VOL. 11, NO. 23, DECEMBER 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

GENERATING AUTOMATIC CERTIFYING REFACTORED ENGINE FOR SOFTWARE LEGACY SYSTEM

M. Srinivas[1], G. Rama Krishna[1] and K. Rajasekhara Rao[2]
[1]Department of Computer Science Engineering, KLEF, K L University, AP, India
[2]Sri Prakash College of Engineering, AP, India
E-Mail: srinu_cse@kluniversity.in

ABSTRACT

Refactoring or Platform migration is a process of improving the underlying design and architecture of legacy systems that subsequently can improve their performance and maintainability. Many of the legacy technologies are no longer supported, hence the need for migration. However, the refactoring tools are not correct in every possible cases and programmers cannot trust them. One has to make sure that the functionality of the legacy system remains intact after going through the process of migration. Hence there is a need to build certified refactoring tools which were useful for industrial developments. In this paper, we will address the complete automated certification mechanism which certifies all the functional components of a service or application and various process involved during the certification phase. We are particularly interested in complex program transformation based on a sequence of refactoring operations provided by eclipse tools.

Keywords: legacy systems, refactored services, migration, certification, eclipse, bugs.

1. INTRODUCTION

In software engineering, series of evolutions downgrade the quality of code [8,9]. Indeed, each modification gets harder to implement. This eventually requires fixing or changing the structure of the program, without changing its behaviour, in order to ease future evolutions [10, 11]. Such architecture modifications are integrated in agile development processes. Unfortunately, tools that change the structure of programs without modifying their behaviour (i.e., refactoring tools) are generally not correct [12]. Indeed, refactoring tools may change the behaviour of the program and so their use requires systematic and extensive testing in order to detect the newly introduced bugs. This is the reason why users do not trust these tools and tend to prefer manual modifications [13, 14]. This paper addresses the correctness of refactoring tools. Since the grammars of various programming are rich, numerous cases must be taken into account in the design of such tools, which makes the task difficult. Moreover, refactoring operations should preserve as much as possible the layout of the source code as well as its comments and its pre-processing directives (macros, pragma, etc.). In addition to a difficult design, the implementation or a refactoring tool is critical. So a theoretical study of a transformation is not sufficient to ensure the quality of the final tool. For this reason, a correct-by-construction approach should work well here. Up to now, this could not be applied to industrial strength. Also, Programming languages are lack of formal semantics, existing formalizations are restricted to a subset of the languages. This proposed work devotes to prove the correctness of refactoring tools which provides powerful mechanisms for automated reasoningand certified code generation. The contributions of this paper are- An extension of the verification of refactoring's to produce verified code; A discussion of patterns of refactoring that are applicable to the verified code itself; Although we restrict our focus to refactoring's, these steps may be followed to produce various other kinds of certified software.

2. RELATED WORK

The intention to build refactoring tools from the verification of refactoring was also expressed in earlier work. A larger development is described by Blazy et al. [19] and Leroy [18], in which a compiler is certified - its frontend compiles a fragment of C into an intermediate language called Cminor, and the backend completes the compilation into PowerPC's assembly language. Okuma and Minamide [17] use Isabelle/HOL to specify and verify a compiler, of which code is then generated and embedded into a larger system that compiles a small functional language into Java bytecode. Garrido & Meseguer [15] and Junior *et al*. [16], using the systems Maude and CafeOBJ respectively. Using interactive theorem provers to build certified programming tools has been attempted for different tools and using different systems. Tourw´e and Mens described three phases of refactoring [20]: perceiving when the refactoring should be applied, identifying which methodology to apply, and finally carrying out the refactoring process. The steps in this simpler model correspond to the Identify, Initiate, and Execute respectively. Demeyer [21] shows that refactoring can have a valuable influence on software performance (e.g. compilers can optimize better on polymorphism than on simple if else statements). Bois and Mens [22] develop a framework for investigating the effects of refactoring on internal quality metrics, but again, they have not provided an experimental substantiation in an industrial environment. Stroggylos *et al*. [7] evaluated source code version control system logs of popular open source software systems to detect changes marked as refactoring's and examined how the software metrics are affected by this process. DuBois *et al*. studied the impact of refactoring on cohesion and coupling metrics in [4] and identified the benefits that can follow, and defined the

application of refactoring could improve selected quality characteristics [5]. Fontana et al. studied the effect of refactoring applied to reduce code smells on the quality assessment of the system [6]. Kataoka and colleagues' introduced a 3-step model [3]: identification of refactoring candidates, validation of refactoring effect, and application of refactoring. This corresponds to the Identify, Interpret Results, and Execute steps respectively. Vakilian *et al.* proposed a compositional model for refactoring (automate individual steps and let programmers manually compose the steps into a complex change) and implemented a tool to support it. Henkel et al. implemented a framework which captures and replays refactoring actions.

3. IMPORTANCE OF LEGACY SYSTEM

In IT Organizations, the term legacy system relates to being a previous or outdated computer system. At times it might likewise have little to do with the age of the framework also. The legacy framework could conceivably be being used. For variety of reasons, a legacy framework might keep on being utilized. The choice to keep the legacy framework may be impacted by monetary reasons like, rate of profitability or merchant lock-in. Some of the most common scenarios that forced the IT organizations for keeping the legacy system include

- The system works satisfactorily, and the owner sees no reason to change it.
- The expenses of renovating or replacing the Legacy systems are restrictive in light of the fact that it is vast, solid, and/or complex.
- Retraining on another framework would be excessive in lost time and money, contrasted with the expected advantages of replacing it (which may be zero).
- The system requires consistent accessibility, so it can't be taken out of administration, and the expense of planning new system with a comparative accessibility level is high.
- The way that System works is not surely knew. Such a circumstance can happen when the Designers of the system have left the association and the system has either not been completely documented or documentation has been lost
- The user expects that the system can easily be replaced when this becomes necessary.

Even though we discussed many reasons to keep the legacy system, there is a definite need to migrate the legacy system (Refactoring). Some of them are legacy technology is no longer supported, Legacy application performance issues, When a merger happens between two large IT organizations, Funding available to migrate the legacy system.

4. OVERVIEW OF REFACTORING THE LEGACY SYSTEM

Refactoring is that the method of improving the inner structure of the code in such how that's doesn't alter the external behaviour of the system [1, 2]. Over the last two decades, many business organizations had noticed that a generous amount of non-trivial legacy software frameworks fail due to unstructured architectural design. Moreover, Research suggests that refactoring is considered a best-method for managing the software system. Indeed, programmers practice regularly with refactoring tools in two different occasions- normal program development phase, whenever and wherever design problems arise. Another is at the time of code duplication, when adding a feature, then the programmer need to remove that duplication using re-factor tool. In addition the key advantages of refactoring are- making software easier to understand, to find defects, improves the design of software, and helps user to program faster. Based on level of automation, refactoring can be categorized into three categories-Fully manual refactoring, Semi-automatic refactoring and automatic refactoring. However, fully manual refactoring and Semi-automatic refactoring tools are underused, because these two refactoring technique sometimes fails to recognize the legacy code and chasing the error messages that leads to more error-prone.

Once there is a necessity to migrate the legacy system into the state of the art web services (Service Oriented Architecture (SOA)) which is the driving factor of migration and web services. We have to make sure that the actual functionality of the legacy system is intact, which we call it as certification. This process of certification involves a lot of manual QA (Quality Assurance) effort, leading to increase in cost and time involved in the whole migration and certification process. The usual Quality Assurance process goes through preparation of functional test cases based on the functional requirement documents (FRD) and inputs from subject matter experts (SME). As we are speaking about legacy systems which are developed years back, we cannot expect all the functional requirement documents to be available keeping track of all the change requests made to the legacy system. Also, it is very difficult to find the subject matter experts as they may not be still with the organization. This enforces us to think about automating the whole process of certification without the need for functional requirement documents or knowledge from a subject matter expert.

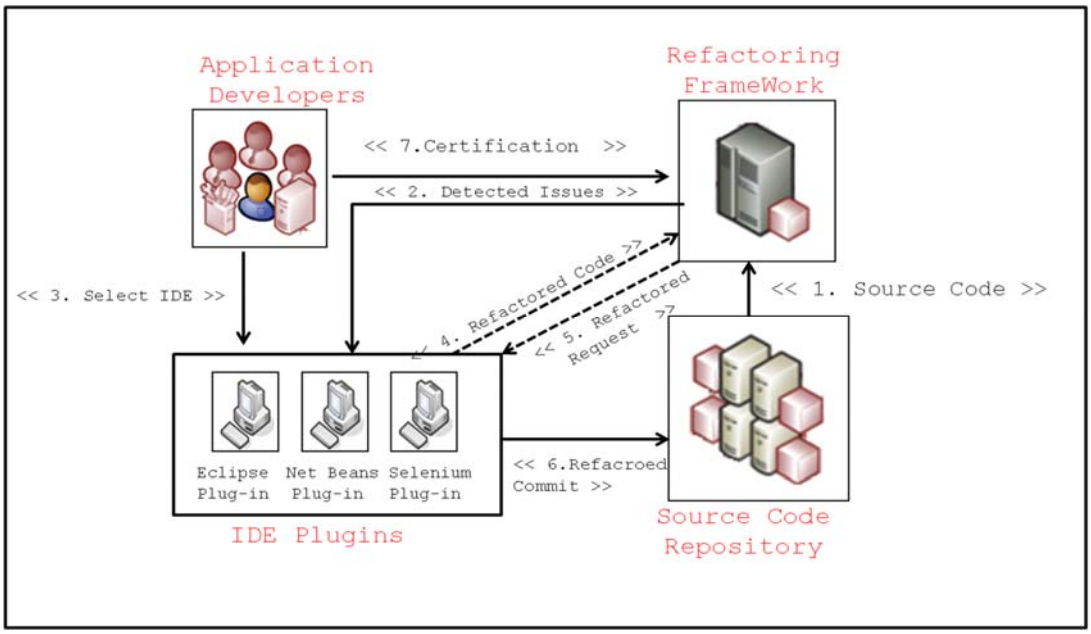


Figure-1. GAFactor refactoring process.

5. GENERATING THE AUTOMATIC REFACTORING (GAFACTOR) TOOL

We designed a novel refactoring tool called GAFactor. This GAFactor system detects a developer's legacy code, reminds to the programmer that the automatic refactoring is available and if the programmer accepts then GAFactor complete the refactoring automatically. GAFactor will overcome the burden of underuse problem that occurs in both manual and semi-automatic refactoring. To use a GAFactor refactoring tool, a developer must recognize that GAFactor tool is available and should select the Network Barrier (switching key) to perform refactor the legacy code. The advantage of using GAFactor refactoring tool over manual refactoring- First, the GAFactor automatically performs static analysis for analyzing the flow of data of the code that saves the programmer from doing error-prone work. Second, The GAFactor is applicable for both application programmers and developers. Third, The GAFactor keeps the 90% of configuration defaults and will not be changed when programmers use the tools until the application programmers press the commit within the tool. This proposed GAFactor Tool uses a component called Switch which helps in toggling between legacy and refactored systems in a convenient and effective manner providing service certification and allowing the client to migrate from legacy to refactored system. The main functionality of this switch component is to migrate from legacy systems to services and provides backward compatibility. Moreover, if the developer wants to invoke or to undo the entire legacy code that has already made then he can use same switching as depicted in Figure-1. The network barrier(Switch) components has several sub components such as Router, Dashboard, Facade, Messenger, Certification, Metrics, which helps in achieving different functionalities of GAFactor as depicted in Figure-2. Second sub-component is facade which provides backward compatibility for legacy applications, whenever there is a need to refactor the legacy systems to services, there also definite need to provide backward compatibility for existing legacy applications.

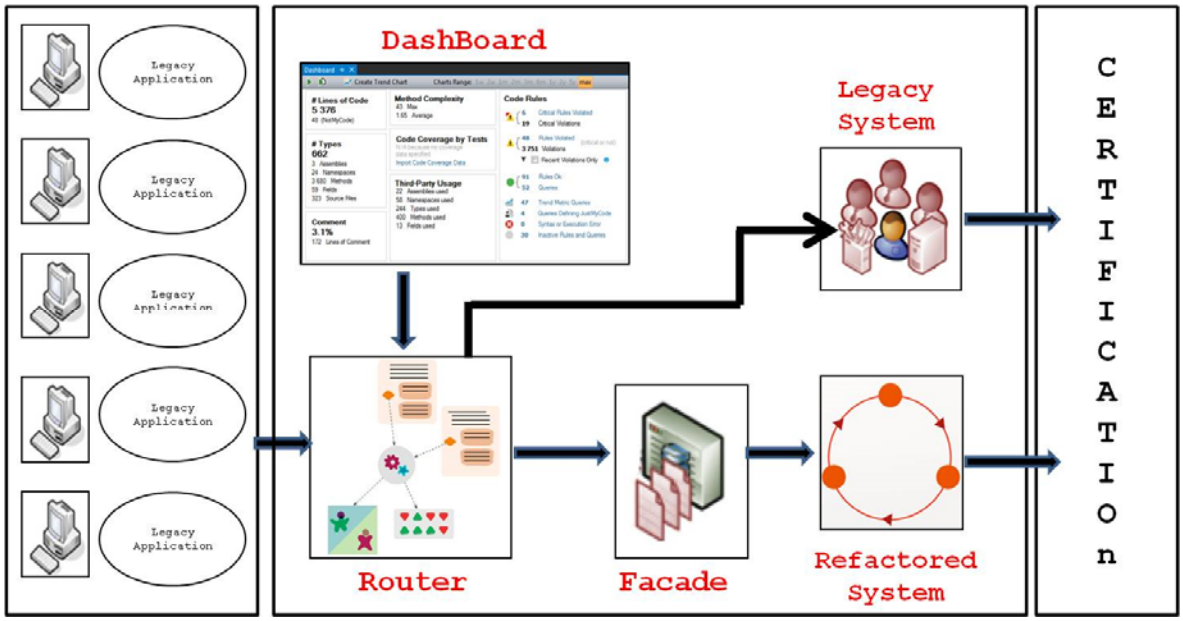


Figure-2. Verifying a refactoring mechanism.

6. ARCHITECTURE FOR CERTIFYING REFACTORED LEGACY SYSTEM:

This section describes architecture for certifying refactored legacy system that shows extended work of previous result for generating the correct code which we had implemented the refactoring (GAFactor) described in the previous section. The following sections elaborate on each step of the certifying refactored process which includes various components Certification Router, Certification Engine, Certification Database, Rules Engine, Certification Dashboard as shown in the Figure-3 Let us discuss about these various components in detail.

Certification Router: Router component acts as a façade between the consumer applications and the actual legacy system. This component clones the actual request coming from the consumer application and routes the actual request to the legacy system and the cloned request to the refactored system. It also stores the actual request in the certification database which later will be used to perform debugging.

Certification Engine: Once the request is sent to both legacy and refactored system, they will hit the backend service and gets the response from the backend service and clones the response computed by each of the legacy and refactored system and stores it in the certification database. Then the legacy system will send the actual response to the consumer application and refactored system does nothing. Once the legacy system response and the refactored system response is stored in the certification database, the certification engine will pick them up, tie them to the actual request and compares the responses. In the process of comparison the certification engine talks to the rules engine to check any particular rules are defined for the service (We will talk about the rules sometime later) and based on that will do the comparison and inserts the results into certification database.

Certification Database: This will hold all the output data of the certification engine, certification router and the certification database. The sample Entity Relation Diagram is illustrated in the Figure-4.Certification_Route_Master contains all the information about the routing details about legacy and refactored systems. Caertification_results_master will have all the certification results and also references request and response masters of the legacy and the refactored applications.

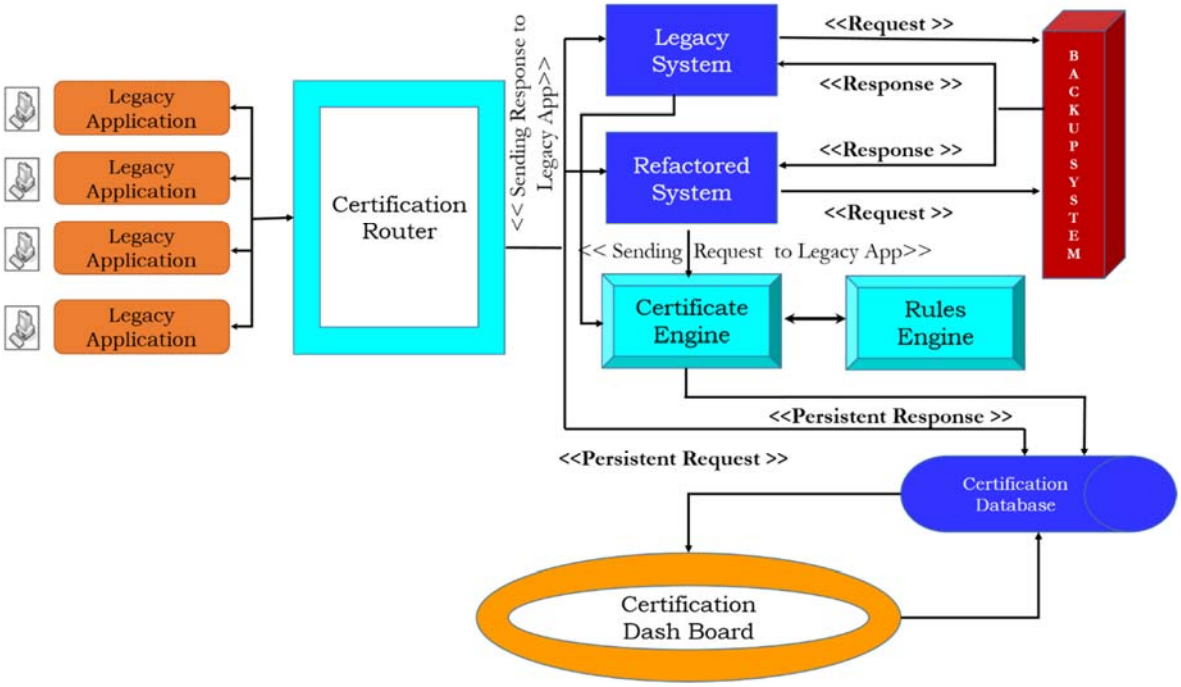


Figure-3. Architecture for automated certification mechanism.

Rules engine**:** Rules engine consists of all the possible deviations of refactored system response from the legacy system response. The possibilities include approved design deviations during the migration phase, known bugs from the legacy application, change requests implemented in the refactored system, improved functionality of the refactored system.

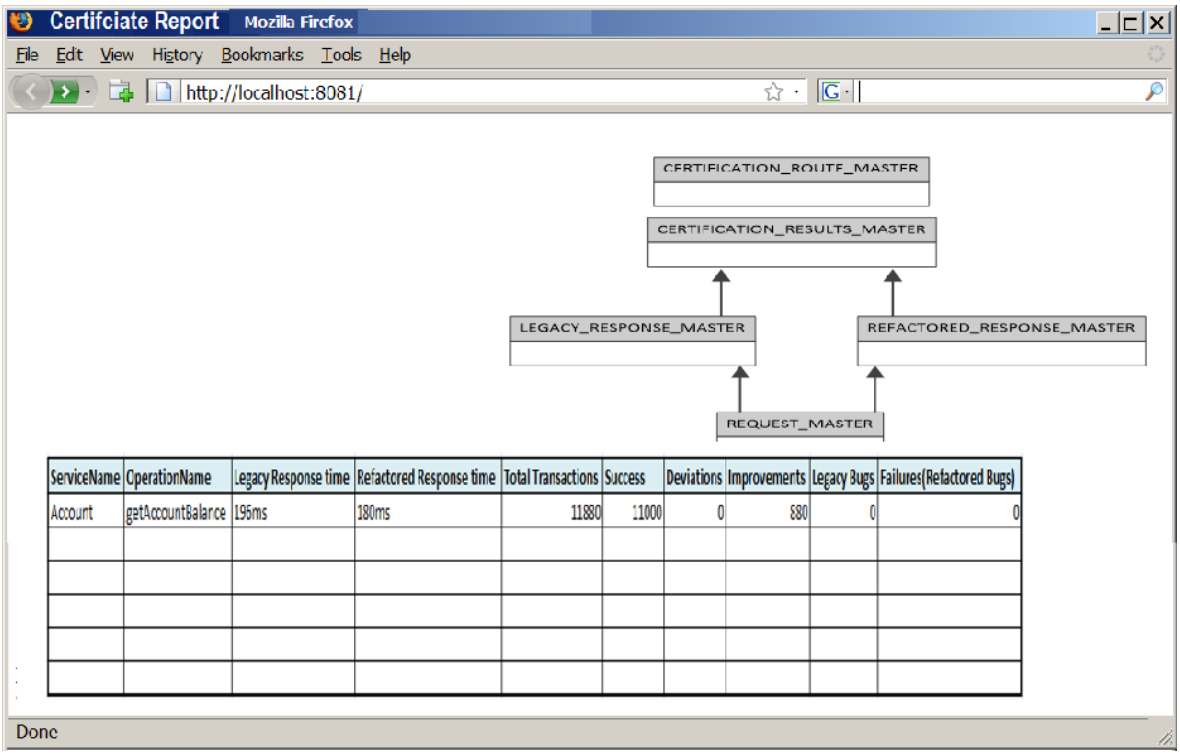
Approved design deviations are the design and architectural changes that were made as part of the migration process. It includes renaming the services, change in the error handling framework, change in the data model of the application to make it in line with the data model of the organization, change in the way we make backend service calls.

Known bugs from the legacy system are the bugs or issues or defects that are identified as part of the independent validation and versification process that will be conducted during any migration phase of the application. Verification and Validation are critical components of a quality management system and are independent procedures that are used together for checking that a product, service, or system meets requirements and specifications and that it fulfils its intended purpose. The words "verification" and "validation" are sometimes preceded with "Independent" (or IV&V), indicating that the verification and validation is to be performed by a disinterested third party. It is sometimes said that validation can be expressed by the query "Are you building the right thing?" and verification by "Are you building it right?" In practice, the usage of these terms varies. Sometimes they are even used interchangeably. IV and V allows one to observe and record any errors or inconsistencies in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways. Most bugs arise from mistakes and errors made by people in either a program's source code or its design, or in frameworks.

When the migration plan is on for the legacy applications all the change requests that are planned for the system functionality will not be made to the legacy system, in turn they are made to the refactored system. These changes requests are to be tracked and added to the rules engine to let know the certification engine about the change request planned for the refactored system.

Improved functionality of the refactored system includes the changes made to the system keeping in mind of the performance issues and other design and implementation challenges encountered in the legacy system. These changes are to be added to the rules engine to let know the certification engine about the improvements planned for the refactored system.

Certification dashboard: Certification Dashboard enables the user to control the level of certification and to view the certification results on the graphical user interface. It also enables the user to get the reports based on services and peak hour - non peak hour certifications. Figure-4 shows sample snapshot of the service certification report. It also allows the user to enable or disable certification by providing a mapping between legacy and refactored applications.



ServiceName	CperationName	Legacy Response time	Refactcred Response time	Total Transactions	Success	Deviations	Improvements	Legacy Bugs	Failures(Refactored Bugs)
Account	gatAccountBalance	195ms	180ms	11880	11000	0	880	0	0

Figure-1. Certification report sample.

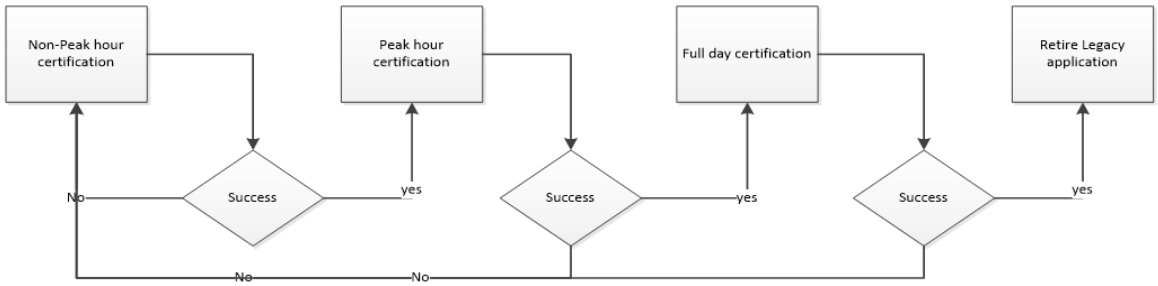


Figure-5. Various phases in enabling certification.

7. ENABLING CERTIFICATION

Enabling certification is definitely a costly process in terms of increasing the load on the productions servers. So, it is very important to control the certification process to make it efficient in terms of quality as well as performance. Various modes of certification include peak hour certification, non-peak hour certification, full day certification, legacy retirement. The first step would be enabling non-peak hour certification. If there are no bugs in non-peak hour certification then we should enable peak hour certification. Once this phase is through we should enable full day certification and eventually retire the legacy application. Please refer Figure-5.

8. SCENARIO OF RESULTS AND CASE

Let us consider Internet Banking application that is migrated to the web services. It may have a whole lot of services. Let us consider three of noteworthy services are mentioned.

a) **Register payee:** This service is to register new payee for fund transfer by a customer

b) **Link other accounts:** Service is to facilitate the link to other account to exiting user id.

c) **Get transaction details:** To provide details of transactions made by the customer to customer executive who tries to read the transaction details.

The response of the legacy and refactored systems can be compared which results success, Failure, bug, timeout, deviation, improvement or change request. The possible types of results are summarized in the Table-1. The issue tracker for various results is shown in Figure-6.

Table-1. Types of possible results.

Type	Description
Success	Both the responses matched and there are no rules defined
Failure	One of the Legacy or the Refactored system is down
Bug	Response did not match even after applying the rules defined
Timeout	One of the Legacy or the Refactored system received timeout from the backend service
Deviation	Responses matched after applying the design deviations defined in rules engine
Improvement	Response matched after masking the fields that are marked as improvement in the rules engine
Change Request	Response will not match as there will be more fields added to the refactored response as part of change request. This has to be certified manually

ServiceName	OperationName	Scenario	Legacy Response	Refactored Response	Type
Account	openAccount	When customer is trying to open ann account	.....	.....	Improvement
..	..	..	..	..	..

Figure-6. Issue tracker.

The different bugs/inconsistencies /deviations are given below.

Improvement: When the customer is trying to add a new payee for funds transfer/to pay bills, and submitting information about payee, there is a constraint nick name of the payee must not exceed 10 characters. If user inputs more than 10 characters and submits the request for adding payee. The responses of legacy as well as refactored system are recorded. Hence there is improvement in system response. See the table below for more details

Deviation: When the customer links his other accounts to existing user Id a message "Account Linked Successfully" is shown. But the data model is changed between the legacy and refactored system. Hence there is a deviation in the refactored system. See the table below for more details

Bug: When customer executive tries to read transactions made by customer with a transaction number that is not updated in the backend system the legacy system fails. This is categorized as Legacy Bug. This will be fixed in the refactored system to show proper message. See the table below for more details. The rules list is summarized in Figure-7.

ServiceName	OperationName	Scenario	Legacy Response	Refactored Response	Type
Account	registerPayee	When customer is trying to register a payee. The payee nick name should not be more than 10 characters but he is trying to enter 15 characters	<Error> <errorID>1002</errorID> <errorMessage>An unkown error occurred</errorMessage> </Error>	<Error> <errorID>1002</errorID> <errorMessage>Payee nick Name should not be more than 10 characters</errorMessage> </Error>	Improvement
Customer	linkAccount	when customer is trying to link other account to his existing account	<linkAccountReply> <message>Account linked succesfully</message </linkAccountReply>	<linkAccountResponse> <message>Account linked succesfully</message </linkAccountResponse>	Deviation
Event	getEventHistory	When a customercare executive tries to read the details of the customer transaction with a transaction number that is not updated	Null Pointer Exception	<getEventHistoryResponse> <errorID>506</errorID> <errorMessage>TransactionID not updated. Please contact CRM Immedietly</errorMessage> </getEventHistoryResponse>	Legacy Bug

Figure-7. Internet banking application.

9. CONCLUSIONS

This paper addresses the complete automated certification mechanism that certifies all the functional components of a service or application and various process involved during the certification phase. This mechanism mainly saves the cost and time compared to the manual

Quality Assurance approach. Moreover, this mechanism does not need any human intervention, will provide a 100% bug free certification which results in higher profits to the organization.

REFERENCES

Srinivas Malladi, *et al*. 2016. GATALSS: A Generic Automated Tool for Analysing the Legacy Software Systems. Research Journal of Applied Sciences, Engineering and Technology. 12(3): 361-365.

Srinivas M., *et al*. 2016. Analysis of Legacy System in Software Application Development: A Comparative Survey. International Journal of Electrical and Computer Engineering (IJECE) 6(1).

Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In ICSM '02: Proceedings of the International Conference on Software Maintenance, pp. 576-585, Washington, DC, USA. IEEE Computer Society.

Du Bois B. 2006. A Study of Quality Improvements by Refactoring. Ph.D. thesis.

Du Bois B., Gorp P.V., Amsel A., Eetvelde N.V., Stenten H., Demeyer S. 2004. A discussion of refactoring in research and practice. Tech. rep.

Fontana F.A., Spinelli S. 2011. Impact of refactoring on quality code evaluation. In: Proceedings of the 4th Workshop on Refactoring Tools. pp. 37-40. WRT '11, ACM.

Konstantinos Stroggyloset al. 2007.Refactoring: does it improve software quality? In Proceedings of the 5th Workshop on Software Quality (WoSQ '07), colocated with the 29th International Conference on Software Engineering (ICSE '07).

M. M. Lehman. 1996. Laws of software evolution revisited. In 5th European Workshop on Software Process Technology (EWSPT'96), volume 1149/1996 of LNCS, pp. 108-124. Springer.

Lorin Hochstein and Mikael Lindvall. 2005. Combating architectural degeneration: a survey. Inf. Softw. Technol. 47:643-656.

Nicolas Anquetil, Simon Denier, Stéphane Ducasse, Jannik Laval, Damien Pollet, Roland Ducournau, Rodolphe Giroudeau, Marianne Huchard, JeanClaude König and Abdelhak Djamel Seriai. 2010. Software (re)modularization: Fight against the structure erosion and migration preparation.

M. Leppanen, S. Makinen, S. Lahtinen, O. Sievi-Korte, A.-P. Tuovinen and T. Mannisto. 2015. Refactoring-a shot in the dark? Software, IEEE. 32(6):62-70.

T. Sharma, G. Suryanarayana, and G. Samarthyam. 2015. Challenges to and solutions for refactoring adoption: An industrial perspective. Software, IEEE. 32(6):44-51.

Gustavo Soares. 2012. Automated behavioral testing of refactoring engines. In Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12: 105-106, New York, NY, USA. ACM.

J. Brant and F. Steimann. 2015. Refactoring tools are trustworthy enough and trust must be earned. Software, IEEE. 32(6):80-83.

A. Garrido and J. Meseguer. 2006. Formal Specification and Verification of Java Refactorings. Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06). 00: 165-174.

A. Junior, L. Silva, and M. Corn´elio. 2007. Using CafeOBJ to Mechanise Refactoring Proofs and Application. Electronic Notes in Theoretical Computer Science. 184:39-61.

K. Okuma and Y. Minamide. 2003. Executing Verified Compiler Specification. Programming Languages and Systems: First Asian Symposium, APLAS 2003, Beijing, China. Proceedings.

X. Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. ACM SIGPLAN Notices. 41(1):42-54.

S. Blazy, Z. Dargaye, and X. Leroy. 2006. Formal Verification of a C Compiler Front-end. Symp. on Formal Methods. pp. 460-475.

Tom Tourw´e and Tom Mens. 2003. Identifying refactoring opportunities using logic Meta programming. European Conference on Software Maintenance and Reengineering, 0:91 100. doi: 10.1109/CSMR.2003. 1192416.

S. Demeyer. 2005. Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls? in Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05. IEEE. pp. 627-630.

Du Bois B. 2004. A Study of Quality Improvements by Refactoring. Ph.D. thesis (2006) Du Bois, B., Gorp, P.V., Amsel, A., Eetvelde, N.V., Stenten, H., Demeyer, S.: A discussion of refactoring in research and practice. Tech. rep.