Methods of detecting spatio-temporal patterns in software development processes

Autoreferat rozprawy doktorskiej

Łukasz Puławski

Maj 2022

1 Wprowadzenie

Eksploracja repozytoriów oprogramowania (ang. *mining software repositories* - MSR) to dziedzina eksploracji danych, w której bada się zbiory danych, powstające podczas rozwoju oprogramowania. Zasadniczo, głównym (choć nie jedynym) źródłem danych w tych badaniach jest system kontroli wersji ([1]), który zapisuje zmiany w kodzie źródłowym programu podczas jego rozwoju. Wśród obszarów badań MSR można wyróżnić dwa związane z tematem rozprawy: Wykrywanie *antywzorców projektowych*, czyli pewnych struktur w kodzie, które stanowią często występujące błędne rozwiązanie typowego zadania programistycznego ([2]) oraz badanie *ewolucji systemu* (ang. *software evolution*), czyli badanie zmian w konstrukcji oprogramowania obserwowanych w czasie ([3]). W pracy przedstawiona jest metoda wykrywania wzorców czasowo-przestrzennych w procesie wytwarzania oprogramowania, co stanowi swoiste połączenie modeli stosowanych w tych dwóch obszarach. Dzięki takiemu podejściu zyskujemy, między innymi, możliwość przewidywania, kiedy i gdzie w przyszłości mogą pojawić się w kodzie antywzorce.

2 Główne wyniki rozprawy

2.1 Reprezentacja struktury kodu programu

W modelu zaproponowanym w rozprawie struktura kodu reprezentowana jest przez skończony multigraf *encji* (ang. *software entity*), w którym wierzchołki

odpowiadają takim konstrukcjom języka programowania jak pakiety, klasy, metody, czy pola, a krawędzie – relacjom faktycznie występującym w kodzie źródłowym, takim jak dziedziczenie klas lub wywołanie metody. Dodatkowo węzły grafu opisane są wartościami *metryk oprogramowania* (ang. *software metrics*), takimi jak np. liczba linii kodu w danej encji lub jej złożoność cyklomatyczna (ang. *cyclomatic complexity*) ([4]). Podobne modele stosowane są w wielu badaniach MSR ([5], [6]), jednak specyficzne uproszczenia zastosowane w pracy, umożliwiają zastosowanie adaptacyjnego algorytmu budowania multigrafu przy sekwencyjnej analizie całej ewolucji systemu. Jest to związane z tym, że zwykle w procesie rozwoju oprogramowania, w danym momencie modyfikowany jest tylko bardzo niewielki fragment kodu źródłowego. Cechę tę określamy jako *lokalność procesu* (ang. *locality property*).

Zakładamy, że rozwój oprogramowania odbywa się liniowo, tzn. główna wersja systemu tworzona jest poprzez następujące po sobie *atomowe modyfikacje kodu źródłowego* (ang. *commits*), oznaczone unikalnym identyfikatorem zwanym *rewizją* (ang. *revision*). Ewolucja systemu reprezentowana jest zatem jako ciąg multigrafów indeksowany liniowo uporządkowanymi rewizjami.

Model reprezentacji opisany jest w podrozdziałach 6.1-6.2, zaś eksperymenty potwierdzające efektywność podejścia adaptacyjnego potwierdzone są eksperymentami opisanymi w dodatkach A.1 i A.2.

2.2 Wykrywanie antywzorców

Przedstawienie struktury kodu źródłowego w formie multigrafu pozwala na przetłumaczenie zagadnień inżynierii oprogramowania i MSR na język teorii grafów. W szczególności wyszukiwanie antywzorców można wyrazić jako wyszukiwanie odpowiednich podgrafów. W najbardziej ogólnym ujęciu jest to zadanie tak trudne jak problem izomorfizmu podgrafu, jednak wzbogacenie modelu o wiedzę dziedzinową z zakresu inżynierii oprogramowania pozwala znacząco ograniczyć przestrzeń poszukiwań dopasowania: W podrozdziale 6.3 przedstawiona jest teoria pozwalająca formalnie opisać złożoność problemu znalezienia podgrafów odpowiadających antywzorcom, oparta o ograniczenie liczby wierzchołków, które mogą być dopasowane do wzorcowego grafu. Wierzchołki takie są połączone specyficzną ścieżką ze specyficznym wierzchołkiem początkowym. Definicja ścieżek oraz wierzchołka początkowego jest właściwa dla każdego typu antywzorca i jest opisana w rozprawie w rozdziale 6.4. Metoda ta ma zastosowanie zarówno w ujęciu statycznym, czyli wyszukania antywzorców w pojedynczej wersji kodu, jak w ujęciu ewolucyjnym,

w którym analizujemy ciągi następujących po sobie rewizji kodu. Połączenie tej koncepcji z właściwością lokalności opisaną w sekcji 2.1 powyżej, pozwala zbudować adaptacyjną heurystykę wykrywania wystąpień antywzorców projektowych w toku ewolucji systemu. Jest ona opisana w rozprawie w podrozdziale 6.3.5, a eksperymenty potwierdzające jej efektywność przedstawione są w dodatkach A.4 i A.5.

W rozdziale 6.4 opisana jest oparta o powyższą heurystykę metoda wykrywania kilku popularnych typów antywzorców: „Swiss Army Knife", „YoYo", „Blob", „Brain Class", „Anemic Entitiy" i „Data Clumps" ([7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]). Zaproponowane podejście zostało przetestowane na siedmiu referencyjnych zbiorach danych, opisanych w podrozdziale 6.7, związanych z rozwojem otwartego oprogramowania tworzonego w języku Java: ArgoUML, Struts, Xerces, JHotDraw, Elasticsearch, Lucene-solr i Wildfly. Zbiorcze wyniki przedstawione są w Tabeli 1. Mimo uproszczeń modelu pozwalających stosować adaptacyjne heurystyki, zaproponowana metoda okazuje się być co najmniej tak dobra jak najlepsze znane obecnie rozwiązania ([20], [7], [21], [2], [22], [13]). Gorsze rezultaty uzyskano tylko w przypadku wykrywania antywzorca BLOB w zbiorach danych ArgoUML i Xerces, gdzie metoda oparta na bardziej skomplikowanym modelu opisana w [20] uzyskała rezultaty odpowiednio: 0,95/0,95 i 0,97/0,84 (precyzja/czułość)

anti-pattern type / test data	Swiss Army Knife	Blob	Data clumps	Base bean	Brain Class	YoYo	Anemic entity
Argo UML	0,78/1,0	0,90/0,76	0,99/0,98	0,71/0,88	N/D	1,0/1,0	1,0/1,0
Elasticsearch	0,78/0,99	0,83/0,9	0,99/0,96	0,71/0,88	0,87/0,84	0,98/1,0	1,0/1,0
JHotDraw	1,0 /0,91	1,0/1,0	0,28/0,0	N/D	0,0/0,0	N/D	N/D
Lucene	0,86/1,0	0,88/0,9	0,99/1,0	0,97/1,0	0,95/0,78	1,0/1,0	N/D
Struts	0,99/1,0	N/D	0,98/0,1	N/D	N/D	N/D	N/D
Wildfly	0,94/1,0	0,92/1,0	0,99/1,0	1,0/1,0	N/D	N/D	1,0/1,0
Xerces	0,8/0,89	0,91/0,69	0,99/0,84	N/D	N/D	0,98/1,0	N/D

Tabela 1: Jakość wykrywania wystąpień antywzorców projektowych w kodzie. W komórce są dwie liczby, które pokazują precyzję/czułość wykrywania antywzorca wskazanego w nagłówku kolumny w zbiorze danych wskazanym w nagłówku wiersza. N/D oznacza brak odpowiednich wystąpień.

2.3 Relacje czasowo-przestrzenne

Fundamentalnym pojęciem w teorii przedstawionej w rozprawie są *relacje czasowo-przestrzenne*, które są oparte na pojęciach relacji przestrzennej i

relacji czasowej opisanych poniżej.

2.3.1 Relacje przestrzenne

Dla dwóch różnych podgrafów możemy znaleźć najkrótszą ścieżkę, która łączy ich dowolne dwa wierzchołki. Jeśli ścieżka taka jest krótka, możemy powiedzieć, że podgrafy te są *bliskie* (ang. *close*), co odpowiada silniejszym zależnościom między odpowiadającymi encjami w kodzie programu. Jeśli ścieżka ta jest długa, albo nie istnieje, możemy powiedzieć, że są one *odległe* (ang. *remote*). W oparciu o tę intuicję w podrozdziale 6.5 wprowadzone jest pojęcie *relacji przestrzennej* (ang. *spatial relation*). Formalizm ten jest rozszerzony w taki sposób, że relacje przestrzenne można także określić dla podgrafów, które same nie występują razem w kodzie w jednej rewizji, pod warunkiem, że istnieje rewizja, w której występują tylko węzły tworzące te podgrafy. Dokładna definicja podana jest w podrozdziale 6.5.1 rozprawy.

2.3.2 Cykl życia i wystąpienia antywzorca

Konkretny podgraf odpowiadający jednemu antywzorcowi możemy zaobserwować w ewolucji systemu w więcej niż jednej rewizji, co odpowiada sytuacji, w której pewna struktura wprowadzona przez programistę do kodu może zostać tam na dłużej. Zbiór wszystkich rewizji, w których występuje dany podgraf nazywamy jego *czasem życia* (ang. *lifetime*). Czas życia można podzielić na maksymalne ciągłe przedziały rewizji, z których każdy nazywamy *wystąpieniem* antywzorca (ang. *occurrence*). Maksymalne – w tym sensie, że podgraf odpowiadający antywzorcowi nie występował w kodzie w pierwszej rewizji poprzedzającej lewy koniec przedziału, nie występował rewizji następującej po prawym końcu przedziału oraz występował we wszystkich rewizjach należących do przedziału. Formalne definicje powyższych pojęć znajdują się w podrozdziale 6.5.4 pracy.

2.3.3 Relacje czasowe

Patrząc na dwa wystąpienia dwóch różnych antywzorców, możemy określić ich wzajemną relację czasową (np. pierwsze wystąpienie nastąpiło bezpośrednio przed drugim). W rozprawie do wyrażania tego typu relacji użyta jest algebra Allena ([23]), która definiuje 13 różnych relacji między przedziałami w zbiorze liniowo uporządkowanym.

2.3.4 Relacje czasowo-przestrzenne

Jeśli dwa wystąpienia dwóch różnych antywzorców są w relacji Allena A, a antywzorce te są bliskie w sensie opisanym powyżej, to powiemy, że są one *A-bliskie*. Jeśli natomiast są odległe, to powiemy, że są *A-odległe*. Są to dwa typy dualnych *relacji przestrzenno-czasowych* (ang. *spatio-temporal relations*). Opis formalizmów związanych z relacjami czasowo- przestrzennymi można znaleźć w podrozdziałach 6.5.5-6.5.7 rozprawy.

2.4 Reguły przestrzenno-czasowe

Jeśli weźmiemy pod uwagę ustalone wystąpienie dowolnego podgrafu W, możemy określić wszystkie jego czasowo-przestrzenne relacje do innych wystąpień antywzorców w całej ewolucji analizowanego systemu. Każda taka relacja opisana jest przez trzy atrybuty: A–typ operatora Allena, $s \in \{\text{“}close\text{”}, \text{“}remote\text{”}\}$ – mówiący o typie relacji przestrzennej oraz $t \in \{\text{“}SwissArmyKnife\text{”}, \text{“}YoYo\text{”}, \text{“}Blob\text{”}, \text{“}BrainClass\text{”}, \text{“}AnemicEntitiy\text{”}, \text{“}DataClump\text{”}\}$ definiujący typ antywzorca, względem wystąpienia którego określamy relację przestrzenną. Dla każdej trójki (A, s, t) możemy określić liczbę relacji czasowo-przestrzennych W, tworząc w ten sposób wektor cech opisujący W. Jeśli powtórzymy tę konstrukcję dla wszystkich wystąpień wszystkich antywzorców, uzyskujemy tablicę decyzyjną, która koduje wszystkie zaobserwowane relacje czasowo-przestrzenne w całej ewolucji systemu. Szczegółowe definicje opisujące konstrukcję takiej tablicy oraz efektywny adaptacyjny algorytm jej budowania w trakcie rozwoju oprogramowania opisane są w rozdziale 6.6 rozprawy.

Wynikiem działania regułowego algorytmu uczenia maszynowego (takiego jak użyty w pracy C4.5 ([24])), zastosowanego do powyższej tablicy decyzyjnej, jest zbiór *reguł decyzyjnych*. Bezpośrednio opisują one wzorce występujące w tabeli, ale pośrednio, z uwagi na sposób jej konstrukcji, opisują one wzorce czasowo-przestrzenne występujące w procesie wytwarzania oprogramowania. Dlatego nazywamy je *regułami czasowo-przestrzennymi* (ang. *spatio-temporal rules*).

2.5 Zastosowania i wyniki eksperymentalne

Reguły czasowo-przestrzenne mają szereg praktycznych zastosowań, które są opisane w podrozdziale 6.7 i których wartość została empirycznie potwierdzo-

dane tren. \ dane test.	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	0,97/0,75	0,99/0,72	1,0/ 1,0	0,99/0,98	1,0/ 1,0	1,0/ 1,0	0,99/0,85
xerces	0,99/0,67	0,99/0,93	0,94/0,65	0,99/0,84	0,9/0,11	0,99/0,63	0,97/0,17
struts	1,0/ 1,0	0,99/ 1,0	0,96/0,74	0,98/0,25	0,0/ 0,0	0,98/0,13	1,0/ 1,0
elasticsearch	0,99/0,67	0,99/0,79	0,98/0,76	0,99/0,88	0,9/0,11	0,99/0,56	0,97/0,17
jhotdraw	0,96/0,35	0,97/0,24	0,57/0,04	0,99/0,99	0,5/0,04	0,99/ 1,0	0,99/0,96
lucene-solr	0,99/0,98	0,98/0,42	0,98/0,98	0,99/0,97	1,0/ 1,0	0,99/0,97	0,99/ 0,9
wildfly	1,0/ 1,0	0,99/ 1,0	1,0/ 1,0	1,0/ 1,0	1,0/ 1,0	0,99/ 1,0	1,0/ 1,0

Tabela 2: Jakość wykrywania obszarów wystąpień dowolnych antywzorców oparty o reguły czasowo-przestrzenne. Wiersze odpowiadają systemom, na którym reguły były uczone, a kolumny systemom, na którym były sprawdzane. W przypadku diagonali, zbiór treningowy był stworzony w oparciu o ewolucję składającą się z 70% początkowych rewizji, a zbiór testowy z pozostałych 30%. W każdej komórce znajduje się para liczb: precyzja/czułość.

na w eksperymentach opisanych w dodatkach A.5-A.7 do rozprawy. Poniżej przedstawione są dwa przykładowe zastosowania.

2.6 Wykrywanie obszarów w kodzie podatnych na występowanie antywzorców

Reguły czasowo-przestrzenne można interpretować w ewolucji dowolnego systemu, dla którego potrafimy wyznaczyć relacje czasowo-przestrzenne. Oznacza to, że można ich użyć do określenia, które obszary kodu (które podgrafy) i w jakim czasie (w jakim zakresie rewizji) mogą zawierać wystąpienia antywzorców. W Dodatku A.6 rozprawy przedstawiono wyniki eksperymentalne, w których reguły czasowo-przestrzenne wykorzystane są w ten właśnie sposób. Przykładowe wyniki przedstawione są w Tabeli 2.

2.6.1 Wpływ reguł czasowo-przestrzennych na jakość wykrywania antywzorców

Skoro reguły czasowo-przestrzenne mogą być użyte do wskazania obszarów, w których mogą występować antywzorce, to można ich użyć do poprawienia jakości wykrywania antywzorców, łącząc w ten sposób ze sobą koncepcje opisane w sekcjach 2.6 i 2.4. W podejściu tym podgraf uznamy za faktyczne wystąpienie antywzorca, jeśli pasuje on do teoriografowego opisu wspomnianego w sekcji 2.2 powyżej oraz jeśli wystąpienie tego grafu znajduje się w obszarze wskazanym przez reguły czasowo-przestrzenne. W takim podejściu

Zbiór	Antywz.	Bez reguł prec./czuł.	Z regułami prec./czuł	Zmiana F1
elastic	BaseBean	0,71 / 0,88	1,0 / 0,54	0,89
argouml	BLOB	0,90 / 0,76	1,0 / 0,76	1,05
elastic	BLOB	0,83 / 0,9	1,0 / 0,88	1,08
Xerces	BLOB	0,91 / 0,69	1,0 / 0,69	1,04
elastic	BrainClass	0,87 / 0,84	1,0 / 0,81	1,05
argouml	SAK	0,78 / 1,0	1,0 / 0,94	1,11
elastic	SAK	0,78 / 0,99	1,0 / 0,99	1,14
Lucene	SAK	0,86 / 1,0	1,0 / 0,92	1,04
Xerces	SAK	0,80 / 0,89	1,0 / 0,65	0,94
Xerces	YoYo	0,98 / 1,0	1,0 / 0,99	1,01
			Średnio	1,04

Tabela 3: Wpływ reguł przestrzenno-czasowych na jakość wykrywania antywzorców projektowych. W tabeli przestawiono tylko te przypadki, w których dodanie tych reguł zmienia jakość wykrywania. Pierwsza kolumna pokazuje jakość wykrywania bez zastosowania reguł-czasowo przestrzennych, a druga – z nimi. SAK = Swiss Army Knife.

może zmniejszyć się liczba błędów pierwszego rodzaju (ang. *false positive*), ale może zwiększyć się liczba błędów drugiego rodzaju (ang. *false negative*). Eksperymentalne wyniki, przedstawione w dodatku A.5, pokazują, że takie podejście podniosło jakość wykrywania średnio o 4% w sensie miary F1. Ich zestawienie przedstawione jest w Tabeli 3.

2.7 Struktura rozprawy

Rozprawa podzielona jest na siedem rozdziałów i dwa dodatki.

Rozdział pierwszy i drugi stanowią wprowadzenie do problematyki pracy, zarysowując motywację oraz podstawowe pojęcia z zakresu procesu rozwoju oprogramowania i eksploracji repozytoriów oprogramowania.

Rozdział trzeci stawia dwie hipotezy (potwierdzone) i określa cztery cele badawcze (osiągnięte):

- H1: Istnienie wzorców czasowo-przestrzennych związanych z powstawaniem antywzorców w procesie rozwoju oprogramowania

- H2: Możliwość znacznego poprawienia efektywności algorytmu wykrywania reguł czasowo-przestrzennych przez wzbogacenie go o wiedzę dziedzinową z obszaru inżynierii oprogramowania

- G1: Stworzenie formalnego modelu opisu antywzorców projektowych i ich ewolucji

- G2: Stworzenie modelu aproksymacyjnego do opisu wzorców przestrzennych

- G3: Stworzenie modelu opisu wzorców czasowo-przestrzennych w procesie rozwoju oprogramowania

- G4: Stworzenie efektywnego adaptacyjnego algorytmu do wykrywania wzorców czasowo-przestrzennych w trakcie rozwoju oprogramowania

Rozdział czwarty objaśnia pojęcia związane z eksploracją danych i uczeniem maszynowym, które związane są z metodami opisanymi w pracy.

Rozdział piąty przedstawia literaturę przedmiotu. Jest on podzielony na dwa podrozdziały, z których pierwszy odnosi się do badań ogólnych związanych z eksploracją danych czasowych (ang. temporal data) i przestrzennych, a drugi – do badań w dziedzinie MSR.

Rozdział szósty stanowi zasadniczą cześć rozprawy i szczegółowo przedstawia teorię wykrywania wzorców czasowo-przestrzennych, nakreśloną powyżej w tym autoreferacie oraz opisuje jej możliwe zastosowania.

Rozdział siódmy stanowi podsumowanie, zawiera wnioski oraz kierunki dalszych badań.

Dodatek A zawiera opis wszystkich eksperymentów wykonanych w ramach pracy.

Dodatek B zawiera instrukcję odtworzenia wyników eksperymentalnych.

3 Podsumowanie i dalsze kierunki badań

W pracy przedstawiono nowatorską teorię relacji i reguł czasowo-przestrzennych w rozwijanym kodzie źródłowym programu oraz metodę ich efektywnego adaptacyjnego wyliczania w trakcie ewolucji systemu. Eksperymenty oparte o dane pozyskane z ewolucji siedmiu systemów napisanych w języku Java i

rozwijanych jako otwarte oprogramowanie, pokazały jej praktyczną przydatność, na przykład do przewidywania gdzie i kiedy w rozwijanymi kodzie mogą pojawić się antywzorce.

3.1 Dalsze kierunki badań

W ocenie autora przedstawiona teoria, poparta empiryczną weryfikacją, daje kilka możliwych kierunków dalszych badań nakreślonych poniżej.

3.1.1 Inne źródła danych

Pierwszym obszarem dalszych badań jest przetestowanie proponowanej metody na systemach rozwijanych w inny sposób. Istnieją badania wskazujące, że oprogramowanie rozwijane z zamkniętym kodem źródłowym w komercyjnych zespołach może ewoluować inaczej niż systemy typu open-source, użyte w eksperymentach w rozprawie ([25], [26], [27], [28], [29]). Można zatem przypuszczać, że zastosowanie zaproponowanej metody może dawać inne reguły czasowo-przestrzenne.

Przy niewielkich modyfikacjach zaproponowany model można zastosować do systemów rozwijanych w obiektowych językach programowania innych nich Java. Dostosowanie go do np. języków funkcyjnych wymagałoby jednak istotniejszej modyfikacji definicji multigrafu i zaproponowania innego modelu zapewniającego adaptacyjne budowanie modelu ewolucji.

3.1.2 Rozbudowany model ewolucji i antywzorca

Uproszczony grafowy model ewolucji i antywzorca pozwala na zastosowanie efektywnych adaptacyjnych algorytmów, co ma istotne znaczenie przy praktycznych zastosowaniach metody. Odbywa się to jednak kosztem nieco gorszej jakości wykrywania wzorców przestrzennych. Wydaje się, że bez utraty adaptacyjności model można rozbudować np. o leksykalne cechy kodu (ang. lexical properties), które są wykorzystywane w niektórych powiązanych badaniach MSR ([30], [31], [32], [33]).

3.1.3 Zastąpienie antywzorców innymi podgrafami

W zaproponowanej teorii antywzorce utożsamialiśmy z pewnymi podgrafami i czasowo-przestrzenne relacje między wystąpieniami takich podgrafów stanowiły podstawę dalszych konstrukcji. Jeśli dla pewnej rodziny podgrafów

potrafimy określić jednolity typ (tak jak typ antywzorca) oraz pojęcie cyklu życia, to możemy w tej metodzie zastąpić pojęcie antywzorca takim podgrafem. Można zatem badać reguły czasowo przestrzenne oparte o podgrafy odpowiadające innym konstrukcjom programistycznym. Szczególny przypadek tego rozwinięcia może stanowić metoda wykrywania błędów opisana poniżej.

3.1.3.1 Wykrywanie błędów

Wykrywanie błędów (ang. *defect prediction*) to jedno z zadań MSR, którego celem jest wskazanie obszarów w kodzie zawierających błędy (ang. *bug/defect*) ([1], [34] [35]). Innym zagadnieniem jest znajdowanie przyczyny błędów (ang. *defect origin*), czyli znalezienie takiej rewizji, która faktycznie wprowadziła błąd do kodu ([36], [37], [38] [39]). Zagadnienie dualne polegające na określeniu rewizji, która błąd naprawia jest zwykle trywialne, ponieważ informacja ta jest wprost zapisana w narzędziach używanych przez programistów. Korzystając zatem z dostępnych metod wykrywania przyczyny błędu możemy określić przedział rewizji, w którym błąd występował. Przedział taki odpowiada pojęciu *wystąpienia* wzorca w teorii przedstawionej w rozprawie. Dlatego jednym z możliwych kierunków dalszych badań jest wykorzystanie reguł czasowo-przestrzenny do wykrywania błędów.

3.1.4 Inny model czasu i relacji czasowych

W pracy zakładamy, że rewizje możemy uporządkować liniowo. Jest to poprawne założenie ([40]), o ile ograniczamy się wyłącznie do rozwoju głównej wersji programu w jednej *gałęzi* (ang. *branch*). Taki model nie jest w stanie odzwierciedlić bardziej złożonego procesu, w którym system rozwijany jest także w innych równoległych gałęziach, a zmiany między gałęziami są łączone (ang. *merge*). Aby odtworzyć taką rzeczywistość w modelu, liniowy wymiar czasu należałoby zastąpić np. odpowiednią logiką temporalną.

Drugi ważny aspekt temporalny związany jest z ograniczeniami teorii Allena, w której potrafimy tylko wyrazić jak umieszczone są przedziały względem siebie, ale nie potrafimy powiedzieć jak są „odległe". W szczególności, dwa przedziały oddzielone jedną rewizją są traktowane identycznie jak dwa przedziały oddzielone tysiącem rewizji. Jednym z możliwych kierunków dalszych badań jest wprowadzenie innego formalizmu, np. opartego o metodę *okna przesuwnego* (ang. *sliding window*).

Literatura

[1] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," 2007.

[2] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha, "A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems," *Software: Practice and Experience*, vol. 49, no. 1, pp. 3–39, Jan. 2019.

[3] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *2009 IEEE International Conference on Software Maintenance*, Sep. 2009, pp. 51–60.

[4] T. J. McCabe, "A complexity measure," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76. San Francisco, California, United States: IEEE Computer Society Press, 1976, pp. 407+.

[5] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Software Is a Directed Multigraph," in *Software Architecture*, I. Crnkovic, V. Gruhn, and M. Book, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6903, pp. 360–369.

[6] R. Dąbrowski, G. Timoszuk, and K. Stencel, "One Graph to Rule Them All Software Measurement and Management," *Fundamenta Informaticae*, vol. 128, no. 1-2, pp. 47–63, 2013.

[7] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[8] N. Moha, Y.-g. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo: IEEE, 2006, pp. 297–300.

[9] R. Wieman, *Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations*. LAP LAMBERT Academic Publishing, Nov. 2011.

[10] D. H. Taenzer, M. Ganti, and S. Podar, "Problems in Object-Oriented Software Reuse," in *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989*, S. Cook, Ed. Cambridge University Press, 1989, pp. 25–38.

[11] A. Stoianov and I. Sora, "Detecting patterns and antipatterns in software using Prolog rules," in *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*. Timisoara: IEEE, May 2010, pp. 253–258.

[12] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st ed. Addison-Wesley Professional, Nov. 2002.

[13] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, "Using History Information to Improve Design Flaws Detection," in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, ser. CSMR '04. Washington, DC, USA: IEEE Computer Society, 2004.

[14] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 350–359, 2004.

[15] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," *Empirical Software Engineering and Measurement, International Symposium on*, vol. 0, pp. 390–400, 2009.

[16] H. Kagdi, M. L. Collard, and J. I. Maletic, "Towards a taxonomy of approaches for mining of source code repositories," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005.

[17] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Feb. 2017, pp. 8–13.

[18] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[19] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Improving the Precision of Fowler's Definitions of Bad Smells," in *2008 32nd Annual IEEE Software Engineering Workshop*. Kassandra, Greece: IEEE, Oct. 2008, pp. 161–166.

[20] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. Essen, Germany: ACM Press, 2012, pp. 278+.

[21] F. Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Information Sciences*, vol. 181, no. 7, pp. 1306–1324, Apr. 2011.

[22] R. P. F. Trindade, M. A. da Silva Bigonha, and K. A. M. Ferreira, "Oracles of Bad Smells: A Systematic Literature Review," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*. Natal Brazil: ACM, Oct. 2020, pp. 62–71.

[23] J. F. Allen, "Maintaining Knowledge About Temporal Intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[24] S. L. Salzberg, "C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993," *Mach Learn*, vol. 16, no. 3, pp. 235–240, Sep. 1994.

[25] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding Change-Proneness in OO Software Through Visualization," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003.

[26] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. W. Weber, "Impact of software engineering research on the practice of software configuration management," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 4, pp. 383–430, Oct. 2005.

[27] B. Fluri, E. Giger, and H. C. Gall, "Discovering Patterns of Change Types," L'Aquila, Italy, Sep. 2008, pp. 463–466.

[28] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Software Maintenance, 2000. Proceedings. International Conference On*. IEEE, 2000, pp. 131–142.

[29] P. L. Li, M. Shaw, J. Herbsleb, B. Ray, and P. Santhanam, "Empirical Evaluation of Defect Projection Models for Widely-deployed Production Software Systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 263–272, Oct. 2004.

[30] M. Kim and D. Notkin, "Program element matching for multi-version program analyses," in *Proceedings of the 2006 International Workshop on Mining Software Repositories - MSR '06*. Shanghai, China: ACM Press, 2006, p. 58.

[31] S. Kim, K. Pan, and E. J. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships," in *Proceedings of the 12th Working Conference on Reverse Engineering*, ser. WCRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.

[32] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, vol. 30. Lisbon, Portugal: ACM, Sep. 2005, pp. 296–305.

[33] N. Moha, Y. G. Gueheneuc, A. F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Form. Asp. Comput.*, vol. 22, pp. 345–361, May 2010.

[34] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, no. 4, pp. 331–344, Dec. 2020.

[35] S. Hosseini, B. Turhan, and D. Gunarathna, "A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction," *IIEEE Trans. Software Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019.

[36] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*. St. Louis, Missouri: ACM, 2005, pp. 1–5.

[37] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: A model to identify how bugs are introduced in software components," *Empirical Software Engineering*, pp. 1–47, 2020.

[38] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[39] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the SZZ algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2018, pp. 380–390.

[40] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 322–333.