I.J. Information Technology and Computer Science, **2016, 12, 1-9**
Published Online December 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2016.12.01

Automation in Software Source Code Development

Henryk Krawczyk and Dawid Zima
Department of Computer Systems Architecture, Gdansk University of Technology, Poland,
E-mail: henryk.krawczyk@eti.pg.gda.pl, dawzima@student.pg.gda.pl

Abstract—The Continuous Integration idea lays beneath almost all automation aspects of the software development process. On top of that idea are built other practices extending it: Continuous Delivery and Continuous Deployment, automating even more aspects of software development. The purpose of this paper is to describe those practices, including debug process automation, to emphasize the importance of automating not only unit tests, and to provide an example of complex automation of the web application development process.

Index Terms—Continuous integration, continuous delivery, continuous deployment, test automation, debug automation.

I. Introduction

According to the Forrester report [1] companies are looking to prioritize innovation through developing software services, but Software development providers can't deliver new services at the rate business leaders want (business leaders want software delivered in six months). Also, according to the report, corporate culture and development process immaturity impedes communication, and slows service delivery, and only a few IT organizations regularly perform advanced continuous delivery practices.

Automation in terms of software development exists in almost all stages of the software development life cycle, independently from the chosen development method. Some of them emphasize automation – like Test-Driven development and many Agile methodologies. Deeper description of different software development models is not in the scope of this paper, and can be found in other publications [2]. The Continuous Integration set of practices are the core of the automation in the software development process. On top of that idea are built other practices extending it: Continuous Delivery and Continuous Deployment. These ideas are the answer to the rapid demand of business for new services – they speed up the process of when the developed application will be deployed to the users. However, not all aspects of software development are subject to automation.

In terms of the software development process (activities strictly related to development) there can be distinguished 3 roles: (1) Development, (2) Validation and (3) Debug. Development is an actual programming effort which ends when a piece of code is committed to the source code repository (whether it is a new feature, bug fix or new test covering some functionality of the software). Validation is responsible for executing and interpreting tests and their results. Debug means analyzing failed tests and error reports to find out root causes in the developed software source code. In small projects all three roles are related to each developer, but in larger projects there may be entire teams specialized in each role.

The purpose of this paper is to describe key practices that are the core of the automation in software development: Continuous Integration, Continuous Delivery and Continuous Deployment emphasizing test and debug process automation, including a simplified example of an automated acceptance test of a web-based application.

The rest of the paper is structured as follows: Section 3 contains detailed description of Continuous Integration practices. Section 4 describes practices extending Continuous Integration, Continuous Delivery and Deployment. Section 5 provides introduction to tests automation with a detailed example of Continuous Integration process with acceptance test automation for web application. Section 6 introduces the debug automation process extending the previous example. Section 7 concludes this publication.

II. Related Work

There are many publications describing and comparing different software development models. In authors other paper [2] they've been discussed including traditional, agile and open source methodologies. Z. Mushtaq et al. [15] proposed hybrid model combining two agile methodologies – Scrum and eXtreme Programming (of which many practices were the origin of the Continuous Integration).

Continuous Integration, Delivery and Deployment has been well described and discussed by M. Fowler [3] [5], J. Humble [6] [7] and D. Farley [6]. Forrester Consulting [1] prepared Continuous Delivery Maturity Assessment Model based on the results of the survey they conducted. A. Miller from Microsoft Corporation [4] has analyzed and presented data from their Continuous Integration process for a period of approximately 100 working days.

Debug automation has not been the subject of many researches, most of them were well described and published by Microsoft researchers [14] [12]. They've shared theirs' experience from more than 10 years of implementing, using and improving overall process of debug automation.

III. Continuous Integration

The Continuous Integration (CI) is a set of practices, known for a long time, but formally introduced as part of the eXtreme Programming methodology, and then well described by Martin Fowler [3]. He has distinguished the 11 most important practices of CI: (1) "Maintain a Single Source Repository", (2) "Automate the Build", (3) "Make Your Build Self-Testing", (4) "Everyone Commits To the Mainline Every Day", (5) "Every Commit Should Build the Mainline on an Integration Machine", (6) "Fix Broken Builds Immediately", (7) "Keep the Build Fast", (8) "Test in a Clone of the Production Environment", (9) "Make it Easy for Anyone to Get the Latest Executable", (10) "Everyone can see what's happening" and (11) "Automate Deployment". It can be concluded in one sentence: the CI set of practices provides rapid feedback about committed change quality and helps to avoid integration problems.

The first CI practice, "Maintain a Single Source Repository" (1), means that there should be a single, centralized, application source-code source behind a version management system (application that allows developers to work in parallel on the same files, allowing them to share and track changes, resolve conflicts etc., i.e.: SVN, GIT, Perforce) that is known to anyone who is involved in the project. There should be distinguished a mainline among other branches, that contains the most up-to-date sources of the project that developers are currently working on. All developers working on the project should commit their changes to the repository. Everything that is needed to build the project, including test scripts, database schemas, third party libraries etc. should be checked-in to that repository. As Martin Fowler says [3], the basic rule of thumb is that anyone should be able to build a project on a virgin machine (fresh setup) having access only to the centralized source code repository. Practice shows that this is not always possible, and sometimes some environment setup on new developer machines are required (i.e. installation of Windows Driver Kit).

"Automate the Build" (2) might be considered as the crucial practice in CI. It means that the process of converting source code into a running system should be simple, straightforward and fully automated (including any environment changes, like database schemas etc.). This is the first step that indicates the quality of change checked-in to the source code repository – if the build was compiling before, and failed to compile after introducing the change, the developer that made the commit should fix the compilation as soon as possible. There are many existing solutions like: GNU Make, Apache Ant for Java projects, NAnt for .Net or MSBuild that allow automation of the build process.

By making the build self-testing (3), there should be low-level tests (i.e. Unit Tests) included in the project, covering most of the codebase, that can be easily triggered and executed, and the result is clear and understandable. If any of the tests failed (a single test case or an entire test suite) the build should be considered as failed. It is important to remember that testing will not tell us that software works under all conditions (does not prove absence of bugs), but will tell us that under certain conditions it will not work. Execution of low-level tests after each check-in allows you to quickly check if the change introduced a regression to the project.

When many developers are working on the same project, developing different components (in isolation) that interact with each other based on the prepared contract (interface) and do not integrate their changes frequently but rather rarely (i.e. once every few weeks), they can experience something called "integration hell" – conflicts, redundant work, misunderstandings on the stage when different components are integrated after being developed in isolation. To resolve these issues, developers should commit to the mainline very often (i.e. every day) (4), literally continuously integrating their changes. This practice allows one to quickly find any conflicts between developers.

Before committing their changes to the repository, developers should: get the latest source code from the repository, resolve any conflicts, and perform build and low-level tests on their development machine. If all steps were successful, then they are allowed to commit their change to the repository, however this is not the end of their work. A dedicated machine (integration machine) detects changes in the source code repository and performs the build (5). Only when the build is successfully completed on the integration machine, can the build be considered as successful, and the developer's work is done.

It is important to maintain the codebase in a healthy state – each compilation break, unit test failure or static source code analyzer error should be fixed as soon as possible by the developers who have broken the build (6). Sometimes, to get quickly back mainline to the successful state, the best way is to revert latest commits to the last known good build.

"Keep the Build Fast" (7) – to be able to provide rapid feedback about committed change quality, build process time should be relatively short. eXtreme Programming methodology tells us that the build should last no longer than 10 minutes.

All tests should be performed in the environment maximally similar to the production environment (8). This means, i.e. using database servers with the same version as on the production, web browsers the same as used by clients etc. Every difference between the test and production environment introduces the risk that developed software will behave differently when deployed to the production environment.

Martin Fowler [3] also pays special attention to the availability of the project executables. They should be easily accessible to anyone who needs them (9) for any purposes – manual tests, demonstrations etc.

According to the exact words of Martin Fowler [3] CI is all about communication, so it is important that everyone involved in the project can see what is happening (10) – what is the current state of the mainline, what is the latest successful build etc. Modern tools supporting the CI process (often called CI servers) provide web based GUI that allows you to display all necessary information.

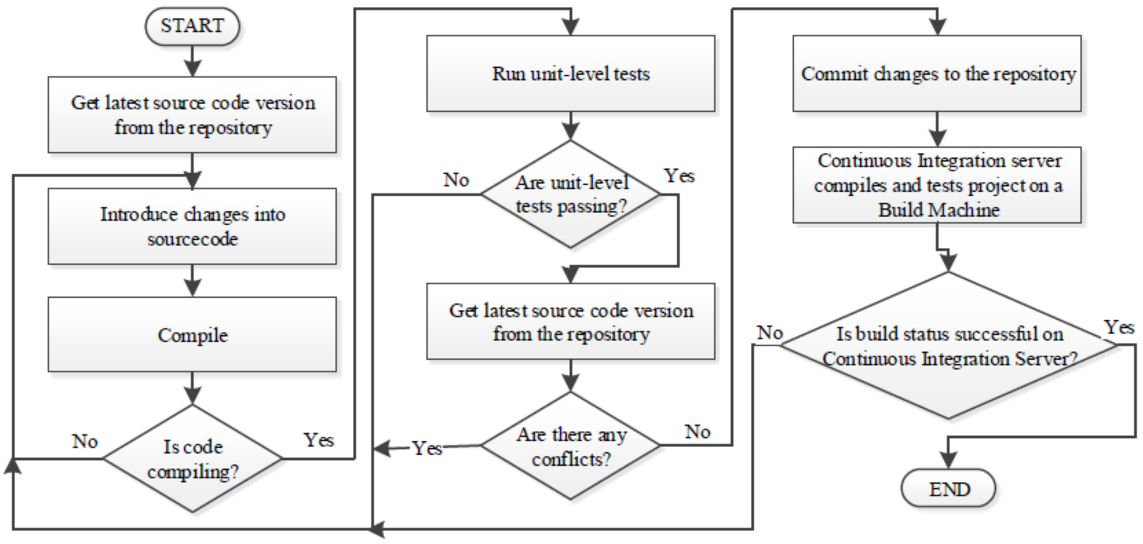
START
Get latest source code version from the repository
Introduce changes into sourcecode
Compile
Is code compiling?
No
Yes
Run unit-level tests
Are unit-level tests passing?
No
Yes
Get latest source code version from the repository
Are there any conflicts?
Yes
No
Commit changes to the repository
Continuous Integration server compiles and tests project on a Build Machine
Is build status successful on Continuous Integration Server?
No
Yes
END

Fig.1. Diagram of the Continuous Integration Process

To perform higher level tests (integration, performance, acceptance etc.) there is a need to deploy the project to the test environment (as previously mentioned, it should be similar to the production environment). So there is a need to automate the deployment process (11). When the deployment automation is used to deploy the project to the production, it is worth to have also an automated rollback mechanism that allows you to revert the application to the last good state in case of any failures. Deployment automation will be elaborated more during the discussion of the Continuous Delivery, Continuous Deployment and the Deployment Pipeline in this paper.

The CI server allows the practical implementation of the CI process. Its main responsibility is to monitor the source code repository, perform build, deploy and test when a change is detected, store build artifacts (i.e. project executables) and communicate the result to project participants. There are many existing commercial and open source CI servers available on the market, offering many collaboration features. The most popular are: TeamCity (JetBrains), Jenkins, Team Foundation Server (Microsoft), Bamboo (Atlassian).

Ade Miller from Microsoft Corporation [4] analyzed data from their CI process. Data was collected for the "Web Service Software Factory: Modeling Edition" project that was released in November of 2007, for a period of approximately 100 working days (4000 hours of development). Developers checked in changes to the repository on average once each day, and the CI server was responsible to compile the project, run unit tests and static code analysis (FxCop and NDepend) and compilation of MSI installers.

During that 100 days, developers committed 551 changes resulting in 515 builds and 69 build breaks (13% of committed changes). According to his analysis, causes of build breaks were: Static code analysis (40%), Unit Tests (28%), Compilation errors (26%), Server issues (6%). The great majority of build breaks were fixed in time less than an hour (average time to fix a CI issue was 42 minutes). There were only 6 breaks that lasted over the night.

He has also calculated the CI process overhead, which in that case was 267 hours (50 for server setup and maintenance, 165 for checking-in, and 52 for fixing build breaks). In hypothetical calculations for an alternative heavyweight process without CI, but with similar codebase quality, he has estimated the project overhead at 464 hours, so in his case the CI process reduced the overhead by more than 40%.

IV. CONTINUOUS DELIVERY, CONTINUOUS DEPLOYMENT AND DEPLOYMENT PIPELINE

Continuous Delivery [5] [6] is the practice of developing software in a way where it is always ready to be deployed to the production environment (software is deployable through its lifecycle and the development team prioritize keeping the software deployable over time spent working on a new feature). Continuous Delivery is built on the CI (adding stages responsible for deploying application to production), so in order to do Continuous Delivery, you must be doing Continuous Integration. Continuous Deployment is a practice built on Continuous Delivery. Each change is automatically deployed to the production environment (which might result in multiple deployments per day). The main difference (and the only one) between Continuous Delivery and Continuous Deployment is that the deployment in Continuous

Delivery depends on business decisions and is triggered manually, and in Continuous Deployment each "good" change is immediately deployed to the production [5] [7].

According to Jez Humble and David Farley [6], Deployment Pipeline is a manifestation of a process of getting software from check-in to release ("getting software from version control into the hands of your users"). A diagram of a Deployment Pipeline has been shown in Figure 3. Each change, after being checked-in to the repository, creates the build that goes through a sequence of tests. As the build moves through the pipeline, tests become more complex, the environment more production-like and confidence in the build's good fitness is increasing. If any of the stages will fail, the build is not promoted to the next one, to save resources and send information to the development team rapidly. Stages common for all project types in the Deployment Pipeline are: commit stage (build compiles, low level unit-tests are passing, code analysis is passing), automated acceptance tests stage (asserts whether the project works on a functional level), manual test stage (asserts whether the system meets customer requirements, finding bugs omitted during the automated tests) and release stage (project is delivered to its users). This pattern does not imply that everything is automated and no user action is required – rather, it ensures that complicated and error-prone tasks are automated and repeatable.

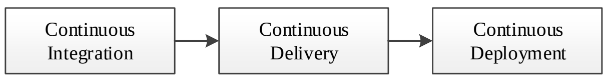

Fig.2. Continuous Integration, Delivery and Deployment relations

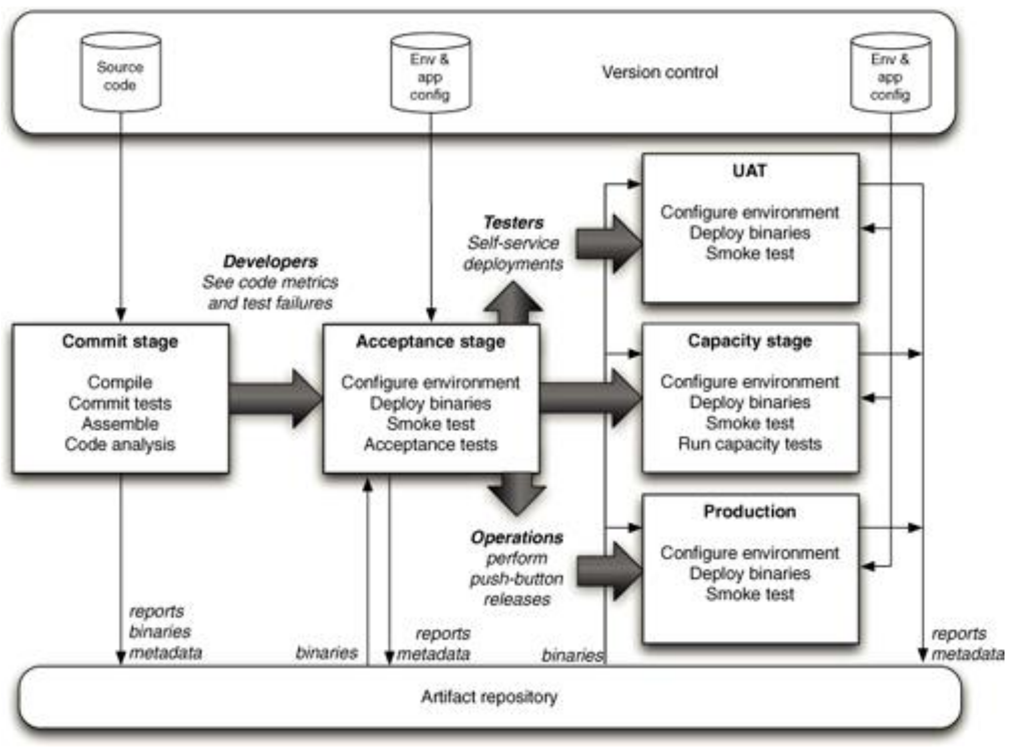

Fig.3. Basic Deployment Pipeline [6]

Jez Humble and David Farley [6] have distinguished a number of Deployment Pipeline practices: (1) Only build your binaries once, (2) Deploy the Same Way to Every Environment, (3) Smoke-Test Your Deployments, (4) Deploy into a Copy of Production, (5) Each Change Should Propagate through the Pipeline Instantly, (6) If Any Part of the Pipeline Fails, Stop the Line.

V. TEST AUTOMATION

A software is tested to detect errors, however the testing process is not able to confirm that the system works well in all conditions, but is able to show that under certain conditions it will not work. Testing may also verify whether the tested software behaves in accordance with the specified requirements used by developers during the design and implementation phase. It also provides information about the quality of the product and its condition. Frequent test execution (i.e. in an automated way) helps to address regressions introduced to source code as soon as possible.

All levels of tests can be automated. Starting from unit tests examining application internals, through the integration tests checking integration between different software components, finishing on acceptance tests validating system behavior. For .Net projects an example of technology that might be used to automate unit tests is xUnit.net [8]. Almost all modern CI servers have a built-in support for the most common test frameworks and all modern frameworks have support for command line instrumentation for automation purposes.

There are many good practices, patterns and technologies related to test development or execution that encourage automation. One of them is colloquially named the "3-As" pattern (Arrange, Act, Assert) suggesting that each test consists of an initialization part (arrange), invoking code or system under test (act) and verifying if the execution results meet the expectations (assert). Another good example is a PageObject [9] pattern which introduces separation between testing code and UI of the application under test (so the change in a tested application UI requires only a single change in the layer of that UI abstraction, not affecting the numbers of tests interacting with that UI through the PageObject layer). Selenium [10] is an example of a technology that allows automation of the interactions with web browsers.

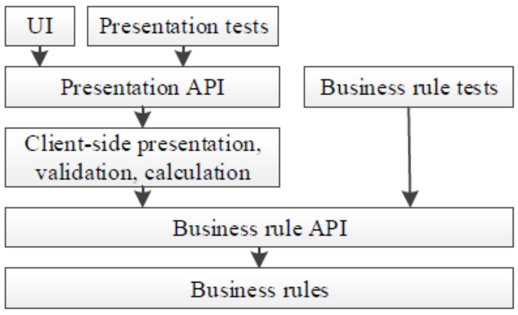

Fig.4. Test Bus Imperative [11]

In terms of automation of higher levels of tests interacting with UI (which might be very time-consuming), there is another pattern that is worth to mention – The Test Bus Imperative [11]. As the author claimed, it is the software development architecture that supports automated acceptance testing through exposing a test bus – the set of APIs that allows convenient access to application modules. So having in an application a presentation API used by both – UI and Acceptance Tests, allows developers to bypass UI to speed-up tests execution, which in consequence allows one to run a higher level of tests much more frequently – i.e. every commit.

In very complex systems, which are developed by multiple teams, with thousands of tests of multiple levels, sometimes information that the test failed is not sufficient. Especially, when one single source code change (maybe a complex one) causes hundreds of tests to fail – but there may be a single root cause for all of those failures. It is a very time consuming task to inspect all of those tests separately, and causes redundant work. Then, it comes in handy to have a debug automation process, which will be described further in this publication.

To have better understanding of how test automation works an example will be considered. For the sake of this example SUT (System Under Test) is a web-based application (hosted on an external HTTP server and accessed by users via web browsers). The system use case that will be covered by the automated acceptance test is very simple: a user using different browsers supported by the application (Firefox, Chrome and Internet Explorer) wants to log in to the application by providing user name and password and clicking "Log in" button. So the automated test must perform the following steps: open web browser, navigate to log in page, enter username and password, click "Log in" button, and validate if the user was correctly redirected to the main page. Development process in terms of this example is: (1) developer commits change to the repository, (2) CI server automatically detects that change, (3) downloads sources and starts a new build (compilation etc.), (4) automatically deploys application to a test environment that is similar to the production one (development HTTP server and database) and performs automated acceptance tests (including the one considered in this example), (5) in case of any failures, a report with test results is generated and presented to the user, otherwise (6) if every step succeeded, the application is deployed to the production. It is worth to mention that tests are written by developers or validation engineers and executed automatically on all provided web browsers. The entire process has been illustrated in Fig. 5.

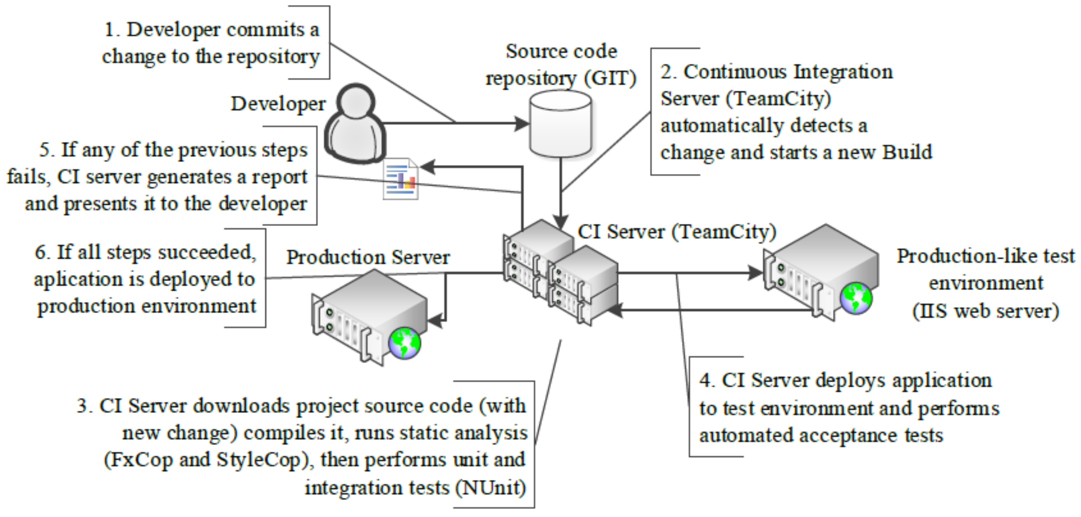
1. Developer commits a change to the repository
Source code repository (GIT)
2. Continuous Integration Server (TeamCity) automatically detects a change and starts a new Build
Developer
5. If any of the previous steps fails, CI server generates a report and presents it to the developer
CI Server (TeamCity)
6. If all steps succeeded, aplication is deployed to production environment
Production Server
Production-like test environment (IIS web server)
3. CI Server downloads project source code (with new change) compiles it, runs static analysis (FxCop and StyleCop), then performs unit and integration tests (NUnit)
4. CI Server deploys application to test environment and performs automated acceptance tests

Fig.5. Continuous Deployment process example

```
Feature: LoginFeature
Scenario Outline: Correct logging in
Given Start a web browser <browser>
  And Go to the log in page
  And enter login 'test' and password 'test'
When press "Log In" button
Then will be redirected to main page and will be
logged in.
Scenarios:
    | browser           |
    | Firefox           |
    | Chrome            |
    | Internet Explorer |
    | PhantomJS         |
```

Listing 1. Login use-case description in Gherkin language

```
public abstract class PageObject : IDisposable
{
  public IwebDriver WebDriver { get; set; }
  public string BaseUrl { get; set; }
  protected PageObject(IwebDriver webDriver){…}
  public void Dispose(){…}
  public void Navigate(string url)
  {
   WebDriver.Navigate().GoToUrl(BaseUrl+url);
  }
}
public class LoginPage : PageObject
{
  public LoginPage(IwebDriver webDriver):base(webDriver)
  {
    Navigate(“”);
  }
  public void InsertUserAndPassword(string user,string p
ass)
  {
   IwebElement loginInput =
    WebDriver.FindElement(By.Id(“login”));
   loginInput.SendKeys(user);
   IwebElement passInput =
    WebDriver.FindElement(By.Id(“password”));
   passInput.SendKeys(pass);
  }
  public PageObject Submit()
  {
    WebDriver.FindElement(By.TagName(“form”)).Submit();
    if(WebDriver.FindElement(By.Id(“login-status”))
     .Text != “OK”) return this;
    return new HomePage(WebDriver);
  }
}
public class HomePage : PageObject
{

public HomePage(IwebDriver webDriver) : base(webDriver)
{}
}
```

Listing 2. Implementation of PageObject pattern using Selenium to interact with web browser

For the sake of this example, as a source code repository GIT has been chosen, and for the CI server, TeamCity from Jetbrains. The TeamCity server has all necessary features built-in, i.e. automated compilation (and deployment) using MSBuild or Visual Studio, running batch scripts, communication with many popular source code repositories (including GIT), running static code analysis (i.e. using FxCop and StyleCop) and unit test runners (i.e. NUnit).

Automation of web application acceptance tests has been accomplished by using a combination of technologies like: Gherkin, SpecFlow, NUnit and Selenium. Gherkin is a business readable language that allows one to specify acceptance criteria using English-like syntax and Given-When-Then patterns (Listing 1.). SpecFlow allows you to generate a test skeleton for a provided gherkin description using (i.e.) NUnit beneath as a test framework (Listing 3.). Generated steps are implemented using Selenium that allows interaction with the web browser via a PageObject abstraction layer (Listing 2.). Listing 2. and 3. are written in C# programming language.

```
[Binding]
public class LoginFeatureStepDefinitions
{
  public PageObject PageObject { get; set; }

  [Given(@”Start a web browser Chrome”)]
  public void GivenStartAWebBrowserChrome()
  {
   PageObject = new LoginPage(new ChromeDriver());
  }

  [Given(@”Start a web browser Firefox”)]
  public void GivenStartAWebBrowserFirefox()
  {
   PageObject = new LoginPage(new FirefoxDriver());
  }

  [Given(@”Start a web browser Internet Explorer”)]
  public void GivenStartAWebBrowserInternetExplorer()
  {
   PageObject = new LoginPage(new InternetExplorerDriver(
));
  }

  [Given(@”Start a web browser PhantomJS”)]
  public void GivenStartAWebBrowserPhantomJs()
  {
   PageObject = new LoginPage(new PhantomJSDriver());
  }

  [Given(@”Go to the log in page”)]
  public void GivenGoToTheLogInPage(){}

  [Given(@”enter login ‘(.*)’ and password ‘(.*)’”)]
 public void GivenEnterLoginAndPassword(string p0,string
p1)
  {
   (PageObject as LoginPage).InsertUserAndPassword(p0,p1)
;
  }

  [When(@”press “”(.*)”” button”)]
  public void WhenPressButton(string p0)
  {
   PageObject = (PageObject as LoginPage).Submit();
  }

  [Then(@”will be redirected do main page (…)”)]
  public void ThenWillBeRedirectedToTheMainPage()
  {
   if (PageObject.GetType()!=typeof(HomePage))
   {
    Assert.Fail();
   }
  }

  [AfterScenario]
  public void TearDown()
  {
   if (PageObject != null)
   {
    PageObject.Dispose();
    PageObject = null;
   }
  }
}
```

Listing 3. Acceptance test implementation using SpecFlow

VI. Debug Process Automation

When the developed application is very complex, consisting of many components with thousands of tests, sometimes information that the test failed may not be sufficient. Especially, after commit failed hundreds of tests at the same time. Inspecting all of them may be a time-consuming task. After all, it may be a single bug that caused multiple tests to fail.

When the software was released to the users, they will probably report some errors (manually or via an automated system). The number of reported errors depends on the quality of the application and the number of the users using the developed application. Manual error report analysis might be a time-consuming task with redundant work, because many of the reported errors will have the same root cause in the application's source code (hundreds of users experiencing the same bug and reporting it via the automated error collection system).

When the inflow of error reports (coming from internal test execution systems or from the users after the software was released) is big, it may come in handy to have some kind of post-mortem debug automation process to reduce the time that developers must spend on bug fixing. The main goal of debug automation is to automatically detect the root cause of a single crash report, and aggregate collections of bug reports of the same bug into buckets to avoid duplicates, thus saving developers' time.

Error reports might be prepared by the crashing application itself when the crash occurs (when developers expect that a crash may have occurred and prepared some kind of exception handling and reporting subsystem) or by the operating system (when the error was unhandled by the application). All of modern operating systems are capable of handling unhandled application exceptions, preparing crash reports consisting of memory dumps or log files from memory analysis (kernel or user memory dumps for Windows, coredumps for Linux, Tombstone files for Android).

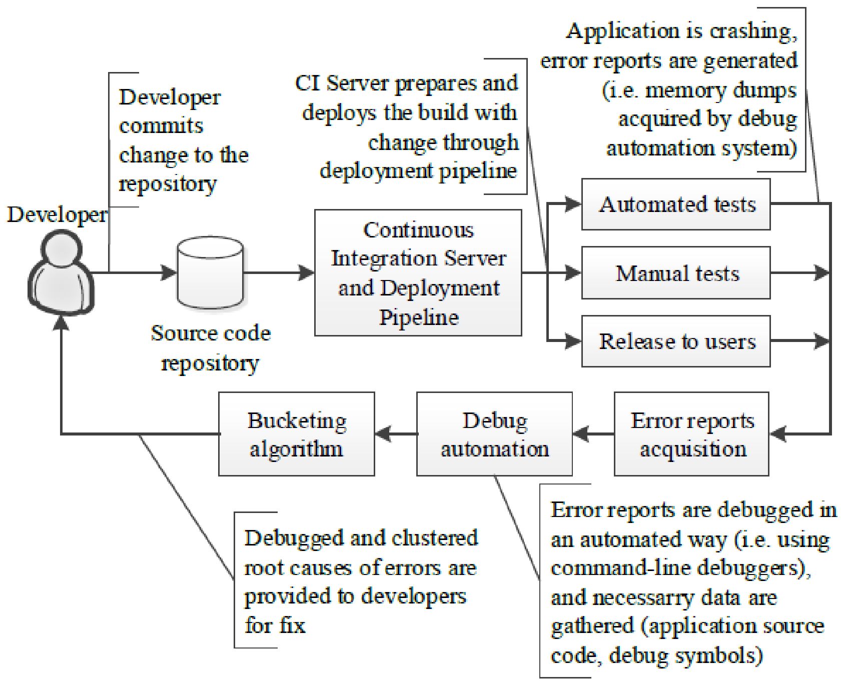
Developer
Developer commits change to the repository
CI Server prepares and deploys the build with change through deployment pipeline
Application is crashing, error reports are generated (i.e. memory dumps acquired by debug automation system)
Source code repository
Continuous Integration Server and Deployment Pipeline
Automated tests
Manual tests
Release to users
Bucketing algorithm
Debug automation
Error reports acquisition
Debugged and clustered root causes of errors are provided to developers for fix
Error reports are debugged in an automated way (i.e. using command-line debuggers), and necessarry data are gathered (application source code, debug symbols)

Fig.6. Example of debug automation process

The Debug automation system should be able to analyze error reports, i.e. using command-line debuggers (KD and CDB in Windows, GDB in Linux). Sometimes it is necessary for the debuggers to have some additional resources like application source code or debug symbols to provide more accurate data. Results of the automated analysis should provide a crash root cause signature that would be used by the bucketing algorithm responsible for clustering duplicated crash reports into buckets representing a single bug in the source code.

An example of a bucketing algorithm may be the one using call stack similarity to indicate if the two crash reports represent the same bug. This similarity may be computed using simple string-like similarity (i.e. Levenshtein distance) or a much more sophisticated method, like the one proposed by the Microsoft Research Team, Position Dependent Model (PDM) that's part of a more complex method called ReBucket [12]. Another example of a bucketing algorithm in Windows might be using the result of "!analyze -v" command in KD or CDB debuggers, providing information like exception code and its arguments or "BUCKET_ID" section [13].

Windows Error Reporting System (WER) [14] is a good example of a large scale debug automation system used at Microsoft. It originated from a combination of diagnosis tools used by the Windows team, and an automatic data collection tool from the Office team. As described in [14], WER is a distributed post-mortem debugging system. When the error occurs on the client machine, a special mechanism that's part of the Windows operating system automatically collects necessary data on the client machine, prepares an error report, and (with user permissions) send that report to WER Service, where it's debugged. If the resolution for the bug is already known, the client is provided with a URL to find the fix.

To have better understanding of how debug automation can reduce redundant work in failing test results analysis, an extended version of the example from the previous section will be considered. An additional assumption to the example provided in the test automation section is that the application under test is written in ASP.NET MVC technology (for the sake of this example). Automated flow has been extended for additional steps: application error acquisition and error correlation, so the entire process is: developer commits change to the repository, CI server detects that change and starts new build, automatically deploys application to test environment and performs automated tests, in case of any failures acquires error reports from the application and performs correlation of the error reports; otherwise, if every step succeeded, the application is deployed to the production. An example of how web application written in ASP.NET MVC technology can handle errors has been presented on Listing 4. Method DumpCallStack sends a prepared text file with call stack of the unhandled exception that occurred which is acquired by the next step of the automated debug process.

Then, after acquiring all error reports with exception call stacks, all of them are compared (i.e. using simple string comparison) to find out how many of them are identical. So the result of this example can be as follows: after submitting a change to the repository 10 tests failed, but debug automation step after analyzing call stacks of those 10 fails finds out that all of them were caused by a single root cause (with one, identical call stack). So the developer, instead of analyzing all 10 test results, has to focus on a single bug represented by a single call stack causing those 10 tests to fail.

```
public class MvcApplication : System.Web.HttpApplication
{
  (…)
  public void Application_Error()
  {
   Exception e = Server.GetLastError();
   Response.Clear();
   Server.ClearError();
   DumpCallStack(e.StackTrace);
  }
}
```

Listing 4. Function in Global.asax of ASP.NET MVC application collecting exception call stack after each failure

VII. Conclusion

Increasing business demand for reducing the time of development and deployment to production of new features, fosters automation in software development, as can be seen in practices like Continuous Integration, Delivery and Deployment. Starting from the compilation, through deployment, tests and debug – almost all stages of iterative development activities might be automated. The core of the automation best practices is mentioned before Continuous Integration, which tends to evolve into an extended version: Continuous Delivery and Continuous Deployment. Complex systems with advanced validation processes (i.e. complex and well-developed automated tests on many levels) needs to have an automated debugging process to reduce redundant work when analyzing failed test results (when a single root cause in the source code caused hundreds of tests to fail).

References

[1] Forrester Consulting, "Continuous Delivery: A Maturity Assessment Model," 2013.

[2] D. Zima, "Modern Methods of Software Development," TASK Quarterly, tom 19, nr 4, 2015.

[3] M. Fowler, "Continuous Integration," [Online]. Available: http://www.martinfowler.com/articles/continuousIntegration.html. [Accessed 3 10 2015].

[4] A. Miller, "A Hundred Days of Continuous Integration," in Agile 2008 Conference, 2008.

[5] M. Fowler, "ContinuousDelivery," 30 May 2013. [Online]. Available: http://martinfowler.com/bliki/ContinuousDelivery.html. [Accessed 28 November 2015].

[6] J. Humble and D. Farley, "Anatomy of the Deployment Pipeline," in Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional, 2010, pp. 105-141.

[7] J. Humble, "Continuous Delivery vs Continuous Deployment," 13 August 2010. [Online]. Available: http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/. [Accessed 22 November 2015].

[8] "xUnit.net," [Online]. Available: http://xunit.github.io/. [Accessed 28 November 2015].

[9] M. Fowler, "PageObject," 10 September 2013. [Online]. Available: http://martinfowler.com/bliki/PageObject.html. [Accessed 12 February 2016].

[10] "Selenium," [Online]. Available: http://www.seleniumhq.org/. [Accessed 12 February 2016].

[11] R. Martin, "The test bus imperative: architectures that support automated acceptance testing," IEEE Software, tom 22, nr 4, pp. 65-67, 2005.

[12] Y. Dang, R. Wu, H. Zhang, D. Zhang and P. Nobel, "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity".

[13] "Using the !analyze Extension," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff560201. [Accessed 22 November 2015].

[14] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009.

[15] Zaigham Mushtaq, M. Rizwan Jameel Qureshi, "Novel Hybrid Model: Integrating Scrum and XP", IJITCS, vol.4, no.6, pp.39-44, 2012.

Authors' Profiles

Henryk Krawczyk: Rector of the Gdansk University of Technology in 2008-2012 and 2012-2016, the dean of Faculty of Electronics, Telecommunications and Informatics in 1990-1996 and 2002-2008, and also Head of the Computer Systems Architecture Department since 1997, received his PhD in 1976 and became a Full Professor in 1996. Current research interests: software development and testing methods, modeling and analysis of dependability of distributed computer systems including emergency situations, defining a new category of so-called approaching threats and determining effective and suitable detection and elimination procedures, analyzing the essential relationship between system components and their user behaviors, developing web services and distributed applications with usage of digital documents.

Dawid Zima: PhD student at the Gdansk University of Technology, Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics. Received his engineering degree in 2012 and master's degree in 2013. Research interests: debug process automation, methods for bucketing error reports based on call stack similarities.

How to cite this paper: Henryk Krawczyk, Dawid Zima, "Automation in Software Source Code Development", International Journal of Information Technology and Computer Science(IJITCS), Vol.8, No.12, pp.1-9, 2016. DOI: 10.5815/ijitcs.2016.12.01