Zeszyty Naukowe Wydziału
Elektroniki i Informatyki
Politechniki Koszalińskiej

ISSN:1897-7421
Nr 15, 2019
Str: 23-34

Daniel Czyczyn-Egird
Wydział Elektroniki i Informatyki
Politechnika Koszalińska
daniel.czyczyn-egird@dce-systems.pl

Prognozowanie defektów w oprogramowaniu z wykorzystaniem modeli predykcyjnych opartych na danych historycznych

Słowa kluczowe: eksploracja danych w oprogramowaniu, modele predykcji defektów, metryki oprogramowania

1. Wprowadzenie

W dobie nieustannego rozwoju komputerów oraz oprogramowania, istnieje duże zapotrzebowanie na nowe i coraz bardziej zaawansowane systemy informatyczne, od których wymaga się oprócz określonych funkcjonalności, także jak najwyższej niezawodności. Niestety zdecydowana większość oprogramowania obarczona jest defektami, które powodują niestabilne działanie określonych funkcjonalności lub też potrafią być przyczyną wadliwego działania całego systemu.

Defekt pojawia się w oprogramowaniu, kiedy osoba tworząca dany system popełnia błąd, który może się pojawić na różnych etapach tworzenia oprogramowania takich jak analiza wymagań, projektowanie dokumentacji systemowej (projekt ogólny/szczegółowy), plan testów, nieodpowiednie skrypty testowe, kod źródłowy etc.

Zatem ważnym aspektem jest proces testowania, podczas którego tester wykonuje określone przypadki testowe i może obserwować czy wyniki tych testów pokrywają się z oczekiwaniami. Wszelkie odstępstwa od oczekiwań są traktowane jako incydenty, które trzeba zbadać i wyjaśnić. Wszystkie wykryte usterki i problemy powinny być zapisywane w systemach do śledzenia błędów (ang. ITS) i / lub w systemach kontroli wersji (ang. VCS) w celu dalszej analizy i próby znalezienia rozwiązania problemu.

Repozytoria danych, w których przechowywane są powyższe informacje, mogą stanowić ciekawe źródło wiedzy dla badaczy i naukowców, którzy zajmują się problematyką procesów ulepszania oprogramowania (ang. SPI) [1] lub zapewnienia jakości (ang. QA).

Artykuł ma na celu przedstawienie ogólnego podejścia do problemu predykcji defektów w oprogramowaniu, w oparciu o modele operujące na informacjach historycznych z repozytoriów danych oraz dokonując ich analizy.

2. Prace powiązane

Istnieje wiele narzędzi wspierających pracę programistów i testerów w ich codziennej pracy nad systemami informatycznymi. Wśród tych narzędzi są takie, które pozwalają na wygodne i szybkie testowanie tworzonych rozwiązań zarówno na poziomie kodu źródłowego jak i na poziomie postkompilacyjnym. Wyłapywanie błędów w oprogramowaniu to rzecz niezmiernie ważna, aby finalnie dostarczać produkty pozbawione defektów. Jednakże ciągłe testowanie i debugowanie systemów wiąże się z poniesieniem wydatków związanych z użyciem zasobów ludzkich (programiści, testerzy) jak i zasobów finansowych [2]. Dlatego też predykcja defektów w oprogramowaniu jest ważna, gdyż ma realne zastosowanie w projektach komercyjnych, powinna pozwalać na wymierne oszczędności. Badania nad predykcją defektów mają zatem coraz to większe zainteresowanie zarówno po stronie praktyków jak i badaczy.

Ramler i Himmelbauer [3] proponują predykcję defektów przy użyciu modeli predykcyjnych powiązanych z systemami oprogramowania na poziomie ich modułów.

Moduły mogą być plikami, klasami, komponentami a także podsystemami danego systemu. Moduły te są opisywane poprzez zestawy atrybutów (np. przez metryki kodu czy liczbę zmian w danej iteracji), które to są dostępne poprzez ekstrakcję ich z różnych źródeł danych takich jak bazy metryk czy repozytoria kodów źródłowych.

Istnieje także wiele prac, w których autorzy oprócz skupiania się wyłącznie na modelach, sporo czasu poświęcają zagadnieniom akwizycji danych z repozytoriów.

Powstało kilka narzędzi przydatnych do zadań związanych z pozyskiwaniem danych oraz wspierających modelowanie predykcyjne (np. predykcję defektów).

Jureczko i Magot [4] przygotowali Quality Spyframework [5] o otwartym kodzie (licencja Apache 2.0 [6]), którego zadaniem było odczytywać i zbierać surowe dane z kodu źródłowego i repozytoriów zdarzeń oraz metryki zdefiniowane przez użytkowników. Projekt skupiał się na dwóch modułach dotyczących pozyskiwania danych oraz raportowania. W ostatniej wydanej wersji framework

pozwalał na odczytywanie metryk z klas dla technologii Java, odczytywanie zdarzeń systemu JIRA [7], a także wpisów z systemu kontroli wersji Subversion (SVN) [8].

D'Ambros i Lanza [9] zaproponowali narzędzie wspierające analizę ewolucji oprogramowania poprzez interfejs sieciowy – Churrasco. Jest to narzędzie o otwartym kodzie źródłowym pobierające i przetwarzające dane z systemu Bugzilla oraz SVN, opierające się na meta-modelu FAMIX, który jest niezależny od zastosowanej technologii programowania. Oprócz tego został wykorzystany obiektowy moduł mapowania relacyjnego (GLORP), moduł ekstrakcji faktów (MOOSE) oraz moduł SVG do wizualizacji.

Madeyski oraz Majchrzak [10] opracowali DefectPrediction for software systems (DePress), specjalny framework, którego celem jest rozszerzalny pomiar oprogramowania oraz integracja danych w celach predykcyjnych (predykcja defektów, predykcja kosztów/nakładów). Framework DePress bazuje na projekcie KNIME [11] i jest jego rozszerzeniem (zestawem wtyczek), pozwalającym budować modele graficzne przepływu danych w graficzny, prosty i przejrzysty dla użytkownika sposób.

Głównym założeniem projektu była predykcja defektów oprogramowania w oparciu o dane historyczne. W tym celu przygotowano zestaw wtyczek odpowiedzialnych kolejno za kolekcjonowanie, transformację oraz analizę danych. Autorzy skupili się na operacjach akwizycji i transformacji danych oraz raportowaniu, jednocześnie zostawiając operacje statystyczne oraz eksplorację danych [12] sprawdzonemu środowisku KNIME, które posiada odpowiednie wbudowane mechanizmy.

Ważnym aspektem, na który autorzy postawili jest także nacisk na archiwizację oraz udostępnianie zbiorów danych na zewnątrz (np. dla innych badaczy, którzy chcieli by testować własne rozwiązania). W tego typu badaniach ważne jest, aby repozytoria danych historycznych oraz przygotowane modele predykcyjne, były w miarę możliwości publiczne (po wcześniejszym procesie anonimizacji treści chronionych komercyjnym prawem autorskim).

3. Prognozowanie defektów w oparciu o dane historyczne

3.1. Predykcja defektów a cykl życia oprogramowania

Defekt może zostać wprowadzony na dowolnym etapie procesu zwanego SLDC (ang. Software Development Life Cycle) [13] dlatego bardzo ważne jest, aby testerzy byli zaangażowani od początku cyklu życia oprogramowania, po to aby wykrywać i usuwać wady.

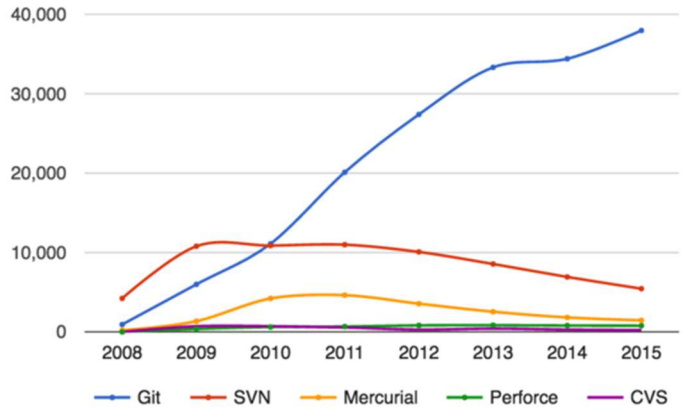
Rys. 1. Diagram cyklu życia procesów wytwarzania oprogramowania

Im szybciej dany defekt zostanie zlokalizowany i naprawiony, tym samym koszt utrzymania jakości będzie mniejszy. Przykładowo jeżeli defekt jest zidentyfikowany w fazie analizy wymagań, wtedy koszt naprawy sprowadza się do zmodyfikowania wymagań na odpowiednim dokumencie. Jednakże jeżeli, wymagania zostaną źle opisane i zaimplementowane, a defekt zostanie wykryty dopiero podczas fazy testowania, wtedy koszt naprawy będzie bardzo wysoki i będzie wiązał się z poprawą wymagań i specyfikacji oraz zmianą w implementacji. Będzie wymagał także dalszego procesu testowania.

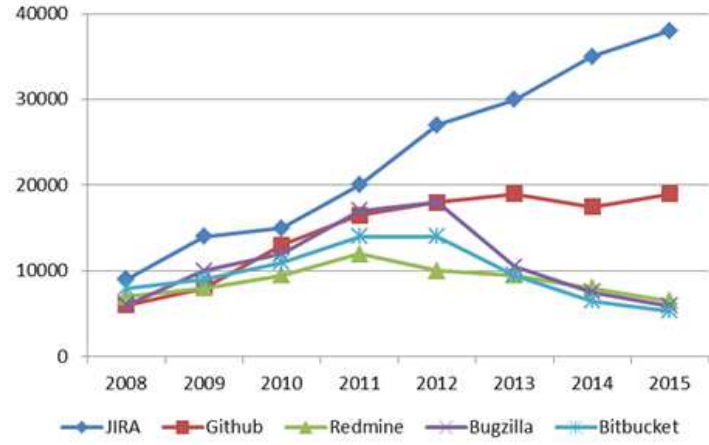
W niniejszym artykule autor skupia się na fazie implementacji i testowania, gdyż z poziomu tych faz możliwe jest uzyskanie odpowiednich danych historycznych z systemów zarządzania konfiguracją oprogramowania oraz systemów śledzenia błędów (oczywiście o ile te są przechowywane i pielęgnowane).

3.2. Główne założenia

Jednym z głównych założeń operacji predykcji defektów jest określenie źródeł danych historycznych, na podstawie których będzie odbywał się cały proces predykcyjny. Interesujące nas dane można uzyskać z systemów zarządzania konfiguracją oprogramowania oraz systemów śledzenia błędów. Obecnie na rynku istnieje wiele typów owych systemów, gdzie jednymi z najbardziej popularnych oraz używanych w profesjonalnych zespołach programistycznych są odpowiednio dla systemów kontroli wersji (pierwsza piątka): Git, Subversion, Mercurial, Perforce, CVS, a dla systemów śledzenia błędów (pierwsza piątka): JIRA, Github, Redmine, Bugzilla, BitBucket. Poniższe diagramy przedstawiają procentowy udział wspomnianych narzędzi (badanie przeprowadzone na podstawie zbadania popularności zapytań według słów kluczowych na portalu dla programistów – stackoverflow.com) [14]:



Rys. 2. Popularność systemów kontroli wersji (VCS)



Rys. 3. Popularność systemów do śledzenia błędów (ITS)

Akwizycja danych z wyżej wymienionych systemów może odbywać się na dwa sposoby: bezpośredni lub pośredni. Pierwszy z nich pozwala na dostęp do repozytoriów systemowych najczęściej poprzez podłączenie się do ich bazy danych lub też za pomocą odpowiednich mechanizmów, które pozwalają na odczyt informacji z tych baz.

W przypadku systemu JIRA, oprócz sparametryzowanych kwerend SQL bezpośrednio wykonywanych na odpowiednich tabelach w bazie danych (silnikiem

bazodanowym obsługującym system JIRA jest MSSQL), dostępne jest jeszcze API, które za pomocą odpowiednich żądań, zwraca oczekiwane wyniki. JIRA oferuje także swój własny mikro-język JIRA Query Language (JQL). Jest to najbardziej elastyczny sposób wyszukiwania danych w JIRA i jest dla wszystkich: programistów, testerów, kierowników projektów, a nawet nietechnicznych użytkowników biznesowych. Ta metoda może być dedykowana dla tych, którzy nie mają doświadczenia z zapytaniami do baz danych, a także dla tych, którzy chcą szybszego dostępu do informacji w JIRA.

Drugi sposób dostępu do danych opiera się na plikach wymiany, które są eksportowane ręcznie z systemów przez odpowiednie interfejsy np. do formatu CSV lub XML. Analiza wyeksportowanych plików może być najprostszym sposobem dostępu do danych oraz często jedynym, jeżeli np. w powodu braku uprawnień nie mamy dostępu bezpośredniego do systemu lub/i jego bazy danych.

W przypadku systemów kontroli wersja, tak samo do wyboru są dwie takie same opcje. Dla przykładu dla Gita lub Subversion, także możemy spróbować podłączyć się do ich baz (plikowych) oraz wyszukiwać interesujące nas artefakty lub też za pomocą odpowiednich narzędzi wyeksportować dane do plików wymiany.

Pobrane dane historyczne powinny być w pewien sposób ujednolicone oraz oczyszczone z niepotrzebnych pól (np. z informacji o autorach danego wpisu), następnie przetransformowane do wspólnego formatu np. formy tabelarycznej, która zostanie w następnych etapach poddana procesom eksploracji.

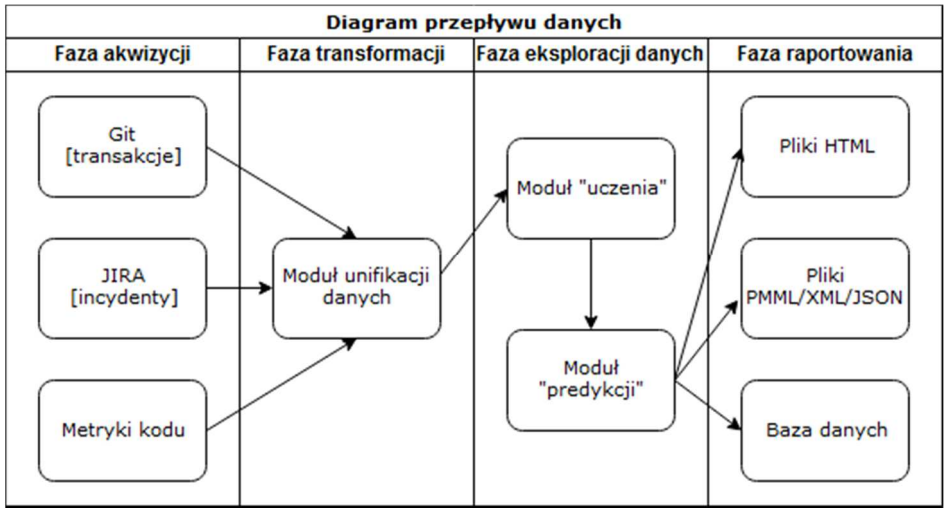
Do pobranych danych historycznych należy dołączyć także metryki kodu źródłowego. Metryki te powinny zostać odpowiednio powiązane z danymi historycznymi, w celu umożliwienia budowy klasyfikatorów predykcyjnych. Bez dołączenia stosownych metryk (np. liczba linii kodu w pliku, liczba klas w pliku, stopień zagnieżdżenia klas), nie było by możliwe zastosowanie modelu predykcyjnego w nowotworzonych systemach, gdzie dane historyczne nie występują.

3.3. Przykładowe podejście do tworzenia modelu predykcji

Jednym ze sposobów budowy modelu predykcyjnego, jest podejście dwu-etapowe, gdzie w pierwszym etapie budujemy oraz trenujemy nasz model, a w drugim etapie jest on używany do predykcji na nowych danych wejściowych. W każdym z dwóch etapów można wyróżnić kilka następujących faz:

1. Faza akwizycji danych – procesy pobierania danych.
2. Faza transformacji danych – procesy ujednolicania i dopasowania danych wejściowych.
3. Faza eksploracji danych – procesy związane z odkrywaniem nowej wiedzy.
4. Faza raportowania – procesy przedstawiania wyników oraz ich archiwizacja.

Z punktu widzenia badawczego, istotnym elementem do dalszych badań, ale także najtrudniejszym, jest dobór odpowiednich klasyfikatorów w fazie trzeciej – w tym celu stosowane są różne techniki z obszarów związanych np. z algorytmami ewolucyjnym, inteligencji roju [15] czy też logiką rozmytą.



Rys. 4. Diagram przepływu danych w procesach uczenia i predykcji

4. Studium przypadku

4.1. Wybór projektu do analizy

Dla poniższego opracowania, jednym z głównych kryteriów wyboru oprogramowania do analizy, był swobodny dostęp do danych archiwalnych zawartych w systemach kontroli wersji oraz w systemach śledzenia błędów. Takie kryterium spełniają projekty o otwartym kodzie źródłowym – wybór padł na system Apache Netbeans [16]. Jest to zintegrowane środowisko programistyczne (IDE) dla języka Java, którego głównym celem jest przyspieszenie budowy aplikacji Java, w tym również usług sieciowych oraz aplikacji mobilnych. Oprócz standardowej funkcjonalności, możliwe jest również rozszerzenie narzędzia o między innymi wsparcie dla języków programowania C i C++, wsparcie dla tworzenia aplikacji w architekturze SOA, użycia XML i schematów XML, BPEL i Java Web Services czy modelowania UML. Tak jak pozostałe produkty fundacji Apache –NetBeans jest rozprowadzany na licencji Apache License. Narzędzie te zostało wydane po raz pierwszy wiosną 1999 roku i jest cały czas nieustannie rozwijane, a jego ostatnia wydana wersja pochodzi z lipca 2019 roku (wersja o numerze 11.1 dla Javy 11). NetBeans jest w całej swojej historii rozwijany przez dużą grupę programistów – zwolenników otwartego oprogramowania (ponad 100 osób zaangażowanych

w projekt, na przestrzeni kilkunastu lat, ostatnie wydania wspierane przez 50 stałych programistów). Fakt tak aktywnego rozwoju wybranego narzędzia niewątpliwie jest spowodowany brakiem ograniczeń w rozwoju tego oprogramowania wraz z nielimitowanym dostępem do repozytoriów kodów źródłowych, zgodnie z polityką licencyjną fundacji Apache.

Repozytoria kodów źródłowych są przechowywane w systemie Git [17], natomiast wykryte defekty są raportowanie w systemie JIRA.

4.2. Akwizycja danych

Kody źródłowe narzędzia NetBeans są magazynowane w systemie Git, ostatnie czasy pokazują, że twórcy oprogramowania powiązani z fundacją Apache, migrują swoje projekty do systemu Git (np. z SVN) jako zyskującego coraz większą popularność. Zatem wybór padł na pobranie danych z systemu Git – w tym celu zostały użyte odpowiednie narzędzia między innymi GraphQL API v4 [18] dla Githuba oraz zestaw skryptów typu bash operujących na systemie Git. Zakres dat dla opisywanych działań to: styczeń 2018 – grudzień 2018.

Tabela 1. Główne informacje z systemu Git

Zakres czasu	01.01.2018 – 31.12.2018
Wydanie	9.0
Transakcje (commity)	560
Zmiany w plikach	2554
Liczba akt. programistów	118

Następnym krokiem była akwizycja danych z systemu JIRA dla zadanego okresu czasu. Z repozytorium JIRA, brane były pod uwagę wszystkie te zdarzenia, które były opisane jako: Bug o statusie Closed/Resolved, wyniku końcowym Fixed/Duplicated. Pominięto parametr priorytetu potrzeby naprawy defektu, w celu uzyskania szerszego zakresu informacji o zdarzeniach.

Tabela 2. Główne informacje z systemu JIRA

Zakres czasu	01.01.2018 – 31.12.2018
Relase	9.0
No. of issues	312
No. of issues (Bug)	128
Improvementtasks	60

W następnym etapie podjęta jest próba sparowania artefaktów pobranych z systemu Git z artefaktami pobranymi z systemu JIRA. Takie parowanie jest możliwe, jeżeli przykładowo w opisach transakcji (Git) występują opisy rozwiązywanych problemów zgłoszonych w systemie JIRA. Przykładowo należy poszukać (np. z użyciem wyrażeń regularnych) ciągu z numerem danego zdarzenia np. „ResolvedissueNETBEANS-953", gdzie po stronie JIRA zdarzenie te opisane jest: „(NETBEANS-953) Profile jdk10 remote platform NullPointerException".

Ostatnim etapem akwizycji danych jest pobranie metryk kodu źródłowego. Można tego dokonać przy pomocy narzędzi dedykowanych pod daną technologię programowania, w przypadku NetBeansa dla Javy, można użyć biblioteki JavaNCSS.

Tabela 3. Wybrane metryki z JavaNCSS

Moduł	Pakiety	Klasy	Metody
org-apache-tools-ant-module	33	45	56
org-netbeans-api-annotations-common	5	17	32
org-netbeans-api-debugger	12	77	152
org-netbeans-api-debugger-jpda	11	34	100
org-netbeans-api-intent	19	39	89
org-netbeans-api-io	20	62	122
org-netbeans-api-progress	5	17	77
org-netbeans-api-search	8	19	39
org-openide-loaders	13	37	99
org-openide-modules	14	29	101
org-netbeans-core-multiview	8	15	45
org-netbeans-core-network	14	29	120
org-netbeans-lib-v8debug	10	19	22
org-netbeans-modules-db	15	49	107

4.3. Transformacja danych

Podczas procesu transformacji i normalizacji, dane (kolumny) zbędne z punktu widzenia predykcji, są usuwane lub wartości im przypisane ustawiane są na zero. Każde artefakty, które zostały pobrane w poprzednim etapie i ze sobą powiązane, otrzymują opis właściwości: HasDefects oraz NoOfDefects z odpowiednimi wartościami opisującymi czy dla danej klasy/metody z kodu źródłowego, występują jakieś defekty, które zostały zgłoszone w systemie JIRA i naprawione w kolejnej transakcji do systemu kontroli wersji (Git).

4.4. Proces predykcji oraz wyniki

Metodologia predykcji powinna opierać się na minimum trzech etapach tj. wyborze atrybutów do predykcji, budowie modelu oraz jego walidacji. W pierwszym etapie wybieramy atrybuty, na których będzie opierał się budowany model. Stosowanie wszystkich atrybutów (danych historycznych oraz metryk) może prowadzić do przeładowania modelu, dlatego zaleca się wybrać tylko te wyróżniające się (można skorzystać np. z eliminacji wstecznej). Kolejny etap to budowa modelu, który może opierać się na jednym klasyfikatorze lub na kilku powiązanych ze sobą, tworząc tzw. klasyfikator hybrydowy. Popularnym klasyfikatorem stosowanym do wszelakich predykcji są drzewa decyzyjne. Dla zadanego przypadku atrybutem predykcyjnym jest HasDefects, który mówi czy w danym zestawie wejściowym znaleziony jest defekt. Zmiennymi niezależnymi są w tym przypadku atrybuty wejściowe wybrane w pierwszym etapie. Walidacja modelu jako ostatni etap, może opierać się przykładowo na walidacji krzyżowej (np. K-fold cross validation), gdzie dane wejściowe dzielone są na dwa segmenty: uczący i walidujący, a cały proces jest powtarzany kilkukrotnie z losowym podziałem (innym za każdym razem). O ile wyniki pracy modelu będą zadowalające, można go próbować zastosować do kolejnych wydań projektowych, zwłaszcza w fazie tworzenia kodu oraz testowania.

Wszystkie operacje związane z modelowaniem oraz testowaniem można przeprowadzić np. w środowisku badawczym KNIME, które ma pewien zestaw modułów wspierających techniki tworzenia modeli predykcyjnych. Pozwala także na zapisanie wyników do bazy danych, czy też eksport do formatu XML jako model PMML (przydatne do dalszych badań).

5. Podsumowanie i przyszłe prace

Artykuł ma na celu przedstawienie ogólnego podejścia do problemu predykcji defektów w oprogramowaniu, w oparciu o modele predykcyjne oparte na atrybutach historycznych uzyskiwanych z systemów kontroli wersji, systemów śledzenia błędów oraz metryk, uzyskiwanych z poziomu kodu źródłowego.

Został przedstawiony ogólny mechanizm procesu predykcji defektów opierający się na analizie wybranych repozytoriów danych projektu Apache NetBeans. Narzędzie to charakteryzuje się długoletnią obecnością na rynku informatycznym (od około 1999 roku), zatem systemy kontroli wersji jak i systemy śledzenia błędów zawierają sporą ilość interesujących danych do eksploracji. Na ich podstawie można zbudować model predykcyjny, który po odpowiedniej walidacji będzie w stanie przewidywać wystąpienia defektów w przyszłości w zadanym projekcie. Co przełoży się na wymierne korzyści w postaci oszczędności czasu (testowanie) i środków finansowych (poprawki).

Temat dotyczący skutecznej predykcji defektów jest tematem bardzo rozwojowym i porusza realne, współczesne problemy rynku informatycznego, dlatego też przewidziane są dalsze prace nad predykcją defektów. Przyszłe plany prac powinny skupiać się na możliwościach własnej, konkurencyjnej implementacji narzędzi wspierających akwizycję, normalizację oraz modelowanie predykcyjne. Oprócz tego istnieje potrzeba budowania bardziej skutecznych modeli predykcyjnych, w tym celu dalsze zainteresowania mogą skupiać się przykładowo na hybrydowych klasyfikatorach predykcyjnych.

Bibliografia

1. Petersen K., Wohlin C.: Software process improvement through the Lean Measurement (LEAM) method, Journal of Systems and Software, vol. 83, no. 7, pp. 1275-1287, 2010.
2. Wojszczyk R.: Quality Assessment of Implementation of Strategy Design Pattern. Advances in Intelligent Systems and Computing, Springer , Vol. 620, pp. 37 - 44, 2018.
3. Ramler R., Himmelbauer J.: Building Defect Prediction Models in Practice, Handbook of Research on Emerging Advancements in Software Engineering, pp. 540-565, 2014.
4. Jureczko M., Magott J.: QualitySpy: a framework for monitoring software development processes, Journal of Theoretical and Applied Computer Science, vol. 6, no. 1, pp. 35-45, 2012.
5. Jureczko M. and Contributors: "Quality Spy.", http://java.net/projects/qualityspy.
6. The Apache Software Foundation, "Apache License, Version 2.0." http://www.apache.org/licenses/LICENSE-2.0.html.
7. JIRA, Atlassian, https://www.atlassian.com/software/jira.
8. SVN, "Enterprise-classcentralized version control ", https://subversion.apache.org/
9. D'Ambros M., Lanza M., Distributed and Collaborative Software Evolution Analysis with Churrasco, Sci. Comput. Program., vol. 75, pp. 276-287, Apr. 2010.
10. Madeyski L., Majchrzak M.: Software Measurement and Defect Prediction with Depress Extensible Framework, Foundations of Computing and Decision Sciences, vol. 39, no. 4, 2014.
11. Berthold M. R., Cebron N., Dill F., Gabriel T. R., Meinl T., Ohl P., Sieb C., Thiel K., and Wiswedel B.: KNIME: The Konstanz Information Miner, Studies in Classication, Data Analysis, and Knowledge Organization (GfKL 2007), Springer, 2007.

12. Czyczyn-Egird D., Wojszczyk R.: The effectiveness of data mining techniques in the detection of DDoS attacks. Distributed Computing and Artificial Intelligence, 14th International Conference, Springer, Vol. 620, pp. 53-60, 2018.
13. Kazim A., A Study of Software Development Life Cycle Process Models, International Journal of Advanced Research in Computer Science, Volume 8, No. 1, 2017.
14. Version Control Systems Popularity in 2016, https://rhodecode.com/insights/version-control-systems-2016 (access on 08.2019).
15. Slowik A., Kwasnicka H., Nature Inspired Methods and Their Industry Applications – Swarm Intelligence Algorithms, IEEE Transactions on Industrial Informatics, Volume 14, Issue 3, pp. 1004-1015, March 2018.
16. Apache NetBeans, https://netbeans.apache.org/
17. GitHub Inc., http://www.github.com.
18. GraphQL API v4, https://developer.github.com/v4/

Streszczenie

W dzisiejszych czasach istnieje wiele metod i dobrych praktyk w inżynierii oprogramowania, które mają na celu zapewnienie wysokiej jakości tworzonego oprogramowania. Jednakże pomimo starań twórców oprogramowania, często w projektach występują defekty, których usuwanie wiąże się często z dużym nakładem finansowym oraz nakładem czasu. Artykuł prezentuje przykładowe podejście do predykcji defektów w projektach informatycznych opierając się na modelach predykcyjnych zbudowanych w oparciu o informacje historyczne oraz metryki produktu, zebrane z różnych repozytoriów danych.

Abstract

Nowadays, there are many methods and good practices in software engineering that are aimed at providing high quality of created software. However, despite the efforts of software developers, there are often defects in projects, the removal of which is often associated with a large financial and time expenditure. The article presents an example approach to defect prediction in IT projects based on predictive models based on historical information and product metrics, collected from various data repositories.

Keywords: data mining, defect prediction models, software metrics