Information and Software Technology 74 (2016) 204–218

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?

Yue Yu*, Huaimin Wang, Gang Yin, Tao Wang

National Laboratory for Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha, 410073, China

ARTICLE INFO

Article history:
Received 7 April 2015
Revised 20 December 2015
Accepted 11 January 2016
Available online 18 January 2016

Keywords:
Pull-request
Reviewer recommendation
Social network analysis

ABSTRACT

Context: The pull-based model, widely used in distributed software development, offers an extremely low barrier to entry for potential contributors (anyone can submit of contributions to any project, through *pull-requests*). Meanwhile, the project's core team must act as guardians of code quality, ensuring that pull-requests are carefully inspected before being merged into the main development line. However, with *pull-requests* becoming increasingly popular, the need for qualified reviewers also increases. GitHub facilitates this, by enabling the crowd-sourcing of *pull-request* reviews to a larger community of coders than just the project's core team, as a part of their *social coding* philosophy. However, having access to more potential reviewers does not necessarily mean that it's easier to find the right ones (the "needle in a haystack" problem). If left unsupervised, this process may result in communication overhead and delayed pull-request processing.

Objective: This study aims to investigate whether and how previous approaches used in bug triaging and code review can be adapted to recommending reviewers for *pull-requests*, and how to improve the recommendation performance.

Method: First, we extend three typical approaches used in bug triaging and code review for the new challenge of assigning reviewers to *pull-requests*. Second, we analyze social relations between contributors and reviewers, and propose a novel approach by mining each project's *comment networks* (CNs). Finally, we combine the CNs with traditional approaches, and evaluate the effectiveness of all these methods on 84 GitHub projects through both quantitative and qualitative analysis.

Results: We find that *CN*-based recommendation can achieve, by itself, similar performance as the traditional approaches. However, the mixed approaches can achieve significant improvements compared to using either of them independently.

Conclusion: Our study confirms that traditional approaches to bug triaging and code review are feasible for *pull-request* reviewer recommendations on GitHub. Furthermore, their performance can be improved significantly by combining them with information extracted from prior social interactions between developers on GitHub. These results prompt for novel tools to support process automation in social coding platforms, that combine social (*e.g.,* common interests among developers) and technical factors (*e.g.,* developers' expertise).

1. Introduction

The pull-based development model [1,2], used to integrate incoming changes into a project's codebase, is one of the most popular contribution models in distributed software development [3]. External contributors can propose changes to a software project without the need to share access to the central repository with the core team. Instead, they can create a *fork* of the central repository, work independently and, whenever ready, request to have their changes merged into the main development line by submitting a *pull-request*. The pull-based model democratizes contributing to open source projects supported by the distributed version control system (*e.g.,* `git` and `Mercurial`): anyone can contribute to any repository *via* submitting *pull-requests*, even if they are not part of the core team.

* Corresponding author. Tel.: +86 13627319266.
E-mail addresses: yuyue@nudt.edu.cn (Y. Yu), hmwang@nudt.edu.cn (H. Wang), yingang@nudt.edu.com (G. Yin), taowang2005@nudt.edu.com (T. Wang).

http://dx.doi.org/10.1016/j.infsof.2016.01.004
0950-5849/

SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Special Focus on Software Engineering for Internet Computing

August 2016, Vol. 59 080104:1–080104:14
doi: 10.1007/s11432-016-5595-8

Determinants of pull-based development in the context of continuous integration

Yue YU[1,2*], Gang YIN[1,2], Tao WANG[1,2], Cheng YANG[1,2] & Huaimin WANG[1,2]

[1]*College of Computer, National University of Defense Technology, Changsha* 410073*, China;*
[2]*National Laboratory for Parallel and Distributed Processing, Changsha* 410073*, China*

Received April 25, 2016; accepted May 30, 2016; published online July 18, 2016

Abstract The pull-based development model, widely used in distributed software teams on open source communities, can efficiently gather the wisdom from crowds. Instead of sharing access to a central repository, contributors create a fork, update it locally, and request to have their changes merged back, i.e., submit a *pull-request*. On the one hand, this model lowers the barrier to entry for potential contributors since anyone can submit *pull-requests* to any repository, but on the other hand it also increases the burden on integrators, who are responsible for assessing the proposed patches and integrating the suitable changes into the central repository. The role of integrators in pull-based development is crucial. They must not only ensure that pull-requests should meet the project's quality standards before being accepted, but also finish the evaluations in a timely manner. To keep up with the volume of incoming pull-requests, continuous integration ($\mathcal{CI}$) is widely adopted to automatically build and test every pull-request at the time of submission. $\mathcal{CI}$ provides extra evidences relating to the quality of pull-requests, which would help integrators to make final decision (i.e., accept or reject). In this paper, we present a quantitative study that tries to discover which factors affect the process of pull-based development model, including acceptance and latency in the context of $\mathcal{CI}$. Using regression modeling on data extracted from a sample of GitHub projects deploying the Travis-CI service, we find that the evaluation process is a complex issue, requiring many independent variables to explain adequately. In particular, $\mathcal{CI}$ is a dominant factor for the process, which not only has a great influence on the evaluation process *per se*, but also changes the effects of some traditional predictors.

Keywords pull-request, continuous integration, GitHub, distributed software development, empirical analysis

Citation Yu Y, Yin G, Wang T, et al. Determinants of pull-based development in the context of continuous integration. Sci China Inf Sci, 2016, 59(8): 080104, doi: 10.1007/s11432-016-5595-8

1 Introduction

Software development process plays a key role in software engineering [1]. There are many important benefits to be gained from building up a mature software process [1, 2], especially for reducing costs and improving software quality. Open source software development exploits the distributed intelligence of participants in internet communities [3]. Currently, the pull-based development model [4] is widely used in distributed software teams to integrate incoming changes into a project's codebase [5, 6]. It is an efficient model for software projects to gather the wisdom from crowds (i.e., the large group of

* Corresponding author (email: yuyue@nudt.edu.cn)

2014 21st Asia-Pacific Software Engineering Conference

Who Should Review This Pull-Request: Reviewer Recommendation to Expedite Crowd Collaboration

Yue Yu*, Huaimin Wang*, Gang Yin*, Charles X. Ling‡
*National Laboratory for Parallel and Distributed Processing,
College of Computer, National University of Defense Technology, Changsha, 410073, China
‡Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7
{yuyue,hmwang,yingang}@nudt.edu.cn, cling@csd.uwo.ca

***Abstract*—Github facilitates the pull-request mechanism as an outstanding social coding paradigm by integrating with social media. The review process of pull-requests is a typical crowdsourcing job which needs to solicit opinions of the community. Recommending appropriate reviewers can reduce the time between the submission of a pull-request and the actual review of it. In this paper, we firstly extend the traditional *Machine Learning* (ML) based approach of bug triaging to reviewer recommendation. Furthermore, we analyze social relations between contributors and reviewers, and propose a novel approach to recommend highly relevant reviewers by mining *comment networks* (*CN*) of given projects. Finally, we demonstrate the effectiveness of these two approaches with quantitative evaluations. The results show that *CN*-based approach achieves a significant improvement over the ML-based approach, and on average it reaches a precision of 78% and 67% for top-1 and top-2 recommendation respectively, and a recall of 77% for top-10 recommendation.**

***Keywords*—Pull-request, Reviewer Recommendation, Comment Network, Social Coding**

I. Introduction

The pull-based software development model [1], compared to traditional methods such as email-based patching [2] [3], makes developers contribute to software projects more flexibly and efficiently. In GitHub, the pull-request mechanism is upgraded to a unique social coding paradigm [4] by integrating multiple social media involving *follow*, *watch*, *fork* and *issue tracker*. The socialized pull-request model is pushing software evolution and maintenance into crowd-based development [5].

A typical contribution process [6] in GitHub involves following steps. First of all, a contributor could find an attractive project by following some well-known developers and watching their projects. Secondly, by forking an interesting one, the contributor implements a new feature or fixes some bugs based on his cloned repository. When his work is finished, the contributor sends the patches from the forked repository to its source by a pull-request. Then, all developers in the community have the chance to review that pull-request in the issue tracker. They can freely discuss whether the project needs that feature, whether the code style meets the standard or how to further improve the code quality. Next, in the light of reviewers' suggestions, the contributor would update his pull-request by attaching new commits, and then reviewers discuss that pull-request again. Finally, a responsible manager of the core team takes all the opinions of reviewers into consideration, and then merges or rejects that pull-request.

As a mushrooming number of developers use the pull-request mechanism to contribute their ideas and suggestions in GitHub, the efficiency of software evolution and maintenance is highly related to the crowd-based review process. However, the discussion among reviewers is time-consuming. Some relevant reviewers may not notice the new pull-request in time. Recommending reviewer will make the review process more effective, because it can reduce the time between the submission of a pull-request and the actual review of it.

A pull-request contains a title and description summarized its contributions of bug fixes or feature enhancements, so it is similar to a bug report in bug tracking systems. To the best of our knowledge, there is very few studies of reviewer recommendation for pull-requests. The most similar researches [7]–[12] are the approaches for recommending developers with the right implementation expertise to fix incoming bugs. For a newly received bug report, these approaches firstly find out some similar historical reports or source code files by measuring text similarity. Then, the expertise of a developer can be learned based on the bug-fixing history, source revision commits or code authorship. If we only focus on the text of pull-requests, these approaches can be extended to assign pull-requests to appropriate developers. However, the social relations between pull-request contributors and reviewers are neglected. Compared to bug fixing, the review process of pull-request is a social activity depending on the discussions among reviewers in GitHub. Thus, social relation is one of key factors of reviewer recommendation.

In this paper, we firstly implement the *Machine Learning* (ML) based approach of Anvik et al. [7], which is one of the most representative work of bug triaging. Furthermore, we analyze social relations among reviewers and contributors, and propose a novel approach of reviewer recommendation. Central to our approach is the use of a novel type of social network called *Comment Network* (*CN*), which can directly reflect common interests among developers. Finally, we conduct an empirical study on 10 projects which have received over 1000 pull-requests in GitHub. As there is no previous work of reviewer recommendation for pull-requests, we design a simple and effective method as a comparison baseline in experiments. The quantitative evaluations show that our *CN*-based approach achieves significant improvements over the baseline and ML-based method.

1530-1362/14 $31.00 © 2014 IEEE
DOI 10.1109/APSEC.2014.57

Wait For It: Determinants of Pull Request Evaluation Latency on GitHub

Yue Yu*†, Huaimin Wang*, Vladimir Filkov†, Premkumar Devanbu†, and Bogdan Vasilescu†
*College of Computer, National University of Defense Technology, Changsha, 410073, China
†Department of Computer Science, University of California, Davis, Davis, CA 95616, USA
{yuyue, hmwang}@nudt.edu.cn, {vfilkov, ptdevanbu, vasilescu}@ucdavis.edu

***Abstract*—The pull-based development model, enabled by git and popularised by collaborative coding platforms like BitBucket, Gitorius, and GitHub, is widely used in distributed software teams. While this model lowers the barrier to entry for potential contributors (since anyone can submit *pull requests* to any repository), it also increases the burden on *integrators* (*i.e.*, members of a project's core team, responsible for evaluating the proposed changes and integrating them into the main development line), who struggle to keep up with the volume of incoming pull requests. In this paper we report on a quantitative study that tries to resolve which factors affect pull request evaluation latency in GitHub. Using regression modeling on data extracted from a sample of GitHub projects using the Travis-CI continuous integration service, we find that latency is a complex issue, requiring many independent variables to explain adequately.**

I. Introduction

The pull-based development model [1] is widely used in distributed software teams to integrate incoming changes into a project's codebase [14]. Enabled by `git`, the distributed version control system, pull-based development implies that contributors to a software project need not share access to a central repository anymore. Instead, anyone can create *forks* (*i.e.*, local clones of the central repository), update them locally and, whenever ready, request to have their changes merged back into the main branch by submitting a *pull request*. Compared to patch submission and acceptance via mailing lists and issue tracking systems, the traditional model of collaboration in open source [3], [9], the pull-based model offers several advantages, including centralization of information and process automation: the contributed code (the patch) resides in the same version control system, albeit in a different branch or fork, therefore authorship information is effortlessly maintained; modern collaborative coding platforms (*e.g.*, BitBucket, Gitorius, GitHub) provide integrated functionality for pull request generation, automatic testing, contextual discussion, in-line code review, and merger.

Pull requests are used in many scenarios beyond basic patch submission, *e.g.*, conducting code reviews, discussing new features [14]. On GitHub alone, currently the largest code host in open source, almost half of the collaborative projects use pull requests exclusively (*i.e.*, all contributions, irrespective of whether they come from core developers with write access to the repository or from outside contributors, are submitted as pull requests, ensuring that only reviewed code gets merged) or complementarily to the shared repository model [14].

While the pull-based model offers a much lower barrier to entry for potential contributors, it also increases the burden on core team members who decide whether to integrate them into the main development branch [14], [17]. In large projects, the volume of incoming pull requests is quite a challenge [14], [21], *e.g.*, Ruby on Rails receives upwards of three hundred pull requests each month. Prioritizing pull requests is one of the main concerns of integrators in their daily work [14].

From a perspective of increasing numbers of pull requests, in this paper we report on a preliminary quantitative study that tries to resolve which factors affect pull request evaluation latency in GitHub. Using regression modeling on data extracted from a sample of GitHub projects using the Travis-CI continuous integration service, we find that:

- latency is a complex issue, requiring many independent variables to explain adequately;
- the presence of CI is a strong positive predictor;
- the number of comments is the best single predictor of the latency;
- as expected, the size of a pull request (lines added, commits) is a positive, strong predictor.

II. The Pull Request Evaluation Process

Continuous Integration: The principal mechanism through which integrators ensure that pull requests can be handled efficiently without compromising quality is automatic testing, as supported by continuous integration (CI) services [14], [17]. Whenever a new contribution is received by a project using CI, it is merged automatically into a testing branch, the existing test suites are run, and the submitter and integrators are notified of the outcomes. If tests fail, the pull request is typically rejected (closed and not merged in GitHub parlance) by one of the integrators, who may also comment on why the contribution is inappropriate and how it can be improved. If tests pass, core team members proceed to do a team-wide code review by commenting inline on (parts of) the code, and including requests for modifications to be carried out by the submitter (who can then update the pull request with new code), if necessary. After a cycle of comments and revisions, and if everyone is satisfied, the pull request is closed and merged. In rare cases, pull requests are merged even if (some) tests failed. Only core team members (integrators) and the submitter can close (to merge or reject—integrators, and to withdraw—submitter) and reopen pull requests.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.42

Reviewer Recommender of Pull-Requests in GitHub

Yue Yu*, Huaimin Wang*, Gang Yin*, Charles X. Ling†
*National Laboratory for Parallel and Distributed Processing,
College of Computer, National University of Defense Technology, Changsha, 410073, China
†Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7
{yuyue,hmwang,yingang}@nudt.edu.cn, cling@csd.uwo.ca

***Abstract*—Pull-Request (PR) is the primary method for code contributions from thousands of developers in GitHub. To maintain the quality of software projects, PR review is an essential part of distributed software development. Assigning new PRs to appropriate reviewers will make the review process more effective which can reduce the time between the submission of a PR and the actual review of it. However, reviewer assignment is now organized manually in GitHub. To reduce this cost, we propose a reviewer recommender to predict highly relevant reviewers of incoming PRs. Combining information retrieval with social network analyzing, our approach takes full advantage of the textual semantic of PRs and the social relations of developers. We implement an online system to show how the reviewer recommender helps project managers to find potential reviewers from crowds. Our approach can reach a precision of 74% for top-1 recommendation, and a recall of 71% for top-10 recommendation.** `http://rrp.trustie.net/`

***Keywords*—Pull-request, Reviewer Recommendation, Social Network Analysis, Distributed Software Development**

I. Introduction

GitHub[1], a popular social coding community [1], attracts a large number of software projects hosted on it. Pull-Request (PR) is the primary method [2], [3] for code contributions from thousands of developers. Currently, it is not uncommon in the popular projects to receive tens of PRs daily covering nearly 60% of code commits from contributors.

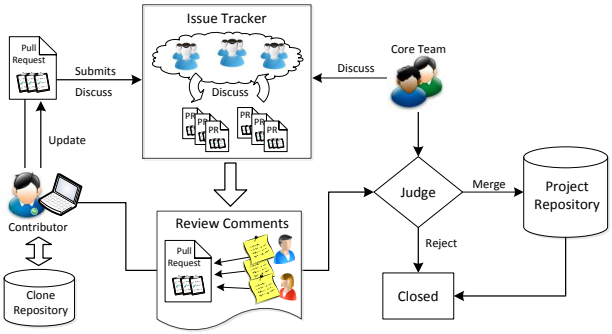

Figure 1. The overview of pull-request mechanism

The overview of PR mechanism is presented as Figure 1. Firstly, a contributor implements some new features or fixes bugs based on his personal repository cloned from the latest version of project repository. When his work is finished, the patches are packaged as a PR submitted to the issue tracker. The system open a new issue for this PR, and then add the issue to an awaiting list to be reviewed. In GitHub, the traditional review process is transformed into a crowdsourcing job. Not only core developers but also external developers in the community can act as reviewers. The reviewers can freely discuss the PR with the contributor and core developers in terms of their interests and expertise. Next, in the light of reviewers' suggestions, the contributor would update his pull-request by attaching new commits, and then reviewers discuss that PR again. Finally, the responsible mangers of the core team take all the opinions of reviewers into consideration, and then merges or rejects that PR. Thus, we can see PR review is an important way to maintain the quality of the software.

Assigning incoming PRs to appropriate reviewers will make PR review more effective, because it can reduce the time between the submission of a PR and the actual review of it. We refer to the period between the time when a PR is submitted into issue tracker and the time when it begins to be discussed by reviewers as review latency. The PRs which have been assigned to reviewers have lower review latency than those without assignment. According to our analysis, the time of the recommended reviewer submitting his first comment on PR is on average 40.8 hours shorter than those without recommendation. However, reviewer assignment is now organized manually in GitHub. As each developer in community has the chance to join the review discussions, the project managers may not completely find out all potential reviewers from crowds.

To reduce this cost, we designed a reviewer recommender to predict appropriate reviewers for incoming PRs in Github. The two key intuitions of our approach focus on the textual semantic of PRs and the social relations of contributors.

- The expertise of a reviewer can be learned from his PR-commenting history. For a newly received PR, the developers who have commented similar PR frequently in the past are the suitable candidates to review the new one.
- The common interests among developers can be measured by social relations between contributors and reviewers in historical PRs. The developers who share more common interests with the contributor are the appropriate reviewers of his incoming PRs.

Thus, we propose a novel approach combining information retrieval with social network analyzing to recommend highly relevant reviewers. We demonstrated the efficiency of our approach on 10 popular projects which have received over

[1]https://github.com/

Exploring the Patterns of Social Behavior in GitHub

Yue Yu, Gang Yin, Huaimin Wang, Tao Wang
National Laboratory for Parallel and Distributed Processing
School of Computer Science, National University of Defense Technology, Changsha, 410073, China
{yuyue, yingang, hmwang, taowang2005}@nudt.edu.cn

ABSTRACT

Social coding paradigm is reshaping the distributed software development with a surprising speed in recent years. Github, a remarkable social coding community, attracts a huge number of developers in a short time. Various kinds of social networks are formed based on social activities among developers. Why this new paradigm can achieve such a great success in attracting external developers, and how they are connected in such a massive community, are interesting questions for revealing power of social coding paradigm. In this paper, we firstly compare the growth curves of project and user in GitHub with three traditional open source software communities to explore differences of their growth modes. We find an explosive growth of the users in GitHub and introduce the *Diffusion of Innovation* theory to illustrate intrinsic sociological basis of this phenomenon. Secondly, we construct *follow-networks* according to the *follow* behaviors among developers in GitHub. Finally, we present four typical social behavior patterns by mining *follow-networks* containing *independence-pattern*, *group-pattern*, *star-pattern* and *hub-pattern*. This study can provide several instructions of crowd collaboration to newcomers. According to the typical behavior patterns, the community manager could design corresponding assistive tools for developers.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics - *Process metrics*; D.2.9 [**Software Engineering**]: Management - *Programming teams*

General Terms

Human Factors, Measurement, Management

Keywords

Behavior pattern, Social network, Social coding, Distributed software development

CrowdSoft'14, November 17, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3224-8/14/11...$15.00
http://dx.doi.org/10.1145/2666539.2666571

1. INTRODUCTION

In recent years, social coding paradigm has been brought into focus in distributed software development for the developers from all over the world. Various kinds of social media [10, 11] are employed in software development, which help building social ties among developers and form different types of social networks. Such social mechanisms can achieve transparency [5] within social coding ecosystem and improve the degree of collaboration in software development.

GitHub[1], a typical social coding community, attracts a large number of users and projects in a short period of time. When launched in 2008, there were only four users [2]. But it seems to rise to fame overnight and increases to more than 3.5 million developers now. GitHub employs several social media such as *follow*, *watch* and *fork*. The developers can track the activities of others and be aware of changes in project using these tools in the community. Many interesting social networks of developers can be constructed. For example, the *follow* relation is created when a developer click the "*follow*" button in the profile of another developer, and then the *follow* relations among developers can form a social network which is called *follow-network* in this paper. Why this new paradigm can achieve such a great success in attracting a large number of developers, and how they are connected in such a massive community, are important questions for understanding such a new paradigm. Many researches are conducted on analyzing the influence of social network in Open Source Software (OSS) communities (see Section 7). However, these work study the network structure [13] of collaboration-oriented social network and collaboration pattern [12] in traditional OSS communities. However, none of them has explored the growth modes of communities and social behavior patterns of developers.

In this paper, we firstly explore the growth curves of GitHub compared to three traditional OSS communities. Then, we construct *follow-networks* from the follow behaviors among developers, which is a typical interest-oriented social network. Finally, we analyze the social behavior patterns among developers by mining the *follow-networks*.

In summary, the following research questions would be answered in this paper:

RQ1: What are the differences between the growth modes of GitHub and traditional OSS communities, and is there any sociological theory that supports the special growth mode of GitHub?

RQ2: Whether or not the social connections among developers form some distinctive behavior patterns in GitHub,

[1]https://github.com

HESA: The Construction and Evaluation of Hierarchical Software Feature Repository

Yue Yu, Huaimin Wang, Gang Yin, Xiang Li, Cheng Yang
National Key Laboratory for Parallel and Distributed Processing
School of Computer Science, National University of Defense Technology
Changsha, China
yuyue_whu@foxmail.com, whm_w@163.com, {jack.nudt, shockleylee}@gmail.com

***Abstract*—Nowadays, the demand for software resources on different granularity is becoming prominent in software engineering field. However, a large quantity of heterogeneous software resources have not been organized in a reasonable and efficient way. Software features, a kind of important knowledge for software reuse, are ideal materials to characterize software resources. Our preliminary study shows that the effectiveness of feature-related tasks, such as software resource retrieval and feature location, will be greatly improved, if a multi-grained feature repository is available. In this paper, we construct a *Hierarchical rEpository of Software feAture* (HESA), in which a novel hierarchical clustering approach is proposed. For a given domain, we first aggregate a large number of feature descriptions from multiple online software repositories. Then we cluster these descriptions into a flexible hierarchy by mining their hidden semantic structures. Finally, we implement an online search engine on HESA and conduct a user study to evaluate our approach quantitatively. The results show that HESA can organize software features in a more reasonable way compared to the classic and the state-of-the-art approaches.**

***Keywords*—Software reuse; Mining Software repository; Feature-ontology; Clustering;**

I. Introduction

Software reuse is widely recognized as an effective way to increase the quality and productivity of software [1]. With the development of software industry, the degree of software reuse is deeper than previous years and the demand for resources on different granularity becomes more prominent. For example, when developing a new large software system, we may reuse some API calls from the third party to accomplish the core functions and the mature open source software as the basic framework. Additionally, some code fragments or components can be reused to meet other additional demands. The reusable resources are multi-grained, consisting of API calls (the finest level of granularity), code fragments, components (higher than API calls) and software systems (much higher than others).

However, considering the large-scale, heterogeneous and multi-grained software resources, it is a great challenge for stakeholders to retrieve the suitable one. With the evolution of open source ecosystems, more than 1.5 million open source software projects are now hosted in open source communities [2]. Reusable resources [3] are manifold, including code bases, execution traces, historical code changes, mailing lists, bug databases and so on. All of these valuable resources have not been reorganized in a reasonable and efficient way to assist in the activities of software development.

As a kind of attributes which capture and identify commonalities and differences in a software domain, software feature [4] [5] is an ideal material to characterize the software resources. Constructing a feature repository of a flexible structure can make a great contribution to multi-grained reuse.

However, classic feature analysis techniques, such as Feature Oriented Domain Analysis (FODA) [6] and Domain Analysis and Reuse Environment (DARE) [7], are heavily relied on the experience of domain experts and plenty of market survey data. Hence, the feature analysis is a labor-intensive and error-prone process.

In recent years, more and more stakeholders develop, maintain and share their software products on the Internet. In order to promote their products to users, project managers write some market-oriented summaries, release notes and feature descriptions on the profile pages via natural language. The large number of online software profiles can be treated as a kind of repository containing a wealth of information about domain-specific features. Although researchers propose several automatic methods to mine features from the web repository [8] [9] [10], the problems have not completely be solved, specifically in organizing features as flexible granularity.

In this paper, we are trying to address the above problems by proposing a novel approach to construct a *Hierarchical rEpository of Software feAture* (HESA). First of all, we extract a massive number of feature descriptions from online software profiles and mine their hidden semantic structure by probabilistic topic model. Then, we present an *improved Agglomerative Hierarchical Clustering* (iAHC) algorithm, seamlessly integrated with the topic model, to build the feature-ontology of HESA. Finally, we implement an online search engine[1] for HESA to help retrieve features in a multi-grained manner, which can support multiple reuse requirements. By conducting a user study, we demonstrate the effectiveness of our system with quantitative evaluations comparing to the classic and the state-of-the-art approaches.

The rest of this paper is or organized as follows. Section II introduces the overview of our work. Section III describes how to construct HESA in detail. Experiments and analysis can be found in Section IV. Finally, we present some related

[1] http://influx.trustie.net

2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications

A Trusted Remote Attestation Model based on Trusted Computing

Yue Yu, Huaimin Wang, Bo Liu, Gang Yin
National Laboratory for Parallel and Distributed Processing
National University of Defense Technology
Changsha, China
yuyue_whu@foxmail.com, whm_w@163.com, jack_nudt@gmail.com

***Abstract*—Traditional security protocols can not be trusted in some application scenarios of high security level because the endpoints integrity is ignored. In this paper, we propose a novel trusted remote attestation model which combines the secure channel and the integrity measurement architecture of trusted computing. We design and implement a prototype system based on a mature security protocol, Transport Layer Security (TLS) protocol, integrated with integrity report provided by trusted platform module (TPM). The TLS protocol guarantees the security of data exchange process and the integrity report of TPM provides the evidence about the trustworthiness and the security state of the communication endpoints. Compared by traditional approaches, our method is more efficient and can be deployed in large scale systems easily.**

Keywords—remote attestation; secure channel; integrity report; trusted computing

I. INTRODUCTION

Modern era, people are more dependent on the Internet than before, and have increasing demand for service provided by the Internet. Some confidential information requires transmitting in secure channel. The traditional security network protocols, such as Security Socket Layer (SSL) protocol and Transport Layer Security (TLS) protocol, just set up a secure channel in which attackers cannot steal or distort transmitting data[4][5].

However, this kind of secure channel is not integrity because of the exclusion of the endpoints secure state. As a result, if the endpoint is invaded by malicious software, it is possible to appear such embarrassing situation that after terminal identity authentication passing through, establishing connection and secure transmission, data is stolen by malicious codes on the terminal. Take the once wide spread virus, Win32.Huhk.d.7607 called "E-band hiding robber", as an example. The virus can infect the main program of IE browser in the system, IEXPLORER.EXE, after into the system through web Trojan. In this way, when the user using IE browser log in the E-bank and trade, the virus can automatically intercept a lot of related information, including the user's payment card number, password, payee name and other sensitive information. The virus invades user terminal, thus bypassing the secure communication channel established by both sides, which bring the great danger to E-bank users. Therefore, using a secure channel to an end-point of unknown integrity is ultimately futile. In the words of Gene Spafford [1], "using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench."

People have realized that in the face of existing security risks and threats, we not only need a top-down security system design, but also need to bottom up ensure the credibility of computing systems from the terminal. Thus the technology of trusted computing (TC) developed rapidly and has also become a hotspot in academia [2][3]. Reporting integrity information to a remote platform is one of the main goals of TC as proposed by the TCG. There is one security chip named Trusted Platform Module (TPM) integrated into mother board of computing platform [6]. Before every component taking control of main CPU, such as BIOS, MBR, OS Kernel, Application and so on, its characteristic code and configure data must be measured, and the measured value is stored into TPM Platform Configuration Registers (PCR). When the computing platform wants to access some resources in remote entity, remote entity can ask the computing platform to give a security status report. Report data includes computing platform's TPM-based identity information and PCR value. Remote entity evaluates the report data and makes decision to allow computing platform to access the resource or not, which is illustrated in Figure 1.

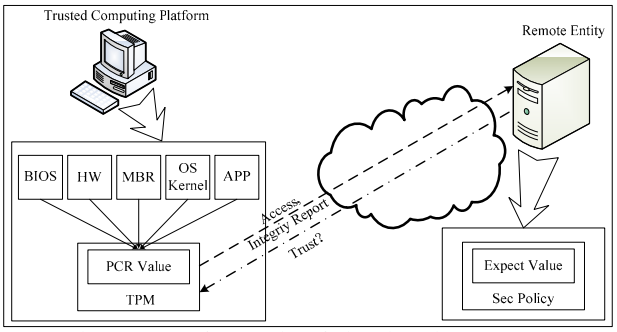

Figure 1. Trusted Computing Platform integrity report

This mechanism has some serious limitations. It has many shortcomings such as inconvenient software upgrading, not adapting to dynamic changes of the system configuration, easy to bind to a special products, and easy to leak platform

978-0-7695-5022-0/13 $26.00 © 2013 IEEE
DOI 10.1109/TrustCom.2013.183

Mining and Recommending Software Features across Multiple Web Repositories

Yue Yu, Huaimin Wang, Gang Yin, Bo Liu
National Laboratory for Parallel and Distributed Processing
School of Computer Science, National University of Defense Technology, Changsha, 410073, China
yuyue_whu@foxmail.com, whm_w@163.com, jack_nudt@163.com

ABSTRACT

The "*Internetware*" paradigm is fundamentally changing the traditional way of software development. More and more software projects are developed, maintained and shared on the Internet. However, a large quantity of heterogeneous software resources have not been organized in a reasonable and efficient way. Software feature is an ideal material to characterize software resources. The effectiveness of feature-related tasks will be greatly improved, if a multi-grained feature repository is available. In this paper, we propose a novel approach for organizing, analyzing and recommending software features. Firstly, we construct a *Hierarchical rEpository of Software feAture* (HESA). Then, we mine the hidden affinities among the features and recommend relevant and high-quality features to stakeholders based on HESA. Finally, we conduct a user study to evaluate our approach quantitatively. The results show that HESA can organize software features in a more reasonable way compared to the traditional and the state-of-the-art approaches. The result of feature recommendation is effective and interesting.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Mining Software Repository; H.3.3 [**Information Storage and retrieval**]: Feature Model, Clustering, Query formulation

General Terms

Algorithms, Human Factors

Keywords

Mining Software Repository, Domain Analysis, Feature Ontology, Recommender System

1. INTRODUCTION

The Internet is undergoing a tremendous change towards the globalized computing environment. With the vision of "Internet as computer"[21][23], more and more software projects are developed, maintained and diffused through the Internet computing environment. The Internet-based software repositories, such as Sourceforge.net[1], Freecode.com[2], Ohloh.com[3] and Softpedia.com[4], have hosted large amounts of software projects, which are fundamentally changing the traditional paradigms of software development. Around the repositories, manifold reusable software resources[13] have been accumulated, including code bases, execution traces, historical code changes, mailing lists, bug databases, software descriptions, social tags, user evaluations and so on.

However, all of these valuable resources have not been reorganized in a reasonable and efficient way to assist in the activities of software development[30]. For example, a large proportion of projects in the above repositories have not been categorized or marked with some effective tags. In Sourceforge.net, there are 39.8% software projects have no category label and in Ohloh.com 61.68% projects have not tagged by users. Table 1 (according to data in mid-2011) presents the details about our statistical results. Considering the large-scale, heterogeneous and multi-grained software resources, it is a great challenge for stakeholders to retrieve the suitable one.

Table 1: Labels in open source communities

Repository	total projects	unique labels	ratio (#label=0)	ratio (#label=0,1)
SourceForge	298,402	363	39.80%	77.00%
Ohloh	417,344	102,298	61.68%	69.89%
Freecode	43,864	6,432	8.61%	20.60%

As a kind of visible attributes which capture and identify commonalities and differences in a software domain, *Feature*[3][15] is an ideal material to represent the software resources. For example, when a company wants to develop a new commercial software product about Video-Player, domain analysts might evaluate user comments to pick out outstanding competing products, analyze and extract the reusable feature assets, combine the related function points and design a novel feature model. Based on the feature model, developers match the features with corresponding software resources including code fragments, components and mature open source software.

However, classic feature analysis techniques, such as Feature Oriented Domain Analysis (FODA)[14] and Domain Analysis and Reuse Environment (DARE)[9], are heavily relied on the experience of domain experts and plenty of

[1]http://sourceforge.net
[2]http://freecode.com
[3]http://www.ohloh.net
[4]http://www.softpedia.com

Internetware '13 October 23-24 2013, Changsha, China

Quality and Productivity Outcomes Relating to Continuous Integration in GitHub

Bogdan Vasilescu[†*], Yue Yu[‡†*], Huaimin Wang[‡], Premkumar Devanbu[†], Vladimir Filkov[†]

[†]Department of Computer Science
University of California, Davis
Davis, CA 95616, USA
{vasilescu, ptdevanbu, vfilkov}@ucdavis.edu

[‡]College of Computer
National University of Defense Technology
Changsha, 410073, China
{yuyue, hmwang}@nudt.edu.cn

ABSTRACT

Software processes comprise many steps; coding is followed by building, integration testing, system testing, deployment, operations, among others. Software process integration and automation have been areas of key concern in software engineering, ever since the pioneering work of Osterweil; market pressures for Agility, and open, decentralized, software development have provided additional pressures for progress in this area. But do these innovations actually help projects? Given the numerous confounding factors that can influence project performance, it can be a challenge to discern the effects of process integration and automation. Software project ecosystems such as GitHub provide a new opportunity in this regard: one can readily find large numbers of projects in various stages of process integration and automation, and gather data on various influencing factors as well as productivity and quality outcomes. In this paper we use large, historical data on process metrics and outcomes in GitHub projects to discern the effects of one specific innovation in process automation: *continuous integration*. Our main finding is that continuous integration improves the productivity of project teams, who can integrate more outside contributions, without an observable diminishment in code quality.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Experimentation, Human Factors

Keywords

Continuous integration, GitHub, pull requests

*Bogdan Vasilescu and Yue Yu are both first authors, and contributed equally to the work.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786850

1. INTRODUCTION

Innovations in software technology are central to economic growth. People place ever-increasing demands on software, in terms of features, security, reliability, cost, and ubiquity; and these demands come at an increasingly faster rate. As the appetites grow for ever more powerful software, the human teams working on them have to grow, and work more efficiently together.

Modern games, for example, require very large bodies of code, matched by teams in the tens and hundreds of developers, and development time in years. Meanwhile, teams are globally distributed, and sometimes (*e.g.*, with open source software development) even have no centralized control. Keeping up with market demands in an agile, organized, repeatable fashion, with little or no centralized control, requires a variety of approaches, including the adoption of technology to enable process automation. Process Automation *per se* is an old idea, going back to the pioneering work of Osterweil [32]; but recent trends such as open-source, distributed development, cloud computing, and software-as-a-service, have increased demands for this technology, and led to many innovations. Examples of such innovations are distributed collaborative technologies like `git` repositories, forking, pull requests, continuous integration, and the DEVOPS movement [36]. Despite rapid changes, it is difficult to know how much these innovations are helping improve project outcomes such as productivity and quality. A great many factors such as code size, age, team size, and user interest can influence outcomes; therefore, teasing out the effect of any kind of technological or process innovation can be a challenge.

The GitHub ecosystem provides a very timely opportunity for study of this specific issue. It is very popular (increasingly so) and hosts a tremendous diversity of projects. GitHub also comprises a variety of technologies for distributed, decentralized, social software development, comprising version control, social networking features, and process automation. The development process on GitHub is more democratic than most open-source projects: *anyone* can submit contributions in the form of *pull requests*. A pull request is a candidate, proposed code change, sometimes responsive to a previously submitted modification request (or *issue*). These pull requests are reviewed by project insiders (*aka* core developers, or integrators), and accepted if deemed of sufficient quality and utility. Projects that are more popular and widely used can be expected to attract more interest, and more pull requests; these will have to be