black-box reuse)*. Często przed użyciem artefaktu trzeba zmodyfikować jego wewnętrzna strukturę. Procedura ta nazywana jest ponownym użyciem typu białej skrzynki (ang. *white-box reuse)*. Konieczność takich zmian może sygnalizować, że artefakt nie został zaprojektowany na odpowiednim poziomie ogólności i/lub możliwości jego konfiguracji nie są wystarczające. Dlatego w takim przypadku warto zastanowić się nad gruntowną przebudową artefaktu. Jest jeszcze przypadek pośredni nazywany ponownym użyciem typu szarej skrzynki (ang. *gray-*

box reuse). Polega on na użyciu artefaktu poprzedzonym aktualizacją jego parametrów konfiguracyjnych, jednak bez dokonywania zmian w jego wewnętrznej strukturze.

3. REPOZYTORIUM JEDNOSTEK PROGRAMOWYCH

Jednostką programową nazywać będziemy dowolny, dający się wyodrębnić i samodzielnie używać fragment kodu. Od jednostek programowych nie wymagamy pełnej możliwości niezależnej instalacji ani samodzielnej kompilacji. Rozważanymi typami jednostek programowych będą funkcje, procedury, metody, klasy, moduły, pakiety i komponenty. W dalszych rozdziałach używane będzie także pojęcie *komponentu*, jednak nie w klasycznym jego rozumieniu. Ze względu na spojrzenie z perspektywy ponownego użycia komponentem nazywać będziemy jednostkę programową lub zbiór takich jednostek mogący podlegać samodzielnej instalacji. Podejście tego rodzaju spotkać można również w pracach Ezrana i Ye [Ezr2002, Ye2001]

Repozytorium jest miejscem przechowywania jednostek programowych przeznaczonych do ewentualnego ponownego użycia. Jak zobaczymy dalej, jednostki te przechowywane są wraz z opisami, pozwalającymi na ich efektywne wyszukiwanie i przeglądanie. Należy zauważyć, że w większości przypadków repozytorium jednostek programowych stanowi niezbędny element procesu wytwarzania oprogramowania zorientowanego na ponowne użycie [Ezr2002]. Oczywiście w przypadku niewielkich zespołów nie posiadających dużej bazy gotowych artefaktów można pozwolić sobie na rezygnację z repozytorium. Decyzja taka powinna być jednak starannie przemyślana. Repozytorium oprócz swojej oczywistej funkcji przechowywania artefaktów stanowi swojego rodzaju bazę wiedzy organizacji, dzięki czemu ułatwia wymianę wiedzy pomiędzy pojedynczymi programistami, zespołami a także uniezależnia proces wytwarzania oprogramowania od “ekspertów”, którzy posiadając na wyłączną własność wiedzę o rozwiązaniach z danego zakresu mogą swoim niespodziewanym odejściem z pracy zakłócić przebieg procesu wytwarzania oprogramowania.

Repozytorium jednostek programowych powinno umożliwić:

- możliwie łatwe wprowadzanie nowych jednostek wraz z ich opisami
- przechowywanie wprowadzonych jednostek
- przeszukiwanie i przeglądanie zgromadzonych jednostek

Repozytorium z definicji pozwala gromadzić informacje o przechowywanych komponentach, w sposób pozwalający ocenić stopień dopasowania opisu danego komponentu do zapytania wydanego przez użytkownika. Przykładem takiego rozwiązania (dla usług WWW) jest zcentralizowany rejestr UDDI [Gra2003]. Wymaga to przechowywania oprócz samych jednostek programowych, także pewnej dodatkowej informacji, stanowiącej opis owych jednostek. Wprowadzane do rejestru informacje na temat jednostki programowej mogą pochodzić z wielu różnych źródeł. Można je wstępnie podzielić na pozyskiwane automatycznie na podstawie kodu samej jednostki (jak np. niskopoziomowy opis interfejsu) oraz na takie, które muszą być dodatkowo wprowadzone wyłącznie w celu wspomagania procesu wyszukiwania. Podział ten, choć na pierwszy rzut oka poprawny i intuicyjny, z czasem okazuje się dość nieostry i w wielu przypadkach może zostać złamany przez niektóre szczególne formy opisu. Np. omawiany w następnych rozdziałach opis jednostek oparty na klasyfikacji za pomocą taksonomii na pierwszy rzut oka wydają się należeć do pierwszej kategorii. Łatwo jednak można wyobrazić sobie klasyfikację dokonywaną automatycznie na podstawie cech interfejsu opisywanej jednostki. Ważniejsze sposoby opisu jednostek programowych przedstawiono na rys. 3.1. Opis na każdym z poziomów przedstawionej piramidy jest zwykle uzupełnieniem opisów leżących na niższych poziomach. Szerokość piramidy proporcjonalna jest do rozpowszechnienia danej metody.

Rys. 3.1 Sposoby opisu jednostek programowych

W przypadku repozytoriów kluczowe jest osiągnięcie kompromisu między prostotą a szczegółowością opisu. W najprostszym przypadku można wyobrazić sobie rejestry przechowujące wyłącznie niskopoziomowe opisy interfejsów. Informacja ta nie będzie jednak wystarczać do wygodnego wyszukiwania jednostek programowych. W przypadku bardziej

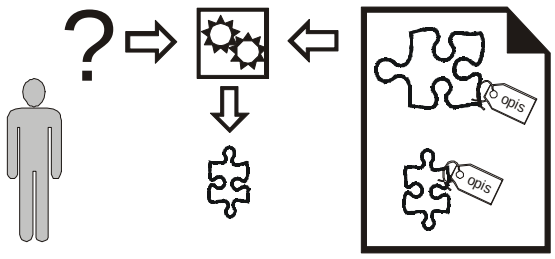
złożonym oczekujemy dodatkowej informacji dotyczącej semantyki komponentu, kontekstu jego użycia oraz jakości oferowanych usług. Niestety informacja ta musi być zwykle wprowadzana ręcznie, może się więc okazać, że jeżeli liczba danych, które trzeba wprowadzić umieszczając jednostkę w repozytorium będzie zbyt duża, programiści będą wykonywać tę czynność niechętnie.

Aby móc efektywnie wyszukiwać komponenty w celu ich powtórnego użycia musimy dysponować czymś więcej niż zwykłym zbiorem jednostek programowych. Użytkownik poszukujący komponentu musi mieć do dyspozycji pewien język zapytań, który pozwoli mu opisać pożądane cechy szukanego komponentu w sposób jak najbardziej precyzyjny. Z drugiej strony od języka zapytań oczekujemy prostoty. Problem polega na tym, że wymagania te są sprzeczne. Moglibyśmy ograniczyć się do prostego języka zapytań, który niestety nie będzie w stanie precyzyjnie oddać wymagań użytkownika, więc zbiór znalezionych komponentów będzie dla niego zupełnie bezużyteczny. Z kolei stworzenie złożonego języka z całym bogactwem konstrukcji opisującego wszelkie możliwe zachowania komponentu spowoduje, że wydawanie takich zapytań będzie trudne i może być co najmniej zniechęcające, a w skrajnych przypadkach wysiłek poświęcony na konstrukcję zapytania będzie dorównywał lub nawet przewyższał wysiłek związany z konstrukcją komponentu. Dlatego kluczem do stosowalności danego języka zapytań wydaje się być znalezienie odpowiedniego kompromisu między tymi dwiema skrajnościami. Od naprawdę dobrych rozwiązań oczekujemy, że pozwolą użytkownikowi poruszać się płynnie pomiędzy tymi punktami, umożliwiając mu dostosowanie szczegółowości zapytań do jego indywidualnych potrzeb.

Ye i Fischer [Ye2001] wyróżnili dwie zasadnicze fazy pracy z repozytorium: wyszukiwanie (ang. *searching*) i przeglądanie (ang. *browsing).*

Proces wyszukiwania polega na dopasowywaniu opisów przechowywanych w rejestrach jednostek programowych do zapytania wydanego przez użytkownika. Od procesu tego wymagamy dwóch podstawowych atrybutów: selektywności (powinny zostać odnalezione jedynie jednostki programowe ściśle pasujące do wydanego zapytania) i akceptowalnej prędkości działania. W ogólnym przypadku oczekujemy, iż rezultatem wyszukiwania będzie częściowo uporządkowany ranking, posortowany według stopnia dopasowania. Należy zauważyć, że tworzenie rankingów jest też możliwe na podstawie różnych metryk stosowanych do komponentów w repozytorium (np. w oparciu o wzajemne relacje użycia

pomiędzy komponentami [Ino2003]). Rankingi tego typu z pewnością mogą wspomagać proces wyszukiwania, jednak ze względu na niezależność od zapytań użytkownika nie stanowią jego integralnej części. Schematycznie proces wyszukiwania przedstawiony jest na rysunku 3.2.



Rys. 3.2. Proces przeszukiwania repozytorium jednostek programowych

Drugą z wyróżnionych faz stanowi faza przeglądania. Polega ona na „ręcznym" przeglądaniu przez użytkownika zbioru jednostek programowych będącego rezultatem procesu wyszukiwania. Proces ten w przypadku większych zbiorów jednostek programowych może być dość czasochłonny i nużący dla programisty, zniechęcając go tym samym do poszukiwania jednostek w celu ponownego użycia. Dlatego istotne jest położenie nacisku na selektywność procesu wyszukiwania, co pozwoli zminimalizować zbiór przedstawiany programiście jako rezultat.

Z wyszukiwaniem i przeglądaniem łączy się jednak pewien problem. Mianowicie, aby w ogóle rozpocząć wyszukiwanie programista musi podejrzewać istnienie jednostki, mogącej pomóc mu w tworzeniu danego fragmentu kodu lub nawet go zastąpić. Niestety w ogólności może to stanowić problem, szczególnie dla programistów początkujących nie posiadających jeszcze odpowiedniego doświadczenia. Pewne rozwiązanie, polegające na konstrukcji środowiska automatycznie przedstawiającego jednostki programowe mające potencjalne zastosowanie w tworzonym właśnie fragmencie kodu (bez inicjacji tej akcji przez programistę) przedstawili Ye i Fischer [Ye2001]. Wydaje się jednak, że przy ograniczonej w sposób oczywisty inteligencji takiego systemu dużo lepsze skutki może przynieść wdrożenie w organizacji odpowiedniego procesu zorientowanego na ponowne użycie [Ezr2002]. Podejście wydaje się być jednak obiecujące, nie tyle ze względu na osiągane w chwili obecnej rezultaty, co dzięki odmiennemu spojrzeniu, które zwalnia programistę z obowiązku systematycznego przeszukiwania repozytorium i pozwala nieco zredukować różnicę wynikająca ze zróżnicowania doświadczenia programistów. Dlatego w dalszej części

pracy pojawią się odniesienia do tego sposobu interakcji z repozytorium, który traktowany będzie jako alternatywny dla procesu wyszukiwania.

Repozytorium jednostek programowych może być albo zcentralizowane albo rozproszone. Zcentralizowane repozytoria mogą być udostępnianie w sieci Internet, mogą również - i jest to przypadek częstszy - stanowić centralny zbiór reużywalnych jednostek programowych w danej organizacji zajmującej się wytwarzaniem oprogramowania. Repozytoria zcentralizowane są z natury prostsze – architektonicznie i konstrukcyjnie – od repozytoriów rozproszonych. Niestety, ilość przechowywanej informacji a przede wszystkim konieczność integracji rozwiązań (np. łączących się firm) może wymagać decentralizacji wolumenu danych oraz wypracowania protokołów komunikacji pomiędzy częściami systemu.

Repozytoria rozproszone możemy podzielić na jednorodne (zbudowane z mniejszych repozytoriów zaprojektowanych do współpracy w systemie rozproszonym) i heterogeniczne. Szczególnie te ostatnie stawiają przed projektantem wiele wyzwań wynikających z konieczności dostosowania do siebie systemów o być może zupełnie różnym charakterze, konstrukcji i filozofii. Repozytoria takie wymagają języka zapytań, w którym zapytanie mogłoby zostać rozproszone i przekształcone na odpowiednie dla danych podsystemów. Musi także radzić sobie z łączeniem napływających rezultatów.

4. PRZESZUKIWANIE REPOZYTORIÓW JEDNOSTEK PROGRAMOWYCH – STAN SZTUKI

Jak już wspomniano w rozdziałach poprzednich, jednym z rodzajów interakcji z repozytorium jednostek programowych jest wyszukiwanie. Polega ono na łączeniu zapytania użytkownika opisującego wymagania dotyczące poszukiwanego komponentu z zawartością repozytorium, w celu otrzymania zbioru jednostek programowych zgodnych z zapytaniem lub ewentualnie rankingu tychże jednostek posortowanego ze względu na stopień dopasowania do zapytania użytkownika. W celu efektywnego przeprowadzenia procesu wyszukiwania, przechowywane jednostki programowe muszą zostać wcześniej opisane, gdyż w ogólności niewiele informacji o jednostce udaje się uzyskać wyłącznie na podstawie kodu źródłowego (który również nie zawsze jest dostępny). W dalszej części rozdziału dokonany zostanie przegląd różnych istniejących metod opisu jednostek programowych. Dla każdego rozwiązania wskazane są jego wady, zalety a także potencjalne możliwości łączenia z innymi formami opisu.

Klasyczne repozytoria jednostek programowych zorientowane na usprawnienie procesu ponownego użycia kodu zwykle zawierają co najmniej kilka rodzajów opisów jednostek programowych [Ezr2002, Ino2003, Naw2004]. Najczęściej spotykane to: unikatowa nazwa komponentu, opis, słowa kluczowe skojarzone z komponentem. Pojawiają się też bardziej zaawansowane formy opisu takie jak taksonomie, tezaurus, czy wreszcie sieci semantyczne. Poniżej przyjrzymy się każdej z wymienionych form opisu.

4.1. Wyszukiwanie w oparciu o nazwę jednostki

Nazwa jednostki programowej pełni (zwykle) podwójną rolę. Z jednej strony stanowi największy skrót funkcjonalności komponentu. Z drugiej strony, w wielu systemach stanowi jednoznaczny identyfikator danego komponentu. Nawet jeżeli ze względów wydajnościowych w konstrukcji danego repozytorium wprowadzono identyfikatory liczbowe lub podobne, zwykle i tak od nazwy oczekuje się unikatowości. Ten rodzaj identyfikacji komponentów jest bowiem dużo bardziej przyjazny dla użytkownika, stąd może on oczekiwać, że interfejs repozytorium pozwoli mu się odwoływać do przechowywanych jednostek programowych właśnie za pomocą nazwy. Niestety, w przypadku dużych repozytoriów oraz wielu programistów pracujących nad jednostkami, zachowanie unikatowego nazewnictwa jednostek wydaje się nierealizowalne. Wprowadza się zatem pojęcie przestrzeni nazw (analogicznie do

rozwiązań stosowanych w językach programowania takich, jak Java[Eck2002], C++ [Str1995], czy nawet w językach dokumentów jak XML [Arc2002] i językach warstwy prezentacji [Fre2004, Pal2004]). Polega ono na prefiksowaniu nazwy komponentu unikatowym identyfikatorem przestrzeni nazw. Oczekuje się przy tym unikatowości nazwy w ramach pojedynczej przestrzeni nazw. W przypadku repozytoriów zawierających wyłącznie jednostki napisane w określonym języku programowania (posiadającym mechanizm przestrzeni nazw) można bezpośrednio posłużyć się mechanizmem przestrzeni nazw oferowanym przez ten język. Jest to jedynie możliwe, w przypadku kiedy przestrzeń nazw przydzielana jest przez twórcę danej jednostki (jak np. w przypadku pakietów języka Java). W przypadku, kiedy dany język programowania nie oferuje takich mechanizmów, twórca repozytoriów powinien sam umożliwić podział zbioru komponentów na przestrzenie nazw w których łatwo byłoby zachować unikatowość nazw pojedynczych jednostek. Rolę takiego podziału może oferować np. opisany niżej podział oparty na taksonomiach.

4.2. Efektywność wyszukiwania na podstawie nazwy

Niewątpliwą zaletą opisu opartego wyłącznie o nazwę jest fakt, że nie musi on być wprowadzany dodatkowo. Języki programowania standardowo wymagają nadawania nazw jednostkom takim jak funkcja, metoda, klasa czy pakiet lub moduł. Nazwy te mogą być użyte bezpośrednio w repozytorium. Dzięki temu właściwie każde repozytorium oferuje przeszukiwanie w oparciu o nazwę.

W celu oszacowania efektywności wyszukiwania opartego wyłącznie na nazwach przeprowadziłem prosty eksperyment. Poprosiłem dziewięciu programistów o wskazanie trzech nazw dla każdego spośród dziesięciu różnych komponentów i funkcji, których działanie było szczegółowo opisane w ankiecie (patrz załącznik C). Eksperyment jednoznacznie wykazał, że nazwa jednostki programowej zdecydowanie nie niesie wystarczającej informacji umożliwiającej efektywne wyszukiwanie jednostek tylko na jej podstawie. Największym problemem jest z pewnością jej skrótowość, która nie umożliwia przedstawienia istotnych aspektów implementacji. Większość ankietowanych miała tendencję do kwitowania różnych odchyleń od standardowej implementacji niewiele mówiącym sufiksem “Ex” (np. ListEx). Okazało się, że programiści używają również do nazywania komponentów różnych języków (ojczysty lub angielski) i stosują różne konwencje nazewnictwa. Warto też dodać, że mimo iż opisane komponenty były nieskomplikowanymi przykładami wybranymi z rzeczywistych projektów, stopień powtarzalności w nazwach był

bardzo niewielki – najbardziej jednolicie nazywany komponent został nazwany podobnie przez 5 programistów, a w przypadku 4 komponentów nikt (!) nie trafił w ani jedno słowo z nazwy oryginalnego komponentu.

4.3. Wyszukiwanie w oparciu o opis komponentu

Drugim najczęściej stosowanym sposobem opisu jednostek programowych jest dodatkowy opis słowny. Opis taki w odróżnieniu od nazwy musi zostać wprowadzony do repozytorium ręcznie. Nie jest to jednak wielki wysiłek. Zasady czytelnego programowania nakazują bowiem i tak komentowanie każdej funkcji, czy klasy. Opisy te mogą być automatycznie (np. za pomocą narzędzi takich jak JavaDoc[Jav2004] lub Doxygen [Dox2004]) lub z udziałem człowieka przeniesione do repozytorium wraz z umieszczaną w nim jednostką. Opis taki może być użyty nie tylko w fazie wyszukiwania, ale przede wszystkim w fazie przeglądania wyników.

Niestety, samo wyszukiwanie na podstawie opisów również nie wydaje się wystarczające. Oczywiście, w porównaniu do wyszukiwania bazującego na nazwach jednostek, istnieje dużo większe prawdopodobieństwo, iż wynik zapytania będzie zawierał jednostkę, o którą chodziło użytkownikowi. Niestety, ponieważ czysto tekstowe przeszukiwanie opisów nie zwraca uwagi na kontekst, istnieje również duże prawdopodobieństwo, że wynik będzie zawierał dużą liczbę jednostek, których użytkownik w ogóle nie chciał. Duże problemy stwarzają też kwestie tworzenia rankingów na zbiorze wyników. Może to powodować, że nadmiernie wydłużony proces przeglądania zniechęci programistów do przeszukiwania repozytoriów i szczególnie w przypadku prostych jednostek będą woleli napisać je sami, rezygnując tym samym z korzyści jakie niesie ponowne użycie.

Kolejnym problemem jest kwestia obszerności opisu. Jeżeli opis będzie zbyt krótki, może okazać się, że nie zawiera on kwestii istotnych dla wyszukującego (pamiętajmy, że punkt widzenia konstruktora reużywalnej jednostki programowej i jej późniejszego użytkownika mogą znacząco się różnić). Jeżeli opis będzie zbyt długi, istnieje ryzyko nie tylko znacznego wydłużenia fazy przeglądania na skutek dłuższego czasu wymaganego do zapoznania się z opisem, ale również zwiększa się prawdopodobieństwo znalezienia jednostki, która nie nadaje się do zastosowań wyszukującego, co dodatkowo zwiększy czas późniejszego przeglądania.

4.4. Wyszukiwanie w oparciu o słowa kluczowe

Wyszukiwanie oparte na słowach kluczowych polega na przypisaniu każdej z jednostek zestawu słów kluczowych. Słowa te przeszukiwane są na zasadzie zbliżonej do omówionych wcześniej opisów. Ich zaletą jest fakt, że pozwalają zaakcentować najbardziej istotne z punktu widzenia umieszczającego komponent w repozytorium cechy jednostki. Dlatego ten rodzaj wyszukiwania stanowi wartościowe uzupełnienie dwóch wcześniej wymienionych, jednak sam w sobie nie tylko niewiele wnosi, lecz bez towarzyszenia innych metod wyszukiwania staje się niewiele bardziej użyteczny niż poszukiwanie wyłącznie po nazwie.

4.5. Taksonomie

W odróżnieniu od przedstawionych wcześniej metod opisu, taksonomie pozwalają na umieszczenie w opisie jednostki programowej wiedzy, która może być przetwarzana nie tylko przez człowieka, ale również do pewnego (choć w wypadku taksonomii bardzo ograniczonego) stopnia także przez komputer. Taksonomia stanowi pewien hierarchiczny zbiór (drzewo) kategorii. Każda z jednostek programowych umieszczonych w repozytorium musi zostać zakwalifikowana do jednej nich. Ponieważ struktura kategorii jest uporządkowana w postaci drzewa, więc przynależność jednostki programowej do pewnej kategorii implikuje także jej przynależność do wszystkich kategorii będących jej przodkami. Stanowi to pewną namiastkę wspomnianej wcześniej możliwości zmiany precyzji zapytania w zależności od potrzeb.

Podejście oparte o taksonomie wymaga zwykle ręcznego określenia przynależności komponentu do poszczególnych kategorii zdefiniowanych przez taksonomię (choć można sobie wyobrazić taksonomie, dla których klasyfikacja mogłaby zostać przeprowadzona automatycznie). Głębokość drzewa taksonomii nie jest jednak zwykle wysoka, nie stanowi to więc poważnego obciążenia dla wprowadzającego daną jednostkę programową do repozytorium.

Taksonomie można oczywiście łączyć. Jeżeli zdefiniujemy wiele taksonomii, z których każda opisywać będzie oddzielne (choć niekoniecznie) aspekty klasyfikowanych komponentów (np. charakter komponentów, lokalizacja geograficzna usługi WWW, jakość oferowanych usług) uzyskamy potężne narzędzie, które pozwoli nam sformułować zapytanie o komponent z uwzględnieniem dogodnych dla nas aspektów. Możemy np. poszukać komponentu słownika

ortograficznego dla systemu Windows, bez konieczności definiowania potężnej taksonomii w której na pewnym poziomie pojawiałby się podział na dostępne systemy operacyjne. Podział taki niewiele ma wspólnego z podziałem ze względu na inne aspekty i przemieszanie tego w jednej taksonomii byłoby co najmniej nieeleganckie. Zamiast tego możemy wyobrazić sobie dwie taksonomie: jedną klasyfikującą komponenty ze względu na funkcjonalność, drugą klasyfikującą ze względu na platformę docelową. Połączenie wyszukiwania za pomocą tych dwóch klasyfikacji pozwoli nam precyzyjnie opisać pożądane cechy komponentu zmniejszając przy tym zdecydowanie rozmiar taksonomii. Podejście polegające na połączeniu wielu taksonomii do opisu jednostek w repozytorium będzie dalej nazywane *taksonomią wielowymiarową*.

Z każdym rodzajem opisu za pomocą predefiniowanego zbioru kategorii wiąże się jednak pewien problem. Załóżmy, że na potrzeby tworzonej przez nas aplikacji potrzebujemy znaleźć komponent słownika ortograficznego, najlepiej jednak takiego, który posiadałby specjalistyczne słownictwo z dziedziny biologii. Co gorsza, moglibyśmy oczekiwać, że nasza aplikacja sama wybierze sobie najbardziej właściwy słownik spośród niezależnie nabytych od innych producentów i zainstalowanych na komputerze użytkownika (lub np. dostępnych w sieci Internet) słowników ortograficznych. Możemy oczywiście założyć, że twórca taksonomii sam przewidzi odpowiednie kategorie pozwalające precyzyjnie opisać komponenty słowników ortograficznych pod tym właśnie kątem. Wydaje się to jednak, szczególnie w przypadku bardziej ogólnych taksonomii, bardzo mało prawdopodobne. W ogólności nie sposób jest przewidzieć wszystkich aspektów, za pomocą których można przeszukiwać repozytoria jednostek programowych, gdyż bardzo często punkty widzenia twórcy taksonomii (lub innej metody klasyfikacji), twórcy jednostki programowej i użytkownika repozytorium znacząco się od siebie różnią.

Chyba najbardziej znanym zastosowaniem taksonomii do wyszukiwania jest rejestr UDDI [Gra2003]. Taksonomie są też podstawą budowy większości nowoczesnych repozytoriów nastawionych na ponowne użycie kodu [Jac2002, Ezr2003].

4.6. Sieci semantyczne

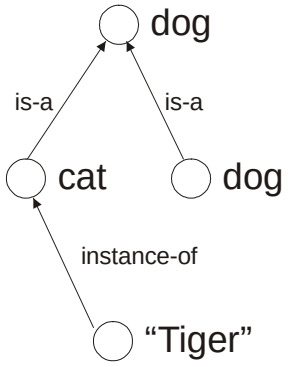
Dużo lepszym sposobem reprezentacji wiedzy od wspominanych wcześniej są sieci semantyczne. Sieć semantyczna łączy w sobie reprezentację wiedzy w sposób czytelny dla człowieka z reprezentacją zrozumiałą dla komputera. Co więcej, jak pokazane zostanie

w dalszych rozdziałach, sieci semantyczne stanowią uogólnienie (a przynajmniej możemy postrzegać je jako uogólnienie) sposobów opisu jednostek za pomocą już przedstawionych technik, a także technik opisu wprowadzonych dalej. Dlatego sieciom semantycznym poświęcono tu więcej uwagi niż pozostałym technikom reprezentacji wiedzy o komponentach.

Przez s*ieć semantyczną* rozumiany będzie dowolny zbiór obiektów reprezentujących pojęcia (same w sobie nie definiowane), oraz zbiór nazwanych, skierowanych bądź nie, relacji pomiędzy tymi pojęciami. Jak łatwo więc zauważyć, sieć semantyczna jest multigrafem, dla którego zdefiniowano funkcję odwzorowującą zbiór krawędzi w zbiór typów relacji. Bardziej formalnie zatem, sieć semantyczną możemy przedstawić jako:

$$N = (C, T, R \subset T \times C \times C)$$

Gdzie C jest zbiorem pojęć, T nazywamy zbiorem typów relacji a R opisuje różnego typu relacje binarne pomiędzy pojęciami(elementami zbioru C). W ogólności można wyobrazić sobie sieć, dla której $T \subset C$, jednak sieci takie nie będą przedmiotem rozważań w tej pracy.



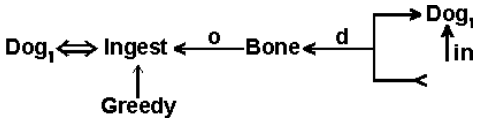
Rys. 4.1. Przykładowa sieć semantyczna

Pojawiły się również pomysły dotyczące wprowadzenia do sieci semantycznych relacji wieloczłonowych [Sow2004]. Ze względu jednak na fakt, że zakres tej pracy ogranicza się do wykorzystania sieci semantycznych w repozytoriach wykorzystywanych do ułatwienia procesu ponownego użycia kodu, gdzie wymagania odnośnie przechowywanej wiedzy nie są aż tak wygórowane, tego rodzaju sieci również zostaną pominięte.

Warto zauważyć, że przy zachowaniu powyższej definicji sieci, istotnym zbiorem jest predefiniowany (zwykle) zbiór typów relacji T. Zbiór ten stanowi o charakterystyce danej sieci. Ze względu na rodzaj relacji i inne cechy wśród sieci semantycznych możemy wyróżnić [Sow2004]:

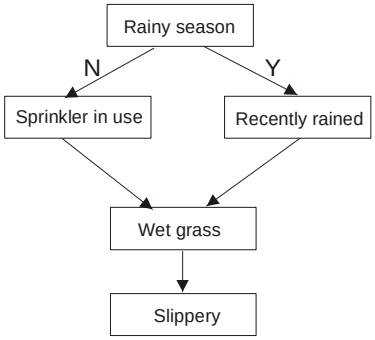
- *Sieci definicyjne* (ang. *definitional networks)* – uwypuklają relacje typu is-a (podtypu). Sieć ta posiada własność dziedziczenia, tzn. wszystkie atrybuty nadtypu są automatycznie dziedziczone przez jego podtypy. Zauważmy, że relacja is-a jest przechodnia. Przykładową siecią definicyjną jest na przykład ta z rys. 4.1.

- *Sieci zapewnieniowe (*ang. *assertional networks)* – używane głównie w celu reprezentacji informacji o różnych pewnikach dotyczących opisywanego świata.



Rys. 4.2. Notacja Schanka jako przykład sieci zapewnieniowej

- *Sieci implikacyjne* (ang. *implicational networks*) – główną relacją jest relacja implikacji (*implies).*



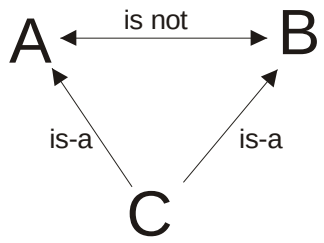
Rys. 4.3. Przykładowa sieć implikacyjna

- *Sieci wykonywalne* (ang. *executable networks)* – zawierają mechanizm wykonawczy, zdolny do rozstrzygania istnienia pewnych relacji lub poszukiwania wzorców w trakcie pracy sieci

- *Sieci uczące się* (ang. *learning networks)* – budują lub poszerzają swoją wiedzę przez pozyskiwanie wiedzy z przykładów. Nowa wiedza może modyfikować istniejącą przez dodawanie lub usuwania relacji oraz przez modyfikowanie wag połączeń (w sieciach z wagami)

- *Sieci hybrydowe* (ang. *hybrid networks)* - stanowią kombinację powyższych typów

Mówiąc o relacjach pomiędzy pojęciami w sieci, należy zwrócić szczególną uwagę na relacje zawierające w swoim znaczeniu negację (jak np. is-not). Dają one możliwość wprowadzenia do sieci sprzeczności, jak to pokazano na rys 4.1.



Rys. 4.3. Sprzeczność w sieci semantycznej

Z rysunku 4.3. można wnioskować[5] m.in.

$$(C \text{ is-a } A) \wedge (C \text{ is-a } B) \wedge \neg(B \text{ is-a } A) \Rightarrow$$
$$(C \text{ is-a } A) \wedge \neg(C \text{ is-a } A)$$

Istnieją różne sposoby radzenia sobie z takimi sprzecznościami. Sowa przedstawia rozwiązanie, w którym niektóre relacje mogą być unieważniane, jako obowiązujące na bardziej ogólnym poziomie, niż relacje bardziej specjalistyczne [Sow2004]. Moim zdaniem, podejście takie jest mylące, a do modelowania związków typu „zwykle jest” należy użyć raczej mechanizmów reprezentowania wiedzy probabilistycznej, sytuacje takie jak z rysunku 4.3. traktując jako ewidentną sprzeczność w wiedzy sieci.

Jasnym jest, że sieć semantyczna w myśl dotychczasowej definicji jest (dla komputera) wyłącznie nic nie znaczącym grafem. Dlatego, aby nadać sieci znaczenia, należy uzupełnić ją o reguły wnioskowania. Np. dla relacji is-a i is-not z rysunku 4.1. (których intuicyjne przeciwieństwo rozważaliśmy dotychczas wyłącznie na podstawie nazw) należy wprowadzić przynajmniej następujące reguły wnioskowania

1. $(X \text{ is-a } Y) \Leftrightarrow \neg (X \text{ is-not } Y)$

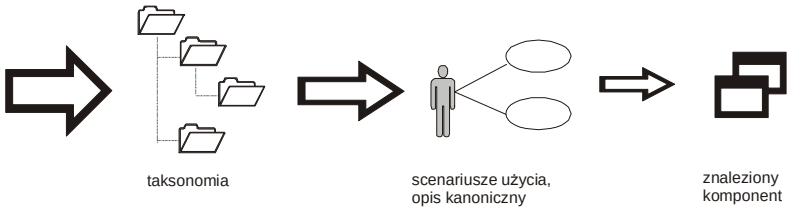
2.$(X \text{ is-a } Y) \wedge (Y \text{ is-a } Z) \Rightarrow (X \text{ is-a } Z)$,

które umożliwią rozumowanie prowadzące do wykazania sprzeczności w rozważanej sieci.

[5] Przy założeniu intuicyjnych reguł wnioskowania

5. PODEJŚCIE KANONICZNE DO WYSZUKIWANIA JEDNOSTEK PROGRAMOWYCH

Jak pokazano w rozdziałach poprzednich, właściwie każda z istniejących metod przeszukiwania repozytoriów jednostek programowych ma swoje wady. Największą z nich jest zdecydowanie często niedostateczna selektywność. Stąd, próbując ograniczyć do minimum wysiłek związany z fazą przeglądania zbioru wyników, w tym rozdziale oraz w dalszej części pracy pokazane są metody, stanowiące dodatkowy, częściowo ortogonalny do pozostałych, sposób wyszukiwania jednostek programowych, pozwalające efektywnie zawęzić zbiór wyników zwracanych na podstawie zapytania użytkownika. Jak już wspomniano, metody te w żaden sposób nie wykluczają stosowania metod opisu takich jak taksonomie, których skuteczności w większości przypadków trudno zaprzeczyć. Podejście takie ułatwi także realizację postulatu postawionego we wstępie, a mianowicie zezwoli użytkownikowi zdecydować o poziomie szczegółowości zapytania, a tym samym podjąć decyzji dotyczącą nakładu pracy w poszczególnych fazach współpracy z repozytorium. To użytkownik będzie bowiem decydował o rodzaju (rodzajach) zastosowanej metody.



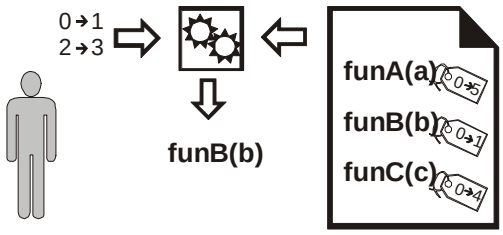
Rys. 5.1. Miejsce omawianych rozszerzeń metod wyszukiwania

Opisywane podejście do wyszukiwania jednostek programowych bazuje na ujednoliceniu opisów jednostki programowej po stronie repozytorium i po stronie zapytania użytkownika. Nie jest to jednak, jak w opisywanych wcześniej metodach, opis słowny czy oparty na taksonomiach. Opis ten stanowić będzie bezpośredni, sformalizowany opis funkcjonalności jednostki programowej przez przedstawienie jej zachowania dla wybranych przypadków użycia. Ponieważ metoda ta nie zakłada dynamicznego zdobywania informacji o jednostce programowej w czasie procesu wyszukiwania (patrz rozdział 5.2), konieczne jest wypracowanie kanonicznej reprezentacji dla owych przypadków użycia, aby umożliwić wprowadzenie wiedzy o jednostce programowej wraz z samą jednostką.

Fakt, iż metoda nie zakłada dynamicznego pobierania danych o jednostce ma bardzo poważne implikacje. Przede wszystkim, repozytorium może być niezależne od używanej do tworzenia jednostek programowych technologii. Oczywiście, dzięki odpowiedniej konstrukcji repozytorium, częściową niezależność można również osiągnąć w przypadku metod zakładających dynamiczne pozyskiwanie wiedzy. Z natury jednak repozytoria tego typu będą czyniły silniejsze założenia i posiadały znacznie większe wymagania co do języka i środowiska wykonywania przechowywanych jednostek. Wymagania te zostaną omówione bardziej szczegółowo w rozdziałach następnych.

5.1. Poszukiwanie podprogramów 1-argumentowych

Omawianie podejścia kanonicznego rozpoczniemy od najprostszego przypadku – poszukiwania pojedynczych, 1-argumentowych funkcji. Od funkcji tych będziemy żądać ponadto, aby wynik ich działania był niezależny od stanu globalnego systemu. Należy zauważyć, że omawiana technika poszukiwania jest również aplikowalna w przypadku metod będących częściami większych jednostek programowych takich jak moduły, klasy czy komponenty, pod warunkiem jednak, że stan nadrzędnej jednostki programowej (ani tym bardziej wspomniany wcześniej globalny stan systemu) nie wpływa na zachowanie funkcji. Pewne odstępstwa od tego założenia można osiągnąć wprowadzając pojęcie warunkowego widma funkcji, omówione na końcu tego rozdziału.



Rys. 5.2. Poszukiwanie podprogramów jednoargumentowych

5.1.1. Konstrukcja opisu i zapytań

Przedmiotem poszukiwań w rozpatrywanej metodzie będzie funkcja w postaci $F : A \to B \cup \{\varepsilon\}$. Zbiór A nazywać będziemy typem argumentu funkcji F, zbiór B typem jej rezultatu. Należy zauważyć, że wszystkie rozpatrywane funkcje traktować będziemy jako funkcje całkowite (tzn. $dom(F) = A$). Jeżeli z logiki aplikacji wynika, że dana funkcja określona jest wyłącznie dla pewnego podzbioru wartości dopuszczalnych dla typu A, rezultatem funkcji będzie w tym przypadku wartość ε,oznaczająca błędne wykonanie,

co pozwoli zachować założenie o całkowitości funkcji. Jest to zgodne z praktyką stosowaną w większości języków programowania, w których nie istnieje mechanizm warunków wstępnych wykonania funkcji, stąd wartości błędne muszą być obsłużone wewnątrz kodu.

Przypadkiem użycia tak zdefiniowanej funkcji F będziemy nazywać pewną asercję w postaci

$$F(a) = b, a \in A, b \in B \cup \{\varepsilon\}$$

Oznacza to, że funkcja F wywołana dla argumentu *a* daje w rezultacie wartość *b*. Zauważmy, że *b* może również przyjąć wartość sygnalizującą błąd. Może to mieć miejsce np. dla przypadku, gdy $F(a) = \frac{1}{a}$, dla której prawdziwe jest stwierdzenie $F(0) = \varepsilon$. Zauważmy, że błąd wykonania w tym przypadku (i bardzo wielu innych) doskonale charakteryzuje istotę funkcji F, stąd przypadki użycia, w wyniku których dochodzi do błędnego wykonania mogą często odgrywać istotną rolę w opisie danej jednostki programowej.

W omawianej metodzie funkcję F charakteryzować będziemy przez zbiór jej przypadków użycia. Zbiór taki, nazywać będziemy *widmem funkcji F*. Widmo takie (dla funkcji jednoargumentowej) ma postać zbioru par $W_F = \{(a,b) : F(a) = b\}$. Niestety, jak łatwo zauważyć, liczba możliwych wartości dla danego typu jest bardzo duża, często nieskończona. Jeżeli zakładamy więc, że daną funkcję charakteryzować będziemy przez zbiór przypadków użycia, musimy w oczywisty sposób ograniczyć się do pewnego skończonego podzbioru dostępnych dla wywołania funkcji wartości. Ponadto, aby efektywnie przeprowadzić łączenie zapytania użytkownika (mającego również postać zbioru przypadków użycia) z informacją przechowywaną w repozytorium, należy zapewnić, że przypadki użycia podawane przez użytkownika będą obliczane dla tych samych wartości argumentu funkcji, co te przechowywane wraz z jednostką programową. W tym celu musimy wprowadzić pojęcie zbioru *wartości kanonicznych* dla typu danych.

Niech Ψ będzie zbiorem typów danych dostępnych w danym języku. Przyporządkujmy każdemu typowi $T \in \Psi$ pewien skończony zbiór wartości C_T, takich, że $C_T \subseteq T$. Zbiór ten nazywać będziemy zbiorem *wartości kanonicznych* typu T, a każdy jego element *wartością kanoniczną typu T*. Formalnie, nic nie stoi na przeszkodzie, żeby zbiór wartości kanonicznych zawierał wszystkie dopuszczalne wartości typu T ($C_T = T$), pod warunkiem, że zbiór ten jest

skończony. W praktyce jednak, zbiór ten powinien być raczej niewielki, aby umożliwić przechowanie przypadków użycia F dla każdej wartości argumentu ze zbioru C_A, stąd pomijając niewielkie typy wyliczeniowe, zwykle stanowi on jedynie niewielki podzbiór wartości T uznanych za „najbardziej charakterystyczne". I tak na przykład, możemy przyjąć, że $C_{\text{int}} = \{-2,-1,0,1,2\}$, choć oczywiście ze względu na subiektywną ocenę „charakterystyczności" danej liczby całkowitej, można wyobrazić sobie wiele innych sensownych zbiorów wartości kanonicznych dla typu całkowitoliczbowego (choć większość z nich z pewnością będzie zawierała wartości {-1, 0 ,1}).

Widmem kanonicznym funkcji F nazywamy widmo W_F, takie, w którym uwzględniono tylko i wyłącznie rezultaty wywołania funkcji dla wszystkich wartości kanonicznych wynikających z typu jej argumentu. Zastosowanie widma tego typu do opisu jednostek programowych pozwala nie tylko zaoszczędzić miejsce (zmniejsza drastycznie liczbę koniecznych do zapamiętania wartości), ale przede wszystkim przez ujednolicenie wartości, dla których określane jest widmo w opisie funkcji z wartościami w zapytaniu użytkownika, ułatwia ujednolicenie owych opisów w procesie wyszukiwania. Bardziej szczegółowo proces ten przedstawiony jest w rozdziale 5.1.2.

Jak łatwo się zorientować zbiór wartości charakterystycznych jest dobierany przez człowieka na podstawie jego subiektywnych odczuć. Co więcej, prawidłowy dobór tych zbiorów jest kluczowy dla skuteczności omawianej metody, co niestety stanowi jej poważną wadę. Należy zauważyć, że w ogólności zbiór Ψ nie musi być ani skończony, ani znany a priori. Na przykład języki obiektowe umożliwiają programiście definiowanie własnych typów. Aby rozważyć możliwość definiowania zbiorów wartości charakterystycznych również dla takich typów, należy wprowadzić dwa dodatkowe pojęcia:

Relacją dziedziczenia na zbiorze Ψ nazywamy relację *is-a*(X,Y), gdzie $X,Y \in \Psi$. Relacja dziedziczenia jest wynikiem wewnętrznych mechanizmów języka umożliwiających wprowadzenie dziedziczenia między typami. Stwierdzenie *is-a*(X,Y) oznacza zatem, że X dziedziczy z Y.

Relacją rzutowania na zbiorze Ψ nazywamy relację *castable*(X,Y), gdzie $X,Y \in \Psi$. Zapis *castable*(X,Y) oznacza, że typ X może być odwzorowany (rzutowany) w typ Y. Jest to relacje ogólniejsza od relacji dziedziczenia. W szczególności zachodzi

$$is\text{-}a(X,Y) \Rightarrow castable(X,Y)\ .$$

Relacja *castable* jest ogólniejszym przypadkiem relacji *is-a*, gdyż może zachodzić również między typami prostymi, dla których nie zdefiniowano hierarchii dziedziczenia (np. $castable(\text{int}, float)$). Oprócz mechanizmów rzutowania, istnienie relacji rzutowania jest efektem stosowania takich konstrukcji, jak specjalne rodzaje konstruktorów, czy definiowane przez programistę własne operatory rzutowania [Str1995]. Należy też zauważyć, że w odróżnieniu od relacji dziedziczenia, prawdą jest, że istnieją typy X i Y takie iż:

$$castable(X,Y) \wedge castable(Y,X), X \neq Y$$

Na przykład w języku Java prawdziwe jest stwierdzenie $castable(\text{int}, Integer)$ ale także $castable(Integer, \text{int})$, choć do wersji 1.5 języka konwersja ta nie była przeprowadzana w sposób automatyczny.

Mając tak określone relacje na zbiorze T, możemy założyć, że:

$$castable(X,Y) \Rightarrow \forall x \in C_X : ((Y)x) \in C_Y$$

gdzie zapis $(Y)x$ należy rozumieć jako rzutowanie elementu x typu X na typ Y (rzutowanie jest realizowalne na mocy założenia). Dzięki temu możliwe jest swoistego rodzaju dziedziczenie wartości kanonicznych pomiędzy typami. Na nieszczęście, zachodzi ono dokładnie w przeciwną stronę do relacji rzutowania typów, stąd nie do końca rozwiązuje problem typów nowo definiowanych, nie przewidzianych w konstrukcji repozytorium. Dla wielu nowo wprowadzonych typów, wprowadzający ręcznie będzie musiał zdefiniować zbiór wartości kanonicznych.

W praktycznych zastosowaniach dobrym pomysłem może być wprowadzenie reguły, że zbiory wartości kanonicznych dla typów wbudowanych definiowane są przy zakładaniu repozytorium, natomiast dla typów użytkownika (obiektowych), zbiory te mogą mieć domyślnie jeden element powstały przez użycie domyślnego konstruktora danego typu, oraz ewentualne wartości uzyskane dzięki wprowadzeniu opisanych wcześniej relacji.

Nawet w tak prostym przypadku jak poszukiwanie podprogramów 1-argumentowych pojawiają się dość istotne problemy związane z podejściem kanonicznym. Jednym z nich jest kwestia identyfikowania błędnego wykonania funkcji. Ten rodzaj rezultatu może objawiać się

na bardzo różne sposoby. W językach obiektowych najbardziej charakterystycznym zachowaniem funkcji w takiej sytuacji jest rzucenie wyjątku. Może się jednak okazać, że funkcja zamiast tego przekazuje umowny wynik, reprezentujący wartość błędną. Jest to w szczególności możliwe (i chętnie stosowane) w przypadku, gdy przeciwdziedzina funkcji nie wyczerpuje całego zbioru B. Jeżeli np. funkcja z założenia produkuje wyniki nieujemne, wartość < 0 (bardzo często –1) reprezentuje jej nieprawidłowe wykonanie. Równie częstym zachowaniem w przypadku funkcji zwracających wskaźnik lub referencję do obiektu jest przekazanie wartości pustej (oznaczanej najczęściej jako *null* lub *nil*).

Dane funkcji mogą (i jest to założenie tego podejścia) być wprowadzane do repozytorium ręcznie. Od założenia tego można jednak częściowo odejść, ułatwiając sobie pracę za pomocą automatów, potrafiących uzyskać widmo funkcji wywołując ją cyklicznie dla wszystkich wartości kanonicznych typu jej argumentu. Niestety działanie to może wymagać nadzoru, jeżeli funkcje w repozytorium zgłaszają wartość błędną inaczej niż przez rzucenie wyjątku. Bez wiedzy o semantyce rozpatrywanej funkcji (a tego opisywana metoda nie zakłada), automat nie będzie w stanie stwierdzić, czy np. liczbę ujemną w rezultacie funkcji traktować jako wynik pożądany, czy może jako sygnalizację błędu.

Jeszcze inny problem pojawia się podczas porównywania ze sobą liczb rzeczywistych. Poleganie na relacji równości może łatwo prowadzić do niepotrzebnego dyskryminowania niektórych podprogramów, ze względu na możliwe niewielkie rozbieżności między wartością oczekiwaną a obliczoną. Problem ten na szczęście można dość łatwo rozwiązać stosując przedziały zamiast dokładnej specyfikacji wartości oczekiwanej.

5.1.2. Proces wyszukiwania

Jak już wspomniano, w omawianym podejściu zarówno opis jednostki programowej (w tym wypadku funkcji jednoargumentowej) po stronie repozytorium jak i zapytanie użytkownika mają postać kanonicznego widma (odpowiednio: właściwego dla przechowywanej funkcji i pożądanego przez użytkownika). Zapytanie użytkownika zawiera dodatkowo nazwę oczekiwanych typów argumentu poszukiwanej funkcji oraz jej rezultatu. Proces wyszukiwania polega zatem na dopasowaniu widm do siebie i przedstawieniu użytkownikowi rezultatu w postaci funkcji pasujących do jego zapytania.

Mówimy, że funkcja $F : A \to B$ określona widmem W_F pasuje do zapytania $\{A_Q, B_Q, W_Q\}$, jeżeli

$$castable(A_{Q,} A) \wedge castable(B_Q, B) \wedge W_Q \subset W_F$$

Ponieważ typy A_Q i A (podobnie jak B_Q i B) mogą być różne, zawieranie $W_Q \subset W_F$ należy testować uwzględniając odpowiednią konwersję pomiędzy typami.

Należy zwrócić uwagę, że użytkownik nie jest zmuszany do podania wartości funkcji dla wszystkich wartości kanonicznych. Może on wybrać te z nich, które jego zdaniem najbardziej charakteryzują poszukiwaną jednostkę programową, jednak w wyborze wartości jest ograniczony w sposób oczywisty do wartości charakterystycznych żądanego typu argumentu.

5.1.3. Widmo warunkowe

Dotychczasowe rozważania prowadzone były przy założeniu, że funkcja F nie zależy od stanu globalnego ani, w przypadku jeżeli jest metodą pewnej klasy, od stanu obiektu tej klasy dla którego jest wywoływana (co w praktyce oznacza żądanie, aby była metodą statyczną niezależną od stanu globalnego ani żadnych pól statycznych). Pojęcie widma warunkowego pozwala (przynajmniej w teorii) nieco osłabić te założenia.

Widmem warunkowym metody F dla obiektu O w stanie S nazywamy widmo jak z rozdziału 5.1.1, z tym jednak, że metoda F nie musi być statyczna i może zależeć od stanu obiektu. Metoda F nie może jednak modyfikować stanu obiektu, dzięki czemu utrzymujemy (niejawnie przyjęte w rozdziałach poprzednich, jako wynikające z faktu niezależności od stanu), że:

$$\forall a_1, a_2 : O.F(a_1) = b_1; O.F(a_2) = b_2 \Rightarrow O.F(a_2) = b_2; O.F(a_1) = b_1,$$

czyli kolejność wywołań przy zadanym stanie obiektu O nie jest istotna dla otrzymywanych rezultatów.

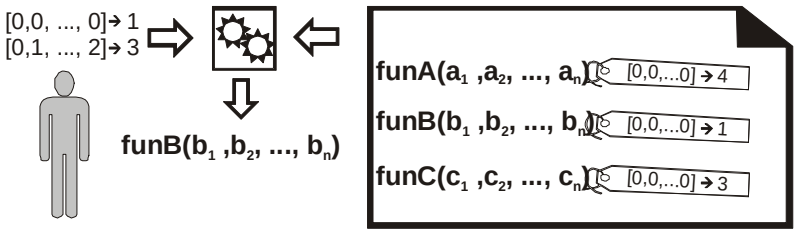
Stan (będący warunkiem, pod którym widmo jest poprawne) można reprezentować na wiele różnych sposobów. Rolę tą może pełnić np. sekwencja wywołań metod przeprowadzająca obiekt ze stanu początkowego w stan S. Nie znając jednak docelowej klasy obiektu O, a więc tym samym metod jakie on posiada nie można z góry podać jaka sekwencja powinna

poprzedzać procedurę pozyskania widma rozpatrywanej jednostki. Trudno też przechowywać sekwencję wszystkich metod pod wszystkimi możliwymi warunkami (w ogólności obiekt może mieć nieskończenie wiele stanów). Dlatego należy traktować wprowadzone pojęcie widma warunkowego jako sygnalizację problemów wyszukiwania metod zależnych od stanu obiektu, który omówiony będzie w rozdziałach następnych.

5.2. Wyszukiwanie podprogramów wieloargumentowych

5.2.1. Dopasowanie z zachowaniem liczby argumentów

Podprogramy jednoargumentowe są rzadkością, stąd metoda poszukiwania tego typu jednostek programowych musi zostać rozszerzona na podprogramy wieloargumentowe, aby w ogóle można było mówić o stosowaniu tego typu podejścia w praktyce.



Rys. 5.3 Poszukiwanie podprogramów wieloargumentowych

Przedmiotem rozważań tego rozdziału będą zatem funkcje postaci:

$$F : A_1 \times A_2 \times ... \times A_n \rightarrow B \cup \{\varepsilon\}$$

Poszukiwanie podprogramów wieloargumentowych za pomocą kanonicznej metody opisu w punktach charakterystycznych niesie ze sobą wiele dodatkowych trudności w porównaniu z opisanym w rozdziale poprzednim poszukiwaniem podprogramów $1-$ argumentowych. Podobnie, jak to pokazano w rozdziale poprzednim, do opisu funkcji zarówno po stronie repozytorium jak i po stronie zapytania użytkownika używać będziemy widma funkcji. Aby jednak dostosować opisywaną metodę do poszukiwania funkcji o wielu argumentach konieczne jest rozszerzenie kilku definicji z poprzedniego rozdziału.

Przypadkiem użycia n-argumentowej funkcji $F : A_1 \times A_2 \times ... \times A_n \rightarrow B \cup \{\varepsilon\}$ będziemy nazywać asercję w postaci

$$F(a_1, a_2, ..., a_n) = b, \text{gdzie } a_1 \in A_1, a_2 \in A_2, ..., a_n \in A_n, b \in B \cup \{\varepsilon\}$$

W inny sposób przypadek użycia przedstawiać będziemy jako ciąg wartości w postaci $(a_1, a_2, ..., a_n, b)$. Istotne przy tym jest zachowanie kolejności argumentów. Przez analogię, widmem n-argumentowej funkcji F nazywać będziemy pewien zbiór

$$W_F = \{(a_1, a_2, ..., a_n, b) : F(a_1, a_2, ...a_n) = b\}$$

Oczywiście, również przez analogię, *widmem kanonicznym n-argumentowej funkcji F* nazywać będziemy widmo W_F takie, że

$$\forall (a_1, a_2, ..., a_n, b) \in W_F : a_1 \in C_{A_1}, a_2 \in C_{A_2}, ..., a_n \in C_{A_n}$$

Zauważmy, że szczególnie w przypadku funkcji o wielu argumentach istotne jest ograniczenie rozmiaru zbioru wartości kanonicznych dla poszczególnych typów, jako że liczba wpisów w pełnym widmie kanonicznym jest iloczynem mocy zbiorów wartości kanonicznych poszczególnych typów argumentów.

O ile rozszerzenie definicji dotyczących poszukiwania metodą kanoniczną, tak aby uwzględniały dowolną ilość argumentów funkcji, jest raczej naturalne, o tyle wymagane zmiany w procesie wyszukiwania są już poważniejsze. Najważniejszym problemem z jakim musi radzić sobie mechanizm wyszukujący w przypadku funkcji o wielu argumentach jest kwestia ich kolejności (Założono, że w zapytaniu użytkownika musi wystąpić dokładnie tyle samo argumentów ile jest w poszukiwanej funkcji. Założenie to zostanie nieco osłabione w dalszej części rozdziału). Nie ma żadnych gwarancji, że użytkownik w zapytaniu określi fragment widma funkcji stosując dokładnie tą samą kolejność parametrów, której użył jej twórca. Dlatego mechanizm wyszukujący podczas dopasowywania do siebie obydwu widm musi uwzględnić wszystkie permutacje argumentów wejściowych zachowujące zgodność typów. Niech $P : N \to N$ będzie permutacją argumentów wejściowych odwzorowującą indeksy argumentów z zapytania o typach $A_1^Q, A_2^Q, ..., A_n^Q$ w indeksy argumentów widma w repozytorium o typach $A_1^R, A_2^R, ..., A_n^R$. Mówimy, że permutacja P zachowuje zgodność typów, jeżeli

$$\forall i = 1..n : castable(A_i^Q, A_{P(i)}^R)$$

Oznacza to tyle, że każdy argument z zapytania musi mieć swój odpowiednik w opisie w repozytorium, przy czym typ argumentu z zapytania musi dać się rzutować w typ

z repozytorium (co jest warunkiem użycia rozpatrywanej funkcji). Oczywiście może się zdarzyć, że właściwe odwzorowanie nie istnieje. Oznacza to jednak, że rozpatrywana funkcja nie może zostać dopasowana do zapytania użytkownika.

Aby sprawdzić, czy funkcja F z repozytorium pasuje do opisu użytkownika, mechanizm wyszukujący musi najpierw znaleźć wszystkie permutacje zachowujące zgodność typów. Następnie dla każdej tego typu permutacji wykonywany jest test zawierania widm (podobnie jak w rozdziale 5.1.1)

$$W_Q \subset W_F$$

Oczywiście, nietrudno zauważyć, że poszukiwanie tego rodzaju może być bardzo czasochłonne. Na szczęście, dużo ścieżek poszukiwań może zostać odciętych zaraz na początku, czy to na skutek niezgodności typów, czy na skutek nieprawidłowych wartości w widmie. Warto w związku z tym odejść od liniowego modelu przetwarzania polegającego na znalezieniu wszystkich dostępnych permutacji argumentów, a dopiero potem na próbie dopasowania tak powstałych widm. Pozwoli to na szybką rezygnację z permutacji, które nie rokują nadziei na znalezienie dopasowania między widmami. Warto również zauważyć, że przeglądając kody różnych programistów można zauważyć pewne określone tendencje w nadawaniu kolejności argumentom. Podobieństwa w kolejności są szczególnie silne między ludźmi pracującymi w tym samym zespole, często wzajemnie przeglądającymi i poprawiającymi swój kod. Dlatego warto podczas poszukiwania właściwej permutacji rozpocząć od sprawdzenia widm dla oryginalnego ustawienia parametrów (jeżeli jest to możliwe ze względu na zgodność typów), gdyż być może pozwoli to natychmiast uzyskać rozwiązanie bez zagłębiania się w dalsze permutacje.

5.2.2. Dopasowanie bez zachowania liczby argumentów

W rozdziale 5.2.1 poczyniono założenie, że funkcja poszukiwana przez użytkownika musi mieć dokładnie tyle argumentów, ile określił on w swoim zapytaniu konstruując fragment widma charakteryzującego poszukiwana jednostkę. Założenie to na pierwszy rzut oka wygląd słusznie, okazuje się jednak zdecydowanie zbyt idealistyczne.

Przyjrzyjmy się funkcji zdefiniowanej następująco:

$$\text{t(double x, int m)} = \begin{cases} \sin(x) \Leftrightarrow m = 0, \\ \cos(x) \Leftrightarrow m = 1 \end{cases}$$

Wyobraźmy sobie, że posiadając repozytorium w którym znajduje się taka funkcja, użytkownik sformułował następujące zapytanie:

```
sin(double), W={(0,0), (1, 1,745e-2)}
```

Mimo, że zapytanie to opisuje funkcje sinus, do której wyliczenia funkcja t znakomicie się nadaje, nie zostanie ona znaleziona, przy użyciu metody opisanej w rozdziale 5.2.1. Wynika to z różnej ilości argumentów wejściowych.

Z problemem tym można sobie radzić specyfikując niektóre parametry jako nieobowiązkowe przy jednoczesnym wyszczególnieniu wartości (jednej lub wielu) domyślnych. W tym wypadku, parametrem nieobowiązkowym jest oczywiście parametr m, dla którego wartości domyślne to 0 i 1. Co więcej, można by pokusić się o rozwiązanie, w którym argumenty tego typu nie będą podlegały ograniczeniom wynikającym ze zbioru wartości kanonicznych dla ich typu, co do pewnego stopnia zapobiegnie ukrywaniu przez widmo kanoniczne pewnych aspektów funkcjonalności.

W wariancie prostszym, problem ten można rozwiązać, polecając programistom dostarczanie prostszych interfejsów do funkcji takich jak *t*. W tym przypadku należało by zdefiniować funkcje *sin(double x) = t(x,0)* oraz *cos(double x) = t(x,1)*, co rozwiązało by problem. Niestety w ogólności nie zawsze jest to możliwe, jeszcze rzadziej jest to rozwiązanie eleganckie.

5.3. Efektywność podejścia kanonicznego

Największą zaletą podejścia kanonicznego, jak i prezentowanych w następnych rozdziałach jest fakt, że pozwala przekazać do repozytorium wiedzę o poszukiwanych jednostkach programowych z innej perspektywy, w wielu wypadkach szybciej i wygodniej niż za pomocą metod klasycznych. Co więcej, w żaden sposób nie wymaga ono rezygnacji z dotychczas stosowanych rozwiązań opartych o taksonomie, opisy słowne czy sieci semantyczne, dzięki czemu może być traktowane po prostu jako uzupełnienie tych form opisu. Z kolei w porównaniu z podejściami reprezentowanymi w rozdziałach następnych, zakładającymi dynamiczne pozyskiwanie wiedzy o jednostkach programowych, podejście kanoniczne

charakteryzuje się prostotą oraz dużą niezależnością od języka programowania, środowiska uruchomieniowego itp.

Mimo to, podejście to posiada także wiele wad. Chyba najważniejszą z nich jest założenie o niezależności poszukiwanych funkcji od stanu. Jest to bardzo silne założenie, powodujące, że metoda ta najbardziej nadaje się do poszukiwania prostych funkcji o charakterze typowo matematycznym. Rozwiązania tego problemu, takie jak widmo warunkowe opisane w rozdziale 5.1.3 należy zaliczyć raczej do rozważań czysto teoretycznych.

Drugą istotną wadą podejścia kanonicznego jest silny wpływ, jaki wywiera na skuteczność metody dobór odpowiednich wartości kanonicznych dla poszczególnych typów danych. Jeżeli wartości tych wybierzemy zbyt mało, widmo funkcji stanowić będzie widok na zbyt wąską funkcjonalność przez nią oferowaną . Odbije się to niekorzystnie na selektywności wyszukiwania. Widać to dobrze na przykładzie wyników uzyskanych za pomocą prototypowego repozytorium (patrz rozdział 7). W repozytorium tym umieszczono metody języka Java, wyszczególnione w załączniku A. Okazało się, że nawet dla tak niewielkiego repozytorium (choć oczywiście odpowiednio dobranego, tak aby przewidzieć pewne problemy) pojawiły się znaczące trudności w opisie funkcjonalności za pomocą widm przy niewielkich zbiorach wartości kanonicznych.

Zapytanie o funkcję inc(int x)[6] opisaną widmem {(0,1), (1,1)} - $C_{\text{int}} = \{0,1\}$, repozytorium zwróciło aż cztery funkcje, z czego tylko jedna z nich realizowała pożądaną funkcjonalność. Były to funkcje inc, power2, exp3plus1 oraz sgnplus1[7], wszystkie posiadające identyczne widmo, mimo zupełnie różnej funkcjonalności. Rozszerzenie zbioru C_{int} o wartość –1 oraz odpowiednie rozszerzenie zapytań pozwoliło wyeliminować funkcję power2, pozostawiając jednak ciągle dwie zbędne funkcje. Podobne problemy pojawią się przy takim zbiorze wartości kanonicznych dla liczb całkowitych dla większości funkcji z załącznika A. Z kolei nadmierne rozszerzanie zbioru C_{int} może prowadzić do znaczącego wydłużenia czasu

[6] nazwy metod podane są wyłącznie dla wygody czytelnika i określają intencjonalne znaczenie funkcji. Podczas eksperymentu nazwy te nie zostały wprowadzone do komputera, więc nie miały żadnego wpływu na zwracane wyniki.

[7] Działanie funkcji podano w załączniku A

obliczeń, czyniąc metodę praktycznie bezużyteczną. Dlatego dobieranie wartości kanonicznych dla typów jest trudnym do oszacowania kompromisem między wydajnością a szybkością obliczeń (zauważmy, że dodanie do zbioru C_{int} wartości 2 powoduje jednoznaczną identyfikację wymienionych funkcji).

Rozwiązanie to posiada także szereg pomniejszych wad, wspomnianych już w rozdziałach poprzednich. Są to problemy z identyfikacją wartości błędnych, argumenty o wartościach domyślnych, problemy z obiektami typów zdefiniowanych przez użytkownika itp. W opisie metody założono także niejawnie, że wszystkie parametry funkcji są typu wejściowego, co w praktyce nie zawsze ma miejsce. Wszystko to sprawia, że metodę należy uznać raczej jako wstęp do rozwiązujących większość tych problemów (choć nie wszystkie), ogólniejszych metod opisanych w rozdziałach następnych. Niestety, za rozwiązanie tych problemów trzeba zapłacić cenę – metody te są silnie związane z konkretna technologią i wymagają obecności specyficznych środowisk uruchomieniowych.

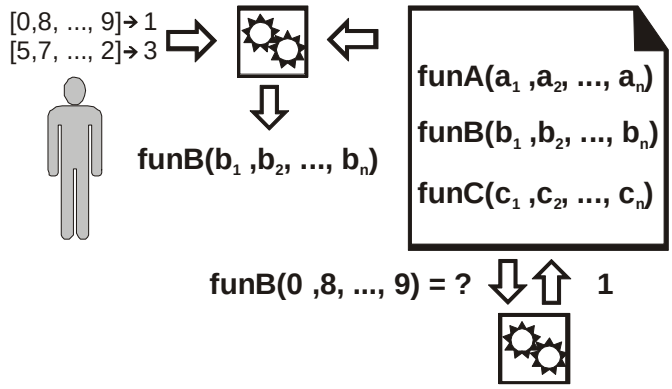
6. WYSZUKIWANIE Z DYNAMICZNYM POZYSKIWANIEM DANYCH O JEDNOSTKACH PROGRAMOWYCH

W rozdziale 5. opisano metody wyszukiwania, które zakładają wprowadzenie danych o przechowywanej jednostce programowej w momencie jej wprowadzania do repozytorium. Oczywiście dane te mogły być wprowadzone ręcznie, lub za pomocą automatu, który np. pozwala na uzyskanie widma zadanej funkcji dla określonego zbioru wartości kanonicznych. Samo jednak repozytorium pozostawało biernym kontenerem danych, posiadającym mniej lub bardziej zaawansowane mechanizmy wyszukiwania.

W tym i następnych rozdziałach omówiono metody wyszukiwania jednostek programowych, które zakładają, iż repozytorium samo, w trakcie procesu wyszukiwania będzie w stanie pozyskać (oraz ewentualnie przechować do późniejszych zastosowań) potrzebne dane o jednostce programowej. Wbrew pozorom różnica ta nie jest jedynie subtelną zmianą w filozofii wprowadzania danych, ale pozwala rozwiązać wiele problemów z metodami opartymi jedynie na wiedzy dostępnej w momencie wprowadzania jednostki do repozytorium. Niestety, metody te wymagają, aby repozytorium posiadało odpowiedni moduł wykonawczy, zdolny załadować rozpatrywaną jednostkę, utworzyć dla niej środowisko uruchomieniowe, po czym odpytać ją o szereg wymaganych parametrów. W niektórych językach programowania czy technologiach komponentowych jest to proste. W innych, może sprawić poważną trudność. Tak czy inaczej sprawia, że repozytorium staje się ściśle zależne od technologii wytwarzania przechowywanych jednostek. Oczywiście można sobie wyobrazić repozytorium posiadające wiele jednostek wykonawczych, dla różnych technologii. Jednak problemy, które to ze sobą niesie, związane z ujednoliceniem typów danych, łączenia wywołań w różnych technologiach itp., są tak duże, że ich rozwiązanie w celu zbudowania repozytorium może okazać się grą nie wartą świeczki. Co więcej repozytorium takie i tak nie będzie się nadawać do przechowywania jednostek programowych stworzonych w technologii nie przewidzianej przez twórców repozytorium.

6.1. Wyszukiwanie funkcji niezależnych od stanu

Wyszukiwanie funkcji niezależnych od stanu przy pomocy podejścia zakładającego dynamiczne pozyskiwanie danych jest w zasadzie podobne do wyszukiwania przy pomocy podejścia kanonicznego opisanego w rozdziale 5.



Rys 6.1. Wyszukiwanie funkcji niezależnych od stanu z dynamicznym pozyskiwaniem danych

Podobieństwo to przejawia się tym, że również i w tym przypadku użytkownik charakteryzuje poszukiwaną jednostkę programową za pomocą ciągu przypadków użycia, tj. pożądanego widma tej funkcji. W odróżnieniu jednak od podejścia kanonicznego, nie jest on w żaden sposób ograniczony jeżeli chodzi o wartości argumentów danego typu. Zakłada się zatem pełną dowolność danych wejściowych.

W dalszej części pracy, o ile nie zaznaczono inaczej, zakłada się także, że każda uruchamialna jednostka programowa zakończy swoje działanie w skończonym czasie. Założenie to można przyjąć bez utraty ogólności rozważań, jako że w konstrukcji repozytorium można przewidzieć pewien czas, po którym wykonywanie jednostki zostanie przerwane, a wynik wykonania zostanie potraktowany jako wynik błędny. Wiąże się z tym kilka zasadniczych trudności, problemy związane ze specyfikacją czasu wykonania nie są jednak przedmiotem tej pracy.

Aby poradzić sobie z taką swobodą w tworzeniu pożądanego widma funkcji, repozytorium nie może się ograniczać do przechowywania obliczonych wcześniej lub wprowadzonych przez użytkownika wartości funkcji dla wybranych kombinacji wartości parametrów wejściowych funkcji. Musi ono być w stanie pozyskiwać wiedzę na temat zachowania rozpatrywanej funkcji automatycznie. W tym celu repozytorium należy wyposażyć w moduł wykonawczy, zdolny do samodzielnego załadowania odpowiednich bibliotek oraz wykonania rozpatrywanych funkcji dla pewnych danych wejściowych. Języki takie jak Java oraz większość technologii komponentowych umożliwiają bardzo prostą realizację tego postulatu. Podejścia takiego można również użyć umieszczając funkcję w dynamicznie ładowanych bibliotekach w dowolnej technologii (np. DLL dla Windows) . W ogólności jednak realizacja modułu wykonawczego może być nietrywialna, lub nawet niemożliwa. Wprowadzenie takiego modułu powoduje również silne uzależnienie repozytorium od technologii (języka,

środowiska uruchomieniowego), w jakiej tworzone są przechowywane jednostki programowe.

Od strony formalnej, wyszukiwanie przy zastosowaniu tego podejścia, również jest bardzo podobne do opisanego w rozdziale 5.2.1. Moduł wyszukujący znajduje wszystkie permutacje argumentów zachowujące zgodność typów, po czym dla każdej takiej permutacji odpytuje moduł wykonawczy o wartość funkcji dla zadanych wartości wejściowych (oczywiście działania te można zrównoleglić w celu jak najwcześniejszego odcięcia gałęzi drzewa poszukiwań). Funkcję w repozytorium określa się jako pasującą do zapytania użytkownika, jeżeli istnieje taka permutacja zachowująca zgodność typów P_C, że dla każdego przypadku użycia podanego w zapytaniu, funkcja ta ma wartość określoną przez te przypadki przy uwzględnieniu permutacji wartości argumentów. Zauważmy, że permutacja P_c jest wspólna dla wszystkich przypadków użycia.

Metoda ta, jest dużo skuteczniejsza od metod opartych o kanoniczny opis jednostek programowych. Przyjrzyjmy się wynikowi zapytania, mającego na celu znalezienie jednoargumentowej funkcji inc(int x), zwiększającej wartość argumentu o 1. Wystarczy zrezygnować ograniczenia się do wartości kanonicznych podając zapytanie o widmo {(0,1), (16,17)} aby otrzymać dokładnie jedną (spośród wymienionych w załączniku A) funkcję – poszukiwaną funkcję zwiększającą argument o 1. Tak naprawdę wystarczyłoby zapytanie {(16,17)} aby otrzymać dokładnie ten sam rezultat, byłaby to jednak przesadna oszczędność.

Równie prosto można wyszukać np. metodę power(x, y). Dla przykładowego widma {(2,4, 16),(3,4,81)} otrzymamy w rezultacie dokładnie jedną funkcję – poszukiwaną funkcję power. I ponownie jak w poprzednim przypadku już jeden z punktów wystarczyłby do precyzyjnego określenia nazwy metody. Warto zwrócić uwagę, że o ile w przypadku wartości (3,4,81) odwzorowanie jest zupełnie jednoznaczne, to dla wartości (2,4,16) mechanizm przeszukujący nie może ustalić poprawnego odwzorowania argumentów (gdyż $2^4 = 4^2 = 16$). Zaproponuje zatem dwa rozwiązania, oba jednak wskazujące na tą samą metodę power.

W przypadku funkcji tak prostych jak inkrementacja, dobór wartości dla których przetestujemy rezultat funkcji jest właściwie dowolny (choć akurat w tym przypadku wartości bliskie 0 nie są najlepsze). Podobny efekt osiągnęlibyśmy więc w przypadku podejścia kanonicznego rozszerzając zbiór C_{int} o wartość powiedzmy 16. Szybko jednak musielibyśmy dodać dowolną liczbę ujemną odkrywając, że po dodaniu do repozytorium funkcji abs(inc(x))

sytuacja znowu jest niejednoznaczna. W ogólności, w przypadku skomplikowanych funkcji jest dużo lepiej pozwolić użytkownikowi wybrać wartości, które najlepiej jego zdaniem charakteryzują poszukiwaną funkcję, niż narzucić mu je z góry.

Na przykładzie funkcji inc(x) widać też dobrze, że przypadki testowe, które na pierwszy rzut oka wydają się dobre, mogą okazać się niewystarczająco selektywne. Dlatego proces wyszukiwania należy traktować jako iteracyjny proces składający się z naprzemiennych faz wyszukiwania i przeglądania.

Oprócz cech wskazanych powyżej, podejście oparte na dynamicznym pozyskiwaniu danych dla funkcji niezależnych od stanu posiada wszystkie wady podejścia kanonicznego. Problemy takie jak problem ilości argumentów czy problem porównywania wartości zmiennoprzecinkowych można rozwiązać dokładnie w taki sam sposób. Nieco więcej uwagi należy się problemowi identyfikacji wartości błędnej. Ponieważ pozyskiwanie danych odbywa się tu automatycznie, problem nabiera na sile. Można go rozwiązać, umieszczając wprowadzoną ręcznie dla każdej funkcji informację, jaki typ wyniku uznaje się za sygnalizujący błąd (np. w postaci asercji lub fragmentu kodu w wbudowanym w repozytorium języku skryptowym). Umożliwi to mechanizmowi wykonawczemu zidentyfikowanie wartości błędnych oraz ujednolicenie ich poprzez np. rzucanie wyjątku.

Opisana technika na pozór wydaje się mieć zastosowanie wyłącznie do metod deterministycznych. W praktyce można się jednak pokusić o wykorzystanie jej również w celu poszukiwania funkcji o charakterze niedeterministycznym. Oczywiście funkcje niedeterministyczne w praktyce nie występują, a funkcje takie jak *random* zawdzięczają swój pseudolosowy charakter skomplikowanej, nieliniowej zależności od stanu globalnego. Z tego powodu wydawać mogło by się, że powinny być raczej przedmiotem rozważań w rozdziale 6.2. Należy jednak zauważyć, że owa zależność od stanu jest jedynie środkiem realizacji celu, a nie wymaganiem samym w sobie, stąd funkcje takie postrzegać należy raczej jako niezależne od stanu, za to produkujące rezultaty o charakterze losowym. Najczęstsze kryteria poszukiwania tego rodzaju funkcji dotyczyć mogą rozkładu produkowanych wartości losowych (patrz np. Mas1996). Jeżeli repozytorium ukierunkowane jest na ten rodzaj poszukiwań, można zrealizować go zastępując operator równości w przypadku użycia operatorem rozkładu (oznaczającym np. równomierny rozkład produkowanych wartości lub rozkład Gaussa o określonych lub nieokreślonych parametrach). W takim przypadku

repozytorium musi mieć moduł zdolny do wielokrotnego wykonywania żądanej funkcji i na podstawie obserwowanych wyników oceny ich rozkładu.

6.2. Wyszukiwanie jednostek programowych oparte o scenariusze użycia

Wszystkie metody wyszukiwania w rozdziale 5, omówione do tej pory miały jedną wspólną wadę, która ograniczała bardzo mocno zakres ich praktycznego użycia. Mianowicie, co wspomniano już wielokrotnie wcześniej, nie uwzględniały faktu posiadania przez system lub jednostkę nadrzędną stanu. W tym rozdziale pokazane zostanie podejście, stanowiące uogólnienie przedstawionych poprzednio. W największym skrócie uogólnienie to sprowadza się do rozszerzenia przedstawionych wcześniej definicji, tak aby uwzględniały sekwencję instrukcji, zamiast pojedynczych wywołań. Jeżeli sekwencja taka zostanie wykonana na pojedynczej instancji obiektu, za pomocą asercji będących częścią sekwencji można monitorować zachowanie stanu obiektu, upewniając się przy tym, że jest ono zgodne z wymaganiami użytkownika zapisanymi w postaci zapytania.

6.2.1. Definicje podstawowe

Dalsze rozważania wymagają pewnego uściślenia definicji stosowanych dotychczas w sposób intuicyjny. I tak, *jednostką programową U* nazywać będziemy w kontekście tego rozdziału klasę lub komponent[8], mogącą podlegać instancjowaniu (tworzeniu egzemplarza o unikalnej tożsamości). Tak zdefiniowana jednostka nie może posiadać stanu. Oczywiście, łatwo zauważyć, że w rzeczywistości np. klasa może posiadać stan, reprezentowany przez pole lub pola statyczne (przynależne do klasy, a nie jej konkretnej instancji). Ze względu jednak na uproszczenie dalszych rozważań, przyjęto, że pola statyczne należą do instancji podobnie jak pola instancyjne. Założenie takie jest zgodne z rzeczywistością, jeżeli mamy do czynienia wyłącznie z jedną instancją danej jednostki programowej (jak to ma miejsce w przypadku klasycznych komponentów [Szy2001]). Ponieważ jednak taka sytuacja ma właśnie miejsce w omawianym procesie wyszukiwania wewnątrz mechanizmu wykonawczego repozytorium, zmuszeni jesteśmy uproszczenie takie zaakceptować.

[8] Na mocy założeń poczynionych we wstępie, komponent traktować będziemy jako niezależnie instalowany zbiór klas, posiadający jednolity interfejs.

Jednostka programowa, stanowi z punktu widzenia tego rozdziału jedynie definicję interfejsu, zatem może być traktowana jako zbiór funkcji.

$$U = \{F_1^U, F_2^U, ..., F_n^U\}$$

Zauważmy, że w oznaczeniu funkcji umieszczono nazwę jednostki programowej, co podkreśla przynależność funkcji do tej jednostki.

Instancją jednostki programowej jest pewien byt (obiekt) posiadający fizyczną, unikalną tożsamość. W dalszej części rozdziału instancja jednostki U oznaczane będą przez I_m^U, gdzie m jest indeksem instancji i stanowi o jej unikalnej tożsamości. Jeżeli tożsamość instancji nie będzie istotna dla rozważań, przyjmiemy uproszczone oznaczenie I^U. Przyjmiemy, że

$$I_m^U = \{S_m, F_1^U, F_2^U, ..., F_n^U\}$$

co oznacza, że instancja posiada przynależny do niej stan S_m, oraz posiada interfejs zgodny z interfejsem jednostki programowej. Z tego powodu jednostkę programową U nazywać będziemy często typem instancji I^U.

Każda z funkcji ma postać podobną do przedstawionej w rozdziale 5.1

$$F_n^U : A_1^n \times A_2^n \times ... \times A_3^n \times S_m \rightarrow S_m^{'} \times (B^n \cup \{\varepsilon\})$$

Zauważmy, że stan instancji jednostki programowej należy traktować jako jeden z argumentów funkcji. Na skutek działania funkcji, może on też otrzymać nową wartość, stąd należy go jednocześnie traktować jako fragment wyniku działania funkcji. W odróżnieniu od podejść nie rozpatrujących stanu, może więc zdarzyć się sytuacja, w której B jest zbiorem pustym.

Funkcje mogą wykazywać cztery rodzaje zależności od stanu instancji. Mogą być:

- Niezależne od stanu. Funkcje takie często nazywane są funkcjami statycznymi, choć pojęcia te nie są do końca tożsame.

- Zależne od stanu, lecz nie wpływające na stan. Stan instancji może być traktowany jako dodatkowy parametr funkcji.

- Wpływające na stan, lecz nie zależne od stanu.

- Wpływające na stan i zależne od stanu.

Dostęp do stanu instancji z zewnątrz możliwy jest wyłącznie za pośrednictwem funkcji tej instancji. Wynika stąd, że powyższa definicja jednostki programowej i instancji uwzględnia wyłącznie funkcje oznaczone z punktu widzenia hermetyzacji jako publiczne. Wynika to z założenia, że funkcje o innym poziomie kontroli dostępu, takie jak publiczne czy prywatne, nie są istotne w procesie ponownego użycia, gdyż nie mogą zostać w żaden sposób wywołane. I ponownie, założenie to jest prawdziwe tylko w części. W przypadku funkcji chronionych[9] możliwe jest odwołanie się do nich przez dziedziczenie z klasy jednostki podlegającej ponownemu użyciu. Ten rodzaj ponownego użycia zaliczyć można do ponownego użycia typu białej skrzynki (patrz rozdział 2). Ze względu jednak na swoją specyfikę nie będzie on tutaj rozpatrywany.

W rozdziale założono również, że funkcja sama w sobie nie może posiadać stanu wewnętrznego. Nie jest to zawsze prawda (np. statyczne zmienne lokalne w C++ [Str1995]), jednak przypadek posiadania przez funkcję stanu jest na tyle rzadki, że można go pominąć dla uproszczenia rozważań, nie odbiegając przy tym zanadto od zastosowań praktycznych.

6.2.2. Wyczerpujące przeszukiwanie przestrzeni scenariuszy

Rozważania na temat podejścia opartego na scenariuszach użycia rozpocznijmy od przypadku najprostszego, w którym przestrzeń dostępnych scenariuszy wykonania przeglądana jest w sposób wyczerpujący. Wprowadźmy następujące definicje:

Wywołaniem funkcji F_n^U dla Instancji I^U w stanie S_m nazywamy odwołanie do funkcji w postaci $F_n^U(a_1 \in A_1^n, a_2 \in A_2^n, ..., a_k \in A_k^n, S_m) = b \in B^n \cup \{\varepsilon\}$ (patrz też rozdział 5.2.2.1). Bezpośrednio obserwowalnym rezultatem takiego wywołania jest jego wynik, $b \in B^n \cup \{\varepsilon\}$ (pamiętajmy, że B może być zbiorem pustym). Rezultatem dodatkowym, jest fakt, iż wywołanie takie przeprowadza instancję I^U ze stanu S_m w stan $S_m^{'}$. Rezultat ten nie może zostać zaobserwowany inaczej, jak przez obserwacje wyników dalszych wywołań

[9] Użyto tu terminologii obiektowej do określania poziomów dostępu do funkcji

funkcji uzależnionych w jakiś sposób od stanu jednostki. Specjalnym przypadkiem wywołania funkcji jest wywołanie, którego spodziewany rezultat nie został wyspecyfikowany. Oznacza on tyle, co akceptacje dowolnego, poprawnego rezultatu wywołania.

Scenariuszem użycia jednostki U, nazywać będziemy uporządkowaną liniowo sekwencję wywołań. *Normalnym scenariuszem użycia*, nazywać będziemy scenariusz użycia rozpoczynający się od wywołania $F_n^U(a_1 \in A_1^n, a_2 \in A_2^n, ..., a_k \in A_k^n, S_0) = b \in B^n \cup \{\varepsilon\}$,

czyli wywołania dla stanu określającego stan początkowy jednostki.

Problem określenia stanu początkowego nie jest do końca jednoznaczny. W pracy tej przyjęto założenie, że jest to stan, który otrzymuje obiekt na skutek utworzenia go w sposób domyślny (np. dla klasy na skutek utworzenia jej instancji za pomocą domyślnego konstruktora). Niestety, nie wszystkie obiekty muszą posiadać konstruktory domyślne. Pojawia się wtedy nie tylko problem z określeniem stanu początkowego, ale w ogóle problem z automatycznym zainstancjowaniem obiektu w celu rozpoczęcia procesu wyszukiwania. Pewnego rozwiązania tego problemu można upatrywać w – podobnie jak to miało miejsce w podejściu kanonicznym – określeniu zbiorów wartości kanonicznych dla poszczególnych typów (w tym wypadku najlepiej jednoelementowych!) i utworzenia obiektu za pomocą konstruktora, któremu przekazano wartości kanoniczne jako argumenty. Niestety, jedna klasa może mieć wiele konstruktorów, co jeszcze bardziej zwiększa i tak już potężną ilość kombinacji do sprawdzenia. Co gorsza nie ma żadnej gwarancji, że konstrukcja obiektu za pomocą wartości kanonicznych powiedzie się. Do tego wszystkiego, konstruktor może nie być widoczny (w sensie hermetyzacji), a konstrukcja obiektu może wymagać np. odwołania do publicznej i statycznej metody klasy. Problem konstrukcji obiektów stanowi zatem jeden z najpoważniejszych problemów związanych z tym podejściem.

W opisywanym podejściu zapytanie użytkownika ma postać zbioru normalnych scenariuszy użycia. Scenariusz taki można traktować jako specyficzny (dość prosty) przypadek testowy dla poszukiwanej klasy, napisany jednak bez znajomości jej ostatecznej struktury, nazw metod ani nawet dokładnych typów parametrów. Specyfikuje on jednak precyzyjnie wymagania użytkownika, który żąda od poszukiwanej klasy przejścia podanego przez niego przypadku testowego. Przykładowe zapytanie służące do poszukiwania klasy reprezentującej stos liczb całkowitych mogło by np. wyglądać tak:

```
push(2); push(10); pop()=10; pop()=2;
```

Zapytanie można by również uzupełnić o specyfikację zachowania klasy w przypadku pobrania nadmiernej liczby elementów:

```
push(2); push(10); pop()=10; pop()=2; pop()=ε
```

W opisywanym tu (najprostszym) przypadku, wyszukiwanie jednostek programowych opisanych w taki sposób polega na wyczerpującym przeszukiwaniu przestrzeni możliwych dopasowań scenariuszy z jednoczesnym wykonaniem testu dla każdego dopasowania. Dla każdej z funkcji wymienionych w zapytaniu poszukiwana jest więc metoda, która może zostać dopasowana do zapytania w sposób analogiczny do tego, jak to opisano w rozdziale 5. Należy zwrócić uwagę, że wszystkie dopasowania muszą być znalezione w odniesieniu do metod jednej klasy. Jeżeli w ramach rozpatrywanej klasy nie udaje się odnaleźć dopasowania dla którejkolwiek z metod, jest ona odrzucana. Zakłada się przy tym, że w ramach pojedynczego zapytania, dla metody o określonej nazwie może zostać znalezione dokładnie jedno odwzorowanie. Oznacza to, że nie rozpatruje się przypadków, w których pojedyncza nazwa po stronie zapytania użytkownika odnosić się może do dwóch lub więcej nazw metod w rzeczywistej klasie. Nakładane jest także ograniczenie dotyczące odwzorowania parametrów metod (patrz rozdziały 5.2.1, 5.2.2) w zapytaniu na rzeczywiste metody. Mianowicie permutacja odwzorowująca musi być stała dla dopasowań metod, które w zapytaniu występują kilkakrotnie. Ograniczenie to spowodowane jest (zupełnie naturalnym) założeniem, że znaczenie parametrów nie zmienia się wraz z kontekstem wywołania metody (w tym wypadku wraz z zmianą stanu obiektu). Przyjrzyjmy się przykładowym dopasowaniom dla rozpatrywanego zapytania i klasy java.util.Stack ze standardowego pakietu języka Java (permutacje odwzorowujące argumenty nie zostały uwzględnione, ze względu na to, że żadna z metod nie ma więcej niż jednego argumentu).

a) `push(int) = void push(Object element), pop() = Object pop()`

b) `push(int) = void push(Object element), pop() = int size()`

Jak widać, dopasowanie scenariuszy jest więc niczym więcej, jak jednoczesnym dopasowaniem wszystkich metod biorących udział w scenariuszu do metod w obrębie jednej klasy. Techniki dopasowania pojedynczych metod zostały szczegółowo opisane w rozdziale 5.

Dopasowanie a) jest wynikiem zastosowania prostych technik dopasowania metod jednoargumentowych, z uwzględnieniem naturalnego wnioskowania

$$castable(\text{int}, Integer) \wedge is\text{-}a(Integer, Oject) \Rightarrow castable(\text{int}, Object)$$

Zwróćmy uwagę, że na etapie znajdowania dopasowań, wiele z nich może być zupełnie pozbawionych sensu. Ma to np. miejsce w przypadku przedstawionego rozwiązania b. Rozwiązania te zostają zwykle odrzucane na etapie wykonywania przypadków testowych reprezentujących zapytanie użytkownika dla rzeczywistych danych. Dla dopasowania b zostanie więc wykonana sekwencja

```
push(2) ; push(10); size()=10,
```

która ze względu na fałszywość ostatniego warunku zostanie odrzucona (metoda size() zwróci w rezultacie 2 zamiast spodziewanych 10)

O ile w przypadku technik przeszukiwania opisanych wcześniej proces wyszukiwania dopasowań i ich weryfikacji dało się w miarę efektywnie zrównoleglić, uzyskując redukcję drzewa poszukiwań, o tyle w przypadku techniki opisywanej w tym rozdziale nie jest to już tak proste. Wynika to z faktu posiadania przez obiekt stanu, który ulega zmianom w miarę wykonywania przypadku testowego. W związku z tym ewentualne nawroty w algorytmie znajdującym dopasowanie wymagały by selektywnego wycofania niektórych ze zmian poczynionych w stanie obiektu. W ogólności jest to niemożliwe bez kopiowania obiektu przed każdą zmianą, co miało by bardzo negatywny wpływ na wydajność. Aby jednak zredukować drzewo poszukiwań już na wczesnych etapach procesu, można zastosować częściową weryfikację znajdowanych dopasowań, polegającą na weryfikacji wyniku pierwszego wykonania. Weryfikacja taka nie kosztuje właściwie nic (w sensie zasobowo – kosztowym), a może pomóc wielokrotnie przyspieszyć proces wyszukiwania.

6.2.3. Metody o nieskończonym lub nieprzewidywalnym czasie działania

Przyjrzyjmy się ponownie zapytaniu opisującemu stos:

```
push(2); push(10); pop()=10; pop()=2; pop()=ε
```

Zapytanie to ujawnia pewien kłopotliwy aspekt – co jeżeli poszukujemy blokującego stosu (lub kolejki FIFO) nadającego się do zastosowań typu producent – konsument? Jak

wyspecyfikować rezultat funkcji oznaczający blokadę? W poprzednich rozdziałach założono, że wszystkie funkcje zakończą swoje działanie w skończonym czasie. Jeżeli jednak niezakończenie działania jest w istocie pożądane, można postąpić na dwa sposoby:

- Jak już napisano w rozdziale 5. można potraktować zakończenie na skutek przekroczenia limitu czasu wykonania jako zwykły błąd wykonania, rozróżnienie jednostek blokujących od nie blokujących przenosząc do fazy przeglądania wyników
- Można wyróżnić specjalną klasę błędu, oznaczającą zakończenie na skutek przekroczenia limitu czasu wykonania.

6.2.4. Ograniczanie drzewa poszukiwań

Przeszukiwanie przestrzeni możliwych dopasowań scenariuszy użycia metodą wyczerpującą jest bardzo czasochłonne. Co prawda, wiele odcięć w przeszukiwanej przestrzeni powstaje na skutek niedopasowania typów, liczby argumentów czy wreszcie w przypadku optymalizacji omówionej na końcu rozdziału 6.2.2. na skutek wykonywania pierwszych metod scenariusza w domyślnym stanie obiektu. Niestety, odcięcia te wciąż mogą okazać się niewystarczające i w przypadku dużych zbiorów jednostek programowych przeszukiwanie ich metodą opartą o scenariusze użycie może być niewystarczająco efektywne dla zastosowań praktycznych. Problem zwiększa się jeszcze ba skutek zwiększania koniecznej do przeszukania przestrzeni rozwiązań na skutek udogodnień wyszukiwania opisanych w rozdziale 5. (w odniesieniu do wyszukiwania metodą kanoniczną), takich jak możliwość dopasowywania metod z różną liczbą argumentów, konieczność wnioskowania o możliwości rzutowania typów itp. Rozwinięcia te nie tylko dają się bezproblemowo przenieść na grunt przeszukiwania w oparciu o scenariusze użycia. Użycie takich udogodnień jak chociażby wspomniana możliwość automatycznego rzutowania typów podczas procesu dopasowywania jest koniecznością, jeżeli w ogóle mamy mówić o praktycznym zastosowaniu omawianej metody. Słabą wydajność metody potwierdzają obserwacje przeprowadzone z prototypowym repozytorium, którego budowa przedstawiona jest w rozdziale 8.1. Co prawda jego konstrukcja zupełnie nie była nastawiona na optymalizację wydajności (z założenia repozytorium służyć miało do badania selektywności i trafności odpowiedzi dla różnych podejść), jednak obserwowane spowolnienie widoczne nawet przy niewielkich zbiorach klas i metod można uznać za co najmniej niepokojące. Z tego powodu konieczne jest

wprowadzenie dodatkowych technik opisu i reguł wnioskowania, pozwalających wydatnie zredukować drzewo poszukiwań. Rozwiązania takie opisane są w następnych podrozdziałach.

6.2.4.1. Rodzaj zależności od stanu

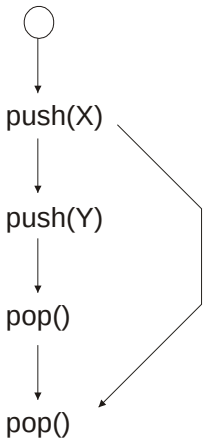
W rozdziale 6.2.1. pokazano cztery rodzaje zależności metody od stanu. Najprostszy, a zarazem całkiem skuteczny sposób ograniczenia przeszukiwanej przestrzeni dopasowań polega na przypisaniu każdej metodzie znacznika, określającego jeden z czterech rodzajów zależności od stanu jednostki nadrzędnej. Również od użytkownika żąda się określenia rodzaju zależności od stanu metod w scenariuszu będącym treścią zapytania. Następnie warunek zgodności metod należy rozszerzyć o warunek równości deklarowanych typów zależności od stanu.

Dodatkową zaletą tej metody, poza niewątpliwym zmniejszeniem liczby koniecznych do przeszukania rozwiązań, jest fakt, iż wymagane opisy jednostek programowych znajdujących się w repozytorium, można przy pewnym wysiłku pozyskać automatycznie przez analizę ich kodu źródłowego. Oczywiście rozwiązanie to wymaga obecności źródeł, lub co najmniej takiej postaci kodu dla której informacje te będą mogły być odtworzone (np. skompilowane klasy Javy umożliwiają dekompilację, należy jednak pamiętać, że rozbudowa repozytorium o takie mechanizmy raczej pozbawiona jest jakiegokolwiek praktycznego uzasadnienia)

Niestety, co jest cechą wspólną omawianych rozwiązań, może się zdarzyć, że zastosowanie tej techniki redukcji spowoduje odrzucenie dobrego rozwiązania. Wyobraźmy sobie klasę reprezentującą tablicę. Tablica ta ma jednak dodatkową funkcjonalność polegającą na zliczaniu liczby odwołań do poszczególnych elementów. Funkcja pobrania elementu zostanie więc przez twórcę tej klasy zakwalifikowana jako modyfikująca i zależna od stanu, natomiast użytkownik poszukujący implementacji zwyczajnej tablicy (do czego rozpatrywany komponent doskonale się przecież nadaje) określi metodę pobrania elementu jako zależną, lecz nie modyfikującą stan obiektu. W ten sposób klasa, która powinna być znaleziona, znaleziona nie zostanie. Problem taki może pojawiać się również w mniej akademickich przykładach, np. wszędzie tam, gdzie w grę wchodzi cachowanie obliczanych na podstawie stanu rezultatów. Do problemu tego wrócimy ponownie w rozdziale 6.2.4.3.

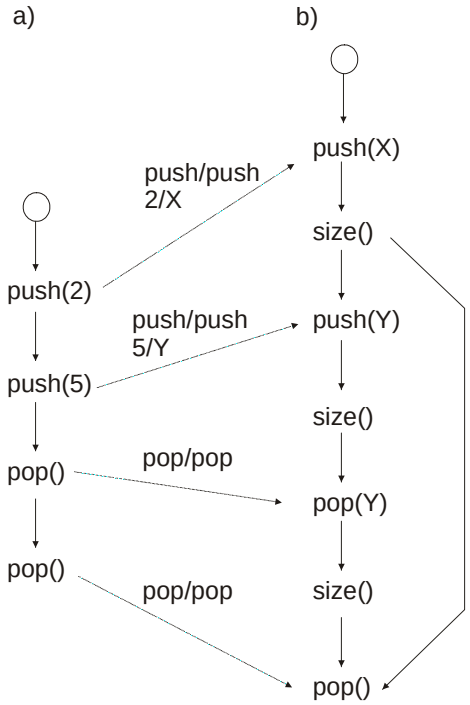
6.2.4.2. Standardowe scenariusze użycia

Jednym z rozpatrywanych podejść pozwalającym na znaczącą redukcję przestrzeni poszukiwań jest rozwiązanie, polegające na opisaniu każdej klasy w repozytorium za pomocą zbioru standardowych scenariuszy użycia tej klasy. Scenariusz taki reprezentowany jest jako częściowo uporządkowany zbiór wywołań funkcji (patrz rozdział 6.2.1), ułożonych tak, aby możliwie najpełniej reprezentować standardowe techniki użycia funkcji.



Rys. 6.1. Standardowy scenariusz użycia prostego stosu

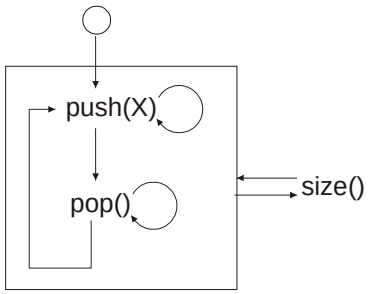
Rys 6.1. prezentuje standardowy scenariusz użycia dla prostej klasy stosu, posiadającej wyłącznie dwie metody. Częściowe uporządkowanie metod ułatwia dopasowanie do scenariusza w zapytaniu użytkownika. Dopasowanie scenariuszy przeprowadzać można fragmentami, pomijając nieobowiązkowe (po stronie repozytorium) funkcje nie zmieniające stanu (jeżeli opis tych funkcji uzupełniono o informację analogiczną do rozpatrywanej w rozdziale 6.2.4.1). Przykładowe dopasowanie między (trochę bardziej skomplikowanym) standardowym scenariuszem użycia stosu i zapytaniem użytkownika pokazane jest na rysunku 6.2.



Rys 6.2. Dopasowanie zapytania a) do standardowego scenariusza użycia b)

Wywołania metody size() mogły zostać pominięte w procesie dopasowania, na podstawie wiedzy, iż nie zmienia ona stanu obiektu[10]. Zapisy nad strzałkami oznaczają podstawienia wykonane w wyniku dopasowania.

Już dla tak prostego przykładu widać, że niektóre zapisy mogą sprawiać kłopoty. Metoda size() pobierająca rozmiar stosu może być wywołana właściwie w dowolnym momencie (w szczególności może być wywołana jako pierwsza lub ostatnia, czego nie uwzględniono na rysunku), nie ma jednak prostego sposobu zapisu tego faktu. Najogólniejszym podejściem byłoby więc zastąpienie częściowo uporządkowanego zbioru wywołań diagramem stanu, np. takim jak pokazano na rysunku 6.3.



[10] Oczywiście, można sobie wyobrazić inny zapis standardowego scenariusza użycia, który umieściłby wywołania metody size() na alternatywnych ścieżkach wykonania.

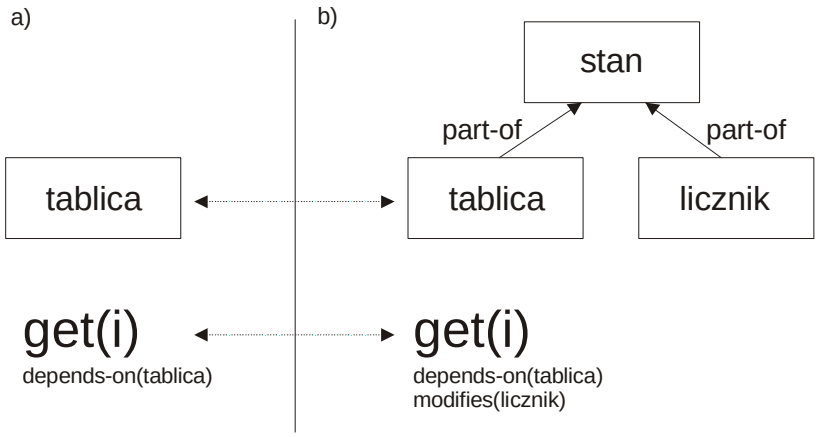
Rys 6.3. Diagram stanu opisujący sposób użycia stosu

Diagramy stanu są jednak w ogólnym przypadku dużo trudniejsze do dopasowywania. Co gorsza, jak widać na rys 6.3., niewiele wnoszą. Zgodnie bowiem z diagramem przedstawionym na tym rysunku już dla tak prostego przykładu właściwie wszystkie metody mogą zostać wywołane kiedykolwiek. Nie stanowi to zatem wielkiego postępu, a raczej krok w tył w stosunku do metody opisanej w rozdziale 6.2.1. Z tego powodu podejścia tego typu zaleca się stosować raczej do opisu wysokopoziomowych interfejsów związanych z procesami biznesowymi, niż do klas ogólnego przeznaczenia.

6.2.4.3. Opis struktury stanu jednostki

Powróćmy na chwilę do przykładu z tablicą zliczającą pobrania elementów z rozdziału 6.2.4.1. Problem ze znalezieniem klasy reprezentującej taką tablicę przy zastosowaniu techniki redukcji drzewa poszukiwań opisanej w tym rozdziale polegał na rozbieżności zdań o charakterze zależności od stanu obiektu funkcji pobrania elementu tablicy. Użytkownik poszukujący zwyczajnej tablicy określił funkcję pobrania jako nie modyfikującą stanu obiektu. Konstruktor tablicy zgodnie z prawdą określił funkcję pobrania jako modyfikującą stan, gdyż jako efekt uboczny zwiększała ona licznik pobrań. Nieporozumienie – prowadzące do odrzucenia klasy tablicy na etapie poszukiwania – nie jest w tym wypadku spowodowane niezrozumieniem przez użytkownika zasad działania poszukiwanej jednostki. Wynika ono z faktu, iż użytkownik zainteresowany jest pewnym podzbiorem funkcjonalności jednostki programowej. Co więcej, funkcjonalność którą zainteresowany jest użytkownik dotyczy nie tylko nie wszystkich metod rozpatrywanej klasy tablicy, ale przede wszystkim jedynie wydzielonej części stanu obiektu. Powodem nieporozumienia jest więc w tym wypadku nadmierne uproszczenie spojrzenia na stan obiektu jako całość.

Pewne rozwiązanie tej kwestii może przynieść dekompozycja stanu obiektu na niezależne części (w tym wypadku tablica oraz licznik pobrań) . Między różnymi częściami wprowadzić można relację zawierania (part-of). Tak opisaną strukturę stanu wprowadza się do repozytorium wraz z jednostką programową. Z kolei użytkownik, konstruując zapytanie dostarcza swój oczekiwany model stanu. Co więcej, przy każdej metodzie zaznacza rodzaj zależności od konkretnego elementu stanu (jeżeli zależność taka ma miejsce). Opis struktury stanu po stronie repozytorium oraz zapytania użytkownika przedstawiony jest na rys. 6.4.



Rys. 6.4. Opis struktury stanu po stronie zapytania (a) i repozytorium (b)

Praktyczne zastosowanie tej techniki ma jednak kilka ograniczeń. Przede wszystkim nakłada na wprowadzającego jednostkę do repozytorium konieczność zdefiniowana struktury stanu. Wymaga też znacznej rozbudowy algorytmów wyszukiwania w celu uwzględnienia wcześniejszej fazy dopasowania do siebie struktur stanu. Po wygenerowaniu możliwych odwzorowań tych struktur (z uwzględnieniem relacji zawierania) konieczne jest też oczywiście uwzględnienie tych odwzorowań podczas dopasowywania do siebie metod. Należy przy tym zwrócić uwagę, iż metoda modyfikująca pewien fragment stanu w oczywisty sposób modyfikuje także wszystkie fragmenty stanu, w których ów fragment się zawiera. Algorytm tego typu (podobnie zresztą jak inne algorytmy opisane w tej pracy) dużo łatwiej (i czytelniej) jest zaimplementować w sposób deklaratywny, gdyż zasady znalezienia właściwego odwzorowania czy to struktur stanów, czy metod mogą być w kategoriach deklaratywnych opisane w sposób niezwykle krótki i intuicyjny.

6.3. Analiza złożoności obliczeniowej

Łatwo wykazać, że decyzyjny odpowiednik problemu przeszukiwania przedstawioną metodą jest NP-zupełny. W przypadku funkcji niezależnych od stanu, wynika to z NP-zupełności następującego problemu:

Dana jest funkcja w postaci $F : A \times A \times ... \times A \rightarrow A$. Dane jest także zapytanie użytkownika o funkcję n-argumentową, której wszystkie argumenty oraz rezultat są typu *A.* Sprawdzić, czy istnieje permutacja argumentów, dla której funkcja *F* pasuje do zapytania użytkownika. Zakładamy przy tym, że sprawdzenie wyniku funkcji F dla zadanego zbioru argumentów odbywa się w czasie stałym.

Szkic dowodu: Przynależność problemu do klasy NP może zostać łatwo wykazana na podstawie modelu NDTM przedstawionego w [Bła1979] oraz faktu, że każda z możliwych permutacji może zostać sprawdzona w czasie wielomianowym.

Dowód NP-trudności przeprowadzimy wykazując wielomianową transformację problemu spełnialności wyrażeń boolowskich w normalnej postaci kanonicznej do omawianego problemu.

Niech będzie dany problem spełnialności pewnego wyrażenia boolowskiego $BE(a_1, a_2, ..., a_n)$, gdzie $a \in \{0,1\}$. Utwórzmy funkcję

$$F(x_1, x_2, ...x_n, x_{n+1}, ..., x_{2n}) = BE(x_1, x_2, ..., x_n), \text{ gdzie } x_k \in \{0,1\}$$

Jak widać, odpowiedź na zapytanie użytkownika $F(0,0,...0,1,1,...1) = 1$ jest równoznaczna pytaniu, czy istnieje takie przyporządkowanie wartości 0 albo 1 do zmiennych wyrażenia *BE*, że równanie to przyjmuje to wartość 1. Ponieważ transformacja dziedzin problemów zachowuje co najwyżej wielomianowy przyrost ich rozmiaru, wynika stąd, że gdyby przedstawiony problem miał rozwiązanie wielomianowe, miałby je również decyzyjny problem spełnialności wyrażeń boolowskich, który na mocy twierdzenia Cooka [Koś1997] jest NP-zupełny.

Podobne rozumowanie można przeprowadzić w wypadku poszukiwania funkcji zależnych od stanu. NP-trudność problemu wynika tu w sposób oczywisty z powyższego twierdzenia. Można jednak wykazać, że problem (odpowiednik decyzyjny) poszukiwania funkcji zależnych od stanu jest NP-zupełny nawet przy założeniu, że wszystkie funkcje są bezargumentowe. Przynależność problemu do klasy NP można wykazać analogicznie jak dla przypadku funkcji niezależnych od stanu. NP-trudność może zostać wykazana przez wykazanie wielomianowej transformacji problemu spełnialności wyrażeń boolowskich do omawianego problemu:

Dana jest jednostka programowa $U = \{F_1^U, F_2^U, ..., F_n^U\}$, przy czym $\forall_{k<n} : F_k : S_U \to S_U$' oraz $F_n : S_u \to \{0,1\}$. Oznacza to, że każda z funkcji danej jednostki jest bezargumentowa, a jedynym rezultatem jej działania jest zmiana stanu jednostki. Zakładamy również, że istnieje jedna dodatkowa funkcja F_n dająca możliwość obserwacji stanu jednostki *U*. Dane

jest również zapytanie użytkownika w postaci scenariusza użycia, takiego jak to opisano w rozdziale 6.2. Sprawdzić, czy jednostka U pasuje do zapytania użytkownika.

Transformacja wielomianowa od problemu spełnialności wyrażeń boolowskich opiera się na podobnej zasadzie co transformacja do problemu poszukiwania funkcji niezależnych od stanu. Niech będzie dany problem spełnialności pewnego wyrażenia boolowskiego $BE(a_1, a_2, ..., a_n)$, gdzie $a \in \{0,1\}$. Utwórzmy jednostkę programową

$$U = (S_{a_1}, S_{a_2}, ..., S_{a_n}, F_1, F_2, ..., F_{2n}, F_{2n+1})$$

taką, że $\forall_{k \le 2n} : F_{\lfloor k/2 \rfloor} : S_{a_{\lfloor k/2 \rfloor}} \to k \bmod 2$ oraz $F_{2n+1} = BE(S_{a_1}, S_{a_2}, ..., S_{a_n})$. Niech zapytanie użytkownika ma postać scenariusza użycia

$$F_1, F_2, ..., F_n, F_{n+1} = 1$$

Sprawdzenie, czy dana jednostka programowa pasuje do tak sformułowanego scenariusza użycia jest więc jednoznaczne z odpowiedzią na pytanie, czy istnieje takie przyporządkowanie wartości 0 albo 1 do zmiennych wyrażenia *BE*, że równanie to przyjmuje to wartość 1. Ponieważ transformacja dziedzin problemów, podobnie jak poprzednio, zachowuje co najwyżej wielomianowy przyrost ich rozmiaru, wynika stąd, że gdyby przedstawiony problem miał rozwiązanie wielomianowe, miałby je również decyzyjny problem spełnialności wyrażeń boolwskich. Problem jest zatem NP-zupełny.

6.4. Ocena efektywności

Przeszukiwanie oparte na dynamicznym pozyskiwaniu danych o jednostkach programowych pozwala rozwiązać wiele problemów pojawiających się w podejściu opartym na opisie kanonicznym. Jednocześnie zachowuje, a wręcz uwypukla najważniejszą cechę przedstawianych w tej pracy metod: pozwala uniezależnić sposób wyszukiwania od intencji twórcy jednostki programowej czy samego repozytorium. Pozwala na to w dużo większym stopniu, uwalnia bowiem użytkownika od ścisłych ograniczeń związanych ze zbiorami wartości kanonicznych, będących przecież częścią repozytoriów a nie samego zapytania. Dodatkowo, podejście to w większości wypadków nie wymaga żadnego dodatkowego opisu jednostek programowych wprowadzanych do repozytorium.

Odmiennym zagadnieniem jest selektywność przedstawianej metody. Jest ona bezspornie bardzo duża, co stanowi zarówno zaletę jak i wadę. Metoda stosowana bez żadnych rozszerzeń (takich jak chociażby możliwość dopasowywania funkcji o różnej liczbie argumentów) może być zdecydowanie za bardzo selektywna. Będzie ona zatem odrzucać rozwiązania potencjalnie użyteczne. Z drugiej strony właściwie nie zdarza się, z wyjątkiem uproszczonych do przesady zapytań lub celowo przygotowanych bliźniaczo podobnych jednostek programowych, otrzymać w rezultacie jednostki programowe, których działanie nie jest zgodne z intencją wydającego zapytanie (z intencją, gdyż w oczywisty sposób wszystkie jednostki są zgodne z zadanym zapytaniem, nie oznacza to jednak, że zapytanie było w stanie oddać intencję użytkownika). Eliminuje to szum informacyjny, znacząco skraca fazę przeglądania repozytorium. Dodatkowo, tak doskonałą selektywność pozwala myśleć o metodzie także w kontekście automatycznego wyboru implementacji.

Przedstawiana metoda ma niestety również sporo wad. Są to wspomniane już wcześniej bardzo silne związanie repozytorium z konkretną technologią, trudności w konstrukcji modułu wykonawczego czy wreszcie kłopoty z tworzeniem instancji bardziej złożonych klas na potrzeby wyszukiwania. Oczywisty jeż tek fakt, że metoda na obecnym etapie rozwoju nie jest w stanie sprostać złożoności niektórych repozytoriów. Chociażby zastosowanie tego rodzaju wyszukiwania do bibliotek typu MFC jest zupełnie nie do zrealizowania. Wynika to chociażby z trudności jakie napotyka się próbując testować interfejsy użytkownika. A że przedstawiana metoda jest w praktyce bardzo podobna (pod niektórymi względami) do testowania automatycznego, wszelkie obszary trudne dla testów automatycznych są jednocześnie trudne dla omawianej metody.

Największą jednak wadą przedstawianej metody jest złożoność obliczeniowa. Łatwo pokazać (patrz rozdział 6.3.), że problem poszukiwania jest NP-trudny. Na szczęście, w praktyce przestrzeń poszukiwań jest mocno ograniczana różne czynniki, a wartości zmiennych względem których złożoność rozwiązań jest wykładnicza zwykle nie osiągają zbyt dużych wartości (liczba różnych metod w przypadku testowym, liczba argumentów metody). Warto tu podkreślić, że przy podejściu opartym na opisach kanonicznych już sam rozmiar opisu był wykładniczy względem liczby argumentów funkcji, ze względu jednak na fakt, że w praktyce liczba argumentów zazwyczaj nie przekracza czterech, podejście to jak najbardziej nadaje się do stosowania w praktyce.

W celu doświadczalnego zbadania efektywności przeszukiwania przy pomocy metody opartej na dynamicznym pozyskiwaniu danych, przeprowadzono prosty eksperyment. Dziewięciu programistów zostało poproszonych o wypełnienie ankiety, której treść przedstawia załącznik D. Podsumowując wyniki tej ankiety, należy stwierdzić, że właściwie w każdym przypadku poszukiwana jednostka programowa została poprawnie i jednoznacznie zidentyfikowana(!). Rozgraniczenie na bardziej i mniej szczegółowe przypadki użycia okazało się nie grać roli, gdyż tylko dwie osoby wypełniły rubryki „b", a ponadto dodatkowe wpisy nie wpływały na wynik zapytania do repozytorium. Jedyne problemy dotyczyły klasy łańcucha (przypadek 2c,d) oraz funkcji abs (przypadek 1b). Łańcuch (w repozytorium: klasa java.lang.String) jako obiekt typu immutable, wymagał dodatkowego założenia, że pierwsze wywołanie metody może zostać przyporządkowane do konstruktora, bowiem 80% przypadków testowych rozpoczynało się metodą set(„xyz"). Dwie osoby nie odnalazły łańcucha w ogóle, jedna ze względów notacyjnych (zapis ABCD.substring(1,3) = ABC), druga ze względu na założenie o statyczności metody (zapis substring('ABCD',1,3)='ABC'). Problem z funkcją abs dotyczył jednej osoby i wynikał z nadmiernej oszczędności w zapytaniu (przypadek a) abs(1)=1), przypadek b) abs(1)=1, abs(-1)=1 – w obydwu przypadkach brak jednoznacznej identyfikacji, funkcja abs znajduje się jednak w wyniku zapytania).

7. INTEGRACJA Z TEST-DRIVEN DEVELOPMENT

Opisywane w rozdziale 6.2. podejście do wyszukiwania jednostek programowych w oparciu o scenariusze użycia może zostać łatwo rozszerzone z sekwencji wywołań (scenariusza użycia) na dowolne przypadki testowe (patrz też rozdział 8). W ten sposób podejście to można łatwo zintegrować z wytwarzaniem oprogramowania metodą *test-first coding* czy *test-driven development*.

Wiele popularnych dziś lekkich metodyk wytwarzania oprogramowania, takich jak chociażby XP [Bec1999] zaleca tworzenie oprogramowania metodą *test-first coding*. Polega ono na wytwarzaniu testów automatycznych przed napisaniem kodu, który testy te miałyby sprawdzać. Kent Beck stwierdza nawet, że żadna funkcjonalność nie powinna nigdy zostać dopisana bez uprzedniego dostarczenia przypadku testowego, który zawiedzie[11]. Podkreśla się, że *test-frist coding* nie jest techniką testowania. Jest ono raczej techniką analizy, gdyż

[11] Dosłownie: *Never write a line of functional code without a broken test case*[Bec2004]

pozwala skupić się na spodziewanych rezultatach oraz spodziewanym zakresie funkcjonalności zanim powstanie choć jedna linia kodu. Według Becka jest ono również techniką projektowania, gdyż stanowi źródło wielu pytań typowo projektowych (chociażby dotyczących interfejsów czy relacji między klasami) jeszcze we wczesnych fazach wytwarzania oprogramowania. Wreszcie w efekcie stosowania *test-frist coding* otrzymujemy nie tylko kod, ale pełen zestaw testów automatycznych – które przecież i tak należałoby w końcu napisać. Stosowanie *test-first coding* sprawia, że projekt systemu staje się bardziej czytelny, klasy słabiej powiązane a programista ma dużo większą i co więcej obiektywnie potwierdzoną pewność poprawnego działania systemu.

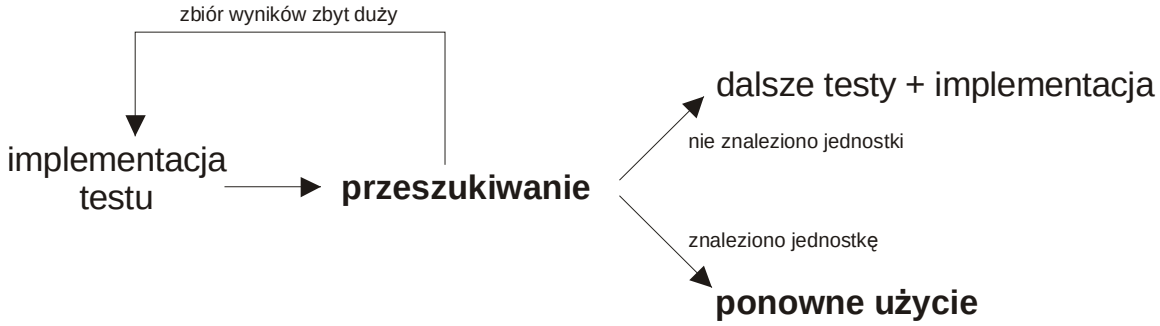
Szczególnym rodzajem *test-first coding* jest podejście zwane *test-driven development* [Bec2002]. Zakłada ono, podobnie jak w przypadku *test-first coding*, że żadna funkcjonalność nie powstanie bez uprzedniego napisania testu, który zawiedzie. Proces wytwarzania testów jest tu jednak z założenia poprzeplatany z procesem wytwarzania oprogramowania. Zaczynamy od napisania prostego testu, po czym tworzymy najprostszą możliwą implementację, która sprawia, że napisany test wykona się poprawnie. Następnie komplikujemy nieco przypadek testowy, próbując „złamać" bieżącą implementację testowanej funkcjonalności. Kiedy się to uda, poprawiamy braki w implementacji i tak dalej. *Test-driven development* zakłada więc, że tworzenie kodu i testów jest procesem iteracyjnym, o bardzo małych przyrostach. Fazy wytwarzania testów i kodu przeplatają się nawzajem, to jednak wytworzenie testu (a dokładniej – jego niepowodzenie) jest tym, co rozpoczyna cały proces i wyzwala następną iterację. Proces kończy się w momencie, kiedy uznajemy, że przypadki testowe wyczerpują pożądaną funkcjonalność implementowanego kodu i oczywiście wszystkie wykonują się bezbłędnie. Ten rodzaj pisania dla wielu osób wydaje się być bardziej intuicyjny, gdyż umożliwia dekompozycję procesu wytwarzania na małe, często bardzo małe fragmenty i lokalne cele do osiągnięcia. Ciekawą dyskusję z Maritnem Fowlerem na ten temat można znaleźć w [Ven2004].

Niestety, mimo iż niezaprzeczalnie korzystne dla jakości kodu, programowanie z uprzednim wytwarzaniem testów nie jest zbyt popularne wśród programistów. Z pewnością sytuacja ta ulega ciągłej poprawie, chociażby na skutek wprowadzenia nauczania tych technik programowania na studiach, wciąż jednak *test-first coding* traktowane jest raczej jako zbędne

obciążenie niż jako sposób poprawy jakości wytwarzanego oprogramowania[12]. Wynika to po części z ogólnie niechętnego podejścia do testów automatycznych, które w najlepszym razie traktowane są jako artefakt do wytworzenia „w wolnym czasie", a w praktyce często są albo szczątkowe albo nie powstają w ogóle. Stąd też w oczywisty sposób pojawia się tendencja do przenoszenia wytwarzania testów na koniec cyklu życia produktu.

W tym kontekście, wydaje się, iż dostarczenie programistom narzędzi umożliwiających wykorzystanie testów jednostkowych do wyszukiwania gotowej implementacji może wnieść znaczącą poprawę w nastawieniu do *test-first coding* ale również do idei usystematyzowanego ponownego użycia. Opisywana metoda wyszukiwania jednostek programowych w oparciu o testy jednostkowe doskonale bowiem integruje się z *test-driven development*. Zmodyfikowany w tym celu cykl pojedynczego kroku TDD zawiera dodatkowo fazę przeszukiwania repozytorium w poszukiwaniu jednostek programowych, które mogły by zostać użyte zamiast implementacji pożądanej funkcjonalności od nowa. Dzięki temu korzyść ze stosowania *test-first coding* widoczna jest natychmiast. Nie tylko gotowy test specyfikuje wymagania wobec tworzonej jednostki programowej. Już po napisaniu najprostszego przypadku testowego można uruchomić przeszukiwanie i ocenić rezultaty. Jeżeli zwrócony zbiór jednostek programowych jest zbyt duży, konieczne jest cofnięcie się do fazy tworzenia testu w celu dokładniejszej specyfikacji wymagań (rozbudowy przypadków testowych). Jeżeli w wyniku otrzymujemy zaledwie kilka (lub nawet żadnej) jednostek programowych i żadna z nich nie wydaje się nadawać do przewidywanych zastosowań, przechodzimy do fazy implementacji. Nie tracimy przy tym nic – w końcu i tak planowaliśmy implementować. Jeżeli jednak okaże się, że otrzymamy w rezultacie jednostkę, która może zostać użyta zamiast planowanej implementacji lub chociażby jako baza, którą po dostosowaniu można będzie wykorzystać (ponowne użycie typu białej skrzynki – patrz rozdział 2) odniesiona została znacząca korzyść.

[12] Opinie te są wynikiem obserwacji własnych jak i rozmów z wieloma osobami zatrudnionym w różnych firmach wytwarzających oprogramowanie na terenie Poznania, Warszawy i Zielonej Góry.



Rys 8.1. Zmodyfikowana iteracja TDD

Tworzenie testów niewielkimi krokami z jednoczesną oceną odpowiedzi repozytorium jest przy opisywanej metodzie dużo bardziej korzystne niż stworzenie wszystkich testów na raz i dopiero wtedy wydania zapytania do repozytorium. Rzadko bowiem zdarza się, że jednostka programowa z repozytorium może zostać wykorzystana bezpośrednio, a jeżeli nawet to jednostki o takim stopniu ogólności są na tyle często wykorzystywane podczas pracy, że programiści są w stanie identyfikować je nawet bez pomocy repozytorium. Z kolei w przypadku jednostek bardziej specjalistycznych, sytuacja w której jakakolwiek z posiadanych już jednostek będzie w stanie od razu przejść pożądane testy wydaje się mało prawdopodobna. Dlatego iteracyjne podejście do tworzenia testów pozwoli zidentyfikować jednostki zdolne realizować tylko fragmenty docelowej funkcjonalności, które jednak łatwo mogą zostać rozszerzone (chociażby przez dziedziczenie lub agregację) o brakujące cechy.

8. INTERKATYWNE REPOZYTORIUM

8.1. Założenia

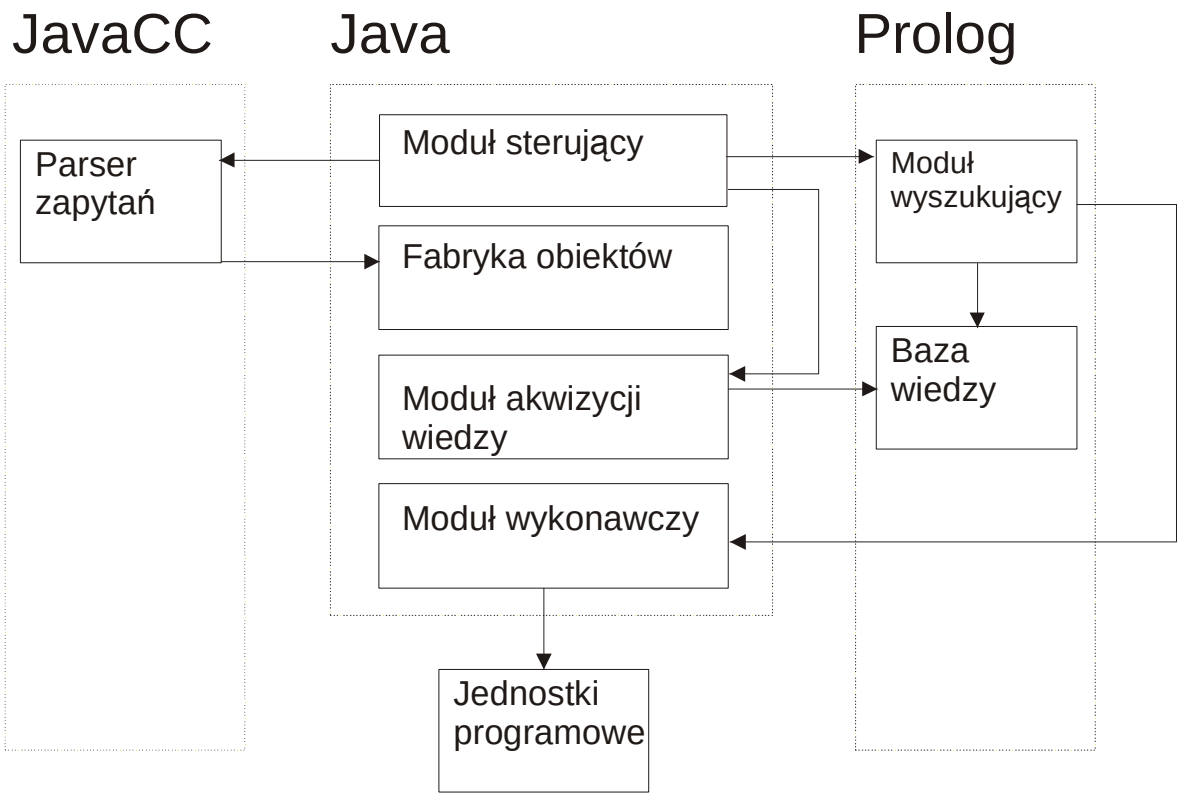
W celu oszacowania efektywności metod opisywanych w rozdziałach 5,6 zbudowałem prototyp repozytorium umożliwiający poszukiwanie jednostek programowych za pomocą tych metod. W założeniu repozytorium miało służyć do oceny efektywności, rozumianej w kategoriach selektywności poszukiwań. W przypadku optymalnym, oczekujemy zatem ze na statystyczne zapytanie użytkownika, system odpowiedzieć powinien wyłącznie jednostkami, które dadzą się użyć bezpośrednio zgodnie z jego (użytkownika) pierwotnymi intencjami. Co więcej, oczekujemy również, ze żadna z pożądanych jednostek nie zostanie pominięta w zaprezentowanym zbiorze wyników. Należy w tym miejscu podkreślić, ze efektywność takiego repozytorium nie może być mierzona w kategoriach adekwatności znalezionych jednostek do rzeczywistego zapytania użytkownika rozumianego wyłącznie przez pryzmat syntaktyki. Zgodność taka jest bowiem zagwarantowana przez poprawność

zaimplementowanego algorytmu. Interesować nas powinno raczej zachowanie repozytorium w zderzeniu z rzeczywistymi problemami, jako że jest ono odzwierciedleniem jakości języka użytego do opisu wymagań stawianych przez poszukującego wobec jednostki programowej.

Budowane repozytorium nie miało odpowiedzieć na pytanie dotyczące wydajności (rozumianej jako szybkość generowania odpowiedzi) opisywanych w pracy metod. Dlatego architektura systemu została wybrana raczej pod katem elastyczności modyfikacji samego jądra odpowiedzialnego za przeszukiwanie, co ułatwiło eksperymenty. Ponadto wybrana architektura miała na celu minimalizacje koniecznych do wprowadzenia danych o jednostce programowej. Pozwalało to na elastyczna modyfikacje zawartości repozytorium, bez konieczności wprowadzania dużej ilości danych ręcznie. Pod tym względem schemat pracy z repozytorium odpowiadał bardziej jednorazowego tworzeniu repozytorium na podstawie jednostek programowych przechowywanych dotychczas w innej formie, niż sukcesywnego tworzenia repozytorium w miarę powiększania się dorobku organizacji.

8.2. Architektura

Architektura repozytorium przedstawiona jest na rysunku 8.1.



Rys 8.1. Architektura prototypowego repozytorium

8.2.1. Jednostki programowe

Prototyp repozytorium został przystosowany do przechowywania informacji o klasach i metodach napisanych w Javie. Wybór tego właśnie języka podyktowany był istnieniem w ramach wirtualnej maszyny Javy mechanizmu refleksji, umożliwiającego pobranie informacji o interfejsach klas po ich załadowaniu. Informacja ta jest w zupełności wystarczająca dla opisywanych metod wyszukiwania, gdyż obejmuje parametry metod wraz z ich typami, typ rezultatu metody oraz jej widoczność (w sensie hermetyzacji). Możliwe jest także pobranie informacji dotyczących hierarchii dziedziczenia oraz implementowanych interfejsów. Ponadto Java dysponuje bogatą biblioteką standardową, zawierającą poza typowymi interfejsami systemowymi także wiele klas narzędziowych oraz przeróżnych struktur danych. Pozwoliło to wykorzystać istniejące standardowe klasy jako zawartość repozytorium, co było korzystne nie tylko ze względu na wymagany do przygotowania takiego repozytorium mniejszy nakład pracy, lecz przede wszystkim ze względu na możliwość sprawdzenia skuteczności metod na rzeczywistych, mających powszechne zastosowanie jednostkach programowych.

8.2.2. Baza wiedzy

Baza wiedzy systemu została zbudowana w oparciu o język Prolog[13]. Język ten pozwala na zapisanie wiedzy w wygodnym formacie, pozwalając jednocześnie na określenie reguł wnioskowania na tej wiedzy. Daje to możliwość chociażby zdefiniowania własności relacji, np. przechodniości relacji dziedziczenia (is-a). Związki pomiędzy jednostkami zostały ograniczone do relacji binarnych. Relacje te zapisane są nie w typowej postaci, w której nazwa relacji jest nazwą predykatu, a termy biorące udział w tej relacji są jego argumentami, np.

```
is-a(‘java.lang.Integer’, ‘java.lang.Object’).
```

ale w formie, w której nazwa relacji stanowi jeden z argumentów predykatu o ustalonej nazwie, np.

[13] Została wykorzystana platforma JIProlog [Jip2004] umożliwiająca ścisłą, dwustronną współpracę pomiędzy Prologiem a Javą.

```
rel(‘java.lang.Integer’, is-a, ‘java.lang.Object’).
```

Celem takiego zapisu było umożliwienie łatwego odzyskana wiedzy o rodzajach relacji zachodzących miedzy określoną jednostką a innymi jednostkami oraz o relacjach zachodzących w repozytorium w ogóle. W przypadku ewentualnej rozbudowy prototypu umożliwi to łatwą integrację z innymi technikami wyszukiwania, takimi jak np. sieci semantyczne i ewentualną budowę w pełni funkcjonalnego interaktywnego repozytorium.

W prototypowym repozytorium wyróżniono kilka rodzajów relacji, istotnych ze względu na implementowane metody wyszukiwania. Najważniejszą relacją repozytorium jest relacja dziedziczenia bezpośredniego. Jej istnienie wynika bezpośrednio z mechanizmu dziedziczenia między obiektami (jeżeli zachodzi między dwoma klasami lub dwoma interfejsami), oraz z faktu implementacji jakiegoś interfejsu (jeżeli zachodzi pomiędzy klasą a interfejsem). Relację tą oznaczono przez is-a-dir (ang. i*s-a direct*). Przykładowe użycie tej relacji to:

```
rel(‘java.lang.Integer’, is-a-dir, ‘java.lang.Number’).
```

Wiedza dotycząca tej relacji te może zostać pobrane automatycznie za pomocą mechanizmu refleksji. Ogólniejszą relacją od relacji is-a-dir jest relacja is-a oznaczająca dziedziczenie (niekoniecznie bezpośrednie). Relacja ta zdefiniowana jest w oparciu o relację dziedziczenia bezpośredniego. Przede wszystkim więc, jest ona nadzbiorem owej relacji:

```
rel(X, is-a, Z):-
    rel(X, is-a-dir, Z).
```

Dodatkowo należy dodać regułę, umożliwiającą rekursywną propagację tej relacji wzdłuż gałęzi drzewa dziedziczenia:

```
rel(X, is-a, Y) :-
   rel(Z, is-a-dir, Y),
   rel(X, is-a, Z).
```

Ze względu na zaimplementowane dalej algorytmy, należy również zaznaczyć, że

```
rel(X, is-a, X).
```

Najogólniejszą z omówionych dotychczas relacji jest relacja castable (patrz rozdział 5.1.1). Relacja ta opisuje możliwość transformacji danych jednego typu w inny typ. Należy zwrócić uwagę, że niezależnie od zapisów w prologu repozytorium musi posiadać taką wiedzę

na temat konwersji, aby miało możliwość jej wykonania podczas procesu przeszukiwania. Jeżeli wiedza ta nie wynika w sposób bezpośredni z mechanizmów języka (w tym przypadku Javy), odpowiednie informacje musza być zaszyte w repozytorium. W przypadku systemów bardziej złożonych informacja taka może być przechowywana w bazie wiedzy w postaci dodatkowych predykatów. W opisywanym prototypie informacja została zaszyta w kodzie samego repozytorium w module fabryki obiektów. Relacja castable zdefiniowana jest na bazie relacji is-a:

```
rel(X, castable, Y) :- rel(X, is-a, Y).
```

(wynika z tego m.in., że rel(X, castable, X)). Informacja ta może być dodatkowo uzupełniona przez ręczne wskazanie możliwości konwersji nie wynikającej bezpośrednio z relacji dziedziczenia. Mogą one dotyczyć np. prostych typów wbudowanych. Przykładowe wpisy to:

```
rel(int, castable, float).

rel(int, castable, 'java.lang.Integer').

rel('java.lang.Integer', castable, int).
```

Dodatkowo wyspecyfikowano regułę, informująca, iż jeśli pewien typ X daje się rzutować do innego typu Z, oraz jeżeli typ Z dziedziczy z typu Y, to X daje się również rzutować do typu Y.

```
rel(X, castable, Y):-
    rel(Z, is-a, Y),
    rel(X, castable, Z),
    X \= Z.
```

8.2.3. Automatyczne pozyskiwanie wiedzy

Po uruchomieniu repozytorium, rozpoczyna ono proces automatycznego pozyskiwania wiedzy o jednostkach programowych. W tym celu moduł pozyskiwania wiedzy poszukuje wszystkich pojęć będących klasami znajdujących się w bazie wiedzy przy pomocy zapytania

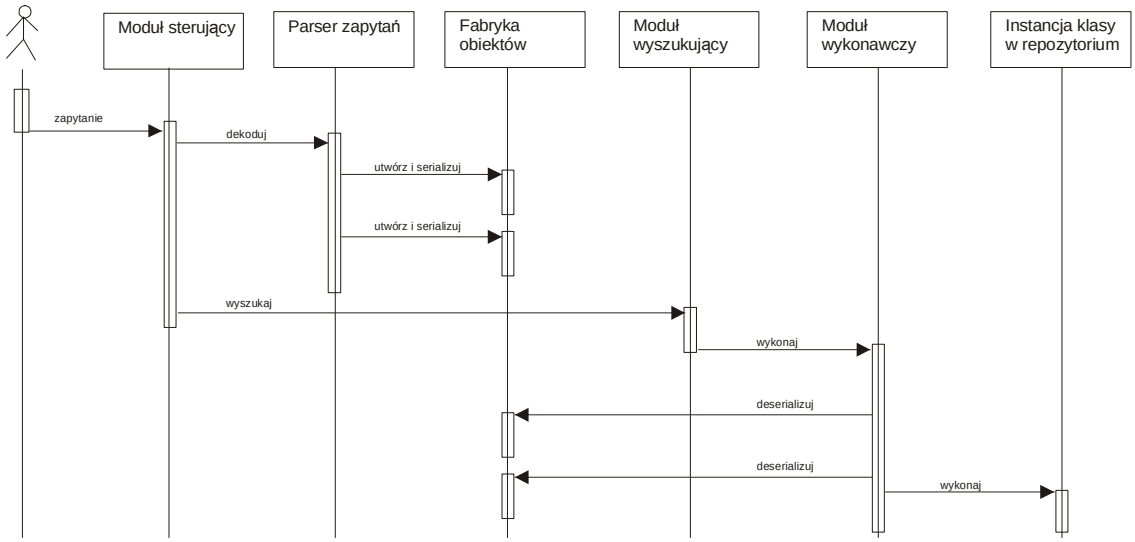
```
?- rel(X, is-a, class).
```

Dla każdej ze znalezionych klas, za pomocą refleksji pobierana jest lista metod publicznych, wraz z ich argumentami i typami rezultatów. Na tej podstawie baza wiedzy uzupełniana jest twierdzeniami opisującymi dokładnie typy i liczbę argumentów. Dla celów komunikacji z użytkownikiem oraz dla mechanizmu wykonawczego przechowywane są również nazwy klas i metod oraz właściwa kolejność argumentów.

Repozytorium pozwala również pozyskać automatycznie wiedzę o relacjach dziedziczenia, na skutek czego wprowadzane są do repozytorium dodatkowe relacje is-a.

8.2.4. Reprezentacja obiektów Javy w Prologu

Ponieważ argumentami wywołań metod w zapytaniu użytkownika są obiekty (podobnie jak wartościami oczekiwanymi) istnieje konieczność przekazania owych obiektów do mechanizmu wnioskującego (patrz też rozdział 8.2.5). Właściwe obiekty tworzone są na podstawie zapisu zapytania do repozytorium w module fabryki obiektów, nadzorowanej bezpośrednio przez parser gramatyki zapytania. Tak utworzone obiekty są następnie serializowane za pomocą standardowego mechanizmu zapisu obiektów. Otrzymany strumień jest dodatkowo kodowany przy pomocy kodowania base64, dzięki czemu z całą pewnością można go w dalszym przetwarzaniu traktować jako poprawny łańcuch znaków. Tak utworzony łańcuch jest przekazywany do mechanizmu wnioskującego jako stała w zapytaniu języka prolog. Na skutek sprzężenia zwrotnego pomiędzy prologiem a Javą, może on być następnie deserializowany w mechanizmie wykonawczym w celu utworzenia poprawnego argumentu wywołania i porównania wyniku rzeczywistego ze spodziewanym. Proces zapytania z uwzględnieniem serializacji i deserializacji obiektów pokazany jest na rysunku 8.2.



Rys. 8.2. Przetwarzanie zapytania z uwzględnieniem serializacji i deserializacji obiektów

8.2.5. Wyszukiwanie

Algorytm wyszukiwania został również zapisany w Prologu. Nie tylko znacząco skróciło to ilość wymaganego kodu, ale również pozwoliło eksperymentować z kodem w celu lepszego zrozumienia wpływu różnych mechanizmów na samo wyszukiwanie. Niezwykle istotny dla implementacji mechanizmu wyszukiwania był mechanizm sprzężenia z Javą oferowany przez platformę JIProlog, który pozwolił na uruchamianie metod przeszukiwanych jednostek za pośrednictwem modułu wykonawczego w ramach procesu wyszukiwania. Istotnym elementem modułu wnioskującego jest term opisujący istotę wzajemnego dopasowania metod[14]:

```
match(Class, Name, Argsno, [T|Types], [No | Mappings], [U |
Used]):-
    has-argsno(Class, Name, Argsno),
    rel(T, castable, MapT),
    has-arg(Class,Name,No, MapT),
    notcontains([U | Used], No),
    match(Class, Name, Argsno, Types,  Mappings, [No, U
|Used]).
```

[14] Ze względu na złożoność predykatów są one często przytaczane w formie uproszczonej. Brakuje chociażby warunku kończącego rekursję czy definicji niektórych użytych predykatów, jeżeli nie są one istotne.

Istotny jest fakt, że prototyp repozytorium wymaga zgodności w liczbach parametrów (zapis has-argsno(Class, Name, Argsno). Uwzględnia za to relację dziedziczenia podczas porównywania typów parametrów. W wyniku działania predykat ten unifikuje nazwę klasy i metody[15] z odpowiednimi wartościami, dla których zachodzi dopasowanie typów parametrów przy dowolnej ich permutacji (patrz rozdział 5.2).

Sama faza wyszukiwania (w oparciu o scenariusze użycia) podzielona została na dwie fazy: wyszukiwania dopasowań i weryfikacji rezultatów (wykonania). Dopasowywanie scenariuszy wykonywane jest przy użyciu opisanego wcześniej predykatu match, z tym, że dopasowywana jest cała lista pojedynczych wywołań metod, a nazwa klasy wiązana jest pomiędzy wywołaniami, co gwarantuje, że wszystkie metody jednego scenariusza zostaną dopasowane do tej samej klasy.

```
context-aware-match(Class, Handler, [ [Name, Argsno, Types,
Values, Result, Mappings] | Tail]):-
    match(Class, Name, Argsno, Types, Mappings),
    context-aware-match(Class, Handler, Tail).
```

Dodatkowo zaimplementowany jest mechanizm pozwalający zweryfikować rezultat wywołania pierwszej metody jeszcze podczas procesu dopasowywania.

Faza wykonania bazuje na dopasowanym przy pomocy predykatu context-aware-match scenariuszu. Wykonuje ona cały scenariusz, weryfikując przy okazji wyniki wykonywania poszczególnych jego kroków.

```
context-aware-exec(Class, Handler, [ [Name, Argsno, Types,
Values, Result, Mappings] | Tail]):-
    makeCall(Handler, Class, Name, Types, Values, Mappings,
Result),
    context-aware-exec(Class, Handler, Tail).
```

Predykat makeCall (implementacja nie zamieszczona) realizuje sprzężenie z modułem wykonawczym i pozwala na wywołanie żądanej metody w wirtualnej maszynie Javy.

[15] Nazwą metody w prototypie repozytorium jest jej sygnatura w postaci nazwa-metody(typ1, typ2, ... typn).

Oba predykaty połączone są w całość procesu wyszukiwania predykatem context-aware-search, dokonującym dodatkowo alokacji obiektu dla każdego wykonywanego scenariusza.

```
context-aware-search(Class, [X | Tail]):-
   context-aware-match(Class, [X | Tail]),
   allocate(Class, Handler),
   context-aware-exec(Class, Handler, [X | Tail]),
   deallocate(Handler).
```

Jak widać referencja do zaalokowanego obiektu przekazywana jest jako uchwyt. Zarządzaniem mapowania uchwytów do referencji w obrębie wirtualnej maszyny i odwrotnie zajmuje się moduł wykonawczy. Utrzymuje on tablicę asocjacyjną mapującą przydzielane z sekwencera uchwyty na referencje.

8.3. Interakcja i język zapytań

Język zapytań do repozytorium został tak stworzony, aby pogodzić trzy czynniki: prostotę wydawania zapytań, elastyczność w wyrażaniu wymagań co do poszukiwanej jednostki oraz prostotę gramatyki języka. Do przetwarzania zapytań na zapytania w prologu użyto generatora parserów gramatyk z wbudowanym analizatorem leksykalnym JavaCC [Jcc2004]

Scenariusz użycia ma postać zbioru wywołań (przypadków użycia) oddzielonych średnikami[16]

```
scenario = usecase ( <SEMICOLON> usecase)* [<SEMICOLON>]
```

Pojedyncze wywołanie metody ma postać

```
usecase = <IDENTIFIER>
        <LBRACKET> construct ( <COMMA> construct)* <RBRACKET>
        [ <EQ> result ]
```

Pierwszy identyfikator jest nazwą metody. Nazwa ta w gruncie rzeczy nie jest istotna dla samego mechanizmu wyszukiwania, jeżeli jednak nazwy dwóch metod są identyczne to podczas wyszukiwania ich nazwy również zostaną powiązane. Podobnie stanie się z permutacjami odwzorowującymi argumenty wywołania w zapytaniu w rzeczywiste

[16] Definicje symboli leksyklanych nie zostały zdefiniowane, jednak ich znaczenie jasno wynika z nazwy

argumenty metody. Construct jest zapisem tworzącym obiekt (lub wartość typu prostego) o następującej gramatyce:

```
construct = <IDENTIFIER> <LBRACKET> [params] <RBRACKET>
```

Identyfikator jest nawą typu (np. int lub java.lang.String). podane parametry są wartościami niezbędnymi w konstrukcji typu (np. int(2)). Zwykle są one odwzorowywane na wywołanie konstruktora, w przypadku typów prostych zastępuje się je odpowiednim przypisaniem po konstrukcji bezparametrowej.

Rezultat wywołania (opcjonalny) może być albo wartością typu construct, albo oznaczeniem wykonania błędnego („@exception”)

```
result = construct | <EX>
```

Parametry konstrukcji obiektu, to również lista oddzielona przeczinkami. Na liście tej mogą wystąpić albo inne konstrukcje typu construct, albo literał liczbowy, zmiennoprzecinkowy lub łańcuchowy.

```
params = param ( <COMMA> param) *

param = construct | value

value = <STRING_LITERAL> | <INTEGER_LITERAL> |
        <FLOATING_POINT_LITERAL>
```

Przykładowe zapytanie do repozytorium może zatem przyjąć postać:

```
Push(int(2)); Push(int(4)); Pop() = int(4); Pop = int(2);
```

lub w bardziej skomplikowanym przypadku, wymagającym konstrukcji obiektu przekazywanego jako parametr

```
Module(my.package.Complex(int(3), int(4))) = int(5);
```

8.4. Integracja z jUnit

W celu pełnego wsparcia dla TDD (patrz rozdział 8) podejście wyszukiwania w oparciu o scenariusze użycia zostało rozszerzone na dowolne przypadki testowe zapisane w postaci testów automatycznych jUnit [Jun2004] . Uogólnienie do dowolnych przypadków testowych oznaczało przede wszystkim odejście od liniowej struktury przepływu sterowania w zapytaniu

użytkownika, jak to miało miejsce w przypadku podejścia opartego o scenariusze użycia (patrz rozdział 6.2). Rozwiązanie to wykluczyło też, lub przynajmniej utrudniło stosowanie metod redukcji przestrzeni poszukiwań z rozdziału 6.2.4

Tworzone przez użytkownika przypadki testowe w przypadku tego podejścia nie różnią się znacząco od zwykłych przypadków testowych tworzonych za pomocą narzędzia JUnit. Najważniejszą różnicę stanowi fakt, że odwołania do metod obiektu testowanego, realizowane muszą być za pośrednictwem metody call z klasy SearchableTestCase z której dziedziczą wszystkie testy.

```
public Object call(String name, Object[] args, Object expectedResult)
      throws InstantiationException, IllegalAccessException
```

Jest to spowodowane faktem, że klasa obiektu, do którego odnoszą się wywołania w tworzonym przypadku testowym nie jest znana, podobnie jak nazwy metod czy kolejność ich argumentów. Wiedza ta zdobywana jest stopniowo, w trakcie wykonywania testów. Mechanizm ten opisany jest szczegółowo dalej. Oczywiście, można wyobrazić sobie podejście, w którym kod przypadku testowego tłumaczony jest na odpowiednie wywołania, jednak w celu przetransformowania zwykłego kodu Javy do postaci wymaganej przez mechanizm wyszukujący konieczna byłaby nie tylko analiza gramatyczna, ale również semantyczna kodu. Do prawidłowego działania mechanizm wyszukujący potrzebuje bowiem informacji o typie wyrażeń będących argumentami wywołania metod, a także spodziewanego typu rezultatu metody. Ponieważ tego rodzaju analiza kodu znacznie wykraczałaby poza zakres tej pracy, z podejścia tego zrezygnowano.

Pewną zaletą pisania przypadków testowych przy użyciu metody call jest brak uporczywych błędów kompilacji, wykazywanych przez środowiska programistyczne, które na bieżąco analizują poprawność kodu (np. Eclipse [Ecl2004]). W przypadku, w którym wyszukiwanie nie przyniesie rezultatów w postaci jednostki programowej nadającej się do ponownego użycia, tak napisany przypadek testowy może łatwo zostać przetransformowany do rzeczywistego przypadku testowego. Transformacja taka może odbywać się na dwa sposoby: przez zmianę klasy nadrzędnej na taką, w której metoda call realizuje bezpośrednie wywołania odpowiedniej metody z testowanej klasy (klasa taka może być dostarczona wraz z narzędziem) lub za pomocą analizy gramatyki i odpowiedniej (dość prostej) transformacji wywołań metody call do rzeczywistych wywołań.

8.4.1. Proces wyszukiwania

Ponieważ wiedza na temat metod, których wywołania użytkownik domaga się od poszukiwanego obiektu pozyskiwana jest na bieżąco w czasie wykonywania przypadku testowego, konieczna była zmiana procedury wyszukiwania w stosunku do tych przedstawionych w rozdziałach poprzednich.

Mechanizm wyszukujący na bieżąco buduje drzewo możliwych stanów. Węzły tego drzewa zawierają informację o wszystkich możliwych stanach, w których obiekt (lub obiekty, jeżeli wywołania zostają dopasowane do różnych klas) mogą znaleźć się podczas wykonywania przypadków testowych przy uwzględnieniu wszystkich możliwych odwzorowań metod użytkownika na metody w repozytorium. Przy każdym wywołaniu metody, liście drzewa rozgałęziają się, tworząc kolejne możliwe stany. Podczas rozgałęzienia uwzględnia się jednak odwzorowania metod, które miały miejsce przy poprzednich wywołaniach dla danej ścieżki (ścieżka od korzenia drzewa do danego liścia symbolizuje jeden z możliwych wariantów wykonania przypadku testowego). Dzięki temu gwarantuje się (podobnie jak w przypadku podejścia z rozdziału 6.2), że dwa wywołania metody z zapytania użytkownika, używające tej samej nazwy metody zostaną wykonane przy użyciu tej samej metody z rzeczywistego obiektu. Jak już wspomniano wcześniej podczas opisywania podejścia opartego o scenariusze użycia, takie wiązanie nazw metod ma nie tylko logiczne uzasadnienie, ale powoduje znaczącą redukcję przestrzeni poszukiwań w przypadku większości przypadków testowych.

Ponieważ w każdym z węzłów musi znajdować się obiekt, którego stan jest poprawny z punktu widzenia ścieżki prowadzącej do tego węzła, pojawia się istotny problem pozyskiwania obiektów znajdujących się w takim właśnie stanie. Ponieważ nie sposób przewidzieć a priori liczby potrzebnych obiektów, nie sposób od samego początku utworzyć odpowiedniej ich liczby w celu sukcesywnej modyfikacji ich stanów wraz z każdym wywołaniem metody. Problem ten rozwiązano przez tworzenie obiektów w stanie domyślnym (przy każdym żądaniu rozgałęzienia), a następnie ich propagację wzdłuż gałęzi drzewa od korzenia do rozgałęzianego liścia. W każdym węźle na obiekcie wykonywana jest odpowiednia operacja, w celu aktualizacji jego stanu, przy założeniu, że obiekt w był w poprawnym stanie w węźle będącym przodkiem rozpatrywanego węzła. W ten sposób gdy obiekt dotrze do liścia, jego stan będzie poprawny i będzie on mógł być poddany kolejnej (ostatniej) modyfikacji celem utworzenia rozgałęzienia.

Na pierwszy rzut oka wydawać by się mogło, że tworzone w taki sposób drzewo natychmiast przyjmie monstrualne rozmiary co uniemożliwi praktyczne zastosowanie opisywanej metody. Jest to jednak prawda tylko w przypadku naprawdę rozbudowanych przypadków testowych. W typowych przypadkach rozmiar drzewa bardzo poważnie redukuje się na skutek braku dopasowania pomiędzy metodami lub dzięki napotkaniu nieprawidłowych wyników działania już w pierwszych wywołaniach Np. dla przypadku testowego na rys 8.3. liczba utworzonych węzłów wyniosła jedynie 475.

```
public void testStack(){

            for (int k=0; k <10; k++){
                  call("push", k, DONT_CARE);
            }
            for (int k = 0; k < 10; k++){
                  call("pop", 9 - k);
            }

}
```

Rys. 8.3. Przypadek testowy dla klasy stosu

Ograniczenia wynikały tu głównie z faktu użycia jedynie dwóch różnych metod mimo aż 20 ich wywołań. Warto zwrócić uwagę, że poszukiwane metody mają dość proste nagłówki (0 i 1 argumentów typu int), co przy uwzględnieniu możliwych rzutowań typów (chociażby na java.lang.Object) daje bardzo dużą liczbę odwzorowań na rzeczywiste metody, co mocno wpływa na rozmiar drzewa stanów.

9. ZAKOŃCZENIE

Ponowne użycie jest kluczowe dla zwiększenia wydajności i jakości procesu wytwarzania oprogramowania. W pracy przedstawiono kilka metod przeszukiwania repozytoriów jednostek programowych, które w połączeniu z tradycyjnie stosowanymi technikami przeszukiwania repozytoriów mogą przyczynić się do poprawy efektywności wyszukiwania i nastawienia programistów do systematycznego ponownego użycia. Wprowadzone metody przeszukiwania łączy nie tylko pewien ciąg uogólnień, który przeprowadza je płynnie od najprostszych wariantów poszukiwania w oparciu o opis kanoniczny do zaawansowanych technik bazujących na scenariuszach użycia. Ich wspólnym celem jest przede wszystkim zmiana sposobu patrzenia na sam proces wyszukiwania. Nie zmuszają one użytkownika repozytorium do podążania ścieżkami wyznaczonymi przez opisujących jednostki programowe. Dają mu możliwość znajdowania własnych, jak najlepiej dopasowanych do jego indywidualnych potrzeb sposobów opisu swoich wymagań, w taki sposób aby minimalnym wysiłkiem znaleźć dokładnie to, co miał na myśli. Dzięki temu, w połączeniu z dotychczas stosowanymi technikami wyszukiwania, pozwoli to wyszukiwać jednostki dużo sprawniej, przyczyniając się tym samym do dalszego zwiększania i tak robiących już teraz wielkie wrażenie korzyści z ponownego użycia.

Najistotniejszą jednak zaletą proponowanych technik przeszukiwania repozytorium (szczególnie przeszukiwania opartego na scenariuszach użycia) jest możliwość jego integracji z podejściem *test-driven development*. Integracja ta, w najgorszym razie nie przynosząca właściwie żadnych strat, prowadzić może do zmiany znaczenia tworzonych przypadków testowych, a przez to redukcji kosztów ponownego użycia oraz zwiększenia znaczenia wciąż często niedocenianych metod tworzenia oprogramowania, zakładających rozpoczynanie implementacji od testów. W rozdziale 8 przedstawiono sposób połączenia TDD z procesami ponownego użycia (przy zastosowaniu wprowadzonych technik przeszukiwania) oraz wynikającą z tej integracji modyfikację pojedynczej iteracji TDD.

Proponowane metody nie pozostają jednak wolne od wad. Badanie wykonane za pomocą prototypowego repozytorium oraz na podstawie ankiet przeprowadzanych wśród programistów wskazują, że mimo ogólnie dobrych rezultatów, metody te wykazują pewne słabości. Bardzo ważnym z praktycznego punktu widzenia problemem jest często nadmierna selektywność metod. Prowadzi ona do odrzucania niektórych jednostek programowych, mimo

iż są one użyteczne z punktu widzenia programisty zadającego zapytanie. Drugim, bardzo istotnym problemem jest złożoność obliczeniowa. W pracy podano dowód, w myśl którego problem znalezienia zbioru jednostek programowych w odpowiedzi na zapytanie użytkownika jest NP-trudny. W praktyce jednak, drzewo przeszukiwań jest dość mocno ograniczane przez warunki wynikające bezpośrednio z definicji samego dopasowania jednostki do zapytania, w związku z czym przeszukiwanie może być przeprowadzone w akceptowalnym czasie. Dodatkowo w pracy podano techniki dalszego ograniczania przestrzeni poszukiwań przez stosowanie różnorodnych opisów jednostek programowych przechowywanych w repozytorium.

Wydaje się, iż mimo ograniczeń wynikających z natury proponowanych metod, a także problemów wydajnościowych spowodowanych NP-trudnością opisywanych technik wyszukiwania, podejścia te mogą przyczynić się do dalszego zwiększenia korzyści z ponownego użycia.

Dalsze prace nad przedstawionymi metodami prowadzone mogą być w wielu kierunkach. W celu dokładniejszego oszacowania wpływu tego rodzaju wyszukiwania na efektywność wytwarzania oprogramowania, konieczne wydaje się opracowanie wspierającego te metody narzędzia, które zostanie zintegrowane z jedną z popularnych platform programistycznych. Umożliwi to prowadzenie dalszych, bardziej zaawansowanych eksperymentów, które pozwolą bliżej przyjrzeć się mocnym i słabym stronom przedstawianego podejścia. Równie istotna, zarówno z praktycznego jak i teoretycznego z punktu widzenia, jest analiza ewentualnych uproszczeń w założeniach, które mogłyby prowadzić do uzyskania wielomianowego algorytmu wyszukiwania, przy jednocześnie niewielkim ograniczeniu praktycznego zastosowania danej metody. Jeżeli okazałoby się to niemożliwe, kluczowe dla sukcesu omawianych metod będzie dokładna analiza przedstawionych podejść do ograniczania przestrzeni poszukiwań oraz ewentualne opracowanie dalszych metod przyspieszających przeszukiwanie. Również metoda oparta na kanonicznym opisie jednostki – mimo, iż posiadająca wiele wad - wydaje się rokować pewne nadzieje. Może ona pozwolić na znaczącą redukcję czasu poszukiwań, pod warunkiem jednak, że zostanie uzupełniona o dodatkowe mechanizmy eliminujące jej słabe strony.

10. LITERATURA

[Arc2002] Arciniegas F., *XML- Kompendium programisty*, Helion, 2002

[Ave2001] Aversano L., Canfora G., de Lucia A., Gallucci P., Web Site Reuse: Cloning and Adapting, 3rd International Workshop on Web Site Evolution, 2001

[Bec1999] Beck K., Extreme Programming, Addison-Wesley Pub Co, 1999

[Bec2002] Beck K., Test-Driven development, Addison-Wesley Pub Co, 2002

[Bec2004] Beck K., Aim,Fire, http://www.computer.org/software/homepage/2001/05Design/

[Bła1979] Błażewicz J., Złożoność obliczeniowa algorytmów i problemów szeregowania zadań , Wydawnictwo Uczelniane Politechniki Poznańskiej, Seria "Rozprawy", Poznań 1979

[Dox2004] Doxygen home page, http://www.stack.nl/~dimitri/doxygen/

[Ecl2004] The Eclipse project, www.eclipse.org

[Eck2002] Eckel B, Thinking in Java, Prentice Hall PTR, 2002

[Ezr2002] Ezran M., Morisio M., Tully C., *Practical Software Reuse*: *the essential guide*

[Fre2004] *Freemarker*, http://freemarker.sourceforge.net

[Gam1994] Gamma E., Helm R., Johnson R., Vlissides J., Design patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Pub Co, 1994

[Gra2003] S. Graham, S. Simeonov, T. Boubez, D. Davis, G. Daniels, *Java. Usługi WWW*, Helion 2003

[Ino2003] Inoue K., Yokomori R., Fujiwara H., Yamamoto T., Matushita M., Kusumoto S., *Component Rank: Relative Significance Rank for Software Component Search,* 25th International Conference on Software Engineering, 2003, str. 14-24

[Jac2002] Jaccheri L., Torchiano M., *Classifying COTS Products*, Software Quality, Springer 2002

[Jav2004] JavaDoc tool home page, http://java.sun.com/j2se/javadoc/

[Jcc2004] JavaCC home page, https://javacc.dev.java.net/

[Jip2004] JIProlog home page, http://www.ugosweb.com/jiprolog

[Jun2004] Junit home page, www.junit.org

[Koś1997] Kościelski A., Teoria obliczeń, Wrocławska Drukarnia Naukowa, 1997

[Naw2004] Nawrocki J., Paliświat B., *Przeszukiwanie repozytoriów jednostek programowych*, artykuł przyjęty na VI Krajową Konferencję Inżynierii Oprogramowania

[Mas1996] Masters T., Sieci Neuronowe w praktyce, WNT 1996

[Pal2003] Paliświat B., Komponenty typu szczelina - wypełnienie w tworzeniu witryn internetowych, I Krajowa Konferencja Technologie Informacyjne, str. 69-74, 2003

[Pal2004] Paliświat B., Nawrocki J., Warstwa prezentacji w wielowarstwowych systemach informacyjnych, II Krajowa Konferencja Technologie Informacyjne, str. 249-256, 2004

[Sem2004] *Semantic Networks*, http://www.duke.edu/~mccann/mwb/15semnet.htm

[Sma1998] Smaragdakis Y., Batory D., Implementing Reusable Object-Oriented Components, Proceedings of the Fifth International Conference on Software Reuse, 1998.

[Som1997] Somerville I., Sawyer P., Requirements Engineering: A Good Parctice Guide, John Wiley and Sons Ltd, 1997

[Src2004] Sourceforge, http://sourceforge.net/

[Str1995] Stroustrup B., *Język C++*, WNT 1995

[Sow2004] Sowa J., *Semantic Networks*, http://www.jfsowa.com/pubs/semnet.htm, 2004

[Szy2001] C. Szyperski, *Oprogramowanie Komponentowe. Obiekty to za mało*, WNT 2001

[Ven2004] Venners B., Test-driven development. A conversation with Martin Fowler, http://www.artima.com/intv/testdriven.html

[Way1994] Wayne C. L., Effects of Reuse on Quality,Productivity and Economics, IEEE Software, Volume 11, Issue 5,, pages 23-30, 1994

[Ye2001] Ye Y., Fischer G., *Context-Aware Browsing of Large Component Repositories*, Proceedings of the 16th Annual International Conference of Automated Software Engineering, 2001

ZAŁĄCZNIKI

A. Funkcje używane podczas testów podejścia kanonicznego i opartego o dynamiczne pozyskiwanie danych dla pojedynczych funkcji

```
public static int dec(int x)
```

Zmniejsza argument o 1

```
public static int inc(int x)
```

Zwiększa argument o 1

```
public static int power2(int x) throws Exception
```

Oblicza x-tą potęgę dwójki. Wyjątek rzucany jest dla ujemnych wartości x

```
public static int power(int base, int x) throws Exception
```

Oblicza base do potęgi x. Wyjątek rzucany jest dla ujemnych wartości x

```
public static float any(int base, int x, float d) throws Exception
```

Oblicza (base * d) do potęgi x. Wyjątek rzucany jest dla ujemnych wartości x

```
public static double sin(double x)
```

Oblicza sin(x)

```
public static double cos(double x)
```

Oblicza cos(x)

```
public static double t(double x, int m) throws Exception
```

Oblicza funkcję $t(\text{double x}, \text{int m}) = \begin{cases} \sin(x) \Leftrightarrow m = 0, \\ \cos(x) \Leftrightarrow m = 1 \end{cases}$

```
public static int sgn(int x)
```

Oblicza sgn(x)

```
public static int sgnplus1(int x)
```

Oblicza sgn(x) + 1

```
public static int exp3(int base)
```

Oblicza $base^3$

```
public static int exp3plus1(int base)
```

Oblicza $base^3 + 1$

B. Klasy i funkcje użyte do testowania podejcia opartego o scenariusze użycia

W prototypowym repozytorium umieszczono następujące standardowe klasy:

- java.util.Calendar
- java.util.Vector
- java.util.ArrayList
- java.util.HashMap
- java.util.BitSet
- java.util.Currency
- java.util.HashSet
- java.util.Random
- java.util.Stack
- java.util.StringTokenizer
- java.lang.String

a także cztery klasy własne:

- pl.poznan.put.cs.paliswiat.reuse.test.Stack – stos nieblokujący
- pl.poznan.put.cs.paliswiat.reuse.test.Fifo – kolejka fifo
- pl.poznan.put.cs.paliswiat.reuse.test.Math – zbiór metod z załącznika A
- pl.poznan.put.cs.paliswiat.reuse.test.ExpList – przykładowa implementacja listy z punktu 8 załącznika C

C. Ankieta służąca do zbadania skuteczności wyszukiwania opartego o nazwę

Podaj po trzy przykładowe nazwy dla klas lub metod, których działanie przedstawione jest w sposób opisowy poniżej. Jeżeli nie masz pomysłu na więcej niż jedną (lub dwie) nazwy, zostaw pozostałe wiersze niewypełnione.

1. Klasa zawierająca podstawowe funkcje trygonometryczne: sin, cos, tan

2. Nazwa metody w słowniku ortograficznym, która zwraca poprawioną wersję słowa jeżeli nie jest ono rozpoznane oraz to samo słowo jeżeli zostało rozpoznane jako poprawne. Jeżeli słowo jest niepoprawne i słownik nie potrafi zaproponować korekty, metoda powinna zwrócić wartość null.

3. Klasa zawierająca funkcjonalność związaną z zarządzaniem użytkownikami w portalu.

4. Klasa zawierająca funkcjonalność związaną ze sprawdzaniem uprawnień w portalu.

5. Klasa reprezentująca listę jednokierunkową lub dwukierunkową. W przypadku rozpoczęcia przechodzenia wstecz po raz pierwszy, lista automatycznie dobudowuje struktury przyspieszające tę operację.

6. Klasa pośrednika szyfrującego/deszyfrującego komunikat SOAP i wzbogacająca go o nagłówki zawierające informacje o użytym trybie szyfrowania.

7. Klasa portletu posiadającego funkcjonalność grupy dyskusyjnej.

8. Lista służąca do automatyzacji pewnych testów, wymagająca wcześniejszego zadeklarowania elementu, który zostanie dodany do listy. W przypadku prób dodania nie zadeklarowanego wcześniej elementu rzucany jest wyjątek

9. Metoda służąca do wcześniejszej deklaracji dodawanego elementu w liście z pytania 8

10. Metoda w servlecie zbierającym statystyki służącym da do przetworzenia obiektu żądania na zapis w logu

D. Ankieta służąca do zbadania skuteczności wyszukiwania opartego o scenariusze użycia

Podaj przypadki testowe, których użył(a)byś w celu wyszukania podanych poniżej metod. W podpunkcie a) podaj przypadki, których użył(a)byś jako pierwsze przybliżenie, aby zorientować się w rozmiarze zwróconych rezultatów. W podpunkcie b) rozszerz przypadek testowy w ten sposób, aby mieć pewność, że znaleziona jednostka programowa będzie jedyną (lub jedną z bardzo niewielu) jednostek działających zgodnie z Twoimi oczekiwaniami. Jeżeli uważasz, że podpunkt b) nie wnosi nic nowego w danym przypadku, zostaw to pole puste. Możesz używać dowolnych nazw metod (nie są one istotne). Testy konstruuj jako sekwencję wywołań w postaci wywołanie = spodziewana wartość oddzielonych średnikami, np. y(3); x(3) = 71. W przypadku klas, załóż, że konstrukcja każdej z nich już nastąpiła (za pomocą bezparametrowego konstruktora), a Twoim zadaniem jest zbudowanie testów do zainstancjowanego obiektu.

1. Pojedyncze funkcje
    a. inc(x) = x + 1
    b. abs(x)
    c. power(x,y) = x^y
    d. 1/x
    e. max(x,y)
2. Klasy
    a. stos
    b. kolejka
    c. łańcuch z możliwością pobierania podłańcuchów
    d. łańcuch z możliwością dopasowywania do wyrażeń regularnych
    e. klasa licznika posiadająca metody zerowania oraz zwiększania o 1 L

f. lista służąca do automatyzacji pewnych testów, wymagająca wcześniejszego zadeklarowania elementu, który zostanie dodany do listy. W przypadku prób dodania nie zadeklarowanego wcześniej elementu rzucany jest wyjątek