DOI: 10.24427/978-83-66391-84-0_4

Rozdział 4
Tworzenie uogólnionej architektury wersji oprogramowania z użyciem podobieństw struktur programowych

Cezary Boldak

Streszczenie: Wersje oprogramowania powstają, aby spełnić różniące się między sobą wymagania użytkowników / klientów. Pomimo istnienia technik umożliwiających generowanie takich wersji z jednego rdzenia, bardzo często powstają one przez kopiowanie i niezależne modyfikowanie wielu wersji kodu. Prowadzi to do duplikacji kodu, tzw. klonów programowych, i jest niezmiernie trudne w zarządzaniu. Architektura takich produktów, nawet jeśli przemyślana i udokumentowana pierwotnie, często ulega degradacji i traci się nad nią kontrolę. Brak znajomości aktualnej i spójnej między wersjami architektury prowadzi do wielu problemów, dlatego istnieją techniki odtwarzania takiej architektury na podstawie kodu źródłowego. Bazują one m.in. na identyfikowaniu odpowiadających sobie modułów programowych w wersjach na podstawie podobieństwa kodu, jednak zmiany naniesione w kodzie pomiędzy wersjami mogą zakłócić odnalezienie takich modułów. Dodanie do takich technik analizy klonów, mogącej zidentyfikować tylko podobne fragmenty i większe struktury kodu, powinno zwiększyć jakość odtwarzanej, uogólnionej architektury. W Pracy zaproponowano metodę uwzględnienia analizy klonów w metodach odtwarzania uogólnionej architektury wersji oprogramowania.

Słowa kluczowe: architektura oprogramowania, klony programowe, wersje oprogramowania

Wprowadzenie

Oprogramowanie jest obecne praktycznie w każdym aspekcie otaczającego nas środowiska. Od dedykowanych komputerów „zawędrowało" nawet wgłąb ludzkiego ciała, i nie jest to z pewnością proces zakończony. Ciągle rosnące zapotrzebowanie na programy spowodowało presję na ich szybsze, i – co z tym często związane – tańsze, dostarczanie. Aby zaspokoić tę potrzebę, w sytuacji braku czasu oraz niedostatku wykwalifikowanych twórców oprogramowania, idzie się na skróty – nowe programy powstają przez modyfikację programów istniejących, o zbliżonych funkcjonalnościach, bardzo często przez zwykłe skopiowanie i zmianę ich kodu. Wkrótce nad kolejnymi wersjami nikt nie panuje – nie wiadomo co w nich jest, dokumentacji

brak lub jest ona dawno zdezaktualizowana wskutek zmian nanoszonych na pierwotną formę, pierwotny zespół wytwórczy dawno już odszedł z firmy. Jeśli jednak zachodzi potrzeba modyfikacji takich wersji, brak wiedzy o konstrukcji programu jest chyba pierwszą przeszkodą, którą trzeba przezwyciężyć. Istnieją oczywiście metody odtwarzania architektury oprogramowania, i choć nie zawsze dają w pełni satysfakcjonujące efekty, to jednak stanowią dużą pomoc, zwłaszcza kiedy są w dużej mierze zautomatyzowanie. Jednak kiedy istnieje wiele wersji programu, w których objawił się (ten sam ?) błąd i kiedy czas „goni", chcielibyśmy prawdopodobnie poprawić go we wszystkich wersjach. Dlatego pożądane byłoby całościowe potraktowanie wszystkich wersji, żeby odnaleźć analogiczne miejsca w nich wszystkich. Możliwości takie oferuje uogólniona architektura, która opisuje wiele wariantów oprogramowania, wskazując podobieństwa i różnice między nimi. Jej istnienie jest już dużo rzadsze niż indywidualnych architektur każdej z wersji, ale i tu zaproponowane zostały techniki jej rekonstrukcji. W opinii autora w zbyt małym stopniu wykorzystują one fakt, że wersje te powstały z jednego kodu i pomimo zmian podobieństwo na poziomie kodu źródłowego przetrwało prawdopodobnie pomiędzy wersjami. W prezentowanej pracy podjeto próbę zaproponowania połączenia analizy klonów programowych, dziedziny zajmującej się właśnie podobieństwami w kodzie źródłowym programu lub programów, z istniejącymi technikami odtwarzania uogólnionej architektury oprogramowania istniejącego w wielu wersjach na podstawie kodu źródłowego tych wersji. Oprócz drobnych modyfikacji tych technik zaproponowano też nowe koncepcje, zmieniające te techniki w większym zakresie.

W podrozdziale 4.1 przedstawiono architektury oprogramowania proponowane przez różnych autorów, w 4.2 wymieniono kilka technik odtwarzania architektury pojedynczych systemów na podstawie ich kodu źródłowego, w 4.3 omawiono koncepcję linii produkcyjnych oprogramowania (SPL), mających usystematyzować wytwarzanie wielu wersji produktów, a w 4.4 – istniejące techniki tworzenia uogólnionej architektury, stosowanej w SPL. Podrozdział 4.5 poświęcono analizie klonów, ze szczególnym uwzględnieniem klonów strukturalnych. Zasadniczą część pracy stanowi podrozdział 4.6: przedstawiono tu propozycje uwzględnienia analizy klonów przy odtwarzaniu uogólnionej architektury. Pracę zakończono podsumowaniem.

4.1. Architektura oprogramowania

Architektura oprogramowania – czym jest ? Każdy jego twórca ma tu pewną intuicję, ale o jednolitą, powszechnie akceptowaną definicję trudno. Można podejść do architektury oprogramowania od bardziej technicznej strony, wyliczając co się musi w niej znaleźć, na poziomie bardziej abstrakcyjnym lub bardziej konkretnym. Według [1] w projektowaniu oprogramowania poziom architekturalny to coś co wychodzi ponad algorytmy i użyte struktury danych, obejmując organizację i globalną strukturę kontroli, protokoły komunikacyjne, synchronizacyjne, dostępu do danych, …

Standard IEEE [2] definiuje architekturę jako fundamentalną organizację systemu wyrażoną poprzez jego komponenty, zależności pomiędzy nimi oraz między komponentami i ich otoczeniem oraz zasady kierujące projektowaniem i ewolucją systemu. Bazując na tej definicji RUP [3] określa architekturę jako zbiór decyzji nt.: organizacji systemu, wyboru jego podstawowych komponentów i oferowanych przez nie interfejsów oraz ich działania i współdziałania, kompozycji tych komponentów w większe struktury, stylu architekturalnego wg którego podejmowane są poprzednie decyzje. Model widoków „4+1" [4] precyzuje jakie artefakty muszą znaleźć się w architekturze oprogramowania w celu udokumentowania perspektywy różnych udziałowców (stakeholders).

Jednak takie techniczne podejście nie jest jedyne. Fawler [5] widzi architekturę bardziej jako konstrukcję społeczną – należą tu rzeczy nie z definicji, ale ponieważ najważniejsze osoby w projekcie sądzą, że powinny się w niej znaleźć. Ponieważ architektura to decyzje, które najlepiej podejmować na początku projektu, to wynika stąd, że architektura to rzeczy, które ludzie postrzegają jako trudne do zmiany (dlatego podejmowane są na początku projektu). Przykładowo schemat bazy danych, determinujący większość kodu w aplikacji, będzie należał do architektury, ale jeśli zmiana takiego schematu jest łatwa i (najlepiej) automatyczna, to raczej będzie to detal a nie element architektury. W taki sposób, dążąc do łatwej zmiany wszystkich kluczowych elementów w projekcie, staramy się pozbawić go architektury ! Jednak jeśli jest mowa o automatycznym konstruowaniu architektury na podstawie kodu źródłowego bardziej właściwe będą tu techniczne definicje architektury określające je jako zbiór komponentów, czasem hierarchicznie uporządkowanych, współpracujących ze sobą i oferujących określone funkcjonalności (usługi) za pośrednictwem interfejsów.

4.2. Odtwarzanie architektury programowania na podstawie kodu źródłowego

W teorii powstanie architektury powinno poprzedzać wytworzenie innych artefaktów, w tym kodu źródłowego. Jednak jeśli nawet tak jest, to rzadko architektura jest uaktualniana w miarę zmian, którym podlega system. Nawet podczas implementacji mogą występować odstępstwa od architektury (czasem konieczne) powodujące rozbieżności między projektem a realizacją.

Istnienie aktualnej (!) architektury, nawet jeśli nie wymagane formalnie, ułatwia wiele zadań: rozbudowę, refaktoryzację, naprawę błędu, wprowadzanie nowych członków do zespołu opiekującego się systemem w fazie poprodukcyjnej. Wynika stąd potrzeba (re)konstrukcji architektury (lub przynajmniej jej elementów) na podstawie dostępnej informacji, którą może być kod źródłowy systemu.

Odtwarzanie architektury oprogramowania rozumiane jako grupowanie (klastering) podstawowych bloków (funkcje, moduły, ...) w większe komponenty może być wykonane przy użyciu wielu technik [6,7]: hierarchicznego grupowania [8],

algorytmów genetycznych [9] i grafowych [10], optymalizacji [11]. Źródłem danych do wyznaczenia architektury może być kod źródłowy i wszelkie statyczne informacje, które można z niego pozyskać (nazwy funkcji/metod, wzajemne wywoływanie, dostęp do danych, ...), ale także dynamiczne informacje uzyskane podczas wykonania zinstrumentowanych programów [12] czy też metainformacje: autorzy kodu czy umiejscowienie/nazwy plików [13].

4.3. Linie produkcyjne oprogramowania (SPL)

Systemy, które odniosły sukces, rzadko ograniczają się do jednej grupy użytkowników. Naturalne jest dążenie do ich adaptacji tak, aby mogły być użyte przez inne grupy użytkowników mających podobne, ale też różne wymagania względem systemu. Niestety, podobieństwo funkcjonalności nie oznacza ich identyczności, co zmusza twórców do opracowania nowych wersji, aby zastosować je wobec nowych wymagań. Nowe wersje mogą powstać na bazie oryginału, przez skopiowanie i zmodyfikowanie źródeł, co stanowić będzie znaczącą „pokusę" wobec znaczących oszczędności czasu i środków. „Pokusa" ta, niestety, „zemści" się prawdopodobnie w przyszłości, kiedy trzeba będzie zarządzać wieloma niezależnymi wersjami podobnego kodu – wszelkie zmiany, poprawki będą musiały być aplikowane do wielu linii bazowych (*baseline*) kodu.

W zorganizowaniu systematycznego procesu produkcji oprogramowania w wielu wersjach, zaproponowano koncepcję linii produkcyjnych oprogramowania (SPL – *software product lines*) [14]. Kolejne wersje systemów powstają poprzez łączenie gotowych komponentów – półproduktów (*assets*) pobranych z repozytorium. Każda wersja systemu może zawierać inny wybór takich komponentów, co jest pierwszym poziomem zmienności na najwyższym, architekturalnym poziomie. Wybór taki nie jest banalny, gdyż mogą pojawiać się zależności pomiędzy komponentami: komponenty wykluczające się, wymagane, alternatywne, ... Jednak taki poziom zmienności nie jest wystarczający, gdyż rzadko kiedy komponenty są na tyle generycznie napisane, aby spełniały wymagania wszystkich wersji produktu. Zwiększenie generyczności komponentu łączy się zwykle z jego większym skomplikowaniem w budowie i użyciu, mniejszą wydajnością. Dlatego tu też pojawią się kolejne wersje – tym razem komponentów – na niższym poziomie abstrakcji. Taką zmiennością komponentów też trzeba zarządzać, aby uniknąć powstania odrębnych wariantów po użyciu antywzorca: programowania w stylu *copy-paste-modify*. Użyć można wielu dostępnych technik: warunkowa kompilacja, metaprogramowania (szablony na wzór C++, klasy generyczne, ...), mechanizmy programowania obiektowego (dziedziczenie i kompozycja), wzorce projektowe [15], czy też dedykowanych rozwiązań typy język XVCL [16] (lub jego następna [17]). Wszystkie te wspomniane wyżej elementy (zestaw komponentów, ich wersje i wzajemne zależności, sposób doboru komponentów w określonej wersji w wariantach produktu) stanowią architekturę linii produkcyjnej oprogramowania, którą można nazwać architekturą uogólnioną, w odróżnieniu od konkretnej architektury określonego wariantu produktu.

4.4. Tworzenie uogólnionej architektury wariantów oprogramowania na podstawie wariantów produktów

Pomimo oczywistych zalet stosowania linii produkcyjnych oprogramowania, często powstaje wiele wariantów systemów informatycznych jako niezależnych produktów, czy to z powodu ich wieku, czy to nieświadomości twórców, czy to nieopłacalności opracowania SPL wobec niedoszacowanych planów rozwoju systemu [18]. Aby choć trochę ułatwić zarządzanie takimi wariantami, ewentualnie wspomóc wytworzenie linii produkcyjnej, wskazane jest opracowanie uogólnionej architektury na podstawie już istniejących wariantów produktów, zwykle mając do dyspozycji ich kody źródłowe [19].

PuLSE [20] używa adapterów (*wrappers*), aby przekształcić komponenty istniejących wariantów systemu w komponenty generyczne, możliwe do wielokrotnego użycia (*reusable*). Nie powstaje uogólniona architektura, ale takie podejście może być wykorzystane jako etap w procesie rekonstrukcji takiej architektury, zapewniając jej elementy składowe.

Metoda MAP (*Mining Architectures for Product Lines*), [21] proponuje procedurę (główne manualną) analizy architektur istniejących produktów w celu oceny, czy zawierają one potencjał do zbudowania z nich linii produkcyjnej oprogramowania. Potencjał ten jest rozpatrywany poprzez wskazanie elementów wspólnych (*commonalities*) i różnych (*variabilities*) na poziomie komponentów. Metoda rozpoczyna pracę od zbudowania modelu implementacyjnego (relację między jednostkami programowymi) każdej z wersji produktów oraz sformowanie (ręczne) komponentów (etap *bottom-up*). Następnie na taki model komponentów mapuje się (z użyciem wiedzy ekspertów) znane style architekturalne (etap *bottom-up*). Efekty pracy – niezależne architektury wersji produktów – są następnie używane do sformułowania wniosków na temat zasadności budowy SPL, jednak nie powstaje tu uogólniona architektura.

W ramach europejskiego projektu CAFÉ [22] uogólniona architektura (nazywana architekturą referencyjną – *reference architecture*) jest konstruowana przez analizę istniejących wariantów systemów, które mają być scalone w jednolitą linię produkcyjną. Jako etap wstępny wykonywana jest (re)konstrukcja architektur tych indywidualnych wariantów. Podobne podejście prezentowane jest w [23], gdzie na początku budowane są architektury pojedynczych członków rodziny oprogramowania sterującego robotami. Następnie budowany jest uogólniony model cech (*feature model*), który jest wzbogacany o możliwości adaptacji w celu zwiększenia ich uniwersalności. Na tej podstawie tworzona jest uogólniona architektura linii produkcyjnej, wraz z zestawem komponentów budujących pojedyncze warianty.

W pracach [24, 25] używa się uogólnionej metody refleksji Murphy'ego, aby porównać warianty oprogramowania na poziomie architektury. Oryginalna metoda refleksji działa dla pojedynczych programów, gdzie rzeczywiste elementy struktury programu mapowane są na hipotetyczny model architektury (przybliżony,

nieaktualny, ...) – w iteracyjnym procesie zmieniającym obydwie strony mapowania przybliżana jest rzeczywista architektura. Modyfikacja w celu zastosowania do SPL zakłada 2 procesy uzgadniania: implementacji z architekturą pojedynczego wariantu oraz architektury pojedynczego wariantu z uogólnioną architekturą SPL. Te 2 procesy wykonywane są sekwencyjnie dla każdego wariantu, przy czym mapowania w danym kroku musi uwzględniać (i ewentualnie modyfikować) poprzednie fazy. Po analizie ostatniego wariantu otrzymywana jest finalna uogólniona architektura SPL. Uogólniona architektura jest rozszerzeniem UML przez wprowadzenie komponentów obowiązkowych, opcjonalnych i wariantowych. W celu identyfikacji odpowiadających sobie jednostek programowanych (np. funkcji) pomiędzy wersjami (co jest potrzebne w uzgadnianiu mapowań pomiędzy krokami sekwencji) zastosowano analizę prostych klonów (podobieństw w kodzie źródłowym).

[26] proponuje technikę wydobywającą informację o zmienności w rodzinie produktów. Mapuje ona zbiór cech (*features*) i ich interakcji na artefakty programowe z założeniem, że zbiór tych cech jest znany. Relacje między cechami określone są przez ich model (graf). Mapowanie pozwala np. na wybór wariantu, który zawiera podzbiór cech najbardziej zbliżony do cech produktu, który powinien być wytworzony, aby proces tworzenia (za pomocą techniki *copy-paste-modify*) zacząć od takiego istniejącego wariantu.

Zaproponowana ostatnio metoda odtworzenia architektury SPL [27] używająca formalnej analizy konceptualnej (FCA – *formal concept analysis*) oferuje praktycznie automatyczne jej zbudowanie na podstawie kodu źródłowego wariantów. Tu również na początku odtwarzane są architektury poszczególnych wariantów. Następnie identyfikuje się występujące w nich komponenty w różnych wariantach. Odpowiedź na pytanie, czy użyte komponenty z poszczególnych architektur stanowią różne warianty jednego komponentu bazowego, uzyskuje się porównując serwisy przez nie oferowane: obowiązkowe (obecne we wszystkich wariantach) i opcjonalne (obecne jedynie w niektórych wariantach). Serwisy te reprezentowane są przez nazwy publicznych klas zawartych w komponencie. Komponenty z różnych architektur łączone są za pomocą hierarchicznego grupowania, używając kosinusowej miary podobieństwa (opartej na nazwach klas), produkując dendrogram. Węzły w dendrogramie spełniające określony warunek (podobieństwo elementów węzła przewyższa średnie podobieństwo w potomkach) wyznaczają komponenty bazowe, liście węzłów stanowią zaś komponenty-warianty komponentu bazowego. Dla komponentów bazowych określa się ich zmienności (*variabilities*): wewnętrzną (jako zbiór obowiązkowych i opcjonalnych klas) oraz zewnętrzną (jako zbiór obowiązkowych i opcjonalnych interfejsów, wymaganych i oferowanych przez komponent). Oferowane interfejsy w komponentach są reprezentowane przez serwisy (klasy) komponentów. Decyzja czy poszczególne interfejsy są uznawane za podobne (odpowiadające sobie) w wariantach komponentów zapada na podstawie podobieństwa kodu źródłowego implementujących je metod. Na końcu określa się komponenty obowiązkowe i opcjonalne w całej uogólnionej architekturze SPL. Dodatkowo określane są zależności pomiędzy komponentami: alternatywa, AND, OR, wymaganie, wykluczanie.

Niektóre warianty produktów mogą zbyt daleko odejść od pierwowzoru, stanowiąc *de facto* odrębny produkt. Próba uwzględnienia takich odstających wariantów (*outliers*) będzie stanowić duży problem przy odtwarzaniu uogólnionej architektury. W [28] proponuje się ich oddzielenie, aby finalna architektura SPL miała lepszą jakość. Na początku pozyskuje się informację strukturalną z każdego z wariantów. Uogólnia się też nazwy klas (manualnie), aby poprawnie identyfikować odpowiadające sobie klasy w różnych wariantach pomimo zmienionych nazw (w procesie powstawania wariantów). Informacje te są porównywane ze sobą pod kątem zmienności, filtrując niepasujące warianty za pomocą formalnej analizy konceptualnej (FCA). Decyzje o wyodrębnieniu komponentu (pakietu Javy) i o wykryciu bytów odstających zapadają na podstawie analizy progów określonych metryk. Wytworzona uogólniona architektura SPL ma postać: macierzy struktury (DSM – *design structure matrix*), widoku pakietów i klas z zaznaczonymi elementami obowiązkowymi i opcjonalnymi, siatki konceptów (*concept lattice*). Metoda jest nieodporna na podobne (a nie identyczne) odpowiadające sobie fragmenty kodu w różnych wariantach.

4.5. Analiza klonów

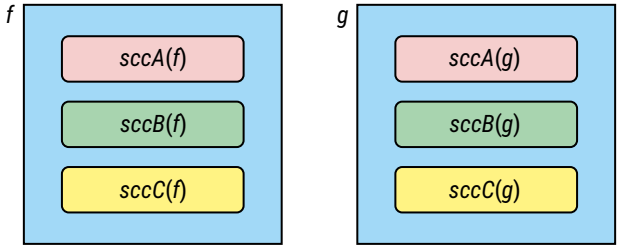
Podobieństwa w systemach informatycznych są powszechne, na różnych poziomach (wymagania, modele, kod źródłowy, …). W tej pracy rozważane będą klony programowe (*code clones*), które występują w kodzie źródłowym oprogramowania. Wg różnych oszacowań w dużych systemach sklonowany kod stanowi 20%–50% całości [29]. Ich źródłem najczęściej jest programowanie w stylu *copy-paste-modify*, choć nie można też wykluczyć podobnych fragmentów napisanych niezależnie. W niektórych przypadkach istnienie klonów jest zamierzone i pożądane [29], jednak w większości sytuacji dąży się do wyeliminowania istnienia klonów, ponieważ podobne struktury programowe są źródłem redundancji, co zwiększa objętość kodu, a co gorsza powoduje problemy w zarządzaniu nim (modyfikacje, naprawy błędów, niezależne refaktoryzacje, …). W tym celu w procesie refaktoryzacji można zaproponować generyczne, adaptowalne do różnych wymagań komponenty, których sparametryzowane użycie w miejsce klonów likwiduje redundancję. Aby jednak rozpocząć takie prace, należy te klony zidentyfikować. Zadanie to stanowi przedmiot licznych prac badawczych [30].

Identyfikacja identycznych fragmentów kodu jest często niewystarczająca. Oczywiście techniki detekcji klonów uwzględniają drobne zmiany w kodzie: usunięcia, wstawienia, zmiany kolejności fragmentów, zmiany nazw. Jednak w przypadku nieidentycznego kodu pojawia się problem, od kiedy uznać, że podobieństwo jest wystarczające, aby uznać, że fragmenty kodu stanowią klon. Proponowane metryki mogą określić wymagany próg podobieństwa, ale ich dobór zawsze jest subiektywny.

Zidentyfikowane, wzajemnie podobne do siebie miejsca w różnych fragmentach oprogramowania stanowią pojedynczą klasę klonów (*clone class*), a wyżej wymienione podobne miejsca są instancjami tej klasy.

4.5.1. Klony strukturalne

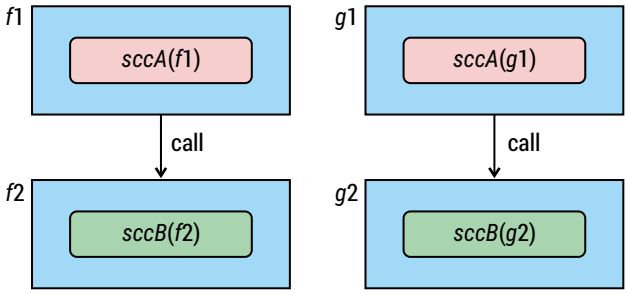
Oprócz prostych klonów zlokalizowanych w pojedynczych jednostkach programowych (metodach, czasem klasach), wyróżnia się klony strukturalne [31]. Pozwalają one wykorzystać fakt, że często obserwowane są powtarzające się takie same sekwencje wystąpienia klonów (dokładniej – instancji klas klonów prostych – *SCC – simple clone class*) w różnych miejscach w kodzie. Przykładowo, jeśli w jednej funkcji *f* występuje sekwencja instancji klas klonów prostych *sccA*(*f*), *sccB*(*f*), *sccC*(*f*) a w innej funkcji *g* taka sama sekwencja instancji klas klonów prostych *sccA*(*g*), *sccB*(*g*), *sccC*(*g*), to można mówić o klonie strukturalnym (rys. 4.1). Wyróżnić można klasy klonów strukturalnych na poziomie funkcji/metod (sekwencje klonów prostych występują w pojedynczych funkcjach/metodach), klas (programowania obiektowego), plików, katalogów, …



RYSUNEK 4.1. **Przykład strukturalnej klasy klonów na poziomie funkcji**

4.5.2. Współpracujące klony strukturalne

Jeśli dodatkowo sekwencje instancji klas klonów prostych występują w odpowiadających sobie konfiguracjach powiązanych strukturach programowych, to można mówić o współpracujących klonach strukturalnych (*collaborative structural clones*) [31]. Przykładowo, jeśli jako struktury programowe weźmiemy funkcje powiązane relacją wywołania, i jeśli znajdziemy dwie pary funkcji: *f1* woła *f2*, zaś *g1* woła *g2*, przy czym w *f1* i w *g1* zlokalizowane są 2 instancje jednej klasy klonów prostych *sccA*(*f1*) i *sccA*(*g1*) zaś w f2 i w g2 zlokalizowane są 2 instancje innej klasy klonów prostych *sccB*(*f2*) i *sccA*(*g2*), to taka konfiguracja stanowi klasę współpracujących klonów strukturalnych (rys. 4.2).



RYSUNEK 4.2. Przykład klasy współpracującego klonu strukturalnego występującego w parach wołających się funkcji

Odnajdywanie klonów strukturalnych, a zwłaszcza współpracujących klonów strukturalnych, może stanowić duże wsparcie do konstruowania generycznych komponentów do różnych zastosowań, np. linii produkcyjnych oprogramowania.

4.6. Analiza klonów w tworzeniu uogólnionej architektury

Zaproponowane ostatnio techniki odtwarzania uogólnione architektury wydają się działać poprawnie na przedstawionych przykładach [25, 27]. Nasuwa się jednak spostrzeżenie, że w zbyt małym stopniu wykorzystują one podobieństwa w kodzie analizowanych wariantów oprogramowania. Podobieństwo takie jest naturalną konsekwencją trybu powstawania tych wariantów: najczęściej *copy-paste-modify*. Rzadko kiedy kod pochodny jest modyfikowany w takim stopniu, żeby nie można było znaleźć podobieństw i skorzystać z nich. Podobieństwa takie mogą wskazać kandydatów dla generycznych modułów programowych na najniższym poziomie (szablony klas i metod/funkcji, klasy i metody generyczne, konstrukcje ART [17]). Pozwolić to może na tworzenie bibliotek używanych potem w wariantach, zmniejszając w nich redundancję i ogólnie podnosząc jakość. Zbuduje się też zasób generycznych modułów o dużym potencjale wielokrotnego użycia także w kolejnych produktach – skoro dowiodły swojej użyteczności w przeszłości. Jednak to tylko początek możliwości. Istniejące techniki w wielu miejscach odwołują się do analogicznych klas, metod występujących w wariantach. Jakże często podobieństwo to jest określane poprzez porównanie ich nazw lub identyczność kodu. Analiza klonów prostych może to zadanie wykonać o wiele lepiej, oferując porównanie zmodyfikowanych fragmentów kodu na poziomie pojedynczych modułów. Jeszcze większe możliwości oferują klony strukturalne, badające konfiguracje klonów prostych i wykazujące podobieństwo modułów na wyższym poziomie abstrakcji. Ale to współpracujące klony strukturalne, które analizują całe struktury modułów, wydają wprost stworzone do użycia w przypadku odkrywania natury wariantów. Ich analiza na poziomie wyższym niż pojedyncze klasy czy pliki wydaje się idealnie pasować do konstrukcji architektury linii

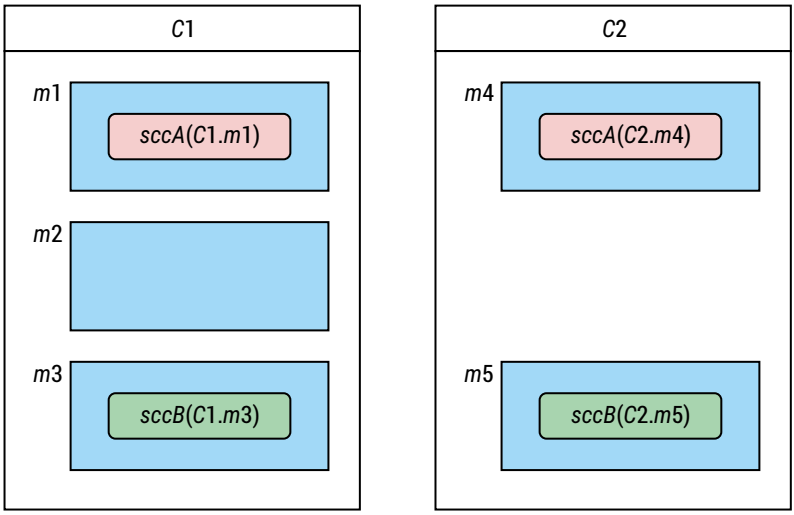
produkcyjnych oprogramowania, gdzie rozpatrywane są całe komponenty, zbudowane przecież ze współpracujących modułów programowych. Zwykle ich identyfikacja jest oddzielnym etapem proponowanych metod – tu gotowe uogólnione komponenty może zaproponować analiza klonów.

4.6.1. Użycie analizy klonów w istniejących technikach tworzenia uogólnionej architektury

Można tu wskazać kilka wymienionych wyżej technik tworzenia uogólnionej architektury, w których analiza klonów mogłaby poprawić działanie.

Technika oparta na uogólnionej metodzie refleksji Murphy'ego [24, 25] używa do znalezienia odpowiadających sobie modułów programowych (np. funkcji) w różnych wariantach produktu analizy klonów prostych. Obecność instancji klonów w porównywanych fragmentach kodu stwierdza się po przekroczeniu przez wybrane metryki (np. liczba linii kodu, instrukcji lub tokenów w zlokalizowanym klonie) określonych progów. Jeśli progi mają zbyt niską wartość, wykrytych klonów jest zbyt dużo, dlatego część z nich odrzuca się przez podwyższenie progu. Z drugiej strony zbyt wysoka wartość progu spowodować może przeoczenie ważnych par (lub zbiorów) podobnych modułów, które są swoimi odpowiednikami w wariantach produktów. Małe klony mogą występować w modułach seriami, przedzielone jednak wstawkami różniącego się kodu wystarczająco dużymi, aby przeszkodzić w uznaniu całości za instancję jednego prostego klonu. Klony strukturalne, skomponowane z sekwencji klonów prostych (próg akceptacji może być niski), stanowią swoisty filtr, odrzucający pojedyncze małe klony i przepuszczający ich sekwencje. Dobór progu akceptacji klonów prostych staje się mniej ważny – może być ustawiony na niższą wartość bez obawy o wskazanie zbyt wielu par (zbiorów) skojarzonych modułów programowych. Zastosowanie analizy klonów strukturalnych w miejsce klonów prostych ma potencjał poprawienia jakości wyprodukowanej uogólnionej architektury.

W [28] występuje manualny proces uogólnienia nazw klas, które podczas tworzenia nowych wariantów uległy zmianie. Klasy o takich samych nazwach w odrębnych wariantach uznawane są za odpowiadające sobie, więc ich ujednolicenie jest kluczowe w identyfikacji uogólnionych komponentów. Jak sami autorzy przyznają, podobne, choć różne fragmenty kodu mogą powodować błędne działanie metody. Remedium na to wydaje się proste – analiza klonów. Odpowiadające sobie klasy o różnych nazwach z łatwością zostaną zidentyfikowane przez analizę współpracujących klonów strukturalnych. Wystarczy jako struktury programowe (wskazane przykładowo na rys. 4.2 jako funkcje) przyjąć metody (i ewentualnie dodatkowo pola), a jako zależności pomiędzy nimi (wskazane na rys. 4.2 jako wywoływanie się) – przynależność do tej samej klasy – rys. 4.3. Byłoby to rozwiązanie automatyczne, w miejsce proponowanego manualnego ujednolicania nazw. Nazwy metod również nie miałyby tu znaczenia – liczyłoby się jedynie istnienie porównywalnych konfiguracji klonów.



RYSUNEK 4.3. Identyfikacja odpowiadających sobie klas *C1* i *C2* o różnych nazwach, metodach ale wykazujących istnienie współpracujących klonów strukturalnych. Takie współpracujące klony strukturalne mogą stanowić klasę klonów klasowych (CCC)

W [27] po odtworzeniu architektury wszystkich wariantów produktów identyfikowane są uogólnione komponenty przez hierarchiczne grupowanie (klastering) konkretnych komponentów z różnych wariantów produktów. Podobieństwo komponentów (potrzebne do wyboru scalanych w danym kroku – najbardziej podobnych) określa się poprzez kosinusową miarę podobieństwa określoną na nazwach klas-serwisów oferowanych przez każdy komponent.

Miara ta, używana w określaniu podobieństwa dokumentów tekstowych, jest wyrażona przez kosinus kąta pomiędzy dwoma wektorami zawierającymi częstotliwość występowania słów w obu dokumentach. Po przystosowaniu do komponentów i klas, dwa wektory reprezentujące komponenty zawierają tyle pozycji, ile jest unikalnych klas (nazw klas w opisywanej metodzie) w obu komponentach. Pozycje wektora zawierają wartości 0 (jeśli komponent nie zawiera klasy przyporządkowanej tej pozycji) i 1 (jeśli komponent zawiera taką klasę).

W wersji użytej w opisywanej metodzie miara będzie niedoskonała, jeśli nazwy klas się zmienią (nie ma tu nawet manualnego ujednolicania nazw znanego z [28]). Ponadto, ta sama nazwa może ukrywać klasy, które zmieniły się wzajemnie tak bardzo, że nie powinny być uważane za podobne.

Tak jak opisano wyżej, odpowiadające sobie klasy można identyfikować poprzez użycie współpracujących klonów strukturalnych (rys. 4.3). Następnie można kontynuować używanie miary kosinusowej opierając się na określeniu odpowiadających sobie klas nie poprzez zgodność ich nazw, ale przez występowaniu w nich tych samych instancji współpracujących klonów strukturalnych. W obu wektorach będą zatem pozycje reprezentujące klasy współpracujących klonów strukturalnych: z wartościami 0 jeśli komponent nie zawiera tego klonu, i 1 jeśli zawiera go.

Zamiast tego można też zaproponować konstrukcję uogólnionych komponentów bezpośrednio z kodów źródłowych komponentów, bez przechodzenia przez fazę odtwarzania ich indywidualnych architektur (jak opisano poniżej).

Dodatkowo w [27] zidentyfikowane uogólnione komponenty opisywane w zakresie zewnętrznej zmienności określonej poprzez interfejsy (wymagane i oferowane). Interfejsy w wariantach komponentów są uznawane za odpowiadające sobie na podstawie podobieństw kodu źródłowego implementujących je metod. Tu również analiza klonów prostych oraz klonów strukturalnych bardzo dobrze spełniłaby swoją rolę.

4.6.2. Współpracujące klony strukturalne w konstrukcji uogólnionych komponentów

Można się pokusić o użycie współpracujących klonów strukturalnych, przeniesionych z poziomu zbioru metod w klasach (rys. 4.3) na poziom współpracujących (połączonych) klas (wraz z ich metodami i polami), aby wspomóc bezpośrednią konstrukcję uogólnionych komponentów bezpośrednio z kodów źródłowych.

Mamy tu do czynienia z dwiema siłami określającymi formowanie uogólnionych komponentów. Po pierwsze, aby komponent był uogólniony, musi mieć swoje instancje (konkretne komponenty) w wariantach produktów, w identycznej lub zbliżonej formie. Po drugie, aby klasy utworzyły komponent, muszą razem wykazywać pewne charakterystyki. Można tu za [32] wymienić: (1) autonomiczność (możliwość użycia bez dodatkowych zależności), (2) zdolność komponentu do łączenia z innymi komponentami (*composability*) (3) specyficzność (*specificity*) rozumianą jako (minimalny) zbiór realizowanych funkcjonalności, współzależnych od siebie i trudnych do rozdzielenia. Gdyby wyrazić te pożądane cechy dobrego komponentu w formie wymaganej (symetrycznej) relacji między jego klasami (definiujących strukturę), można by połączyć te 2 siły właśnie w analizę współpracujących klonów strukturalnych. Tylko klasy pozostające w takiej relacji byłyby rozpatrywane jako miejsce wystąpienia instancji współpracującego klonu strukturalnego, a sam klon wyznaczyłby uogólniony komponent.

Przykładowo, wywoływanie metody jednej klasy przez drugą mogłoby definiować symetryczną relację między nimi. Wyrażałaby ona dążenie do formowania autonomicznych komponentów, ograniczając ich zależność od zewnętrznych modułów. Jednak prawdopodobnie skutkowałaby formowaniem zbyt dużych komponentów, w skrajnym przypadku jednego komponentu zbudowanego ze wszystkich klas systemu – przecież klasy komponentu są używane (wołane) z innych komponentów, a wyżej wymieniona relacja chciałaby umieścić 2 klasy: wołaną i wołającą w jednym komponencie. Innym przykładem potencjalnej relacji jest korzystanie przez obie klasy z tego samego zasobu. Taki rodzaj relacji reprezentowałby dążenie do formowania spójnych komponentów, w których klasy/metody mają te same zależności. Jednak formowane komponenty mogłyby być tu zbyt małe, ignorując zależności innego rodzaju (tak jak jak rozważane wcześniej wołanie jednej klasy z drugiej). Jednym słowem

określenie takiej symetrycznej relacji między klasami, która skutkowałaby łączeniem klas w dobrej jakości komponenty a jednocześnie stanowiłaby podstawę do użycia współdzielonych klonów strukturalnych, wydaje się bardzo trudne.

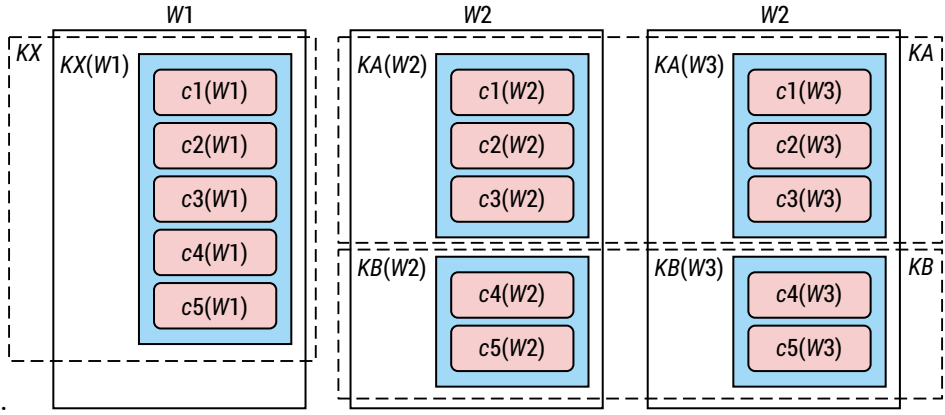
Jednak skoro określenie struktury za pomocą symetrycznej relacji jest tak trudne, to można by ją wyznaczyć w inny sposób – bezpośrednio za pomocą technik formujących komponenty w indywidualnych architekturach [32]. Klasy włączone do komponentów stanowiłyby struktury, w których szukałoby się współpracujących klonów strukturalnych pomiędzy wariantami. Znalezienie klasy takich klonów równałoby się wyodrębnieniu uogólnionego komponentu.

4.6.3. Bezpośrednie tworzenie uogólnionej architektury z uogólnionych komponentów

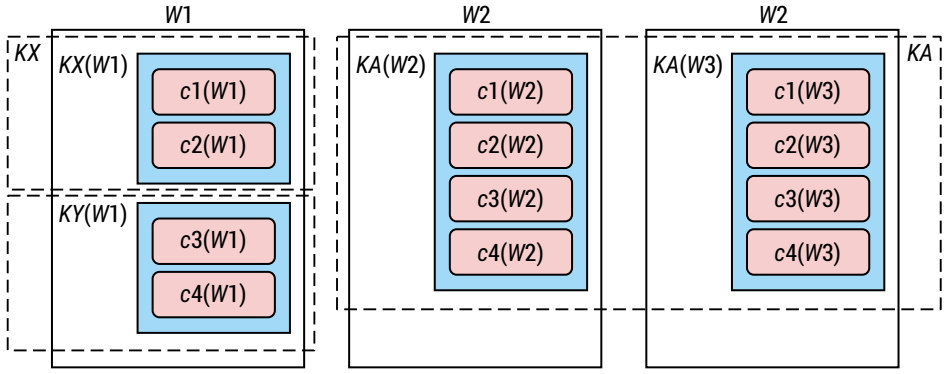
Opisane w podrozdziale 4.4 metody odtwarzania uogólnionej architektury wariantów oprogramowania zwykle zaczynają od odtworzenia indywidualnych architektur każdego wariantu. Dopiero z nich wydobywana jest informacja o elementach wspólnych i zmiennych. Na tym też etapie dochodzi do identyfikowania uogólnionych komponentów [27], gdzie konkretne komponenty wariantów są łączone w procesie hierarchicznego klasteringu. Podejście takie można określić mianem „**architektury najpierw**" (ARF – *architecture recovery first*). Ewentualne błędy popełnione na etapie formowania konkretnych komponentów skutkują błędnym wyodrębnieniem komponentów uogólnionych. Można sobie wyobrazić sytuację, gdzie konkretny komponent *KX*(*W1*) w jednym wariancie *W1* powstał przez połączenie zbyt wielu klas, które w innych wariantach *Wx* (x =/= 1) znalazły się w dwóch odrębnych komponentach *KA*(*Wx*) i *KB*(*Wx*). Taki zbyt duży komponent, albo zostanie zaliczony do uogólnionego komponentu *KA,* albo *KB*, albo też spowoduje powstanie uogólnionego komponentu *KX* (obecnego tylko w jednym wariancie *Wx*) – rys. 4.4. Taki „uogólniony" komponent w zasadzie nie zasługuje na to miano, gdyż jest obecny tylko w jednym wariancie. Jego potencjał powtórnego wykorzystania (*reusability*) jest ograniczony – może on znaleźć zastosowanie tam, gdzie powinny zostać użyte 2 niezależne komponenty *KA* i *KB*, konkurując z nimi. Jednak dwa niezależne komponenty będą tu lepsze niż jeden zespolony, gdyż bardziej uniwersalne, możliwe do zastosowania niezależnie od siebie, kiedy funkcjonalność oferowana przez drugi z pary jest niepotrzebna. Dlatego istnienie *KX* jest nadmiarowe.

Może też być tak, że w wariancie *W1* powstaną 2 odrębne, zbyt małe komponenty *KX*(W1) i *KY*(*W1*), które w innych wariantach *Wx* zostaną włączone do jednego komponentu *KA*(*Wx*). Przy formowaniu uogólnionych komponentów, do KA zostanie włączony albo *KX*(*W1*), albo *KY*(*W1*), albo żaden. Niewłączone komponenty z *W1* prawdopodobnie sformują niezależne uogólnione komponenty *KX* lub/i *KY* obecne tylko w *W1* – rys. 4.5. Można tu przez analogię z poprzednim przypadkiem stwierdzić, że 2 mniejsze komponenty będą lepsze niż jeden, zespolony. Jednak fakt, że w większości przypadków klasy *c1-c4* zostały włączone

do jednego komponentu pozwala sądzić, że jest to bardziej powszechna i ogólna sytuacja, zaś ich rozdzielenie w wariancie *W1* jest albo wynikiem błędu, albo bardzo szczególnej konfiguracji.



RYSUNEK 4.4. Zbyt duży komponent *KX*(*W1*) w wariancie *W1*. Budujące go klasy wchodzą w skład mniejszych komponentów (*KA*(*Wx*) i *KB*(*Wx*), x=2,3) w pozostałych wariantach

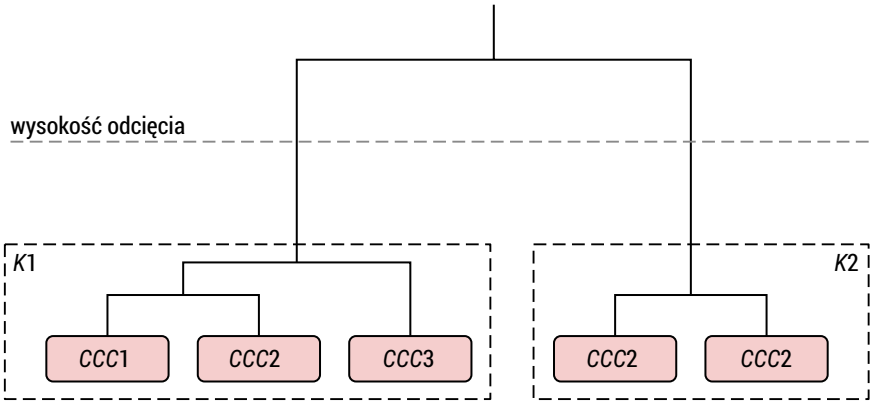


RYSUNEK 4.5. Zbyt małe komponenty *KX*(*W1*) i *KY*(*W1*) w wariancie *W1*. Budujące je klasy wchodzą w skład pojedynczego komponentu *KA*(*Wx*), x=2,3, w pozostałych wariantach

Łączenie komponentów między wariantami produktów na zasadzie: maksymalnie 1 komponent z każdego wariantu, nie pozwala na naprawienie błędów z etapu budowania niezależnych architektur wariantów produktów – podejmowane decyzje mogą być niespójne. Budowanie konkretnych komponentów dla każdego wariantu nie bierze pod uwagę ich późniejszego scalania – zamiast celu globalnego realizowany jest szereg celów lokalnych.

Zamiast tego można zaproponować bezpośrednią budowę komponentów poprzez hierarchiczne grupowanie klas klonów klasowych (CCC – *class clone class*). Takie klasy klonów klasowych powstawałyby jako współpracujące klony strukturalne na poziomie klas (rys. 4.3). Proces hierarchicznego klastrowania mógłby przebiegać

analogicznie jak w [32] i opierać się na maksymalizacji globalnej funkcji dopasowania oceniającej jakość podziału systemu (linii produkcyjnej oprogramowania) na komponenty (komponenty uogólnione). Wychodząc od pojedynczych CCC jako początkowych komponentów uogólnionych (liści w dendrogramie), w każdym kroku algorytmu łączone byłyby te liście lub węzły dendrogramu (rys. 4.6), dla których funkcja dopasowania osiągałaby maksymalną wartość. Wymagałoby to oczywiście sprawdzenia wszystkich potencjalnych złączeń. Wysokość odcięcia, decydująca o ostatecznym podziale na komponenty (powstałe w wyniku spłaszczenia gałęzi istniejących na wysokości odcięcia – rys. 4.6, jest ustalana w miejscu gdzie połączenie węzłów skutkuje gorszym komponentem wynikowym.



RYSUNEK 4.6. Formowanie uogólnionych komponentów (*K*1 i *K*2) z uogólnionych klas – klas klonów klasowych (CCC). Dendrogram – wynik hierarchicznego klastrowania – jest obcinany na wyliczonej wysokości odcięcia wyznaczając jednopoziomowe komponenty uogólnione

W cytowanej pracy globalna funkcja dopasowania liczy komponenty, których lokalna funkcja dopasowania przekracza wybrany próg. Lokalna funkcja dopasowania powinna osiągać wysokie wartości dla komponentów, które są autonomiczne, specyficzne (atomowe) i zdolne do komponowania z innymi (*composable*). Metryki używane do mierzenia tych charakterystyk są wyliczane na podstawie: liczby oferowanych przez komponent interfejsów, spoistości (LCC – *loose class cohesion*) i współzależności (*coupling*). Wyliczenie lokalnej funkcji dopasowania dla uogólnionych komponentów wymagałoby uogólnienia metryk, zaproponowanych oryginalnie dla indywidualnych komponentów. Uogólnienie to mogłoby przyjąć postać średniej, wartości maksymalnej lub minimalnej w każdym z wariantów.

Podejście takie, gdzie analiza klonów programowych wykonywana jest przed tworzeniem (tu – pojedynczej) architektury to może być określone mianem „**klony najpierw**" (CF – *clones first*). W wyniku bezpośredniej konstrukcji ujednoliconej architektury, bez przechodzenia przez architektury indywidualne, większa byłaby szansa na podjęcie optymalnej decyzji dotyczącej formowania uogólnionych komponentów, unikając problemów opisanych wyżej.

Podsumowanie

W pracy przedstawiona została analiza możliwości użycia analizy klonów w tworzeniu uogólnionej architektury wariantów oprogramowania na podstawie kodów źródłowych. Zaproponowane zostały (w podrozdziale 4.6) autorskie pomysły na zmodyfikowanie istniejących technik tworzenia takiej architektury poprzez uwzględnienie informacji o podobieństwach w kodzie pomiędzy wersjami programów. Architektura taka może być użyta przy konstruowaniu ujednoliconej linii produkcyjnej oprogramowania (SPL), ale też w innych zadaniach, jak remodularyzacja, wyodrębnianie serwisów (np. w celu przeniesienia oprogramowania do środowiska chmurowego).

Analiza klonów może być zastosowana do zmodyfikowania istniejących metod tworzenia uogólnionej architektury – wszędzie tam, gdzie zachodzi potrzeba odwołania się do odpowiadającego sobie kodu pomiędzy wariantami. Może też posłużyć do zaproponowania całkiem nowych mechanizmów, takich jak bezpośrednie formowanie uogólnionych komponentów z uogólnionych klas, zidentyfikowanych przy użyciu koncepcji współpracujących klonów strukturalnych – bez pośredniego etapu budowy indywidualnych architektur wariantów.

Propozycje zostały tu przedstawione w formie koncepcji i wymagają doprecyzowania i eksperymentalnej weryfikacji. Jednak ich opracowanie oparte zostało na istniejących i opublikowanych rozwiązaniach w zakresie analizy klonów i tworzenia uogólnionej literatury, co pozwala mieć nadzieję na pozytywne rezultaty. Obecnie trwają prace na zaimplementowaniem wybranych technik w celu oceny korzyści z użycia informacji o klonach w tym zadaniu (głównie w oparciu o modyfikację algorytmu zaprezentowanego w [27]).

Badania zostały zrealizowane w ramach pracy nr WZ/WI-IIT/3/2020, sfinansowanej ze środków Ministerstwa Nauki i Szkolnictwa Wyższego w Polsce.

Bibliografia

[1] Garlan D., Shaw M., *An Introduction to Software Architecture*. School of Computer Science, Carnegie Mellon University, 1994

[2] *IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*, IEEE Std, 2000, 1471–2000

[3] Kruchten P., *The Rational Unified Process: An Introduction*, Addison-Wesley Professional, 2004

[4] Kruchten P., *Architectural Blueprints – The "4+1" View Model of Software Architecture*, IEEE Software, 1995, 12 (6), pp. 42–5

[5] Fowler M., *Who Needs an Architect?* IEEE Software, 2003, 20(5), pp. 11–13

[6] Koschke R., *Architecture Reconstruction. Software Engineering*, International Summer Schools, ISSSE 2006–2008, Revised Tutorial Lectures, 2009, pp. 140–173

[7] Garcia J., Ivkovic I., Medvidovic N., *A comparative analysis of software architecture recovery techniques*, 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 486–496

[8] Maqbool O., Babri H.A., *Hierarchical Clustering for Software Architecture Recovery*. IEEE Transactions On Software Engineering, 2007, 33(11), pp. 759–780

[9] Praditwong K., *Solving software module clustering problem by evolutionary algorithms*, 8th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2011, pp. 154–159

[10] Sartipi K., Kontogiannis K., *A graph pattern matching approach to software architecture recovery*, IEEE International Conference on Software Maintenance, 2001, pp. 408–419

[11] Mitchell B.S., Mancoridis S., *On the Automatic Modularization of Software Systems Using the Bunch Tool*, IEEE Transactions On Software Engineering, 2006, 32(3), pp. 193–208

[12] Xiao C., Tzerpos V., *Software clustering based on dynamic dependencies*, Ninth European Conference on Software Maintenance and Reengineering, 2005, pp. 124–133

[13] Anquetil N., Lethbridge T., *Recovering Software Architecture from the Names of Source Files*, Journal of Software Maintenance Research and Practice, 1999, 11(3), pp. 201–221

[14] Clements P., Northrop L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002

[15] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[16] Zhang H., Jarzabek S., *XVCL: a mechanism for handling variants in software product lines*, Science of Computer Programming, 2004, 53(3), pp. 381–407

[17] Adaptive Reuse Technology, http://art-processor.org

[18] Weiss D.M., Lai C.T.R., *Software Product-line Engineering*: *A Family-based Software Development Process*. Addison-Wesley, 1999

[19] Krueger C., *Easing the Transition to Software Mass Customization*, [in:] van der Linden F. (eds) Software Product-Family Engineering. PFE 2001. Lecture Notes in Computer Science 2290. Springer, 2002

[20] Bayer J., Flege O., Knauber P., Laqua R., Muthig D., Schmid K., Widen T., DeBaud J.M., *PuLSE: methodology to develop software product lines*, Symposium on Software reusability, 1999, pp. 122–131

[21] Stoermer C., O'Brien L., *MAP – mining architectures for product line evaluations*, Working IEEE/IFIP Conference on Software Architecture, 2001, pp. 35–44

[22] Pinzger M., Gall H., Girard J.-F., Knodel J., Riva C., Pasman W., Broerse C., Wijnstra J.G., *Architecture recovery for product families*, Software product-family engineering. Springer, 2004, pp. 332–351

[23] Kang K.C., Kim M., Lee J., Kim B., *Feature-oriented re-engineering of legacy systems into product line assets–a case study*, International Conference on Software Product Lines (SPLC2005). Springer, 2005, pp. 45–56

[24] Frenzel P., Koschke R., Breu A.P.J., Angstmann K., *Extending the Reflexion Method for Consolidating Software Variants into Product Lines*, 14th Working Conference on Reverse Engineering (WCRE 2007), 2007, pp. 160–169

[25] Koschke R., Frenzel P., Breu A.P., Angstmann K., *Extending the reflexion method for consolidating software variants into product lines*, Software Quality Journal, 2009, 17 (4), pp. 331– 366

[26] Linsbauer L., Lopez-Herrejon R.E., Egyed A., *Variability extraction and modeling for product variants*, Software & Systems Modeling 16, 2017, pp. 1179–1199

[27] Shatnawi A., Seriai A.-D., Sahraoui H., *Recovering software product line architecture of a family of object-oriented product variants*, Journal of Systems and Software 131, 2017, pp. 325–346

[28] Lima C., Assunção W.K., Martinez J., Mendonça W., Machado I.C., Chavez C.F.G., *Product line architecture recovery with outlier filtering in software families: the Apo-Games case study*, Journal of the Brazilian Computer Society 25, article 7, 2019

[29] Jarzabek S., *Software Similarities and Clones: A Curse or Blessing?* International Conference on Enterprise Information Systems (ICEIS 2020), Keynote, 2020

[30] Hammad M., Basit H., Jarzabek S., Koschke R., *A Systematic Mapping Study of Clone Visualization*, Computer Science Review 37, 2020, pp. 1–55

[31] Basit H.A., Jarzabek S., *Data Mining Approach for Detecting Higher-level Clones in Software*, IEEE Transactions On Software Engineering, 2009, 35(4), pp. 497–514

[32] Kebir S., Seriai A.-D., Chardigny S., Chaoui A., *Quality-centric approach for software component identification from object-oriented code*, Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), 2012, pp. 181–190

[33] Faust D., Verhoef C., *Software product line migration and deployment*, Journal of Software Practice and Experiences 33(10), 2003, pp. 933–955

[34] Mancoridis S., Mitchell B., Chen Y.-F., Gansner E., *Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures*, IEEE International Conference on Software Maintenance (ICSM'99), 1999, pp. 50–59