ZAŁĄCZNIK 4a

Autoreferat

Imię i Nazwisko: Stanisław Jarząbek

Edukacja

Stopień doktorski (1979), Wydział Matematyki i Mechaniki, Uniwersytet Warszawski, *Tytuł pracy doktorskiej*: "Generowanie optymalizatorów dla klasy języków pośrednich meta-translatora", promotor: Zdzisław Pawlak

Tytuł magisterski (1972), Wydział Matematyki i Mechaniki, Uniwersytet Warszawski, *Tytuł pracy magisterskiej*: "k-Maszyny produktowe", promotor: Zdzisław Pawlak

Historia zatrudnienia

1992-teraz Associate Professor, Department of Computer Science, School of Computing, National University of Singapore (stała pozycja tenure w 1997)

2000-06 Adjunct Associate Professor, Department of Electrical & Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada

1999 staż naukowy (sabbatical): **styczeń-czerwiec:** Visiting Scientist, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany; krótkie wizyty: prof. Mehdi Jazayeri, Technical University of Vienna i prof. Carlo Ghezzi, Politecnico de Milano; **lipiec-listopad**: Visiting Associate Professor at Software Engineering Group, Department of Electrical and Computer Engineering, University of Waterloo, Canada

1990-92 Kierownik Badań Naukowych (Research Manager), CSA Research Pte. Ltd. w Singapurze i Adjunct Senior Lecturer, National University of Singapore

1984-89 Assistant Professor, Department of Computer Science and Systems, McMaster University, Hamilton, Ontario, Canada

1982-84 Lecturer, University of Maiduguri, Nigeria

1972-82 Naukowiec, Pracownia Meta-Translatorów, Instytut Maszyn Matematycznych, IMM, ul. Krzywickiego 34, Warszawa.

Wskazanie osiągnięcia

Tytuł osiągniecia naukowego:

Metody utrzymywania i ewolucji programów bazujące na wielokrotnym wykorzystaniu programów.

Reuse-based software maintenance and evolution

Jako osiągnięcie naukowe, o którym mowa w Ustawie z dnia 14 marca 2003 r. o stopniach naukowych i tytule naukowym oraz o stopniach i tytule w zakresie sztuki, uzyskane po otrzymaniu stopnia doktora, stanowiące znaczny wkład autora w rozwój określonej dyscypliny naukowej wskazuję jednotematyczny cykl następujących publikacji:

[1] Jarząbek, S. "Design of Flexible Static Program Analyzers with *PQL*," *IEEE Transactions on Software Engineering,* March 1998, pp. 197-215.

[2] Jarząbek, S. and Huang, R. "The case for User-Centered CASE Tools," *Communications of ACM,* August 1998, pp. 93-99.

[3] Jarząbek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246; artykuł został nagrodzony ACM SIGSOFT distinguished paper award.

[4] Basit, H.A., Rajapakse, D.C., and Jarząbek, S. "Beyond Templates: a Study of Clones in the STL and Some General Implications," *Int. Conf. Software Engineering, ICSE'05,* St. Louis, USA, May 2005, pp. 451-459.
[5] Basit, H. A., Jarząbek, S. "Data Mining Approach for Detecting Higher-level Clones in Software," *IEEE Trans. on Soft. Eng.*, July/August 2009 (vol. 35 no. 4) pp. 497-514; Published online January 2009.
[6] Jarząbek, S, Pettersson, U. and Zhang, H. "University-Industry Collaboration Journey towards Product Lines," *Int. Conf. on Software Reuse, ICSR'2011,* S. Korea, June 2011, pp. 223-237.
[7] Jarząbek, S. and G. Wang "Model-based Design of Reverse Engineering Tools", *Journal of Software Maintenance: Research and Practice*, No. 10, 1998, John Wiley & Sons, pp. 353-380
[8] Zhang, H. and Jarząbek, S. "XVCL: A Mechanism for Handling Variants in Software Product Lines," special issue on Software Variability Management of Elsevier's journal *Science of Computer Programming*, Volume 53, Issue 3, Dec. 2004, pp. 381-407.
[9] Pettersson, U., and Jarząbek, S. "Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach," *ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2005, Lisbon, pp. 326-335.
[10] Jarząbek, S. "Genericity - a "Missing in Action" Key to Software Simplification and Reuse," *13th Asia-Pacific Software Engineering Conference, APSEC'06*, IEEE Comp. Soc., 6-8 December 2006, Bangalore, India, pp. 293-300.
[11] Rajapakse, D. and Jarząbek, S. "Towards generic representation of web applications: solutions and trade-offs," *Software, Practice & Experience*, Volume 39 Issue 5, April 2009, pp. 501 – 530, Published Online: 27 Nov 2008.
[12] Rajapakse, D.C. and Jarząbek, S. "Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis," *Int. Conf. Software Engineering, ICSE'07,* Minneapolis, USA, May 2007, pp. 116-125.

Powyższe publikacje dokumentują mój dorobek naukowy w zakresie następujących zagadnień:

1. Tradycyjne metody i narzędzia wspomagające utrzymywanie programów.
2. Wielpkrptne wykorzystanie programow.
3. Strategiczna modernizacja procesów biznesowych i programów.
4. Rozpoznawanie i analiza podobieństw w programach (software clone detection).

Monografia: S. Jarząbek *Reuse-based Software Maintenance and Evolution*, CRC, 1997 traktuje temat habilitacji całościowo. Z uwagi na jej rozmiar (390 stron), monografia nie stanowi przedmiotu ewaluacji mojego dorobku, a jest załączona jedynie w celu utrzymania kompletności dokumentacji mojego dorobku w zakresie tematu habilitacji.

Omówienie celu naukowego ww. prac i osiągniętych wyników wraz z omówieniem ich ewentualnego wykorzystania.

Odnośniki [1]-[12] kierują do powyższych publikacji, a pozostałe do listy publikacji w Załączniku 6a zatytułowanym „Wykaz publikacji".

1. Metody i narzędzia wspomagające utrzymywanie i modernizację programów.

1.1 Wstęp i motywacja.

Utrzymywanie istniejących programów (software maintenance and evolution) obejmuje usuwanie błędów (corrective maintenance), adaptację programów do nowych platform (adaptive maintenance), rozszerzanie programów o nowe funkcje w odpowiedzi na nowe wymagania użytkowników (perfective maintenance), i modernizację programów w celu ułatwienia wprowadzania zmian, aby umożliwić wielokrotne wykorzystanie komponentów programowych (preventive maintenance; software modernization; software re-engineering). Statystyki wykazują, że tylko około 20 procent wysiłku utrzymania programów pochłania usuwanie błędów. Istotą programów jest to, że służą nam one w dziedzinach życia, które się szybko rozwijają. Temu rozwojowi towarzyszy rozwój wymagań programów i potrzeba rozszerzania programów o

nowe funkcje. Dlatego też koszty rozszerzania programów o nowe funkcje dominują ponad kosztami innych czynności związanych z utrzymaniem programów.

Historycznie, koszty utrzymania programów rosły szybciej od kosztów rozwijania nowych programów, pochłaniając w niektórych firmach do 80 procent budżetu programowego, a przeciętnie wynosząc ok. 60 procent budżetu programowego. Ulepszenia metod utrzymywania programów mają więc większy wpływ ekonomiczny niż podobne ulepszenia metod rozwijania nowych programów. Wbrew tym statystykom, widzimy większe zainteresowanie i więcej wdrożonych metod i narzędzi dla rozwijania nowych programów niż dla utrzymywania istniejących programów. Wśród programistów rozwijanie nowych programów cieszy się też większym prestiżem i popularnością niż praca przy utrzymywaniu starych programów, która jest często powierzana niedoświadczonym programistom.

Nieudolnie wprowadzane zmiany programów powodują erozje architektury programu, co z kolei prowadzi do jeszcze większych kłopotów w utrzymywaniu programu w przyszłości. Ten negatywny cykl wzajemnych odziaływań prowadzi często do sytuacji, w której programu nie da się już dłużej utrzymywać, i trzeba się go pozbyć, albo - jeśli to jest niemożliwe z uwagi na przetrwanie firmy – zmodernizować.

W latach 1995-2000 pracowałem nad metodami i narzędziami wspomagającymi rozumienie i utrzymywanie programów.

1.2 Statyczna analiza programów.

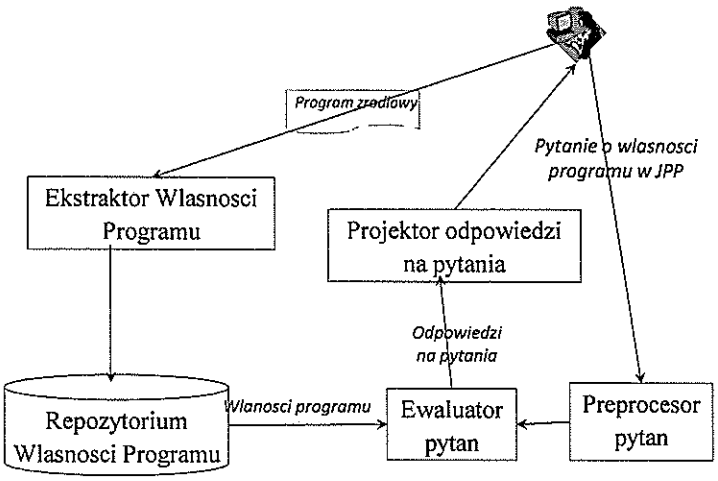
Publikacje: [1,2,13,14]

Zrozumienie programu jest nieodzownym warunkiem jego zmian. Potrzebne zmiany są komunikowane przez użytkowników w terminologii wymagań programu. Programiści muszą przetłumaczyć tak sformułowane zmiany na język kodu. Zadanie to nie jest łatwe, szczególnie przy braku odpowiedniej dokumentacji programu. Pierwsza trudność polega na zlokalizowaniu kodu, który należy zmodyfikować, aby uzyskać pożądaną zmianę wymagań. Problem ten nosi nazwę lokalizacji cech programu (feature location). Druga trudność to rozumienie konsekwencji zmian kodu. Z uwagi na skomplikowane połączenia logiczne pomiędzy modułami (np. relacje przepływu kontroli i danych w programie) rozumienie wszystkich konsekwencji zmian kodu często wymaga analizy wykraczającej poza lokalny punkt programu, w którym planujemy wprowadzić zmiany (ripple effects of changes). Statyczna analiza programów pomaga w radzeniu sobie z tą drugą trudnością.

Wiedza o programie potrzebna do wprowadzania zmian może być wyrażona w postaci pytań (program queries) o własności programu, jak na przykład:

- Które procedury są wołane (bezpośrednio lub pośrednio) z procedury „P"?
- Które zmienne są modyfikowane w procedurze wołanej z procedury „P"?
- Czy jest możliwość wykonania instrukcji numer 100 po instrukcji numer 20 w jakimś (jednym z wielu) cyklu wykonania programu?
- Na które instrukcje programu może mieć wpływ modyfikacja kodu w instrukcji numer 10?

W literaturze znajdujemy szereg propozycji Statycznych Analizatorów Programów (SAP), które potrafią automatycznie odpowiedzieć na klasy pytań o własności programów. Ogólny schemat działania narzędzi SAP jest przedstawiony na Schemacie 1.



Schemat 1. Schemat działania Statycznego Analizatora Programów (SAP)

Mój wkład w dziedzinie statycznej analizy programów polegał na podejściu do konstrukcji narzędzi SAP w oparciu o koncepcyjne modele wiedzy programowej (model-based design) i na wysokopoziomowym języku do formułowania pytań programowych. Metoda opiera się na spostrzeżeniu, że interesującą klasę własności programu możemy opisać przy pomocy relacji pomiędzy elementarnymi jednostkami programu takimi jak procedury, funkcje, metody, klasy, zmienne, czy instrukcje programu. Przykładami relacji są: relacja wywołania pomiędzy procedurami albo relacja modyfikacji czy użycia pomiędzy procedurami i zmiennymi. Narzędzie SAP oblicza własności programu wedle zadanego koncepcyjnego modelu wiedzy programowej, i następnie odpowiada na zadane pytania wyrażone w pol-formalnym Języku Pytań Programowych (JPP), odwołując się do repozytorium własności programu. Modele własności programów, które zaproponowałem były oparte na modelu Jednostkowo-Relacyjnym (Entity-Relationship), natomiast JPP przypomina SQL, z tym że jest językiem wyższego poziomu niż SQL, z uwagi na odwołania do modelu Jednostkowo-Relacyjnego.

W porównaniu do prac wcześniej opublikowanych, narzędzia SAP skonstruowane według powyższego planu wyróżniają się prostotą i naturalnością pojęć, i elastycznością w dostosowaniu narzędzia do nowych języków programowania i nowych rodzajów pytań programowych.

Oprócz formalizmów i pracy teoretycznej, ten nurt mojej pracy zaowocował prototypem narzędzia SAP dla języka COBOL, który mogliśmy przetestować na przykładach programów [1].

1.3 Zwrotna inżynieria programów.

Publikacje: [7,15-18]

Brak aktualnej dokumentacji programu jest norma w przypadku programów żywych, rozwijających się wraz z coraz to nowymi wymaganiami użytkowników. Celem zwrotnej inżynierii jest odtworzenie dokumentacji, zazwyczaj w postaci modeli UML takich jak diagramy klasowe (class diagrams) czy diagramy zmian stanów programu (state transition diagrams). W tym zagadnieniu, zaproponowałem system pisania narzędzi zwrotnej inżynierii w oparciu o modele wyspecyfikowane w języku JPP, uogólniając metody analizy programów i konstrukcji narzędzi SAP, opisanych w podrozdziale 1.2 [7].

2. Wielokrotne wykorzystanie programów.

2.1 Wstęp i motywacja.

Najbardziej pożądanym atrybutem programów jest ich użyteczność. Drugim ważnym elementem jest koszt ich produkcji i utrzymania. Produktywność programowania to temat rzeka, łączy w sobie wiele wątków takich jak techniki i narzędzia programowania, metody zarządzania projektami i ludzkie zdolności i umiejętności.

W tradycyjnych dziedzinach inżynieryjnych, standaryzacja komponentów i procesów produkcyjnych umożliwiły automatyzację, która była kluczem do zwiększonej produktywności. Przykładowo, wprowadzone przez Forda automatyczne linie produkcyjne umożliwiły tanią produkcję samochodów. Automatyzacja produkcji nieodmiennie łączy się z architekturą (makro-wzorcem) produktów i standaryzacja komponentów, których projekt (nie fizyczny przedmiot) jest wykorzystywany do produkcji wielu egzemplarzy samochodów.

W poszukiwaniu rozwiązań dla zwiększenia produktywności programowania, warto zauważyć, że w większości projektów mniej niż 10 procent problemów jest nowych, wymagających kreatywnych zdolności ludzkiego umysłu dla ich rozwiązania. Reszta to rutynowe działania - powtarzanie starych schematów, rozwiązywanie dawnych problemów w trochę zmienionym kontekście, i rozwijanie komponentów programowych podobnych do tych, co już zrobiliśmy w poprzednich projektach.

Nie możemy zautomatyzować zadań, które wymagają kreatywności, i nie tu należy się spodziewać drastycznego zwiększenia produktywności. Ale możemy zautomatyzować rutynowe zadania. Zważywszy, że automatyzacja może dotyczyć 90 procent pracy projektowej, wszelki postęp w tym kierunku zaowocuje znacznymi oszczędnościami kosztu rozwoju programów.

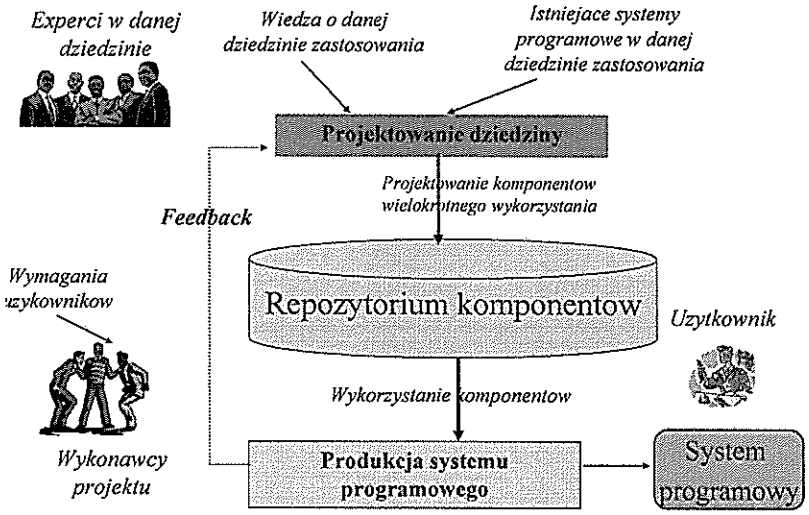
Rozwiązań dla zwiększonej produktywności należy szukać tam gdzie znajdujemy dużo podobieństw i powtórzeń w strukturze produktu, kodzie, bądź też w procesie produkcyjnym. Nie dziwi więc fakt, że w programowaniu, duże nadzieje wiąże się z możliwością wielokrotnego wykorzystania kodu, bez czego nie może być automatyzacji. Uwagę często się kieruje na rodziny systemów programowych, które są budowane dla podobnych celów, i przez to wykazują wiele podobieństw. Dzisiejsze firmy często rozwijają takie właśnie rodziny systemów, na przykład warianty systemu zarządzania przedsiębiorstwem instalowane w różnych firmach: wszystkie takie systemy są podobne do siebie, i jednocześnie różnią się wymaganiami specyficznymi dla danej firmy. Taka właśnie sytuacja stwarza możliwość zautomatyzowanych rozwiązań gwarantujących wysoką produktywność, w oparciu o wielokrotne wykorzystanie programów. Kierunek badań nad skutecznymi metodami wielokrotnego wykorzystania programów w kontekście rodzin podobnych systemów nosi nazwę Programowych Linii Produkcyjnych (PLP). Trend ten zdominował w przeciągu ostatnich dwóch dekad badania akademickie nad wielokrotnym wykorzystaniem programów i przemysłową praktykę.

System automatycznej produkcji kompilatorów, tzw. meta-kompilator, jest klasycznym przykładem takiego rozwiązania. Mamy setki języków programowania i komputerowych architektur, a kompilator jest potrzebny dla każdej ich kombinacji. Kompilatory tworzą ciekawą rodzinę programów. Bez zastosowania technik wielokrotnego wykorzystania programów (w tym przypadku komponentów kompilatora takich jak parser lub analizator semantyczny) pisanie kompilatorów wymaga ogromnych nakładów pracy, której dużą częścią jest powtarzanie tych samych schematów w zmiennych formach. W odróżnieniu od tego naiwnego i kosztownego podejścia, meta-kompilator automatycznie generuje kompilator na podstawie opisu-specyfikacji języka źródłowego i docelowej architektury komputera, na którym programy mają być wykonywane. Dzieje się to dzięki generycznym komponentom kompilacyjnym (takim jak generyczny lexer, parser lub analizator semantyczny), które po dostosowaniu mogą służyć przy budowie wielu kompilatorów. Budowa takich generycznych komponentów stała się możliwa dzięki formalizacji języków programowania przy pomocy gramatyk regularnych, BNF, gramatyk atrybutowych i innych notacji. Algorytmy analizy leksykalnej, syntaktycznej i semantycznej pisane są w formie generycznej, parametryzowanej przez specyfikacje języka programowania. Meta-kompilator dostosowuje komponenty do potrzeb danego języka programowania i komputera.

Meta-kompilator oferuje imponującą produktywność, pozwalając z paru stron specyfikacji otrzymać kompilator obejmujący tysiące linii kodu. Niestety, trudno jest zbudować podobnie sprawne rozwiązania w innych informatycznych, które są mniej stabilne i nie dają się sformalizować tak elegancko jak języki programowania.

W ostatnich dwóch dziesięcioleciach powstała koncepcja Programowych Linii Produkcyjnych, PLP (Software Product Lines) jako uogólnione rozwiązanie oparte na idei wielokrotnego wykorzystania programów: PLP to rodzina systemów programowych, z których każdy oprócz wspólnych wymagań spełnia również specyficzne wymagania konkretnego użytkownika. Jednocześnie wszystkie te systemy powstają i są utrzymywane ze wspólnej bazy specjalnie zaprojektowanych generycznych komponentów wielokrotnego wykorzystania.

W PLP, mamy dwa logicznie odrębne procesy uwidocznione na poniższym schemacie, mianowicie projektowanie dziedziny i produkcja systemów programowych dla użytkowników.



Schemat 2. Programowanie systemów przy pomocy Programowych Linii Produkcyjnych

Projektowanie dziedziny to tworzenie architektury i generycznych komponentów wielokrotnego wykorzystania. Produkcja systemów odbywa się przy wykorzystaniu architektury i generycznych komponentów. Całość odbywa się w ścisłej współpracy i wymianie informacji pomiędzy projektowaniem dziedziny i produkcją programów.

Zazwyczaj komponenty trzeba intensywnie zmieniać w trakcie ich wykorzystania przy produkcji nowych systemów. Zadaniem technik zarządzania zmiennością programów jest, na ile to możliwe, zautomatyzować proces dostosowania komponentów do nowego kontekstu, w którym pragniemy komponenty wykorzystać. Korzyści płynące z PLP zależą w dużym stopniu od jakości technik zarządzania zmiennością programów. Moje zainteresowania podejściem PLP skupiły się na technikach zarządzania zmienności programów.

2.2 Zarządzanie zmiennością programów.

Publikacje: [3,4,6,8-12,19-34]

Popularne techniki zarządzania zmiennością programów obejmują preprocesory (oferujące makro-instrukcje typu #ifdef, #define), pliki konfiguracyjnych parametrów, techniki parametrycznego programowania takie jak w schematach C++ lub generycznych klasach w języku Java, i systemy składania programów takie jak Ant lub *make*. Analizy przemysłowych rozwiązań PLP wykazują, że popularne techniki pozostawiają wiele do życzenia w kontekście PLP [65,68]. Każda z tych technik radzi sobie tylko z niektórymi rodzajami zmienności, co czyni nieodzownym stosowanie wielu technik razem. Techniki te nie były jednak zaprojektowane, aby razem pracować, co prowadzi do kłopotów z ich integracja. Automatyzacja wielokrotnego wykorzystania komponentów przy użyciu tych technik jest niemożliwa, a interwencje manualne prowadzą do trudnych do wykrycia błędów. W ostatecznym rachunku, rozwiązanie, które miało służyć usprawnieniu produkcji, może okazać się trudne i kosztowne w użyciu. W konsekwencji, rozwiązania bazujące na popularnych technikach nadają się dla małych systemów, gdzie liczba zmiennych cech nie jest duża, i wpływ zmiennych cech na komponenty stosunkowo nieskomplikowany. Rozwiązania takie wielce się komplikują w przypadku dużych systemów, charakteryzującymi się wieloma zmiennymi cechami.

W odpowiedzi na powyższe problemy, wraz z zespołem zaproponowaliśmy i rozwinęliśmy uniwersalną technikę zarządzania zmiennością programów XVCL (XML-based Variant Configuration Language) [3,4,6, 26-34,45].

XVCL wziął inspirację z Frame Technology™ (Bassett, P. *Framing Software Reuse: Lessons From the Real World* , Prentice Hall, 1997) osiągającej imponującą produktywność w przemysłowych PLP. Paul

Bassett, wynalazca zasad Frame Technology i współzałożyciel firmy Netron w Toronto zajmującej się wdrożeniami Frame Technology w przemyśle, był naszym współpracownikiem.

XVCL pracuje na zasadzie generacji programów z meta-komponentów zorganizowanych hierarchicznie dla potrzeb wyodrębnienia struktur architektonicznych i kodu sposobnych do wielokrotnego wykorzystania. XVCL umożliwia formowanie meta-komponentów, ich parametryzację ułatwiającą adaptację kodu w czasie wielokrotnego wykorzystania, i hierarchiczną organizację komponentów, bez potrzeby do uciekania się do innych technik.

Podstawowa zasada XVCL jest szeroko pojęta parametryzacja. Podczas gdy parametrami w programowaniu generycznym są typy danych (C++, Java) lub funkcje (języki funkcjonalne), w XVCL parametrem może być dowolna struktura programowa bez względu na jej charakter i rozmiar – zmienna, stała, fragment kodu, moduł lub pod-system.

Przedmiotem wielokrotnego wykorzystania w trakcie generacji kodu są meta-komponenty, które tworzą struktury niezależne od struktur języka programowania takich jak procedura, funkcja, metoda klasowa, lub klasa, i niezależne od fizycznych formacji kodu takich jak plik źródłowy lub kartoteka. Stwarza to możliwość oddzielenia problemów implementacyjnych od problemu wielokrotnego wykorzystania komponentów: Obydwa aspekty projektowania programu mogą być zoptymalizowane bez konfliktów i kompromisów.

Szereg eksperymentów wykazało zdolność XVCL do zarządzania zmiennością programów w najbardziej skomplikowanych sytuacjach zmiennych cech, jako efektywnej i samowystarczalnej techniki do budowania PLP. W laboratoryjnych i przemysłowych projektach typowo uzyskujemy 60-90 procent kodu zawartego w komponentach wielokrotnego wykorzystania [3,4,6,9,10,28,31,34].

XVCL daje dobre wyniki w rękach małej grupy wysoko wykwalifikowanych ekspertów. Gorzej jest z integracja XVCL ze standardowymi przemysłowymi procesami produkcyjnymi. Pomimo skromnego i niewyrafinowanego mechanizmu, XVCL zmienia sposób widzenia programu, i wymaga od programistów myślenia na dwóch płaszczyznach jednocześnie: o konkretnym programie, który budujemy dla użytkownika, i o meta-programie, który zawiera w sobie możliwości nieskończonej ilości wariantów programów jakie można z niego wygenerować. Narzędzia umożliwiające analizę meta-programów i rozumienie procesu generacji wariantów programowych z meta-representacji programu w XVCL mogą pomóc w tym względzie, ale dalsze badania są potrzebne do wypracowania skutecznych rozwiązań.

Mój zespół zaprojektował XVCL we współpracy z firmą w Toronto Netron, Inc., którego jeden ze współzałożycieli, Paul Bassett, byl partnerem w naszym projekcie. XVCL był stosowany przez współpracującą z nami firmę ST Electornics Pte Ltd w Singapurze i doświadczenia z największego projektu z XVCL zostały udokumentowane w publikacjach [6,8,9]. Badania nad przemysłowymi projektami PLP wykonywałem we współpracy z firmą Wingsoft założona przez grupę naukowców na Uniwersytecie Fudan, wykorzystując kod systemu rozwiniętego przez te firmę.

3. Utrzymywanie programów bazowane na wielokrotnym wykorzystaniu programów.

Utrzymywanie i ewolucja programów i wielokrotne wykorzystanie programów są uważane za dwie osobne dziedziny wiedzy i praktyki programowani. Poświęcone są im osobne książki lub rozdziały w tekstach inżynierii programowania, a badacze naukowi publikują prace w osobnych magazynach i spotykają się na osobnych konferencjach.

Przy uważnym spojrzeniu, dostrzegamy jednak analogię pomiędzy kluczowymi czynnościami, które towarzyszą utrzymywaniu programów i adaptacji komponentów programowych do powtórnego wykorzystania w nowych kontekstach. W obydwu przypadkach mamy do czynienia ze zmianami programowi i konieczności rozumienia konsekwencji zmian – zarówno pożądanych jak i niepożądanych. Różnica polega na tym, że w przypadku utrzymywania programów program podlegający zmianom zwykle nie jest zaprojektowany z myślą o łatwości wprowadzania zmian, natomiast komponenty wielokrotnego wykorzystania powinny z zasady umożliwiać łatwość zmian.

Przy użyciu metody XVCL, różnica pomiędzy utrzymywaniem i wielokrotnym wykorzystaniu programów zanika: organizacja programu w hierarchie XVCL'owych meta-komponentów i instrumentacja kodu przy pomocy XVCL'owych instrukcji umożliwia łatwiejsze wprowadzanie zmian do programów zarówno przy utrzymywaniu programów, jak i przy adaptacji komponentów programowych w czasie ich wielokrotnego wykorzystania. Łatwość wprowadzania zmian wynika z ujawnienia w sformalizowanej postaci sieci wpływu zmian na programy. Efekt ten pomaga programistom w sytuacji, gdy program został podzielony na komponenty wielokrotnego wykorzystania, jak i w sytuacji programów monolitycznych, których jakość została zdegradowana w wieloletnim procesie ich utrzymywania, przypadek często spotykany w praktyce przemysłowej.

XVCL umożliwia stopniowa modernizację starzejących się programów, w celu poprawienia ich jakości, komponentyzacji, i umożliwienia wielokrotnego wykorzystania komponentów programowych.

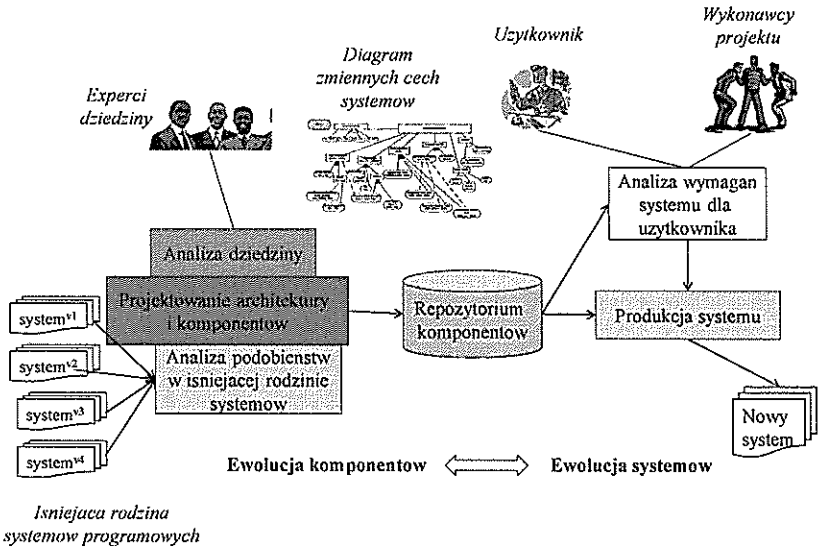
4. Modernizacja starzejących się programów.

4.1 Wstęp i motywacja.

Przymysłowe doświadczenia wskazują, że jakość programów intensywnie zmienianych z biegiem lat staje się coraz niższa, struktura programu staje się coraz mniej widoczna, a dokumentacja nieaktualna. Wynikiem tego jest rosnąca trudność dostosowywania takich programów do zmian w wymaganiach programów. Zastąpienie starzejących się programów nowymi często nie jest możliwe z uwagi na koszt takiego rozwiązania, lub z powodu braku pełnej dokumentacji programu. W powyższej sytuacji, firma zarządzająca starzejącym się programem może uciec się do modernizacji programu w celu poprawienia jego jakości.

Modernizacja jest też często niezbędnym krokiem poprzedzającym adaptację programów do nowych platform (na przykład .NET, JEE lub SOA). W ostatnich 20 latach byliśmy świadkami eksplozji nowych platform (i również ich kolejnych wersji) co spowodowało zapotrzebowanie na automatyczne metody modernizacyjne i adaptacyjne programów.

Modernizacja jest również niezbędna w sytuacji, gdy firma decyduje się na przebudowę istniejącej rodziny programów na Programowa Linie Produkcyjna (PLP), w celu umożliwienia ekonomicznego rozwoju nowych programów i utrzymywania bazy kodu w oparciu o techniki wielokrotnego wykorzystania programów. Cykl modernizacyjny prowadzący do zbudowania PLP ukazuje poniższy schemat.



Schemat 3. Cykl procesów w Programowej Linii Produkcyjnej (PLP)

Celem analizy dziedziny jest rozpoznanie i opisanie podobieństw i różnic pomiędzy istniejącymi systemami tworzącymi rodzinę, którą chcemy przekształcić na PLP. Taka analiza może być wspomagana przez

automatyczne analizatory podobieństw. Diagram zmiennych cech (feature diagram) jest graficzna notacja pozwalająca na wyszczególnienie podobieństw i różnic charakteryzujących systemy w danej dziedzinie. Na tej podstawie jesteśmy w stanie określić, które komponenty programowe są dobrymi kandydatami do repozytorium komponentów wielokrotnego wykorzystania.

Z upływem czasu, PLP ewoluuje, komponenty wielokrotnego wykorzystania podlegają zmianom i indywidualne systemy zmieniają się wraz z nowymi wymaganiami użytkowników. Jest zadaniem technik zarządzania zmiennością programów, aby radzić sobie z problemami ewolucji PLP w systematyczny sposób.

Niezależnie od celu modernizacji, techniki pomocne w modernizacji są podobne i obejmują statyczną analizę kodu, automatyczne odtwarzanie dokumentacji programu (zwrotna inżynierie) i analizę podobieństw (powtórzeń) w programach. Pierwsze dwie techniki omówiłem w Rozdziale 1, a swój wkład w analizę podobieństw przedstawiam w pod-rozdziale 4.3. Poniższy pod-rozdział omawia moją pracę w zakresie planowania projektów modernizacyjnych.

4.2 Strategiczna modernizacja procesów biznesowych i programów.

Publikacje: [35-39]

W początkach ery komputeryzacji, komputery były używane do automatyzacji niektórych czynności wchodzących w skład istniejących procesów biznesowych (np. wczytywanie i magazynowanie danych). Wraz z postępem komputeryzacji, stało się jasne ze znacznie większe możliwości usprawnień kryją się w bardziej radykalnym podejściu, mianowicie w zmianie i przedefiniowaniu procesów biznesowych wykorzystując w pełni możliwości oferowane przez komputery (business process re-engineering). Klasycznym przykładem są internetowe systemy rezerwacji biletów samolotowych lub zakupy artykułów na Amazon lub eBay.

Drastycznym zmianom biznesowych procesów towarzyszy potrzeba stworzenia programów wspomagających nowe procesy. Aby zwiększyć ekonomiczny efekt usprawnień, wskazane jest, aby nowe programy pisać na bazie istniejących programów funkcjonujących w danej firmie. Do tego potrzebne są metody i narzędzia dla modernizacji programów.

W ramach tego tematu, rozwinąłem zintegrowaną platformę do modelowania procesów biznesowych i programów wspomagających te procesy, włączając zależności pomiędzy obydwoma modelami. Celem zintegrowanego modelowania jest to, aby transformacjom procesów biznesowych mogły towarzyszyć transformacje modeli programów. Zintegrowana platforma, którą zaproponowałem służyła planowaniu i zarządzaniu projektami modernizacyjnymi.

W zastosowaniu do projektów modernizacji programów, rozszerzyłem system analizy programów SAP (opisany w pod-rozdziale 1.2) o nowe możliwości definiowania transformacji programów na poziomie modeli: system zwrotnej inżynierii był używany do odtworzenia modeli na podstawie analizy kodu, a transformacje modeli były wyrażone w języki JPP. Po dokonaniu transformacji kod był generowany z nowych modeli używając tradycyjnych metod generacji kodu.

4.3 Rozpoznawanie i analiza podobieństw w programach.

Publikacje: [5,40-42]

Unikanie powtórzeń jest jedna z zasad zdyscyplinowanego, oszczędnego i inteligentnego projektowania programów. Zasada ta ma na celu skrócenie czasu potrzebnego na napisanie programu, jak również uproszczenie struktury programu i ułatwienie jego utrzymywania. Procedury, klasy, dziedziczenie i parametryzacja dostarczają środków pomocnych w unikaniu zbędnych powtórzeń.

Może więc zdziwić fakt, że w wielu programach więcej niż 60 procent kodu zawartych jest w podobnych do siebie strukturach [3,4,6,9,10,28,31,34]. Nawet biblioteki klas zaprojektowane przez ekspertów zawierają wiele powtórzeń: 68 procent kodu biblioteki buforów w Javie [3] i 60 procent kodu kontenerów w STL (C++) [4] zawiera się w podobnych strukturach. Powtórzenia rzadko są udokumentowane, natomiast świadomość ich obecności jest niezbędna w czasie utrzymywania programu.

Kolejne wersje programów – np. wersje Linux'a dla różnych komputerów - często powstają przy użyciu techniki Kopiuj-Wklej-Modyfikuj (K-W-M). Rodzinę wersji programów cechują duże podobieństwa pomiędzy członkami rodziny, szczególnie wersjami niezbyt odległymi od siebie w historii ewolucji programu.

Rodziny wersji programów powstałe przy pomocy K-W-M często dają początek Programowej Linii Produkcyjnej (PLP), w której wszystkie wersje stare i przyszłe są tworzone ze wspólnej bazy komponentów wielokrotnego wykorzystania. Szczegółowa znajomość podobieństw pomiędzy członkami rodziny programów jest niezbędna dla identyfikacji kandydatów na komponenty wielokrotnego wykorzystania. Szczegółowa znajomość różnic pomiędzy nimi jest niezbędna do parametryzacji komponentów w celu ułatwienia przyszłej adaptacji komponentów w kontekście zmiennych wymagań na potrzeby tylko niektórych użytkowników.

Z uwagi na powyższe zastosowania, wiele automatycznych metod rozpoznawania podobieństw w programach zostało zaproponowanych w literaturze (dziedzina badań nad podobieństwami nosi nazwę software clone detection and analysis). W kontekście wielokrotnego wykorzystania programów, metody analizy bazowane na podobieństwie kodu zasługują na szczególną uwagę.

Badania nad rozpoznawaniem podobieństw skupiły się na podobnych fragmentach kodu. Trzeba jednak zauważyć, że przedmiotem wielokrotnego wykorzystania programów powinny być wzorce programowe tworzące większe struktury programowe niż fragmenty kodu. Stąd wraz z doktorantem Hamidem Basitem, badania skierowaliśmy na rozpoznawanie powtarzających się wzorców programowych. Wychodząc z pojęcia struktury programowej, jako konfiguracji fragmentów kodu, zdefiniowaliśmy podobieństwo struktur programowych w terminach podobieństwa kodu w ich fragmentach i podobieństwa ich konfiguracji. Metoda rozpoznawania strukturalnych podobieństw przebiega w następujących fazach:

Najpierw tekst programu jest analizowany w celu wyodrębnienia podstawowych jednostek leksykalnych (tokens). Następnie program w postaci łańcucha jednostek leksykalnych poddany jest analizie powtórzeń wykorzystując algorytmy kombinatoryczne, wyszukując maksymalne powtórzenia ciągów leksykalnych [5]. Powtórzenia te odpowiadają podobnym fragmentów kodu. W kolejnej fazie analizy, algorytmy wykrywania informacji (data mining) odnajdują podobne konfiguracje fragmentów, jako pierwszy szczebel podobnych struktur. Ostatnia analiza jest powtarzana w nadziei na odkrycie strukturalnych podobieństw wyższego rzędu, w których elementami konfiguracji są struktury rozpoznane w poprzedniej fazie.

Powyższą metodę rozpoznawania strukturalnych podobieństw stosowaliśmy z powodzeniem do analizy otwartych systemów skali przemysłowej (miliony rozkazów) [41,42].

W kontekście modernizacji rodzin programów przy użyciu metody PLP, strukturalne podobieństwa są manualnie transformowane w meta-komponenty wielokrotnego wykorzystania, budowane przy pomocy XVCL.

5. Badania w ostatnich latach.

W ostatnich latach kontynuowałem badania nad strukturalnymi podobieństwami i nad metodami ułatwiającymi modernizację programów.

W temacie strukturalnych podobieństw, kontynuowałem współpracę z Hamidem Basitem, formułując formalne podstawy strukturalnych podobieństw w oparciu o teorie grafów, dokonując eksperymentalnych badań ukazujących pożyteczność rozpoznawania strukturalnych podobieństw, prowadząc systematyczne badania metod wizualizacji powtórzeń w programach [62], i definiując nowe typy strukturalnych pobieństw (jest to temat bieżącej pracy doktorskiej Kuldeep'a Kumar'a pod moim kierunkiem). Dokonaliśmy również przeglądu metod wizualizacji powtórzeń programowych, co udokumentowaliśmy w artykule wysłanym do ACM Surveys.

W temacie modernizacji, pracowałem z moim doktorantem Xue Yinxing'iem i kolega z wydziału Xing Zhenchang'iem nad problemem rozpoznawania różnic w podobnych strukturach i odkrywaniu kodu implementującego specyficzne cechy programów (feature location). W tych badaniach łączyliśmy metody analizy podobieństw i PLP rozwinięte w moich projektach, z metodami różnicowania grafów zaproponowane przez Xing'a w jego doktoracie [59,61,63,66].

Ostatnio, swoje zainteresowania zarządzaniem zmienności programów rozszerzyłem na testowanie. Doktorantka Suria Asaithambi pracuje nad generycznymi bibliotekami testów dla platformy Android, w których podobne testy są zastępowane generyczna reprezentacja w XVCL, w celu kompresji i uproszczenia bibliotek testów.

--- Koniec ---

Stanisław Jarząbek