View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Research Papers in Economics

The Annals of "Dunarea de Jos" University of Galati
Fascicle I – 2009. Economics and Applied Informatics. Years XV - ISSN 1584-0409

Automated Web Applications Testing

Alexandru Dan CĂPRIȚĂ
caprita_dan@yahoo.com
Dunărea de Jos University

Abstract. Unit tests are a vital part of several software development practices and processes such as Test-First Programming, Extreme Programming and Test-Driven Development. This article shortly presents the software quality and testing concepts as well as an introduction to an automated unit testing framework for PHP web based applications.

Keywords: software quality, continuous integration, unit testing.

JEL Code: L86

1. Introduction

In the past couple of decades a quality revolution, has been spreading fast throughout the world with the explosion of the Internet. Global competition, outsourcing, off-shoring, and increasing customer expectations have brought the concept of quality to the forefront. Developing quality products on tighter schedules is critical for a company to be successful in the new global economy. PHP unit testing, as part of the dynamic quality analysis for php web based application, involves writing PHP scripts specifically to test other PHP scripts. This type of testing is referred to as *unit testing* because the test method is designed to test individual code units, like classes and methods, one at a time. PHPUnit is an elegant solution to writing these tests, and it follows an object-oriented development approach.

2. Software quality and testing

Quality is a complex concept - it means different things to different people, and it is highly context dependent. A quality factor represents a behavioral characteristic of a system. Some examples of high-level quality factors are correctness, reliability, efficiency, testability, maintainability, and reusability. A quality criterion is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. Highly modular software allows designers to put cohesive components in one module, thereby improving the maintainability of the system.Various software quality models have been proposed to define quality and its related attributes. The most important ones are the ISO 9126 and the CMM[1]. The document ISO 9126 defines six broad, independent categories of quality characteristics:

- Functionality,
- Reliability,
- Usability,
- Efficiency,
- Maintainability,
- Portability.

[1] *Capability Maturity Model - is a model for judging the maturity of the software processes of an organization and for identifying the key practices that are required to increase the maturity of these processes.*

The CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. In the CMM framework, a development process is evaluated on a scale of 1–5, commonly known as level 1 through level 5. For example, level 1 is called the initial level, whereas level 5—optimized—is the highest level of process maturity.

In the field of software testing, there are two well-known process models: the test process improvement (TPI) model and the test maturity model (TMM). These two models allow an organization to assess the current state of their software testing processes, identify the next logical area for improvement, and recommend an action plan for test process improvement.

Testing plays an important role in achieving and assessing the quality of a software product. On the one hand, we improve the quality of the products as we repeat a test–find defects–fix cycle during development. On the other hand, we assess how good our system is when we perform system-level tests before releasing a product. Thus, as Friedman and Voas have succinctly described, software testing is a verification process for software quality assessment and improvement. Generally speaking, the activities for software quality assessment can be divided into two broad categories:

- **Static Analysis**: it is based on the examination of a number of documents, namely requirements documents, software models, design documents, and source code. Traditional static analysis includes code review, inspection, walk-through, algorithm analysis, and proof of correctness. It does not involve actual execution of the code under development. Instead, it examines code and reasons over all possible behaviors that might arise during run time. Compiler optimizations are standard static analysis;
- **Dynamic Analysis**: dynamic analysis of a software system involves actual program execution in order to expose possible program failures. The behavioral and performance properties of the program are also observed. Programs are executed with both typical and carefully chosen input values. Often, the input set of a program can be impractically large. However, for practical considerations, a finite subset of the input set can be selected. Therefore, in testing, some representative program behaviors are observed and a conclusion about the quality of the system id reached. Careful selection of a finite test set is crucial to reaching a reliable conclusion.

By performing static and dynamic analyses, practitioners want to identify as many faults as possible so that those faults are fixed at an early stage of the software development. Static analysis and dynamic analysis are complementary in nature, and for better effectiveness, both must be performed repeatedly and alternated.

Two similar concepts related to software testing frequently used by practitioners are *verification* and *validation*. Both concepts are abstract in nature, and each can be realized by a set of concrete, executable activities:

- **Verification**: helps in evaluating a software system by determining whether the product of a given development phase satisfies the requirements established before the start of that phase. A product can be an intermediate product, such as requirement specification, design specification, code, user manual, or even the final product. Activities that check the correctness of a development phase are called verification activities;
- **Validation**: helps in confirming that a product meets its intended use. Validation activities aim at confirming that a product meets its customer's expectations. In other words, validation activities focus on the final product, which is extensively tested from the customer point of view. Validation establishes whether the product meets overall expectations of the users. Late execution of validation activities is often risky by leading to higher development cost. Validation activities may be executed at early stages of the software development cycle. An example of early execution of validation activities can be

found in the eXtreme Programming (XP) software development methodology. In the XP methodology, the customer closely interacts with the software development group and conducts acceptance tests during each development iteration.

Verification activities aim at confirming that one is *building the product correctly*, whereas validation activities aim at confirming that one is *building the correct product*.

3. Testing models, activities and levels

In order to test a program, a test engineer must perform a sequence of testing activities. These *activities* are explain bellow and are referring on a single test case:

- **Identify an objective to be tested**: the *objective* defines the intention, or *purpose*, of designing one or more test cases to ensure that the program supports the objective. A clear purpose must be associated with every test case;
- **Select inputs**: selection of test inputs can be based on the requirements specification, the source code, or expectations. Test inputs are selected by keeping the test objective in mind;
- **Compute the expected outcome**: the third activity is to compute the expected outcome of the program with the selected inputs. In most cases, this can be done from an overall, high-level understanding of the test objective and the specification of the program under test;
- **Set up the execution environment of the program**: prepare the right execution environment of the program. In this step all the assumptions external to the program must be satisfied (e.g. network connection available, database server running);
- **Execute the program**: the test engineer executes the program with the selected inputs and observes the actual outcome of the program. To execute a test case, inputs may be provided to the program at different physical locations at different times. The concept of *test coordination* is used in synchronizing different components of a test case;
- **Analyze the test result**: The final test activity is to analyze the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome. The complexity of comparison depends on the complexity of the data to be observed. The observed data type can be as simple as an integer or a string of characters or as complex as an image, a video, or an audio clip. At the end of the analysis step, a test verdict is assigned to the program. There are three major kinds of test verdicts, namely, *pass*, *fail*, and *inconclusive* (further tests are needed to be done to refine the inconclusive verdict into a clear pass or fail verdict).

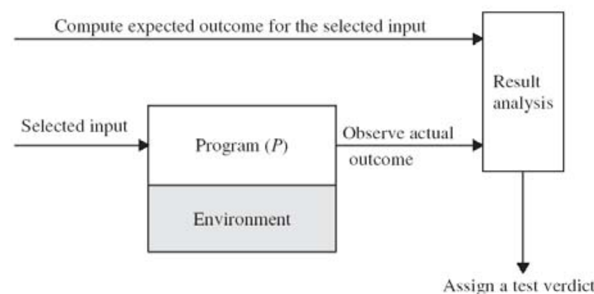

Figure 1. Different activities in program testing

A *test report* must be written after analyzing the test result. The motivation for writing a test report is to get the fault fixed if the test revealed a fault. A test report contains the following items to be informative:

a) Explain how to reproduce the failure;
b) Analyze the failure to be able to describe it;
c) A pointer to the actual outcome and the test case, complete with the input, the expected outcome, and the execution environment.

Testing is performed at different *levels* involving the complete system or parts of it throughout the life cycle of a software product. A software system goes through four stages of testing before it is actually deployed. These four stages are known as *unit*, *integration*, *system*, and *acceptance* level testing. The first three levels of testing are performed by a number of different stakeholders in the development organization, where as acceptance testing is performed by the customers.

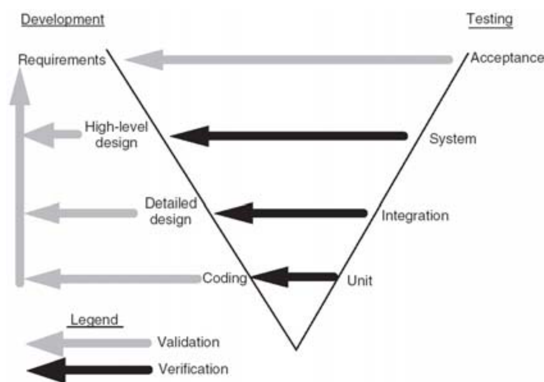

Figure 2. Development and testing phases in the V model

In *unit testing*, programmers test individual program units, such as a procedures, functions, methods, or classes, in isolation. After ensuring that individual units work to a satisfactory extent, modules are assembled to construct larger subsystems by following integration testing techniques. *Integration testing* is jointly performed by software developers and integration test engineers. The objective of integration testing is to construct a reasonably stable system that can withstand the rigor of system-level testing. *System-level testing* includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing. System testing is a critical phase in a software development process because of the need to meet a tight schedule close to delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

Regression testing is another level of testing that is performed throughout the life cycle of a system. Regression testing is performed whenever a component of the system is modified. The key idea in regression testing is to ascertain that the modification has not introduced any new faults in the portion that was not subject to modification. To be precise, regression testing is not a distinct level of testing. Rather, it is considered as a subphase of unit, integration, and system-level testing. In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software. Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry. It is desirable to select a subset of the test cases from the existing pool to reduce the cost. A key question is how many and which test cases should be selected so that the selected test cases are more likely to uncover new faults.

After the completion of system-level testing, the product is delivered to the customer. The customers perform their own series of tests, commonly known as *acceptance testing*. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing. A key notion in acceptance testing is the

customer's expectations from the system. By the time of acceptance testing, the customer should have developed their acceptance criteria based on their own expectations from the system. There are two kinds of acceptance testing as explained in the following:

- User acceptance testing (UAT),
- Business acceptance testing (BAT).

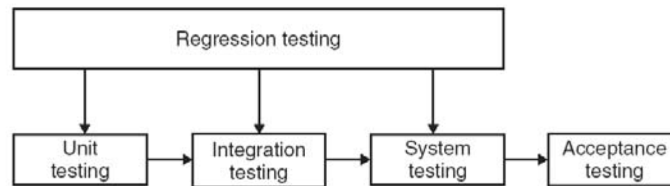

Figure 3. Regression testing at different software testing levels

User acceptance testing is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed off as meeting user needs. On the other hand, BAT is undertaken within the supplier's development organization. The idea in having a BAT is to ensure that the system will eventually pass the user acceptance test. It is a rehearsal of UAT at the supplier's premises.

A software development process generates a large body of information, such as requirements specification, design document, and source code. In order to generate effective *tests* at a lower cost, test designers analyze the following *sources of information*:

- Requirements and functional specifications,
- Source code,
- Input and output domains,
- Operational profile,
- Fault model.

Apart from the traditional testing techniques, various new techniques necessitated by the complicated business and development logic were realized to make software testing more meaningful and purposeful. Some of the popular testing techniques, mainly based on the sources of information for test design, are *black-box* testing (functional), *white-box* testing (structural) and *gray-box* testing (functional and structural).

A. *Black-box* or *functional testing* is one in which test conditions are developed based on the program or system's functionality; that is, the tester requires information about the input data and observed output, but does not know how the program or system works. Just as one does not have to know how a car works internally to drive it, it is not necessary to know the internal structure of a program to execute it. The tester focuses on testing the program's functionality against the specification. With black-box testing, the tester views the program as a black-box and is completely unconcerned with the internal structure of the program or system. Some examples in this category include: decision tables, equivalence partitioning, range testing, boundary value testing, database integrity testing, cause-effect graphing, exception testing, limit testing, and random testing.

A major advantage of black-box testing is that the tests are geared to what the program or system is supposed to do, and it is natural and understood by everyone. This should be verified with techniques such as structured walkthroughs, inspections, and joint application designs (JADs). A limitation is that exhaustive input testing is not achievable, because this requires that every possible input condition or combination be tested. In addition, because there is no

knowledge of the internal structure or logic, there could be errors or deliberate mischief on the part of a programmer.

B. In *white-box* or *structural* testing test conditions are designed by examining paths of logic. The tester examines the internal structure of the program or system. Test data is driven by examining the logic of the program or system, without concern for the program or system requirements. The tester knows the internal program structure and logic, just as a car mechanic knows the inner workings of an automobile. Specific examples in this category include basis path analysis, statement coverage, branch coverage, condition coverage, and branch/condition coverage.

An advantage of white-box testing is that it is thorough and focuses on the produced code. Because there is knowledge of the internal structure or logic, errors or deliberate mischief on the part of a programmer have a higher probability of being detected. One disadvantage of white-box testing is that it does not verify that the specifications are correct; that is, it focuses only on the internal logic and does not verify the logic to the specification. Another disadvantage is that there is no way to detect missing paths and data-sensitive errors. A final disadvantage is that white-box testing cannot execute all possible logic paths through a program because this would entail an astronomically large number of tests.

C. The *gray-box* testing (functional and structural) is a combination of black- and white-box testing. The tester studies the requirements specifications and communicates with the developer to understand the internal structure of the system. The motivation is to clear up ambiguous specifications and "read between the lines" to design implied tests. One example of the use of gray-box testing is when it appears to the tester that a certain functionality seems to be reused throughout an application. If the tester communicates with the developer and understands the internal design and architecture, many tests will be eliminated, because it may be possible to test the functionality only once.

Unit testing is the process of executing a functional subset of the software system to determine whether it performs its assigned function. It is oriented toward the checking of a function or a module. White-box test cases are created and documented to validate the unit logic and black-box test cases to test the unit against the specifications. Unit testing, along with the version control necessary during correction and retesting, is typically performed by the developer. During unit test case development it is important to know which portions of the code have been subjected to test cases and which have not. By knowing this coverage, the developer can discover lines of code that are never executed or program functions that do not perform according to the specifications.

4. PHPUnit framework

Even good programmers make mistakes. The difference between a good programmer and a bad programmer is that the good programmer uses tests to detect his mistakes as soon as possible. This explains why leaving testing until just before releasing software is so problematic. Most errors do not get caught at all, and the cost of fixing the ones found is so high that not all of them can be fixed at that moment. Testing with PHPUnit is not a totally different activity. The difference is between testing, that is, checking that the program behaves as expected, and performing a battery of tests, runnable code-fragments that automatically test the correctness of parts (units) of the software. These runnable code-fragments are called *unit tests*.

PHPUnit framework has to simultaneously resolve a set of *constraints*, some of which seem always to conflict with each other. Tests should be:

- Easy to learn to write: if it's hard to learn how to write tests, developers will not learn to write them;
- Easy to write: if tests are not easy to write, developers will not write them;
- Easy to read: test code should contain no extraneous overhead so that the test itself does not get lost in noise that surrounds it;
- Easy to execute: the tests should run at the touch of a button and present their results in a clear and unambiguous format;
- Quick to execute: tests should run fast so they can be run hundreds or thousands of times a day;
- Isolated: the tests should not affect each other. If the order in which the tests are run changes, the results of the tests should not change
- Composable: any number or combination of tests should be able to run together. This is a corollary of isolation.

There are two main clashes between these constraints:

- Easy to learn to write versus easy to write: tests do not generally require all the flexibility of a programming language. Many testing tools provide their own scripting language that only includes the minimum necessary features for writing tests. The resulting tests are easy to read and write because they have no noise to distract developers from the content of the tests. However, learning yet another programming language and set of programming tools is inconvenient;
- Isolated versus quick to execute: the results of one test should have no effect on the results of another test, each test should create the full state of the world before it begins to execute and return the world to its original state when it finishes. However, setting up the world can take a long time: for example connecting to a database and initializing it to a known state using realistic data.

PHPUnit features include but not limited to: complete port of JUnit 3.8.1 to PHP 5, integrates ideas from TestNG, supports database testing (port of DbUnit), supports Code Coverage Analysis (utilizing the Xdebug extension for PHP) and can generate reports based on this information, supports the calculation of software metrics, supports storing test result and code coverage data in a Test Database, supports Incomplete Tests (port of junitour) and the skipping of tests, supports generation of test code skeletons for existing code, supports logging of test execution in an XML, JSON (JavaScript Object Notation) or TAP (Test Anything Protocol), integrates with Selenium RC for web application user interface testing, integrates with Apache Maven, CruiseControl, and Parabuild for Continuous Integration, integrates with Phing.

One of the most time-consuming parts of writing tests is writing the code to set the world up in a known state and then return it to its original state when the test is complete. This known state is called the *fixture* of the test. PHPUnit supports sharing the setup code. Before a test method is run, a template method called setUp()(creates the objects against the test will run). Once the test method has finished running, whether it succeeded or failed, another template method called tearDown() is invoked (clean up the objects). The steps needed to installing PHPUnit are as follows:

a) Install PEAR[1]:
   C:\Php>go-pear.bat
b) Use PEAR to "discover" the pear.phpunit.de channel:
   C:\Php>pear channel-discover pear.phpunit.de
c) Install PHPUnit and any dependencies it needs:
   C:\Php>pear install phpunit/PHPUnit

[1] ***P**HP **E**xtension and **A**pplication **R**epository - a framework and distribution system for reusable PHP components*

5. Case Study: BankAccount

In this example a BankAccount class is defined. It requires methods to get and set the bank account's balance, as well as methods to deposit and withdraw money. It also specifies the following two conditions that must be ensured:

- The bank account's initial balance must be zero;
- The bank account's balance cannot become negative.

```
<?php
class BankAccount{
   protected $owner;
   protected $IBAN;
   protected $balance = 0;

   function __construct($owner, $IBAN){
      $this->owner = $owner;
      $this->IBAN = $IBAN;
   }

   public function getBalance(){
      return $this->balance;
   }

   protected function setBalance($balance){
      //if ($balance >= 0){
         $this->balance = $balance;
      //}else{
      //    throw new BankAccountException(
             $this->getBalance());
      //}
   }

   public function depositMoney($balance){
      $this->setBalance($this->getBalance() + $balance);
      return $this->getBalance();
   }

   public function withdrawMoney($balance){
      $this->setBalance($this->getBalance() - $balance);
      return $this->getBalance();
   }
}
class BankAccountException extends Exception{
   public function errorMessage(){
      $errorMsg = 'Error because current balance is ';
      $errorMsg .= $this->getMessage().'!';
      return $errorMsg;
  }
}
?>
```

Example 1. The BankAccount class

The first condition is tested by testBalanceIsInitiallyZero() that uses getBalance() method of the BankAccount to obtain the actual balance value and check it against the expected value (i.e. zero). The methods that ensure the second condition is passed are written in a such a way that they raise a BankAccountException when they are called with illegal values that would violate the conditions.

```
<?php
require_once 'PHPUnit/Framework.php';
require_once 'BankAccount.php';

class BankAccountTest extends PHPUnit_Framework_TestCase{
   protected $ba;

   protected function setUp(){
      $this->ba = new BankAccount('UGAL','RO57TREZ3065003
         XXX000095');
   }

   public function testBalanceIsInitiallyZero(){
      $this->assertEquals(0, $this->ba->getBalance());
   }

   public function testBalanceCannotBecomeNegative(){
      try {
         $this->ba->withdrawMoney(1);
      }
      catch (BankAccountException $e){
         $this->assertEquals(0, $this->ba->getBalance());
         return;
      }
      $this->fail();
   }

   public function testBalanceCannotBecomeNegative2(){
      try {
         $this->ba->depositMoney(-1);
      }
      catch (BankAccountException $e){
         $this->assertEquals(0, $this->ba->getBalance());
         return;
      }
      $this->fail();
   }
}
?>
```

Example 2. Tests for the BankAccount class

The PHPUnit command-line test runner can be invoked through the phpunit command:

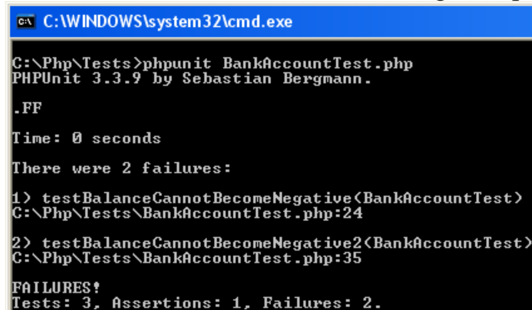

Figure 4. Tests run for the BankAccount class

The last two tests fails because the setBalance() didn't raised a BankAccountException exception as expected when the current balance is set to a negative value.

For each test run, the PHPUnit command-line tool prints one character to indicate progress:

.	- printed when the test succeeds;
F	- printed when an assertion fails while running the test method;
E	- printed when an error occurs while running the test method;
S	- printed when the test has been skipped;
I	- printed when the test is marked as being incomplete or not yet implemented.

PHPUnit distinguishes between failures and errors. A failure is a violated PHPUnit assertion such as a failing assertEquals() call. An error is an unexpected exception or a PHP error. Sometimes this distinction proves useful since errors tend to be easier to fix than failures. If the list of problems is big, it is best to tackle the errors first and see if there are any failures left when they are all fixed.

6. Conclusion

In the course of development for any reasonably complex application issues like bugs, logic errors, and collaboration problems can appear. How these issues are handled can make the difference between a successful development cycle with happy developers and an overdue, overbudget application with an employee-turnover problem. A series of tools can help to better manage projects and track project's progress in real time. These tools, when combined, form a programming technique called *continuous integration*. Any continuous integration project includes four main components: revision control, unit testing, deployment, and debugging.

So quality management is used to decreases production costs because the sooner a defect is located and corrected, the less costly it will be in the long run. With the advent of automated testing tools (like PHPUnit), although the initial investment can be substantial, the long-term result will be higher-quality products and reduced maintenance costs.

This article present unit testing, the basic level of testing. The advantage of unit testing is that it permits the testing and debugging of small units, thereby providing a better way to manage the integration of the units into larger units. In addition, testing a smaller unit of code makes it mathematically possible to fully test the code's logic with fewer tests. Unit testing also facilitates automated testing because the behavior of smaller units can be captured and played back with maximized reusability.

References

1. *Kshirasagar Naik, Priyadarshi Tripathy – "Software Testing and Quality Assurance. Theory and Practice", John Wiley & Sons, 2008;*
2. *William E. Lewis – "Software Testing and Continuous Quality Improvement (Second Edition)", CRC Press , 2005;*
3. *Kevin McArthur – "Pro PHP: Patterns, Frameworks, Testing and More", Apress, 2008;*
4. *Sebastian Bergmann – "PHPUnit Manual", http://www.phpunit.de, 01.2009;*
5. *http://en.wikipedia.org/wiki/Unit_testing, 01.2009;*
6. http://en.wikipedia.org/wiki/Continuous_integration 01.2009