Politechnika Wrocławska

Wrocław, 27 września 2022

dr hab. inż. Lech Madeyski, prof. uczelni
Katedra Informatyki Stosowanej
Wydział Informatyki i Zarządzania
Politechnika Wrocławska
✉: lech.madeyski@pwr.edu.pl
🏠: https://madeyski.e-informatyka.pl

Recenzja rozprawy doktorskiej mgr. Łukasza Puławskiego pt. "Methods of detecting spatio-temporal patterns in software development processes" (Metody wykrywania wzorców strukturalno-czasowych w procesie wytwarzania oprogramowania)

1 Wstęp

Niniejsza recenzja została opracowana w odpowiedzi na list Pana prof. dr. hab. Andrzeja Tarleckiego, Przewodniczącego Rady Naukowej Dyscyplin Matematyka i Informatyka, która wyznaczyła mnie na recenzenta rozprawy doktorskiej w przewodzie doktorskim mgr. Łukasza Puławskiego, którego promotorem jest Pan prof. dr hab. Dominik Ślęzak.

2 Problematyka badawcza, jej znaczenie i aktualność

Tematyka rozprawy mgr. Łukasza Puławskiego wiąże się z bardzo aktywnie rozwijającym się nurtem inżynierii oprogramowania, jakim jest eksploracja repozytoriów oprogramowania (ang. *mining software repositories*), czego wyrazem jest m.in. bardzo prestiżowa konferencja Mining

Software Repositories (MSR), klasyfikowana wg rankingu CORE na poziomie A, organizowana razem z kluczową konferencją inżynierii oprogramowania, czyli International Conference on Software Engineering (ICSE). Praca Doktoranta skupia się na styku dwóch obszarów nurtu MSR. Pierwszy to wykrywanie antywzorców projektowych, czyli pewnych struktur w kodzie, które stanowią często występujące błędne rozwiązania typowego zadania programistycznego. Drugi to badanie ewolucji oprogramowania (ang. *software evolution*), czyli badanie zmian w konstrukcji oprogramowania obserwowanych w czasie.

Rezultatem pracy jest metoda wykrywania wzorców przestrzenno-czasowych w procesie wytwarzania oprogramowania, co stanowi interesujące połączenie obu tych obszarów. Zaletą zaproponowanego podejścia jest w szczególności możliwość lepszego zrozumienia jak oprogramowanie jest rozwijane, jak również przewidywania, gdzie i kiedy w przyszłości mogą pojawić się antywzorce w kodzie. Podsumowując tematykę prac Doktoranta uważam za bardzo ciekawie i trafnie dobraną, ważną, a także aktualną, pomimo, iż sam doktorat był chyba realizowany dość długo. Miałem okazję natrafić na publikacje Doktoranta w bazie DBLP z lat 2011, 2012 i 2017, a w tekście pracy wspomniana jest również publikacja z 2009 roku [1–5]. Ciekawe, że w treści rozprawy nie pojawia się referencja do najnowszej (widocznej w bazie DBLP) publikacji Doktoranta z 2017 roku pt. „Temporal Relations of Rough Anti-patterns in Software Development" [5], która wydaje się być ściśle związana z tematyką rozprawy. Przyznam też, że prace Doktoranta już dość dawno zwróciły moją uwagę nie tylko (niestety) rzadko spotykaną w tym obszarze badawczym polską afiliacją autora, ale właśnie interesującym umiejscowieniem tematyki badań na styku dwóch obszarów. Przypuszczam, że gdyby prace Doktoranta ukazały się w czasopismach lub materiałach konferencji z obszaru inżynierii oprogramowania lub jeszcze lepiej nurtu MSR (np. na konferencjach MSR, czy PROMISE), to byłyby łatwiejsze do zauważenia przez innych badaczy z tego kręgu, a ich oddźwięk międzynarodowy mógłby być większy niż dwa autocytowania w bazie Scopus, czy dwa zidentyfikowane przeze mnie cytowania w Google Scholar.

3 Szczegółowe omówienie i krytyczna analiza treści rozprawy

Rozprawa napisana jest w języku angielskim, składa się ze słownika pojęć, siedmiu rozdziałów, bibliografii obejmującej aż 371 pozycji literaturowych oraz dwóch załączników i jest bardzo obszerna, liczy aż 246 stron.

W rozdziale 1, Doktorant wprowadza w tematykę rozprawy, w szczególności w terminy wzorców strukturalnych (w niektórych miejscach określanych również mianem przestrzennych lub statycznych) w strukturze oprogramowania i wzorców czasowych (temporalnych) w procesie wytwarzania oprogramowania. Wskazuje również, że praca skupia się na detekcji obu typów wzorców i wprowadza termin wzorca strukturalno-czasowego opisującego zmiany statycznych wzorców w czasie, uprzedzając czytelnika, że podstawą będzie ścisły model matematyczny opisujący takie wzorce. Istotną częścią tego rozdziału jest wskazanie tego co, zdaniem Doktoranta, stanowi kluczowy wkład Doktoranta, w szczególności wprowadzenie koncepcji wzorców projektowych przybliżonych (ang. *rough software patterns*), aby radzić sobie z problemem wyrażania i detekcji konceptów z obszaru struktury oprogramowania, które w praktyce mogą się między sobą trochę różnić, a także automatyzacji procesów identyfikacji i odkrywania wzorców strukturalnych i czasowych z wykorzystaniem prostych algorytmów uczenia maszynowego, bazując na koncepcji reguły strukturalno-czasowej.

W rozdziale 2, Doktorant wprowadza podstawowe koncepcje opisując m.in. ewolucję złożonych

struktur w czasie zauważając przy tym trafnie, że oprogramowanie stanowi dobry przykład złożonego obiektu, którego rozwijanie może być traktowane jako ewolucja. Omawia również organizację procesu wytwarzania oprogramowania wybierając postrzeganie tego procesu z perspektywy technicznej, czyli poprzez dane (np. kod źródłowy), które są tworzone i utrwalane (ang. *persist*) w trakcie realizacji procesu bez odnoszenia się do metodyki, czy sposobu w który te dane zostały stworzone lub celu ich stworzenia. Doktorant wskazuje zatem jakie dane są przechowywane w różnych systemach (np. systemach kontroli wersji (ang. *version control systems*), czy śledzenia zagadnień (ang. *issue trackers*)) i jakie zdarzenia są zapamiętywane. Informacje zawarte w systemie śledzenia zagadnień dotyczą akcji podejmowanych w trakcie tworzenia oprogramowania z punktu widzenia funkcjonalności, którą ma dostarczać oprogramowanie oraz ewentualnych zgłaszanych problemów (ang. *bugs*) i ich rozwiązywania. Natomiast logi z systemu kontroli wersji mogą być postrzegane jako historia aktywności w ramach procesu wytwarzania oprogramowania widziana właśnie z perspektywy technicznej.

Warto zauważyć, że Doktorant poczynił jawnie pewne założenia upraszczające. Oprogramowanie w systemie kontroli wersji jest rozwijane w wielu gałęziach (ang. *branches*), jednak Doktorant poczynił założenie, że analizuje tylko proces wytwarzania oprogramowania w głównej gałęzi (ang. *main development branch*). Przyjął też, że wszystkie rewizje kodu się kompilują, bo jeśli się nie kompilują, to są traktowane tak, jakby nie istniały. Ponadto przyjął, że zmiany w głównej gałęzi nie mogą być później usuwane lub modyfikowane np. poprzez łączenie wielu zmian w jedną (ang. *squash*). Powyższe założenia wydają się być dopuszczalne, ale brakuje bardziej szczegółowego omówienia zagrożeń wynikających z przyjęcia powyższych założeń, jak bardzo są one poważne i jakie ewentualnie środki zostały podjęte, jeśli zostały, by je zminimalizować.

Kolejnym, daleko idącym założeniem upraszczającym jest przyjęcie w badaniach bazujących zarówno na danych z systemu kontroli wersji jak i systemu śledzenia zagadnień, że da się powiązać wszystkie zmiany w systemie kontroli wersji z zagadnieniami w systemie śledzenia zagadnień. W ogólnym przypadku założenie to może być postrzegane jako nieco bardziej poważne, ale problem jest nietrywialny, a próba jego rozwiązania sama w sobie może być tematem odrębnej rozprawy doktorskiej. Niemniej, znowu warto byłoby omówić bardziej szczegółowo zagrożenia wynikające z przyjęcia tego założenia.

Doktorant trafnie zauważa, że metryki kodu źródłowego można podzielić na takie, które dotyczą pojedynczej encji (ang. *entity-centric*) i takie, które dotyczą relacji lub współpracy pomiędzy encjami (ang. *relation-centric*), co ma oczywiście swoje implikacje.

Doktorant przywołuje termin wzorców projektowych, odwołując się do bodaj najbardziej wpływowej pozycji literaturowej, czyli książki tzw. bandy czworga (ang. *Gang of Four (GoF)* [6]). W tym momencie jednak pojawia się pytanie, czy znowu nie mamy do czynienia z pewnym uproszczeniem. Czy faktycznie wszystkie wzorce projektowe można skatalogować w ramach klasyfikacji GoF?

Doktorant przywołuje również termin antywzorców powołując się na książkę Browna i in. [7] oraz powiązanych z nimi przykrych zapachów kodu (ang. *code smells*) postrzegając je jako coś w rodzaju antywzorców, ale na wczesnym etapie rozwoju. Wszystkie one są postrzegane jako przykłady statycznych, czy strukturalnych wzorców. Kolejnym założeniem poczynionym przez Doktoranta było ograniczenie się jedynie do programów napisanych w języku Java, co sprawia, że każda klasa dziedziczy (bezpośrednio lub pośrednio) z klasy `java.lang.Object` i nie jest możliwe dziedziczenie z więcej niż jednej klasy, czyli pojedyncze drzewo dziedziczenia reprezentuje kod źródłowy. W związku z tym jeśli podgrafy są od siebie oddalone, to odpowiadające im

struktury kodu są luźno powiązane, co prowadzi doktoranta do ważnych koncepcji relacji oddalenia (ang. *remoteness*) oraz bliskości (ang. *closeness*) fragmentów kodu źródłowego, a relacje są nazwane relacjami strukturalnymi, czy przestrzennymi (ang. *spatial relations*).

W rozdziale 2, doktorant wprowadza również wymiar temporalny procesu wytwarzania oprogramowania związany z czasem, który mija podczas zespołowego rozwijania systemu i jest on ściśle powiązany ze zmianami zachodzącymi w wersjonowanym repozytorium kodu. Pojawia się zatem ważna koncepcja wzorca czasowego (ang. *temporal pattern*), który dotyczy zmian kodu, które można zaobserwować na przestrzeni odpowiednio długiego czasu. Ewolucja kodu jest postrzegana jako sekwencja stanów kodu źródłowego odzwierciedlona w postaci zmian kodu w wersjonowanym repozytorium, a formalnie jako ciąg multigrafów (indeksowanych kolejnymi liniowo uporządkowanymi rewizjami w wersjonowanym repozytorium kodu) reprezentujących różne rodzaje zależności pomiędzy encjami kodu (referencje do innej klasy, rozszerzenie innej klasy, wywołanie innej metody), a każdy multigraf reprezentuje pojedynczą wersję oprogramowania.

Powyższe autorskie pomysły Doktoranta wydają się być naturalnymi wyborami. Podsumowując ten rozdział, Doktorant przedstawił wstępnie bazowe koncepcje dotyczące domeny badawczej doktoratu w ciekawy, klarowny i oryginalny sposób, łącząc zjawiska zachodzące w obszarze inżynierii oprogramowania z pomysłami na ich zamodelowanie w bardziej formalny (niż często ma to miejsce w inżynierii oprogramowania), czyli matematyczny sposób, a przyjęte założenia upraszczające można uznać za dopuszczalne. Jednocześnie warto zauważyć, że zagrożenia płynące z przyjęcia tych założeń, ocena jak bardzo zagrożenia te są poważne i jakie ewentualnie środki zostały podjęte, by je zminimalizować, mogłyby być szerzej omówione. W pracach naukowych, to zwykle ma miejsce w rozdziale omawiającym zagrożenia dla wiarygodności uzyskanych rezultatów (ang. *Threats to validity*).

W rozdziale 3 Doktorant przedstawia praktyczną motywację i kontekst swoich badań. Pierwszą motywacją była potrzeba stworzenia (pół-)automatycznej metody detekcji wczesnych symptomów złych struktur w kodzie źródłowym. Autor wychodzi od stwierdzenia, że refaktoryzacja kodu może być opisana jako przekształcenie złego kodu w lepszy kod i stwierdza, że programiści zwykle rozpoczynają aktywności związane z refaktoryzacją późno, gdy złe struktury w kodzie są już powszechne i bardzo złożone, a co za tym idzie trudniejsze do wyplenienia. Odnosząc się do tego stwierdzenia Doktoranta, wypada zauważyć, że wcale tak nie jest jeśli programiści używają np. bardzo znanej praktyki TDD (ang. *test-driven development*), gdzie refaktoryzacja jest integralną aktywnością w procesie tworzenia oprogramowania od samego początku i jest powtarzana w tzw. cyklu TDD (test, kod, refaktoryzacja) wiele razy dziennie. Natomiast zgodzę się, że stosowanie praktyki TDD wymaga od programistów pewnej biegłości, a co za tym idzie nie każdy programista jest w stanie jej używać i nie w każdym przypadku. Warto nadmienić, że refaktoryzacja jest również dobrze wspierana przez powszechnie wykorzystywane przez programistów zintegrowane narzędzia/środowiska IDE (ang. *Integrated Development Environment*), które są używane do rozwijania oprogramowania. Dzięki temu refaktoryzacje są łatwiejsze do przeprowadzania. Narzędzia IntelliJ firmy JetBrians są rutynowo używane przez programistów Javy, a niedawne badania wśród użytkowników opublikowane w materiałach prestiżowej konferencji ESEC/FSE w 2021 roku [8] pokazały, że w ostatnim miesiącu tylko 2.5% użytkowników nie przeprowadzało refaktoryzacji kodu (20.4% przeprowadzało raz lub dwa razy w miesiącu, 36.9% przeprowadzało refaktoryzację w każdym z tygodni w ostatnim miesiącu, a 40.6% przeprowadzało ją prawie każdego dnia w ostatnim miesiącu). Refaktoryzacja wydaje się zatem być dość często stosowana, nawet jeśli niekoniecznie jest częścią wspomnianego wcześniej podejścia TDD. Niemniej nie przekreśla to sensowności motywacji autora by stworzyć taką metodę, która

(jak pisze Doktorant) będzie stanowiła dobrą podstawę do zbudowania narzędzia wczesnego ostrzegania o tym, że stosowane rozwiązania projektowe czy architektoniczne dryfują w złym kierunku, a zatem wczesnego podjęcia działań naprawczych.

Drugą motywacją Doktoranta (powiązaną jednak z pierwszą) była potrzeba stworzenia (pół-)automatycznej metody pomagającej zredukować koszt długu technicznego. W szczególności, korzystając z możliwości detekcji wzorców temporalnych odnoszących się do antywzorców, czy przykrych zapachów kodu, metoda ta mogłaby pomóc radzić sobie z problemem stopniowej degradacji projektu systemu (ang. *design rot*, *source code decay*). Doktorant uznaje, że niektóre z zaproponowanych metod mogłyby być użyte do budowy modelu predykcji, czy pojawiający się dług techniczny może być zaciągany, czy może powinien być już spłacany. Odnosząc się do tego, wydaje mi się, że decyzja o tym, czy dług techniczny będzie zaciągany, czy też należy się teraz skupić na jego spłacaniu, nie zależy li tylko od tego co znajdziemy w obszarze kodu (a na nim bazują metody proponowane przez Doktoranta). Poza tym dług techniczny w kodzie, który nie jest modyfikowany bywa często akceptowany i nie spłacany, bo np. nie stwarza dużego ryzyka występując we fragmencie systemu, który nie jest modyfikowany.

Trzecią motywacją, bardzo ciekawą, była potrzeba zaproponowania metody opisu wzorców projektowych w sposób formalny, a zarazem przybliżony, co wychodzi naprzeciw specyfice wzorców projektowych, których ujęcie w sztywne ramy modelu matematycznego mogłoby nieść ryzyko utraty ważnej abstrakcji pojęciowej. Tę, moim zdaniem nie tylko ciekawą, użyteczną, ale również oryginalną ideę Doktorant nazwał przybliżonym wzorcem projektowym (ang. *rough software pattern*) i przedstawił w Rozdziale 6.3.2.

Czwartą motywacją była potrzeba wsparcia, a w szczególności poprawienia efektów powszechnie stosowanych w firmach przeglądów kodu, a proponowane badania mogą prowadzić do stworzenia narzędzia, które byłoby w stanie wskazywać recenzentom te fragmenty kodu gdzie problematyczne struktury mają dużą szansę się pojawić, a w rezultacie wesprzeć osoby przeglądające kod.

W rozdziale 3 Doktorant formułuje również hipotezy: H1 (*There are statistically significant temporal patterns in software development process that can be used to predict the appearance of anti-patterns*) i H2 (*Incorporation of expert knowledge can produce more efficient data mining algorithms in the domain of software development process*). Ich sformułowanie może jednak budzić pewne wątpliwości. Przykładowo w hipotezie H1 nie bardzo wiadomo jakie zmienne łączy jaka relacja i co to właściwie znaczy "statystycznie istotne wzorce czasowe w procesie wytwarzania oprogramowania". Rekomendacje odnośnie skutecznego formułowania hipotez (np. rekomendacje Elseviera) sugerują też m.in. by hipotezy były formułowane w sposób maksymalnie prosty i zwięzły, bez niejasności, czy założeń odnoszących się do wiedzy czytelników.

Hipoteza H2 z kolei jest sformułowana dość ogólnie, bo dotyczy w zasadzie wszystkich możliwych algorytmów eksploracji danych w domenie procesów związanych z wytwarzaniem oprogramowania. Pokrycie tak szerokiego obszaru przez jeden doktorat stanowiłoby wyzwanie.

W rozdziale 3 Doktorant formułuje też cele (ang. *goals*) dotyczące wprowadzenia: formalnego modelu antywzorców projektowych i przykrych zapachów kodu i ich ewolucji (G1), przybliżonego modelu służącego reprezentacji statycznych wzorców projektowych (G2), modelu reprezentującego temporalne relacje pomiędzy ewoluującymi strukturami kodu źródłowego (G3) oraz algorytmu powalającego na efektywną lokalizację wzorców strukturalnych i temporalnych (G4).

Podsumowując, mimo pewnych uwag dotyczących uzasadnienia niektórych motywacji badań, a w szczególności uwag odnoszących się do sposobu sformułowania hipotez, rozdział 3 (choć mam

wrażenie, że nie aż tak dobrze napisany jak poprzedni) precyzuje motywacje oraz to czego czytelnik może oczekiwać od rozprawy.

W rozdziale 4, Doktorant omówił krótko metody uczenia maszynowego bez nadzoru i z nadzorem, specyfikę zadań predykcyjnych (klasyfikacji) i opisowych (np. odkrywania sekwencji (ang. *sequence discovery*), czy eksploracji reguł asocjacyjnych (ang. *association rule mining*)), pojęcia klasyfikatora regułowego, zbiorów przybliżonych jako podejścia, które pozwoli poradzić sobie z ryzykiem utraty abstrakcji pojęciowych obecnych w inżynierii oprogramowania (wzorców, czy przykrych zapachów kodu), algebry Allen'a jako formalizmu pozwalającego wyrazić relacje temporalne oraz miary oceny modeli (w szczególności w przypadku klasyfikacji binarnej i wieloklasowej).

W rozdziale tym można znaleźć drobne potknięcia językowe, czy notacyjne, opisane w rozdziale 4 recenzji, ale poza tym rozdział spełnia swoje zadanie i Doktorant umiejętnie prowadzi czytelnika wykazując ogólną wiedzę teoretyczną kandydata w zakresie poruszanych zagadnień.

Rozdział 5 stanowi przegląd literatury związanej z dysertacją i jest podzielony na dwie części. Część pierwsza dotyczy klasycznych i abstrahujących od domeny metod eksploracji danych, które są użyteczne z punktu widzenia tematyki rozprawy i jej lepszego zrozumienia. Część druga skupia się na badaniach w obszarze eksploracji repozytoriów oprogramowania i procesów wytwarzania oprogramowania.

Moje główne zastrzeżenie dotyczy tego, że Doktorant nie próbował, choćby częściowo, bazować na metodyce systematycznego przeglądu literatury (ang. *Systematic Review* lub *Systematic Literature Review*). Jest ona formą badania drugiego rzędu (ang. *secondary study*), czyli takiego, które poddaje przeglądowi wszystkie podstawowe badania/publikacje naukowe (ang. *primary studies*) odnoszące się do konkretnych pytań badawczych w celu zebrania/integracji/syntezy wyników badań. Systematyczny przegląd wyróżnia to, że używa dobrze zdefiniowanej metodyki do identyfikacji, analizy i interpretacji wszystkich dostępnych źródeł informacji w szczegółowo udokumentowany, a zatem ułatwiający ewentualne uaktualnienie, sposób. Systematyczne przeglądy (mające swoje źródła w medycynie i nurcie nazwanym *Evidence-Based Medicine*) przeniesione zostały na grunt inżynierii oprogramowania przez Kitchenham i in. [9, 10]. Szybko stały się one fundamentem jednego z najważniejszych nurtów inżynierii oprogramowania ostatnich lat, czyli *Evidence-Based Software Engineering*. Systematyczne przeglądy są publikowane przez najbardziej prestiżowe czasopisma inżynierii oprogramowania.

W związku z tym, że przedstawiony przegląd ma jednak raczej charakter przeglądu ad hoc niż systematycznego, nie ma pewności na ile reprezentuje on kompletny stan sztuki. Wydaje się, że pewne ściśle związane z tematyką rozprawy pozycje literaturowe Doktorantowi umknęły np. sporo opublikowanych systematycznych przeglądów literatury np. „Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study" [11], „Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review" [12], „Machine learning techniques for code smell detection: A systematic literature review and meta-analysis" [13], „A systematic review on the code smell effect" [14], oraz „A review of code smell mining techniques" [15]. Inną cenną brakującą pozycją literaturową, opublikowaną w 2021 roku w najbardziej prestiżowym czasopiśmie inżynierii oprogramowania, czyli *IEEE Transactions on Software Engineering*, jest praca [16] całościowo opisująca stan sztuki w obszarze badań na przykrymi zapachami kodu od roku 1990. Nie natknąłem się też na książki Smith'a pt. *Elemental Design Patterns* [17] oraz Sarcar'a pt. *Java Design Patterns: A Hands-On Experience with Real-World Examples* [18]. Z drugiej strony liczba referencji użytych w pracy wskazuje, że przegląd literatury wcale nie jest płytki, a Doktorant włożył w niego sporo wysiłku. Zatem uwagę, by korzystać (na tyle na ile to możliwe) z systematycznego podejścia do

przeglądu literatury, proponuję potraktować jako sugestię na przyszłość.

Rozdział 6 stanowi kluczową część rozprawy. Zaprezentowany został formalny model (ang. *software snapshot*), w postaci multigrafu, umożliwiający reprezentację statycznej struktury oprogramowania w danym punkcie czasu, zawierający informacje o wszystkich pakietach, klasach i metodach oraz ich własnościach i wzajemnych relacjach, będący swego rodzaju reprezentacją stanu złożonego obiektu. Zaprezentowany został również sposób przedstawienia ewolucji oprogramowania (ang. *software evolution*), czyli zmian w jego strukturze, które zachodzą w trakcie realizacji procesu wytwarzania oprogramowania w postaci serii modeli reprezentujących stany systemu zmieniające się w czasie (etykietowne czasem).

Doktorant omówił proces parsowania danych zawartych w systemach takich jak wersjonowane repozytorium kodu, a w szczególności stojący za nim formalizm, aby móc skonstruować statyczną reprezentację struktury oprogramowania w danym punkcie czasu, czyli dla konkretnej rewizji kodu, opisując ten proces w odniesieniu do bibliotek (stanowiących tzw. kontekst), plików źródłowych, pakietów, klas, pól i metod, istniejących zależności, a także samej ewaluacji metryk kodu z wykorzystaniem ogólnodostępnego narzędzia Checkstyle, które będąc narzędziem służącym do statycznej analizy kodu dostarcza pewnych (niekoniecznie najbardziej rozbudowanych) możliwości obliczania metryk kodu. Pobieranie danych dotyczących ewolucji oprogramowania wykorzystuje fakt, że kolejne jego rewizje są z dużym prawdopodobieństwem do siebie bardzo podobne i w związku z tym nie ma konieczności parsowania całego kodu w kolejnej jego rewizji/wersji (ang. *revision*), lecz wystarczy sparsować pliki, na które wpłynęła ostatnia zmiana (ang. *commit*). Oznacza to, że budowa reprezentacji struktury oprogramowania w kolejnej wersji nie wiąże się z wysokim kosztem (jest dość efektywna), jeśli mamy już zbudowaną reprezentację struktury oprogramowania w poprzedniej wersji. Ta adaptacyjna metoda konstrukcji statycznej reprezentacji struktury oprogramowania w kolejnych wersjach stanowi fundament na którym, w koncepcyjnie podobny i efektywny sposób, zrealizowano adaptacyjną metodę wyszukiwania wzorców oprogramowania.

W rozdziale 6.4.1 możemy zaobserwować pewne decyzje Doktoranta, które można uznać za niewystarczająco uzasadnione lub dyskusyjne np. jednym z warunków uznania metody za efektywnie trywialną jest warunek by miała ona maksymalnie pięć linii kodu. W tym miejscu pojawia się naturalnie pytanie dlaczego akurat pięć. Zresztą kolejnym warunkiem wskazanym przez Doktoranta jest, by metoda miała cyklomatyczną złożoność nie większą niż trzy. Również tutaj pojawia się pytanie dlaczego akurat trzy. Podobna sytuacja ma miejsce w przypadku uznania (przez Doktoranta) konstruktora za złożony (np. warunek 2, że liczba linii kodu konstruktora przekracza 150% liczby pól klasy lub warunek 3, że cyklomatyczna złożoność konstruktura przekracza 150% liczby pól klasy).

Bazując na decyzjach, których uzasadnienie mogłoby być klarowniejsze, Doktorant definiuje klasę jako anemiczną (ang. *anemic entity/ data class*) gdy, w szczególności, klasa taka ma więcej niż osiem pól i więcej niż osiem metod (dlaczego właśnie osiem?).

W rozdziale 6.4.2 Doktorant w podobny sposób definiuje wzorzec Swiss Army Knife (SAK), a mianowicie zakłada, że klasa spełnia warunki tego wzorca, gdy ma więcej niż sześć metod, liczba linii kodu klasy przekracza 150, suma cyklomatycznych złożoności metod klasy przekracza 30, suma wartości NPath w klasie przekracza 120, liczba efektywnie nietrywialnych (wg Doktoranta) metod zawartych w klasie przemnożona przez średnią wartość metryki NPath metod przekracza 160, liczba metod wywoływanych przez obce wywołania, czyli z innej hierarchii klas, przekracza dwa, a liczba obcych wywołań przekracza siedem. Brak jednak klarownego uzasadnienia dlaczego

akurat takie, a nie inne, wartości zostały przyjęte.

Ten sam problem pojawia się w rozdziale 6.4.3, gdzie Doktorant definiuje wzorzec God Class (także znany jako Blob), wskazując, że mamy do czynienia z tym wzorcem, gdy klasa wywołuje więcej niż siedem efektywnie trywialnych (wg Doktoranta) metod innych klas, gdy proporcja liczby efektywnie trywialnych metod innych klas wywoływanych przez daną klasę do liczby innych metod innych klas wywoływanych przez daną klasę przekracza 0.6, gdy suma cyklomatycznych złożoności nietrywialnych metod przekracza 55 oraz gdy wartość metryki *tight class cohesion* nie przekracza 0.3.

Tego samego typu problem pojawia się w przypadku wzorców Brian Class, Base Bean, YoYo, czy Data Clumps. Konkludując, brakuje zarówno klarownych uzasadnień podejmowanych decyzji jak i dyskusji zagrożeń z tych decyzji wynikających.

W rozdziale 6.4.8, Doktorant przedstawia rezultaty detekcji wzorców na przykładowych projektach korzystając z dwóch miar jakości klasyfikacji: precyzji (ang. *precision*) i czułości (ang. *recall*). Niestety miary te nie są zbyt szczęśliwie dobrane. Każda z tych miar bazuje tylko na dwóch (z czterech) ćwiartek tablicy/macierzy pomyłek (ang. *confusion matrix*). Na podstawie tych niesatysfakcjonujących metryk trudno w pełni ocenić jakość klasyfikacji. Nawet gdyby została użyta miara F, bazująca na kombinacji obu powyższych miar, nie byłaby ona satysfakcjonująca, gdyż odzwierciedlałaby jedynie trzy z czterech ćwiartek tablicy pomyłek. Sugeruję skorzystanie z nieco bardziej wyrafinowanych miar oceny modeli klasyfikacji (np. miary *Matthews Correlation Coefficient* (MCC) bazującej na wszystkich czterech ćwiartkach tablicy pomyłek i rekomendowanej w najbardziej prestiżowym czasopiśmie z inżynierii oprogramowania *IEEE Transactions on Software Engineering* przez Shepperda i in. w bardzo zbliżonej domenie aplikacyjnej [19]).

Kolejnym problemem, o którym jednak jawnie wspomina Doktorant w rozdziale 6.4.8, jest fakt, że są metody stanowiące stan sztuki w obszarze detekcji wzorców, które uzyskują lepsze rezultaty. To oczywiście nie jest specjalnie dobra wiadomość, ale ponownie sugeruję skorzystanie z nieco bardziej wyrafinowanych miar oceny modeli klasyfikacji (np. wspomnianej miary MCC) i porównanie rezultatów uzyskanych przez Doktoranta z metodami stanowiącymi stan sztuki (przynajmniej w tych pracach w których jest dostęp do niezbędnych szczegółów np. w postaci odpowiednich miar, czy kompletnych tabel pomyłek). Być może wtedy rezultaty uzyskane przez Doktoranta prezentowałyby się lepiej na tle metod stanowiących stan sztuki.

W rozdziale 6.5, Doktorant definiuje formalne konstrukcje pozwalające specyfikować relacje przestrzenno-czasowe pomiędzy instancjami wzorców w procesie ewolucji oprogramowania bazując na wspomnianych na początku koncepcjach oddalenia i bliskości wystąpień wzorców. W szczególności pokazuje jak relacje przestrzenne, czasowe i przestrzenno-czasowe mogą być efektywnie, w sposób adaptacyjny, wyliczane.

W rozdziale 6.6, Doktorant definiuje idee reguł przestrzenno-czasowych i, co jest warte docenienia, skupia się na efektywnej, adaptacyjnej eksploracji reguł przestrzenno-czasowych w procesie ewolucji oprogramowania.

W rozdziale 6.7 Doktorant przedstawił empiryczną ewaluację zaproponowanego przez siebie formalnego ramowego rozwiązania (ang. *formal framework*), umożliwiającego eksplorację reguł przestrzenno-czasowych w celu predykcji wystąpień antywzorców w przyszłości na bazie reguł przestrzenno-czasowych uzyskanych na podstawie ewolucji oprogramowania do rewizji r, która jest traktowana jako aktualna rewizja w danym momencie. Doktorant bardzo krótko omówił również inne możliwości zastosowań przedstawionego rozwiązania np. do predykcji defektów

oprogramowania.

Do rozdziału 6 można mieć kilka zastrzeżeń:

- Zaproponowane ramowe rozwiązanie zostało empirycznie sprawdzone na danych z realnych projektów. Jednak dane z tych projektów użyte do analizy są bardzo stare (np. w projekcie ArgoUML analizowano ewolucję w latach 1998-2011, w projekcie Xerces2j w latach 1999-2008, a w projekcie Struts1 w latach 2000-2008). W związku z tym można mieć wątpliwości, czy odzwierciedlają one zarówno to jak aktualnie wytwarzane jest oprogramowanie, jak i ewolucję samego języka programowania na przestrzeni kilkudziesięciu lat, co (jak wskazano np. w pracy Grodzickiej i in. [20]) może mieć wpływ na detekcję brzydkich zapachów kodu.
- Zastosowane miary oceny modeli (precyzja i czułość w Tabelach 6.3, A13-A22) nie bazują na wszystkich (lecz tylko dwóch z czterech) ćwiartkach tablicy pomyłek. Podobny problem dotyczyłby również miary F bazującej na trzech z czterech ćwiartkach tablicy pomyłek. Trudno zgodzić się zatem ze stwierdzeniem (w rozdziale 4.1.4), że "Both accuracy and F-measure are generic measures that take all factors into account and show how well the classifier behaves in both accepting and rejecting instances.". W przypadku miary dokładności (ang. *accuracy*) pojawia się z kolei poważny problem w przypadku niezbalansowania danych. Jeśli np. 95% komponentów nie zawiera defektu czy antywzorca, a tylko 5% zawiera, to naiwna strategia predykcji wszystkich komponentów jako wolnych od defektów/antywzorców pozwoliłaby na osiągnięcie wyniku na poziomie 95%, a z pewnością nie byłby to użyteczny system predykcyjny. Z tego jednak Doktorant zdaje sobie sprawę pisząc "In general, however, F1-score is less biased by uneven distribution of decision classes.".
- Praca omawia szczegółowo formalizmy stojące za proponowanym ramowym rozwiązaniem, niemal zupełnie pomijając sposób pozyskania danych. Krótka wzmianka o tym pojawia się w załączniku. Jednak biorąc pod uwagę publikowane ostatnio prace, wyniki wydają się mocno zależeć od sposobu pozyskania danych, więc w tym kontekście ocena przyszłej użyteczności wyników może być trudna.

Rozdział 7 zawiera konkluzje, odniesienie do hipotez i celów badawczych postawionych w rozdziale 3 oraz kierunki dalszych badań.

Sformułowanie hipotezy H1 (*There are statistically significant temporal patterns in software development process that can be used to predict the appearance of anti-patterns*) budzi pewne wątpliwości. Np. nie jest jasne co znaczy "statystycznie istotne wzorce czasowe w procesie wytwarzania oprogramowania" i czy (oraz ewentualnie jak) Doktorant chce przetestować statystyczną istotność. Dodam, że testy statystyczne się nie pojawiają.

Trudno też w pełni zgodzić się ze stwierdzeniem, że eksperymenty opisane w załączniku A.6 udowadniają, że możemy efektywnie przewidywać gdzie i kiedy instancje wzorców projektowych będą znalezione (*The experiments described in Appendix A.6 prove that we can efficiently predict where and when certain instances of design anti-patterns will be found*). Wyniki eksperymentów przeprowadzonych przez Autora mogą ewentualnie uprawdopodobnić inną tezę, która mogłaby zastąpić H1, że z wykorzystaniem proponowanego podejścia można "efektywnie" przewidywać gdzie i kiedy instancje wzorców projektowych będą znalezione. Oczywiście przy ważnych założeniach upraszczających o których Autor wspomina.

Sformułowanie hipotezy H2 (*Incorporation of expert knowledge can produce more efficient data mining algorithms in the domain of software development process*) wydaje się być bardzo ogólne

i dotyczyć wszystkich możliwych algorytmów eksploracji danych w domenie procesów związanych z wytwarzaniem oprogramowania, mimo iż Autor skupia się tylko na pewnym wycinku tej domeny.

Natomiast cele, które postawił sobie Autor, czyli stworzenie formalnego modelu antywzorców projektowych / przykrych zapachów kodu i ich ewolucji (G1), przybliżonego modelu służącego reprezentacji statycznych wzorców projektowych (G2), modelu reprezentującego temporalne wzorce (G3) oraz algorytmu powalającego na efektywną lokalizację wzorców strukturalnych i temporalnych (G4), generalnie można uznać za osiągnięte.

Zostały również wskazane ciekawe kierunki dalszych badań. Pozostaje mieć nadzieję, że przynajmniej niektóre z nich zostaną podjęte przez Autora, bo póki co w niewielkim stopniu udało się zachęcić innych, by budowali na rozwiązaniach zaprezentowanych przez Doktoranta.

4 Jakość redakcji rozprawy, w tym poprawności językowej

Tytuł rozprawy poprawnie odzwierciedla jej zawartość. Wykaz terminów używanych w pracy (umieszczony na początku rozprawy) zawiera linki do stron na których terminy te się pojawiają, co ułatwia lekturę rozprawy. Poprawność językową rozprawy można ocenić wysoko. Jakość redakcji rozprawy jest odrobinę niższa. Szczególnie końcówka pracy zawiera więcej drobnych potknięć.

Przykładowe potknięcia:

- “in figures 2.1.” – jest tylko jeden rysunek 2.1.
- $X = \{(x \in U : d(x) = d_i\}$ (s.57) – brak nawiasu zamykającego.
- “It doesn’t however take into consideration” – w formalnym tekście rozprawy doktorskiej unikałbym form ściągniętych (ang. *contractions*).
- “2) 2.” (s.99)
- “In this section we...” (s.156), “In Section 6.6.3 we...” (s.170)
- “Let us consider the following quantitative and quantitative spatio-temporal rules” (s.158)
- Pod rysunkiem B.1 znalazł się spory, nieskomentowany fragment (bardzo małą czcionką) rozpoczynający się od “`Sys. Vari, setEnv.bat, unzip - see Section B.3. SerHistExp = class SerializeHistoryExperiment...`”
- “Figure fig:Dependencies of experiment outlines” (s.246) – “fig:Dependencies” prawdopodobnie należałoby otoczyć komendą `ref`.

5 Oryginalny wkład Autora i główne uwagi polemiczne

Na oryginalny dorobek Doktoranta przedstawiony w rozprawie składają się:

1. Analiza aktualnego stanu wiedzy na podstawie przeglądu literatury.

2. Zrealizowanie postawionych celów pracy, a w szczególności stworzenie formalnego modelu antywzorców projektowych i ich ewolucji (G1), przybliżonego modelu służącego reprezentacji statycznych wzorców projektowych (G2), modelu reprezentującego temporalne wzorce (G3) oraz algorytmu powalającego na efektywną lokalizację wzorców strukturalnych i temporalnych (G4).
3. Implementacja zaproponowanego rozwiązania ramowego.
4. Empiryczna ewaluacja zaproponowanego rozwiązania (w szczególności umożliwiającego predykcję gdzie i kiedy wystąpienia pewnych antywzorców projektowych mogą się pojawić w trakcie ewolucji realnych systemów oprogramowania) i porównanie z wynikami uzyskanymi przez rozwiązania znane z literatury.

Powyższe elementy oryginalnego dorobku Doktoranta mają różną wagę oraz swoje mocne i słabe strony.

Przegląd literatury (punkt 1) ma bardziej charakter przeglądu ad hoc niż systematycznego. W związku z tym w bogatej literaturze przedmiotu związanej z tematyką rozprawy można wskazać pewne pozycje literaturowe, które Doktorantowi umknęły. Pomimo tego liczba referencji użytych w pracy wyraźnie wskazuje, że przegląd literatury wcale nie jest płytki.

Kluczowy, oryginalny i bardzo znaczący wkład Autora wiąże się z zaproponowaniem szeregu ważnych formalizmów i realizacji sformułowanych jawnie celów pracy w postaci nowych modeli i algorytmów (punkt 2). Podejście przedstawione przez Doktoranta, wynikające z pomysłu przeniesienia zagadnień występujących w świecie inżynierii oprogramowania na język teorii grafów, uważam za bardzo cenne. Przyjemną cechą zaproponowanego rozwiązania jest również pewna jego elastyczność. Może być ono zastosowane również w innym (bardzo ważnym z praktycznego punktu widzenia i intensywnie eksplorowanym badawczo) obszarze predykcji defektów oprogramowania.

Na tej formalnej bazie powstała implementacja (punkt 3), a następnie została przeprowadzona empiryczna ewaluacja (punkt 4). Do obu tych elementów można mieć pewne zastrzeżenia.

Kod zaproponowanego rozwiązania faktycznie jest dostępny (co jest niewątpliwą zaletą), ale to tysiące linii kodu w archiwum zip (bez historii), praktycznie bez dokumentacji, niedostępnych w publicznym wersjonowanym repozytorium kodu (którego znaczenie w procesie wytwarzania oprogramowania jest wiele razy podkreślane w tekście rozprawy). To wszystko sprawia, że dość trudno będzie innym badaczom rozbudować zaproponowane rozwiązanie.

Uzyskane rezultaty empirycznej ewaluacji zaproponowanego rozwiązania nie do końca wskazują na jego wyższość nad innymi. Trudno oprzeć się wrażeniu, że to co proponuje Doktorant jest zbudowane na solidnych, formalnych postawach (stanowiących główną wartość i treść pracy), a szczegóły związane z jego implementacją i sama empiryczna ewaluacja zostały potraktowane z nieco mniejszą atencją. Co za tym idzie potencjał zaproponowanego rozwiązania być może nie został w pełni wykorzystany.

Empiryczna ewaluacja zaproponowanego rozwiązania została przeprowadzona na przykładowych projektach. Jednak dane z tych projektów użyte do analizy są bardzo stare. W związku z tym można mieć wątpliwości, czy odzwierciedlają one zarówno to jak aktualnie wytwarzane jest oprogramowanie, jak i ewolucję samego języka programowania na przestrzeni kilkudziesięciu lat, co może mieć wpływ na działanie zaproponowanych rozwiązań na aktualnie rozwijanym kodzie. Dyskusja tego typu zagrożeń dla wiarygodności wyników badań byłaby cenna.

Zastosowane miary oceny modeli (precyzja i czułość) nie są zbyt szczęśliwie dobrane. Nie bazują one bowiem na wszystkich ćwiartkach tablicy pomyłek (podobnie zresztą jak miara F). Na podstawie tych niesatysfakcjonujących metryk trudno w pełni ocenić jakość klasyfikacji.

Idąc dalej, wydaje się, że zagrożenia płynące z przyjęcia szeregu założeń upraszczających (szczegóły w rozdziale 3 recenzji) powinny być szerzej omówione (jak to się zwykle czyni) w rozdziale dotyczącym zagrożeń dla wiarygodności uzyskanych rezultatów (ang. *threats to validity*). W szczególności ważne jest jak bardzo zagrożenia wynikające z przyjętych założeń są poważne.

W pracy jest załącznik (ang. *appendix*) w którym pojawiają się dane, co do których mamy chyba tylko informację że “są "otagowane" przez eksperta”, natomiast nie bardzo wiemy w jaki sposób zostały one pozyskane — czy przez przegląd zupełny kodu, przez przeglądanie próbek kodu, podpowiedzi tzw. doradców (ang. *advisorów*) (z czym mamy dość często do czynienia w literaturze przedmiotu)? Czy zaangażowany był ekspert z doświadczeniem w tego rodzaju systemach i jak dużym oraz czy pracował z kodem któregoś z nich? O danych właściwie wiemy raczej niewiele, podczas gdy ostatnio publikowane systematyczne przeglądy literatury wskazują, że wyniki bardzo mocno zależą od sposobu pozyskania danych, więc w tym kontekście ciężko ocenić przyszłą użyteczność prezentowanych wyników.

6 Podsumowanie

Podsumowując stwierdzam, że:

1. Rozprawa stanowi oryginalne rozwiązanie problemu naukowego w zakresie eksploracji repozytoriów oprogramowania, na co składają się osiągnięcia Doktoranta wymienione w rozdziale 5 niniejszej recenzji. Osiągnięcia w zakresie wykrywania wzorców czasowo-przestrzennych w procesie wytwarzania oprogramowania łączą dwa nurty badań: wykrywania antywzorców projektowych, czyli struktur w kodzie, które stanowią często występujące błędne rozwiązanie typowego zadania programistycznego oraz badania ewolucji systemów, czyli zmian w konstrukcji oprogramowania obserwowanych w czasie. Dzięki temu możliwa jest predykcja kiedy i gdzie w przyszłości mogą pojawić się w kodzie antywzorce, czy nawet defekty oprogramowania, co ma duże znaczenie praktyczne.
2. Osiągnięcia zostały częściowo opublikowane w pracach, których jedynym autorem jest Doktorant, w szczególności w rozdziale ze znanej serii *Transactions on Rough Sets* oraz materiałach czterech konferencji. To oznacza pozytywną weryfikację osiągnięć Doktoranta przez specjalistów.
3. Rozprawa dowodzi, że Doktorant posiada ogólną wiedzę teoretyczną w dyscyplinie informatyka.
4. Rozprawa oraz publikacje naukowe Doktoranta, których jest jedynym autorem, dowodzą, że wykazał się On umiejętnością samodzielnego prowadzenia pracy naukowej.

Na podstawie powyższych punktów podsumowania niniejszej recenzji stwierdzam, że recenzowana praca doktorska spełnia ustawowe wymagania stawiane rozprawom doktorskim. W związku z tym wnioskuję o przyjęcie rozprawy doktorskiej i dopuszczenie Pana magistra inżyniera Łukasza Poławskiego do publicznej obrony.

Bibliografia

[1] Łukasz Puławski. „Discovering Implicit Dependencies Between Source Code Artifacts by Mining Software Development Processes". W: *Proceedings of XVIIIth Concurrency, Specification and Programming (CS&P) Workshop*. 2009.

[2] Łukasz Puławski. „Software Defect Prediction Based on Source Code Metrics Time Series". W: *Transactions on Rough Sets XIII*. Red. James F. Peters, Andrzej Skowron, Chien-Chung Chan, Jerzy W. Grzymala-Busse i Wojciech P. Ziarko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 104–120.

[3] Łukasz Puławski. „An automatic approach for detecting early indicators of design anti-patterns". W: red. Saeko Matsuura Maria Virvou. T. 240. Frontiers in Artificial Intelligence and Applications. 2012, s. 161–170.

[4] Lukasz Pulawski. „Automatic Forecasting of Design Anti-patterns in Software Source Code". W: *Proceedings of the 21th International Workshop on Concurrency, Specification and Programming, Berlin, Germany, September 26-28, 2012*. Red. Louchka Popova-Zeugmann. T. 928. CEUR Workshop Proceedings. CEUR-WS.org, 2012, s. 312–323.

[5] Łukasz Puławski. „Temporal Relations of Rough Anti-patterns in Software Development". W: *Rough Sets*. Red. Lech Polkowski, Yiyu Yao, Piotr Artiemjew, Davide Ciucci, Dun Liu, Dominik Ślęzak i Beata Zielosko. Cham: Springer International Publishing, 2017, s. 447–464.

[6] Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[7] William J. Brown, Raphael C. Malveau, Hays W. McCormick III i Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[8] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin i Mohamed Wiem Mkaouer. „One Thousand and One Stories: A Large-Scale Survey of Software Refactoring". W: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, s. 1303–1313. ISBN: 9781450385626. DOI: 10.1145/3468264.3473924. URL: https://doi.org/10.1145/3468264.3473924.

[9] Barbara A. Kitchenham, Tore Dybå i Magne Jørgensen. „Evidence-Based Software Engineering". W: *ICSE'04: International Conference on Software Engineering*. 2004, s. 273–281.

[10] Barbara Kitchenham, David Budgen i Pearl Brereton. *Evidence-Based Software Engineering and Systematic Reviews*. CRC Press, 2016.

[11] Frederico Caram, Bruno Rafael de Oliveira Rodrigues, Amadeu Campanelli i Fernando Silva Parreiras. „Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study". W: *International Journal of Software Engineering and Knowledge Engineering* 29 (lut. 2019), s. 285–316. DOI: 10.1142/S021819401950013X.

[12] Ahmed Al-Shaaby, Hamoud Aljamaan i Mohammad Alshayeb. „Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review". W: *Arabian Journal for Science and Engineering* 45 (sty. 2020), 2341–2369. DOI: 10.1007/s13369-019-04311-w.

[13] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi i Qing Wang. „Machine learning techniques for code smell detection: A systematic literature review and meta-analysis". W: *Information and Software Technology* 108 (2019), s. 115 –138. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.12.009.

[14] José Amancio M. Santos, João B. Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos do Nascimento, Mydiã Falcão Freitas i Manoel Gomes de Mendonça. „A systematic review on the code smell effect". W: *Journal of Systems and Software* 144 (2018), s. 450 –477. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.07.035.

[15] Ghulam Rasool i Zeeshan Arshad. „A review of code smell mining techniques". W: *Journal of Software: Evolution and Process* 27.11 (2015), s. 867–895. DOI: 10.1002/smr.1737.

[16] Elder Vicente de Paulo Sobrinho, Andrea De Lucia i Marcelo de Almeida Maia. „A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where". W: *IEEE Transactions on Software Engineering* 47.1 (2021), s. 17–66. DOI: 10.1109/TSE.2018.2880977.

[17] Jason Mc C. Smith. *Elemental Design Patterns*. Addison-Wesley Professional, 2012.

[18] Vaskaran Sarcar. *Java Design Patterns: A Hands-On Experience with Real-World Examples*. 2nd. Apress, 2018.

[19] Martin Shepperd, David Bowes i Tracy Hall. „Researcher Bias: The Use of Machine Learning in Software Defect Prediction". W: *IEEE Transactions in Software Engineering* 40.6 (2014), s. 603–616. DOI: 10.1109/TSE.2014.2322358.

[20] Hanna Grodzicka, Arkadiusz Ziobrowski, Zofia Łakomiak, Michał Kawa i Lech Madeyski. „Code Smell Prediction Employing Machine Learning Meets Emerging Java Language Constructs". W: *Data-Centric Business and Applications: Towards Software Development (Volume 4)*. T. 40 of book series Lecture Notes on Data Engineering and Communications Technologies. Cham: Springer International Publishing, 2020, s. 137–167. ISBN: 978-3-030-34706-2. DOI: 10.1007/978-3-030-34706-2_8. URL: https://doi.org/10.1007/978-3-030-34706-2_8.

.....
Lech Madeyski