I.J. Information Technology and Computer Science, **2015, 03, 54-59**
Published Online February 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2015.03.08

Investigating into Automated Test Patterns in Erratic Tests by Considering Complex Objects

Akram Hedayati[1]
Mazandaran University of Science and Technology, Iran
Email: akrm.hedayati@gmail.com

Maryam Ebrahimzadeh[2], **Amir Abbaszadeh Sori**[3]
Mazandaran University of Science and Technology[2], Amirkabir University of Technology[3]
Email: mary.ebrahimzade@gmail.com, Email :a.abbaszadeh.s@aut.ac.ir

Abstract- Software testing is an important activity in software development life cycle. Testing includes running a program on a set of test cases and comparing seen results with expected results. Automated testing encompasses all automation efforts across software testing lifecycle, with focus on automating system testing efforts and integration. Automated testing brings plenty of benefits that speeding up test running time, increasing accuracy of testing process and minimizing costs in different parts of system are three superior features of it. Maintenance and development of test automation tools are not as easy as traditional testing due to unexplored issues which need more examinations. Automated test patterns have been presented to mitigate some problems happening by automated testing and improve efficiency. This paper aims to investigate into automatic testing and automated test patterns. Also, demonstrates behaviour of applying an automated test pattern on a complex object. Results show during choosing an automated pattern to run, we should consider test structure especially level of test object complexity otherwise inconsistency may happen.

Index Term- Test, Erratic Test, Fixture, Fixture Fresh Pattern, Automated Testing, Automated Test Pattern

I. INTRODUCTION

Each software product has its own particular audiences. For instance, a computer game software targets different range of users from users of a banking software. Therefore, an organization which is in charge of writing a software product should evaluate and assure whether or not that software is acceptable for its stakeholders. On the other side, quality of software is becoming dominant success criterion in the software industry [1]. Software testing is a set of striving for such assessments including quality of software and has become an essential part of an agile process[2]. Software testing is very labor intensive and expensive, roughly, half of the cost of a software system development is spent on testing process [3-5]. Software automated testing is popular research problem in the computer application research area and is becoming the most disputed subject in software industry. Two main factors including increased complexity of systems and short product release schedules make task of testing process challenging [6, 7]. To clarify, take into account this fact that there are many systems and projects that are developed in a distributed manner at different places in the world. They are frequently updated and need to be tested at various integration levels. It is not an odd occasion if you figure out there are more than thousands of entities, components, requirements, test cases and subsystems in subject of change and upgrade [8].

In fact, this large amount of numbers is so common phenomenon in new era of software and technology. To make our discussion more specific, in each section, there would be large number of test cases that should be run for each iteration or release. In a perfect test execution situation, every test cannot be executed on a daily basis because of complexity of the systems under test and execution time of the test cases. The complexity in terms of large sizes and internal dependencies of industrial systems are affecting all aspects of software development and test. To overcome these issues and have efficient tests, demand for automated tasting and test management are arising [8]. If the testing process be automated, cost of development dramatically can be reduced. On the other side, developing test data is one of the problematic issues. Test data generation refers to define set of input data which satisfy testing criterion. In this regards, there are some tools for this aim that help programmers and developers to generate test data[9]. Writing and maintaining a good test code needs a lot of efforts[10] [11]. For example ,adding a parameter to a constructor class indicates revisiting and updating all tests that created an instance of that class[11]. Another survey indicate that test code should be treated with the same importance as application code and considers the following features: 1-tests should have a clean design that facilitates code reuse without duplication. 2- tests should be easily adaptable to modifications in the application3- tests should be easy to create. 4- tests should have a low maintenance cost[12]. Cost of updating existing tests along with cost of writing automated tests are two main part of software cost[11].

Examinations show that in comparison with past, complexity and size of software product are continuously increasing [10].One of the main purpose of designing test patterns is reducing test programming. By reducing test programming time, test engineers are given

this opportunity to focus more on other specific aspects of tests rather than spending much time on coding. As a result, efficiency would be improved [11] . Automated testing is intended to reduce test execution effort, time and increase accuracy of validation.

There are many issues regarding automated unit and acceptance tests in agile software development, but in recent 10 years they have attracted more attention in software companies [11]. The larger scale, the more efforts required to assure product quality, even in a case that automated testing tools are used because test patterns need to be input manually. This process requires huge amounts of man-hours specifically when product types vary, have a lot of subsystem or test patterns need to be modified for associated signal changes[10]. An important point is that typical development and test cycles must be considered from various point of view of software quality assurance in order to be aligned with business process testing software. In automated testing, monitoring processes including quality assurance, maintainability, optimization, accuracy, modularity, context, synchronization, documentation should be done [13].

The rest of this paper is arranged as follows: section 2 gives brief information about background of automated testing. Section 3 introduces specific description of major concepts in automated testing. In following, related work is presented in section 4. Section 5 presents our investigation of applying one type of automated test pattern on erratic test and demonstrates how potential problem happens. A summary of conclusion is given in section 6 as well.

II. Background

A. Automated Testing

Bertolino [14] identifies test automation this way: ”far-reaching automation is one of the ways to keep quality analysis and testing in line with the growing quantity and complexity of software”. Karhu et al. [15] note that ”automated software testing may reduce costs and improve quality because of more testing in less time, but it causes new costs in, for example, implementation, maintenance, and training”. They continue by stating that ”automated testing systems consist of hardware and software and suffer from the same issues as any other systems”. In software testing, automation includes development and execution of test scripts in order to measure validity of requirements by using automated testing tools [16].

In some methods of writing automated unit testing, test cases are written after code therefore these unit tests are put under software configuration management together with production code. In test-driven development (TDD), programmers write test cases first then implement code which successfully passes the test case[2]. Automated test reduces the cost of running tests by means of fully automated tests, repeatable tests and robust tests. Automated test significantly avoids happening defects by emphasis on repetitive regression testing. Consequently, reduces costs of removal of added new defects [17].

In comparison with traditional test, automated unit testing has 5 main advantages such as speed, accuracy, precision, efficiency, skill-building which are abbreviated as SPAES. Speed means thousands of test cases can be performed with very high speed. Precision exists in automated test because the result of automated test is the same whenever it is running while in traditional test, results in each running time may have differences. Accuracy is another advantage that says there is no human error in automated testing. Automated test can be run for long time continuously and we call this feature as efficiency[18] .

Automated testing offers numerous benefits to any software organization, including finding defects cost-effectively early in the development cycle, providing rapid feedback,and giving developers the courage to refactor their code[12] .

When tests are supposed to run for several times, it seems automated test is a good option to use but some paradoxes are raising here [16]. One of those paradoxes happens when automated test makes no change in data or running paths. On the other hand, we encounter bug reduction. In this case, it is a crucial issue that this bug reduction does not guarantee the reduction of total number of bugs in considerable system[17]. Improving capability of test depends on error detection and correction, therefor by adding new changes they can be modified and improved. [17]

Automated data are changing in each running time of automated tests. During these tests, developers should measure and evaluate consistency of automated code and structures by appropriate and consistent tools. For this purpose, automated testing tools are designed and used to provide more precise results, enhance product quality, increase testing productivity and speed test execution up [19]. Code analysis tools, test management tools, functional testing tools (also called capture/playback tools), detecting memory leaks tools and generating test data tools are tools which are broadly used for aforementioned goals [18]. For example, automated recorded test includes tools which let make a documentation of all user interaction with tested application and software and store them in file or database for future use [17] . When tests lack organized structures and have low quality, tests are not as efficient as expected. Furthermore, if tests after run phase, cannot not detect any error, it does not mean considerable system is free error, it could be consequence of incomplete and defective tests. Additionally, when system is subject to change, maintenance of automated test data files becomes harder.[16]. Test maintenance is costly in case of playback methods. Even though a minor change occurs in GUI, the test script has to be rerecorded or replaced by a new test script.

One of challenges in testing process is that as applications changes over time, tests become more difficult to maintain. Design patterns help us minimize maintenance costs by making tests easier to update and

more adaptable to application changes [12]. By using principles of development and design of applications, automated test development can become a well factored art form that is adaptable to application changes. This adaptability results in lowered maintenance costs and easier test creation [12]. Maintenance problems can be addressed by using design patterns and treating test code with the same importance as application code[12].

B. Test Automation Manifesto

Test automation manifesto indicates principles that result in automated tests that are easier to write, read, and maintenance [11]. Principles of the manifesto including bunch of questions about basic foundation of tests, integrating new behaviours of test, distinguishing components which can be designed automatic, which kind of automatic test is applicable, etc [11]. The answers of these primary questions are proposed as "test automation manifesto". Based on this manifesto, the most important qualities to consider into automated testing is demonstrated as follows:

Concise: the simplest possible form and yet comprehensive. Self-checking: reporting tests results without human interpretation. Repeatable: tests run many consecutive times with no human intervention. Robust: the test results are independent of changes in external environment. Sufficient: tests verify all the requirements of the software being tested. Necessary: all contributing things to have desirable test behaviours should be included in it. Clear: every statement is easy to understand. Efficient: tests run in a reasonable amount of time. Specific: each test failure refers to a specific piece of malfunction. Independent: each test can be run by itself or in any order with other set of tests. Maintainable: tests are easy to understand, modify, upgrade and extend. Traceable: to and from the code it tests and to and from the requirements.[11]

III. Terminology

In this part, we identify several major concepts in inspecting automated test code together two kind of test in software testing.

A. Fixture

Technically, fixture is an instance of associated testcase class used as pre-conditions of the test. Test runtime environment is central part of system, interacts with other parts in order to manage, schedule and run tests in an appropriate manner. In addition, provides suitable user interfaces (UI) for creating and running tests.

Every test consists of four distinct phases that are run in sequence: fixture setup, exercise SUT, result verification, and fixture teardown[17]. In the first phase, system under test (SUT) and every required thing are created and put into a state which is required to run the SUT. Another term, we set up the test fixture. In exercise SUT phase, the test is run and we interact with the SUT. In third phase, required actions are done in order to see expected results and behaviours are observed or not. In last phase, the test fixture is torn down to put the world back into the state in which has been found it [2, 17].

B. Smell

An automated test is a program that checks another program. Consequently, it is vulnerable to the same design problems as application code. These vulnerabilities, often referred as code smells. Another words, a smell is a symptom of an underlying problem in code. Generally speaking, smells are described as a set of problems in test codes. Developers use them as checklists to analyse test codes [2].

Code smells include test code duplication, tight coupling between the application and tests, and long test methods [12] . There are three common kinds of smells. First, code smell refers to obvious problems while reading or writing test code. Second, behaviour smells are smells we encounter while compiling or running tests and are much harder to ignore. Meanwhile developers automate, maintain and run tests, code and behaviour smells are typically paid attention [17]. a lot of examinations and experiences show that main root causes of behaviour smells are slow tests, erratic tests, fragile tests, assertion roulette, manual intervention [17]. The third kind of smells is projects smell that inspects test smells from project manager or customer point of view because they are indicators of the overall health of a project [17] .

C. Debt

Debt is generally considered as a bad state. Project debt arises when a job is not delivered in due time or is not done as enough. Test debt: most projects are in test debt. They test too little and infrequently also. Moreover, prevents happening errors and build quality in them are not done regularly[17]. Automation debt is another sort of debt which even more projects are in automation debt. Due to little automated testing, they have to spend lots efforts on manual testing methods. Technical test debt is another state which most projects consider automated testing are in this state because their tests need a lot of efforts related maintenance issues [17].

D. Pattern

The next term which is described here is pattern. A "pattern" is a recurring solution to a recurring problem.in another way, they are set of periodic solutions for upcoming problems of automatic tests [17].

E. Slow Test

Slow tests are kind of tests which take long enough to run. This explains why when test developers make a change to the SUT, they don't execute tests every time. Slow test reduces productivity and cause a lot of explicit and implicit cost into system and project [17]. The causes of slow tests could be either in the way we built the SUT or the way tests are coded. Main root cause of slow tests is that many tests are interacting with databases to write in or read from it in order to setup a

fixture or verify results. Running these tests with slow components take about 50 times longer to run than if the same test is supposed to run against in-memory data structures. A possible solution is replacing the slow components with a test double[17] . Another factor which makes slow test is general fixture. Because each time a fresh fixture is built, each test is constructing a large general fixture and it includes many more objects. Consequently, it takes longer than usual to construct. To reduce this time and avoid rebuilding it for each test, we can use the general fixture as a shared fixture but unless we make this shared fixture immutable. On the other hand, this is likely to lead erratic tests and so should be avoided. Another way is reducing the number of fixtures being set up by each test. Also, tests need a long delay to ensure consistency and synchronization between underlying processes or threads of system to lunch, run and verify. As a result, in a big scale, these waiting times in each test, significantly slows down overall execution time .to address this problem, maybe we are obliged to avoid asynchronicity in tests by testing the logic synchronous. Another state which slow test happens is when there so many tests to run regardless of how fast they execute. Also, we may have many overlaps between them. To take over this state, if possible, we can break system into a number of fairly independent subsystems or components along with subset suite of suitable cross-section of tests to run by a logical schedule [17].

F. Erratic Test

Results of some tests depend on some factors such as who is running them or when tests are run. As a result, they provide different outcomes, pass or fail. There are some tests that if run for several times, provide different results and behave erratically. The results are affected by external factors including environment, who is running them or when tests are run. Maybe it sounds logical to remove the failing tests from the test suites but this leads lost test. On the other side, keeping the erratic tests may either interfere with other issues which the same tests are expected to detect and obscure resolving errors or even cause additional failures than expected. [17] there are many ways which cause erratic tests. Hence, they are a little hard to trouble shoot. Interacting tests, unrepeatable test, test run war, nondeterministic test and resource optimism are some sort of test that their performances provide erratic test.

Interaction among tests may cause erratic tests. In this case, if many tests run in sequence and use same objects, even if one test fails, consecutive failures or cascading errors will happen in other tests for no evident reason, because they depend on other tests' side effects. [17] as fig (1) shows in execution of a sequence of tests, test 2 failure may leave processobject1 in state that causes test n to fail.

Unrepeatable test is referred to tests that their results at first run time are different via results of subsequent test runs. They affect each other results to some extent. [17]

The next cause is test run war. In this state, if many test runners use some shared external resource such as a database, object, file, random results may happen and we call it a test run war [17]. As we can see in fig (2), while many parallel test runners are using processobject1, test 1 may fail whereas test 2 pass it successfully at the same time.

Non deterministic tests are tests which are dependent on non-deterministic inputs. These tests pass at some times, but if they run at another time, they face failure. This is due to lack of date and time control [17].

Resource optimism describe tests which their results depend on where or where they are run so we have nondeterministic results. Based on some non-ubiquitous external resources tests either fail or pass[17].

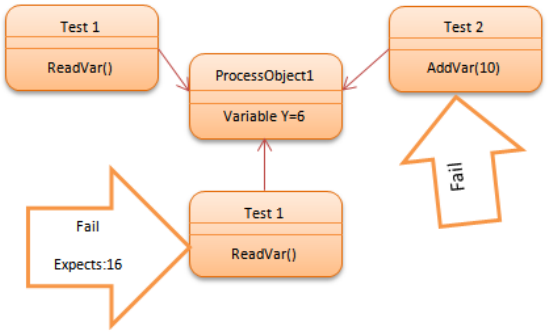

Fig. 1. interacting tests- cascading errors

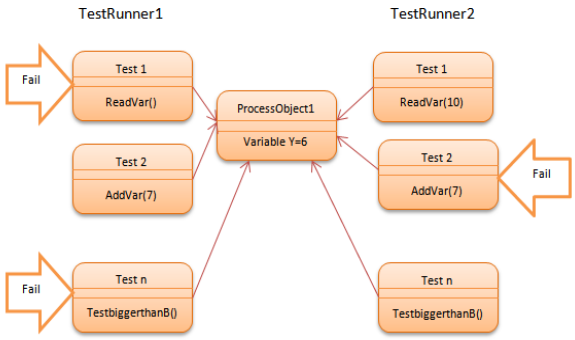

Fig. 2. test run war- run-time behaviour of executing multiple tests at the same time

IV. Related Works

G.meszaros is one of experts in agile software development specifically in automated test patterns and test code designs. He has invented effective mechanisms to facilitate writing and running of tests in terms of test automation framework and automated test patterns in order to achieve automated testing benefits[17] [11] . Shared test fixture pattern and fresh fixture pattern are two type of automated patterns which are used for solving stated problems of slow tests and erratic tests respectively [17] . In following, we describe these patterns.

A. Shared Test Fixture

To run an automated test, we need a text fixture that is completely deterministic and well understood. Setting up a fresh fixture as explained before, may take long time

than usual especially when we are dealing with complex system state stored in a test database.

In this pattern, a standard fixture is created, a shared fixture is a fixture which is created by one test and is reused by other tests simultaneously. Even, it can be either a prebuilt fixture that is reused by one or several tests in many test runs. Additionally, that fixture lasts longer than a single testcase object. By employing this way, many tests reuse the same standard test fixture between themselves more and more without tearing down and recreating it. Shared test fixture pattern by reducing fixture setup overhead helps to improve test run times in slow tests [17] . On other hand, it is obvious that if results of test depend on outcomes of other tests, shared test fixture bring about interaction among tests that may cause erratic tests. Also, we should consider shared fixture in order to be applicable for all tests, is bound to be more complicated than the minimal fixture needed for a single test. Greater complexity causes another type of text named fragile fixture. Other disadvantages of this pattern is elaborated in [17].

B. Fixture Fresh Pattern

Fixture fresh pattern is used for avoiding erratic tests. Every test needs a test fixture. Fixture defines state of test runtime environment before running time. Making decision upon using prebuilt fixture or creating a new one is one of key test automation decisions. In this approach, only a single run of a test will use fixture and it will be torn down after finishing. Hence, tests are completely independent. Fixtures which are left over by other test runs are not used by other tests. In [17] it is mentioned that whenever we want to avoid interdependencies among tests, it is the right time to use fixture fresh pattern but in next section, we want to describe a condition that it is impossible to take advantage of the pattern on it.

V. Research Methodology

In previous sections, we went through different kind of tests such as slow tests and erratic tests and elaborated how automated test strategy patterns mitigate the effect of those tests. Furthermore, previous studies demonstrate that whenever we want to avoid interdependencies between tests, we are allowed to use fresh fixture pattern and no other limited condition mentioned. In this part, we want to add a significant criterion while making decision upon using fresh fixture pattern for tests. Regarding our finding, if we cannot modify slow and complex tests in erratic tests by using simple objects or smaller codes, applying fresh fixture pattern may result in interacting tests, test run war or other issues we encountered in erratic tests. Consequently, tests fail. Due to these observations, we claim that considering structure of test code is a determining factor in choosing pattern.

Here, we want to establish validity of our claim by Reductio Ad Absurdum proof method in mathematical logic. This proof is represented as set of true statements or premises which are built upon axioms or theories, we notate premises as s, along with preposition we want to prove, p.

If $s \cup \{p\} \vdash f$ then $s \vdash \neg p$ (1)

If $s \cup \{\neg p\} \vdash f$ then $s \vdash p$ (2)

Based on notations number 1 or 2 , we bring into account p, or the negation of p, with s. By further examination, if above predicate result in logical contradiction f, then we can conclude that the statements in s lead negation of p, or p itself, respectively.

According to what was said, our statement says our erratic test has one or more complex objects, and we want to prove fresh fixture pattern for this test is not possible. For proof, the claim is negated to assume fresh fixture pattern is acceptable for this object. As we described earlier, in fresh fixture pattern each test creates its fixture and tears it down after single run and fixture is rebuilt for different times. Since each test fixture is initialized in each runtime, in turn it implies process of fixture setup is not time-consuming. If fixture setup process be time-consuming result in tests take a long time to do. As a result, it contradicts one of the main benefits of automated tests, reducing runtime and increasing speed, as we introduced before. As a result, when we suppose fixture setup process does not take long time, it indicates that the same test including the fixture is not slow. This means that test does not include any complex databases or codes while this statement results in contradiction to first proposition. The contradiction means that it is impossible for an object to be both complex and simple at the same time. Consequently, it follows that the assumption fresh fixture pattern is usable for complex tests must be false and hence proving the claim.

An example of a complex object is shown in fig (3).as seen, a number of tests are supposed to run. Nested loops and databases through object make it complex. While test 1 is inserting a variable into database, test 2 is updating the same database. Both want to access same resource simultaneously thus this coincidence makes test 2 failure. According to fixture fresh pattern, each test uses a single fixture for single runtime. If each test has fixture setup time and its process takes longer than expected, this significantly slows automated testing process down and brings further potential problems. Another word, speed falls and time ups.

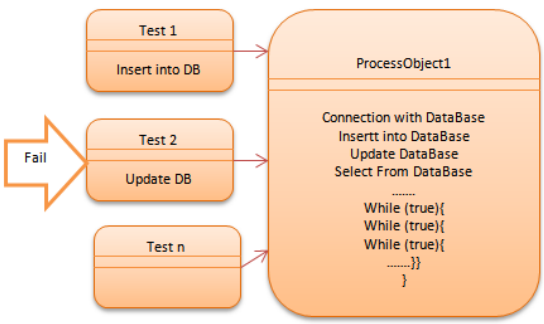

Fig. 3. an example of complex object using fixture fresh pattern

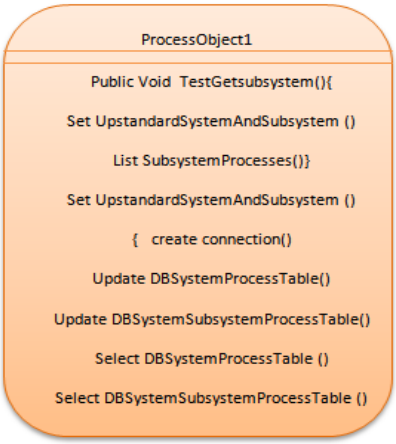

Fig. 4. pseudo code of a complex object

We have shown above explanation by piece of pseudo code as shown in fig (4). Suppose plenty of tests are scheduled to run. Also have interactions among each other and their results affect other test results and since fixture should be built for any test, each fixture setup needs a lot of time. This condition negates reducing time in automated testing and states there is no possibility to run fixture fresh pattern in test complex objects.

VI. Conclusions

When we introduce automation, we create more software that must be coded, debugged, and maintained. In spite of benefits of automated testing, more time is added to test schedule for automation activities that if carefully not managed, we will encounter a negative return in automation investment. Automated test patterns has been developed as solutions to mitigate potential caused problems and reduce costs of implementation. Based on our examination, we conclude each automated pattern is not applicable for any kind of test. Making decision on using an appropriate automated pattern needs an exhaustive investigation that embrace all properties of test such as level of test complexity, contributed objects, number of running time of each single test, test interactions.

References

[1] L. J. Osterweil, *et al.*, "Strategic directions in software quality," presented at the ACM Computing Surveys, 1996.
[2] F. Lanubile and T. Mallardo, "Inspecting Automated Test Code:A Preliminary Study," *Springer*, pp. 115-122, 2007.
[3] D. Alberts, "The economics of software quality assurance," presented at the AFIPS : AFIPS Joint Computer Conferences, 1976.
[4] G. J. Myers, *Art of Software Testing*: Wiley 1979.
[5] M. J. Harrold, "Testing: A roadmap," presented at the International Conference on Software Engineering, 2000.
[6] S. Jinhui, *et al.*, "Research progress in software testing," presented at the Acta Scientiarum Naturalium Universitatis Pekinensis, 2005.
[7] G. Todd, *et al.*, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, pp. 184-208, 2001.
[8] S. Eldh, *et al.*, "Towards Fully Automated Test Management for Large Complex Systems," presented at the International Conference on Software Testing, Verification and Validation, 2010.
[9] B. korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering* vol. 16, pp. 870-879 1990.
[10] T. Kataoka, *et al.*, "Test Automation Support Tool for Automobile Software," *SEI TECHNICAL REVIEW*, pp. 79-83, OCT 2013.
[11] G. Meszaros, *et al.*, "The Test Automation Manifesto," in *Extreme Programming and Agile Methods, Xp/Agile Universe 2003*. vol. 2753, ed: Springer, 2003, pp. 73-81.
[12] M. Rybalov, "Design Patterns for Customer Testing " presented at the Pacific Northwest Software Quality Conference 2004.
[13] L. G. Hayes, *The Automated Testing Handbook*: Software Testing Institute, 2004.
[14] A. Bertolino, "Software Testing Research: Achievements, Challenges,Dreams," presented at the IEEE Computer Society, Washington, DC, USA, 2007.
[15] K. Karhu, *et al.*, "Empirical Observations on Software Testing Automation," presented at the International Conference on Software Testing Verification and Validation, Denver, CO, 2009.
[16] S. A. Adnan, "Continuous Integration and Test Automation for Symbian Device Software," Bachelor, Helsinki Metropolia University of Applied Sciences, Helsinki 2010.
[17] G. Meszaros, *XUnit test patterns : refactoring test code*, 2007.
[18] J. D. McCaffrey, *.NET Test Automation Recipes: A Problem-Solution Approach*. NY, 2006.
[19] E. Dustin, *Effective software testing : 50 specific ways to improve your testing*: Addison-Wesley Longman Publishing, 2002.

Authors' Profiles

Akram Hedayati: Born in 1988, she holds a Bachelor's degree in Computer engineering from Ferdowsi University of Mashhad, Iran, in 2010. She also obtained a Master's degree in Information Technology Engineering majoring in Management Information Systems at Mazandaran University of Science and Technology, Iran, in 2013. Her research interests include Information Systems, Enterprise Architecture and Software Engineering.

Maryam Ebrahimzadeh: She received her M.Sc. degree of Information Technology Engineering at Mazandaran University of Science and Technology, Iran, in 2013. She received her B.Sc. degree in Computer Engineering from the same university. Her research focus is on Information Systems.

Amir Abbaszadehsouri: Born in 1990, Mazandaran, Iran. He received Bachelor's degree in Software Engineering from Technical and Vocational University, Tehran, Iran, in 2012 and Master's degree in Computer Networks from Amirkabir University of Technology (Tehran Polytechnic). His current research interests include Computer Networks, Multimedia Software and Future Tech.

