Automated test methods

Mgr. Libor Vaněk
September, 2010

Table of Content:

1 Introduction

System development nowadays more then ever starts to look for automated test methods. There are several main drivers for this trend - need for faster design-develop-test-analyze cycle, push for higher quality, increasing complexity of systems and their integration and last but not least ever-raising costs of manual testing.

Software testing itself is very wide field with topics ranging from compliance frameworks (ISO 9000), software development methodology (test-drive development), test data selection methods (Boundary testing) to many various tools and frameworks.

In this paper I want to present several most commonly used methods for automated testing and help reader to understand when they are suitable for use and what are their limitations.

2 Test methods

2.1 Record & replay

This test method is one of the oldest one used already for several decades. As such it's often considered as obsolete and incompatible with modern software development methodologies - like agile or TDD. Most of currently used software methodologies require writing test cases/scripts describing business logic. This may not be always feasible - e.g. business logic may be entangled with UI, not clearly known even to business users (legacy systems), not technically possible (no access to application API) or simply too time & money consuming. Also in case of refactoring (typically parts of) legacy systems it increases risk of not capturing all aspects of system and focus is mainly on ensuring regression testing. [RR]

Record & replay method as name suggest consist of two stages:

1. **Record** - when user performs designated actions with system and a component records his activities and reactions of a system.
2. **Replay** - when recorded activities are replayed from a repository and system reactions are verified against stored ones.

Main advantages are that record phase may be performed by existing system users or staff without any IT-specific knowledge. This allows recording large number of real-life test cases to be created.

Major limitation of this methodology is necessity to have an existing application. Also new application must not differ significantly from current application - otherwise recorded test cases will not be applicable. This principally forbids this methodology to be used for "green field" projects - even though it's possible to create and record activities using UI mock-ups.

2.1.1 Common issues with record & replay method

Record and replay method has over years lot of bad reputation as obsolete method. Many times this may be due to incorrect use of it. Therefore it's important to be aware of most common issues and limitations.

2.1.1.1Functional sensitivity

If functionality (behavior) of the system is modified (e.g. due to requirements change) test cases related to it will fail. This is generally

common for all test method but in record & replay method may be more difficult to identify affected test cases and fix them.

2.1.1.2 Interface sensitivity

Typically used test tools simulate user activities on the user interface ("robot user"). Even minor change of the user interface may cause test to fail even if human tester would consider test as pass. To overcome this issue many tools offers various methods to make test cases more robust (e.g. identifying UI elements by name/ID/label instead of position on screen) - but these add complexity to test case creation.

2.1.1.3 Data & context sensitivity

Record and replay test method is very sensitive to setting up same starting point (precondition) for every test run. This may complicate the situation when we are dealing with highly integrated environment or results depend on a date & time. Modern test tools offers methods to make test cases less sensitive by e.g. selecting which parts of system's response may be ignored or how to verify it using some formula. Another approach is to replace integration with external system by simulation stubs. But - same as in previous case - these methods increases usage complexity.

2.1.2 Types of record & replay usage

There are 2 main approaches how record & replay method can be used. Each one is suitable for different use case and with different benefits and limitations.

2.1.2.1 "Robot user"

This usage involves recording test cases based on how user interacts with the system via User Interface. Usually this means testing of complete system. This approach is ideal for legacy systems where we need to ensure regression testing and other testing methods are not suitable (e.g. lack of source codes, license issues etc.).

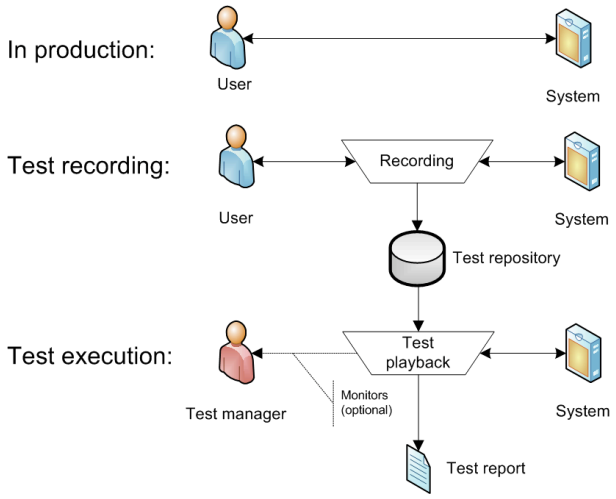

Figure 1 - "Robot user" record & replay method

On figure above is depicted method of operating of this approach. During test recording system records into test repository User's activities with System ("request") and System's reactions ("response"). This repository is then used to replay test scenarios by performing "request" and comparing System's "response" with stored one. Some test tools allows Test manager to monitor test progress.

This method can be used only in case that during development process user interface will not be significantly changed. This can be used as an advantage in case when UI change is considered as undesired effect (e.g. bug-fix releases of legacy systems).

2.1.2.2 Application built-in record & replay

This "robot user" is not the only way how to do record & replay style tests. When system architecture enables it, it's possible to build-in record & replay directly as integral part of system. There are several possible approaches - most common is shown on figure bellow:

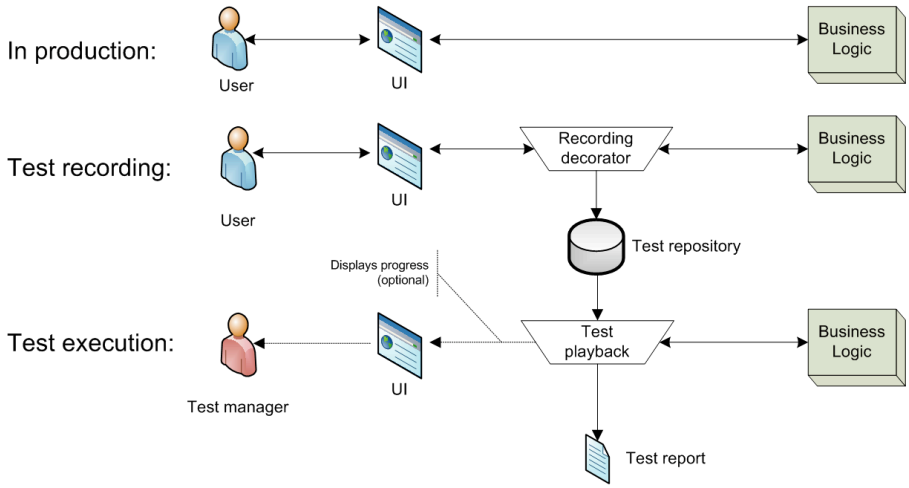

Figure 2 - Record decorator usage

This method uses recording decorator placed between system User Interface and Business Logic. This of course requires clear separation of these blocks in architecture and available API or hooks which recording decorator and test playback component can use.

Main advantage of this approach is that it can be applied to system regardless of technology of user interface - therefore ideal for projects preserving business logic and changing UI layer (e.g. migration from "fat client" to web-based UI). Is also more robust them "robot user" approach as the recording and replay is more closely tied with system itself and less sensitive to it's changes. When technology allows it (proper screen I/O hooks available), playback component can display test progress to Test manager on the UI.

Commonly used JEE application offers ideal separation of UI and business logic. Perfect place to put recording decorator and test playback component would be EJB container. Unfortunately JEE standard think of this option therefore it's up to use proprietary EJB extension (if available).

2.1.3 Critical success factors

When considering using record & replay method it's crucial to be aware of several success factors. [RR]

1. **Context independence** - system must be able to be configured to known starting point including data and relevant external system
2. **Means to initialize system** - test tool must be able to setup system to these starting point before every test run
3. **Functionality stability** - record & replay method delivers best results when majority of system functionality

remains unchanged during development cycle - as any test cases affected by changed/new functionality needs to be re-recorded and manually verified.

4. **User interface insensitivity** - tests must be recorded in a way that (planned) changes to the UI will not affect their replay and will cause false-negative (or - worse - false positive) results.
5. **Separate UI and business logic tests** - try to separate tests focusing on business logic (whose should be recorded in as UI-insensitive way as possible) and UI tests (to verify UI regressions). Different sets of tests with different sensitivity can be very useful to do "defect triangulation" (narrowing down whether defect is related to UI or business logic and in which specific component).
6. **Limited test's lifetime** - recognize that test cases, especially "robot user" test cases have a limited lifetime. They will not survive certain kinds of change of the UI or business logic. Therefore have prepared strategy, time and budget to identify those test cases and either discard, rerecord or replace them by different kind of tests.

2.1.4 Method suitability

Record & replay test method performs best in these scenarios:

- Refactoring legacy system where JUnit-style, hand-written tests, are too difficult and/or there is need to have regression tests in place.
- Hand-written test scripts are not feasible (too time/money consuming).
- There are not enough skilled people to create hand-written test scripts available.

Record & replay should be avoided when:

- Is not possible to fixate behavior of the system (freezing data which system uses).
- Behavior of system is expected to significantly change during development process.
- Automated tests should serve as part of system functional specification and there is no existing system that can be used for recording test cases.

2.2 Keyword driven testing

Keyword-driven testing (also sometimes known as Table-drive testing) refers to an application-independent automation framework. Such a framework requires the development of keywords and data tables (independent from the test automation tool which then executes them) and automation scripts (that runs the tested system).

Keyword driven testing distinguishes between two stages in test creation - planning and implementation. [KDTW]

2.2.1 Planning stage

This phase consists of create tables containing action (keyword), input data and expected result - all in one record. Table also provides any additional data that are needed as input to the system and (if suitable) the benchmark information that will be used to verify the state components or system in general.

Example of such a table is bellow:

Window	**Component**	**Action**	**Data**	**Expected value**
LoginPage	DomainComboBox	Select	CompanyDomain	
LoginPage	UserIDTextBox	EnterText	<username>	
LoginPage	UserPasswordTextBox	EnterText	<password>	
LoginPage	LoginButton	VerifyAction		"loggedIn"

Notice that data captured in table does say little what User Interface to be used and how test should be executed. Basically it captures what should be performed and what is expected result of these actions.

One of the advantages of this method is that keyword-tables can be automatically converted into plain-text which then can be integral part of software specification readable for business users. This may be enhanced further by having table translation table for keyword ("Action" in example above). E.g. second row from example above would be transformed into "On page LoginPage with component UserIDTextBox do enter text <username>". The resulting text may be created even more readable by implementing several simple heuristics - e.g. combining actions on same Window/Component into single sentence.

Entries in "Data" column which are in brackets (e.g. "<username>") refers to external data sets. This approach allows that

same test case will be run with multiple data sets. Example of such a data set is in table bellow:

LoginDataSet	
Username	**Password**
user01	PassWord1
user02	passW0rd

2.2.2 Implementation stage

This stage is dependent on specific framework and system that is used. Consists of writing scripts that will interpret keyword-tables, fill-in data from data-tables (where necessary) and verifies expected results (where wanted).

Bellow is example in pseudo-code of such a script activity:

```
Main record process module:
Verify "LoginPage" Exists (Attempt recovery if not)
Set focus to "LoginPage"
Verify "DomainComboBox" component exists. (Attempt
recovery if not)
Find "Type" of component "DomainComboBox" (It is a
ComboBox)
Call the module that processes ALL ComboBox
components
     (similar for "UserIDTextBox" and
"UserPasswordTextBox")
Verify "LoginButton" component exists (Attempt
recovery if not)
Find "Type" of component "LoginButton" (It is a
Button)
Call the module that processes ALL Button components

ComboBox Component Module:
Validate the action keyword "Select"
Call the Textbox.Select function with "CompanyDomain"
as parameter

ComboBox.Select Function:
Verify that first parameter is valid ComboBox option.
(Record an error if not)
Select this option

Button Component Module:
Validate action keyword "VerifyAction".
Call ComboBox.VerifyAction function (to be component-
instance specific)
```

```
Compare output value with expected output "loggedIn"
Record success or failure
```

Pseudo-code above mixes for easier understanding general implementation and execution from sample tables. In reality the code will be written independently from content of keyword- and data-tables.

2.2.3 Critical success factors

Theses are critical success factors for this method:

- **Test case development** (table filling) **and automation** (integration of test framework, writing test scripts) **must be fully separated** - it is crucial to separate these two activities. Fundamentally they require different skill sets (testers are not - and should not try to be - developers) but more importantly it's critical to keep test cases independent from underlying framework/system technology and on the other side preserve framework/system integration and scripts from being "intoxicated" by specif test cases.
- **Test cases must have clear and differentiated scope** - it is important that test cases have a clearly differentiated scope and that they not deviate from that scope. It's more useful to create several smaller, but single functionality-focused test cases then fewer complex ones
- **Proper level of abstraction must be selected** - test cases must be written at the right level of abstraction. For some cases it's better to write test cases at higher business level, some other at detailed UI component level or even to mix both approaches. It's important that test framework supports this level of flexibility

2.2.4 Method suitability

Advantage of this method lies in separation of roles in test case creation. Once testers create or learn keyword vocabulary, they can start designing tests without having any knowledge about test framework, scripting language or (to some extent) detailed implementation of tested system itself.

Testers can start developing tests (tables) without functional application - as long as preliminary higher-level requirements or design guidelines are determined. All the testers needs are reliable definition of basic interface structure (component types, main screens

definition) and functional flows definition from business perspective. Using these testers can write most of, if not complete, keyword- and data-table test cases.

Main challenge of using keyword driven testing approach is that initial phase is very time-consuming, especially when requirements tend to evolve during defining first flows and test cases. Once test process is established, initially heavy investment leads to very robust tests which can be used for regression testing even as new significant features are added to system or user interface is changed - all that is typically needed are modifications of test framework integration or slight tuning of test scripts.

2.3 Test Driven Development (TDD)

Test Driven Development (TDD) is related with boom of agile programming methodology - Extreme Programming (XP) concept. Target of TDD is to have complete system test coverage - every line of code and required system functionality to be covered by a test case.

There are two main underlying principles:

1. Never write a single line of code unless you have a failing automated test.
2. Eliminate duplication.

The whole development process consists of many short development cycles. [TDD]

2.3.1 Steps of TDD process

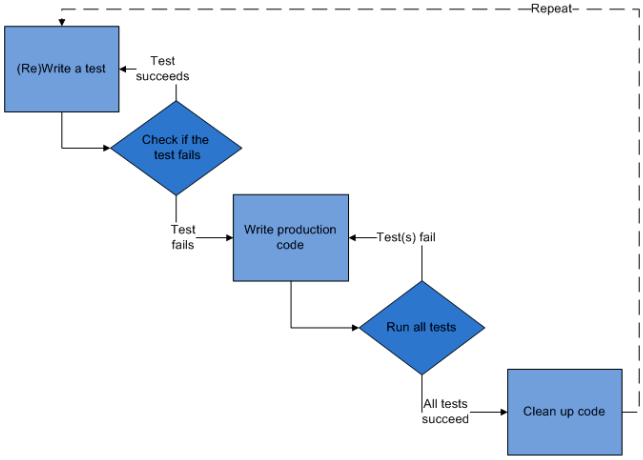

Figure 3 - TDD development cycle

The development cycle consists of these steps:

2.3.1.1Step 1 - Add a test

TDD cycle starts with writing new test case - either for new required functionality or code improvement write a test case. This test case must inevitably fail - if test succeeds then either required functionality exists or the test is contains bug.

Writing test case requires developer to have clear and detailed understanding of specification and requirements. Studying use cases or user stories covering requirements, expected exceptions and their handling can bring this knowledge. This is main differentiator of TDD from other software development & testing method - developer is forced to learn business-required system behavior before writing single line of code. This itself leads to code more aligned with expectations of business users.

2.3.1.2Step 2 - Run all tests and see if the new one fails

This step validates test case is written correctly and to lower risk of false-positive test results (test pass even when expected functionality is not present or working correctly). This raises confidence in quality of test cases.

2.3.1.3Step 3 - Write some code

Next step is for developer to write some code that will implement required functionality and pass the test. Written code is expected to be crude and not "elegant" or in alignment with code quality standards in place. This is accepted as it will be improved and in step 5.

Very important point is to write only and only code related to failed test case. Developer must not implement neither any additional functionality nor improve code for which doesn't exist failing test case.

2.3.1.4Step 4 - Run the automated tests and see them succeed

In this step developer checks all test cases to be confident that written code passes all tests and doesn't have any negative side effects (causing some other test cases to fail).

2.3.1.5Step 5 - Refactor code

Final step is to clean up written code according to code quality standards, adding proper logging etc. One of the important concepts is to remove duplicated code, which may be introduced, but more importantly to remove any "magic numbers" (or strings) that were introduced in code to pass test cases and replace then with references to real testing data sets.

Refactorization (clean-up) of test case written in step 1 is also being done in this step.

Whole process of course consists of re-running all test cases (as many times as necessary) to make sure that any changes doesn't cause any test to fail.

2.3.1.6Repeat

With each development cycle systems is pushed closer to desired overall functionality. The size of each cycles should be relatively small (rule of thumb - 1 to 10 edits for each cycle). Agility is the focus - instead of extensive debugging it's proffered to revert changes and either try different approach or retry implementation in some later cycle.

When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself (unless testing library is the goal - e.g. library is suspected to be buggy) but instead each cycle should add new functionality to overall system.

2.3.2 Critical success factors

Theses are critical success factors for this method:

- **Management support of TDD approach** - it's important to gain understanding and support of entire organization, esp. of management, how and why TDD steers development process. Without that it may seem that too much time spent on writing test cases that often fail and need to be rewritten.
- **Focus on tests' quality** - test case quality and their overall maintenance must become integral part of project overhead. Low quality tests often generating false negative reports will tend to be ignored by developers and real failures may not be detected. Tests should be also written for as easier maintenance as possible - e.g. re-using (sharing) error strings, usage (creation) of libraries. This is done mainly in step 5 (Refactor code).
- **Oldest tests are most precious ones** - adding tests in later TDD cycles for already developed functionality in required coverage and depth is very difficult and almost never done. Therefore the original tests created in early development phases are becoming more precious as time goes by. If a poor architecture, a poor design or a poor testing strategy leads to a late change causing many of

existing tests fail, it is crucialthat they are all fixed - instead of deleting, disabling or crudely alterning them without understanding their full scope and impact.

- **Test's blind spot** - tests as usually written by same developer who writes the code that is being tested. Code and test therefore may share same blind spots - e.g. certain input parameter values are not being tested/checked, misinterpreted specification will lead to both wrong test case and wrong code. Therefore it's important to incorporate into project additional review methods - e.g. peer programming, code/test review audits.
- **False sense of security** - TDD method leads to creating large number of passing unit tests. This may bring false sense of security which may result in underestimating other QA activities like integration testing, UI usability testing etc

2.3.3 Method suitability

TDD is ideal method for new greenfield-type project because it relies on having every implemented functionality covered by test case. Therefore it's not suitable to use TDD for evolution of a system that was not so far developed by TDD method.

TDD usage is closely tied with extensive use of a suitable version control system (SVN, GIT etc.). When test(s) fails unexpectedly, reverting the code to the last version (that passed all tests) is in many cases more productive then extensive debugging.

TDD impact on overall system development process is larger then changing code and test development methods. It also help to steer system design. Focus on test cases from before a single line of system code is written leads to focus on designing how users will use each functionality - therefore developer has also to think about interfaces (either UI or system integration) before their implementation.

2.4 Behavior Driven Development (BDD)

BDD is approach heavily based on TDD. Is sometimes described as second generation of agile development & testing methodology. Due to many common aspects with TDD I will not elaborate this method in detail, instead focus on difference between TDD and BDD.

Main focus of BDD is on obtaining clear understanding of expected system behavior based on discussion with stakeholders (typically, but not exclusively, target business users). Test cases are captured using more natural language (then in TDD) that is easier to read for non-developers. BDD encourages developers to use their native language in combination with domain driven design language and thereby minimizing translation between "technical language" (according to which is code written) and domain language (used by business, users, project management etc.).

2.4.1 BDD method principles

BDD method uses these main principles:

- **Establish** (and document) **goals** of different stakeholders required for a envisioned system implementation.
- Based on these goals **draw features** necessary **to achieve these goals** using feature injection methodology (see bellow).
- Involve stakeholders into the whole system development process.
- Use examples to describe behavior of system (or set of components).
- Convert these examples into automated tests to provide quick feedback and regression testing.
- In describing system behavior and test cases, use word "should" (or "should not") - instead of verb "test" (as in TDD) - e.g. write "Should fail on invalid email address format" instead of "Test validity of user email format". This alone leads to better understanding of expected behavior and also helps to understand when test case is no longer valid from business perspective. [BDDI]
- Use word "ensure" to describe direct responsibility of system. It's crucial to differentiate desired outcome of component(s)/code in question from a side-effect (e.g. from other elements of code/component).
- Extensive use of mocked-components as temporary stand-ins for not yet existing parts of system.

2.4.2 Feature injection

Organization typically got several visions how to deliver a value to it's business - how to make, save and/or protect money. Once primary stakeholder indentifies vision as suitable to current (or

expected) conditions for implementation, he brings in additional incidental stakeholders.

Each of the stakeholders has different sets of goals how to achieve this vision - e.g. regulatory requirements from compliance department stakeholder, integration with social networks from marketing dept. These goals are then transformed into broad themes or feature sets defining how to achieve them - e.g. "audit every transaction", "public API". These can be then elaborated into further details describing features, user interface requirements etc.

Following this process ensures that every system requirement can be tracked to specific goal and vision. This is crucial especially when prioritizing features to be implemented (or dropped to over-time/over-budget issues).

2.4.3 Method suitability

BDD is as TDD suitable ideally on greenfield projects where development starts more or less from scratch. As BDD is focused even more then TDD on involvement of business, it's necessary that whole organization including all stakeholders is willing to embrace this development style. It's not suitable for projects where business users expect just to approve set of system requirements and then having system delivered as "turn-key solution".

For business-oriented users and stakeholders BDD brings visibility into the whole system development process and enables them to participate on steering it.

3 Other testing methods

In chapter 3 I described methods focused on creating test cases and test data based on detailed knowledge of system functionality, often very closely tied with development of tested system itself.

In many cases there is need to test system without these detailed knowledge. Reasons may range from simply validation testing to detect any potential blind spot over penetration testing to black-box testing where testers are unable to gather any information about system's internal implementation.

3.1 Fuzzy testing

Fuzzy testing method lies in feeding tested system with invalid, unexpected, pseudo-random or random input data. System failure can then reveal hidden defect. By failure is understood system crash, failure of built-in assumptions or any other unexpected behavior (memory leak, infinite loop etc.). Failure doesn't mean rejecting invalid input and raising proper error message.

Theoretically can be fuzzy testing used to detect false-positives (system accepts invalid input which should have been rejected) but practically it's very hard to detect these situations. For this tests is more suitable boundary testing.

3.1.1 Generating test data

Simplest approach is to send to system random set of bits as input. Due to extremely large of possible inputs (exponential to the length of input) there are typically used some heuristic methods like randomly mutating known (valid) input.

Most successful usage of this method comes from using knowledge about input structure, typically based on specification. This involves creating model-based test data generator on specification walk-through and then adding anomalies in the data contents, structures, messages, and sequences. This "smart fuzzy testing" technique is also known as robustness testing, syntax testing, grammar testing, and (input) fault injection.

Another options is to capture input data to an existing system (like record phase in Record & replay method) and heuristically try to recognize internal structure. As example of this heuristic can be Sequitur [SEQ] which tries to built a grammar from any input stream.

3.1.2 Minimizing failure data set

One of the main problems with this method is in case of detecting a failure to find out what is minimal input data set causing this failure. Minimal data set is important for developers to be able effectively localize and fix the bug.

For solving this issues were developed several algorithms and tools. Most commonly used one is named "Delta" [DEL]. Delta uses this algorithm:

1. Split input into atomic elements (e.g. lines of code)
2. Set whole input as group.
3. Split every current group into 2 groups.
4. For every group try if after removing it is input still "interesting". If yes, remove the group from input permanently.
5. Repeat steps 3-4 until group contains only single atomic element.

Core idea of the algorithm lies in "outsourcing" of the detection of "interestingness of input". In our case it's if system on given output fails. Theoretically algorithm can be improved by adding heuristics what can be removed "easily" (comments) and what must be removed in blocks (e.g. removing variable definition should lead to removing all uses of the variable). Nevertheless even such simple, brute force algorithm is surprisingly effective and effort given into given into implementing such heuristics doesn't usually pay off.

3.1.3 Method suitability

Fuzzy testing usually tends to find simple failures in the system. It's typically used to increase robustness of system by non-functional tests, especially for system exposed to Internet. Fuzzy testing can help to detect potential DoS holes, memory leaks and other stability issues.

3.2 Boundary value testing (BVT)

BVT method uses black-box approach to generating test cases. Main assumption in BVT is that the majority of program errors occurs at input (or output) boundaries - places where the algorithm/formula/data manipulation must change the behavior in order to system to produce correct result.

3.2.1 Single variable BVT

The basic procedure for BVA starts by identifying the boundary values, typically from the input point of view. All of these boundary values will be incorporated into the set of test cases. In addition to those values, values near the boundaries will be tested. In addition to the boundary and boundary-adjacent values, the BVT method also includes some nominal value of input (or output) in the test case data set. The baseline BVA procedure is best illustrated by the following example.

Consider a system with a single input variable *N* that has an output defined only for values of *N* in the range $a \leq N \leq c$. The test case data set would be at minimum the set of values baseline = *{a, a+, b, c-, c}* where *a+* is a value just greater than *a*, *c-* is a value just less than *c*, and the value *b* is some nominal value that lies between *a+* and *c-*. If error handling is critical to the software under test BVT will extend value by values outside the allowable range. The baseline tests identified above are augmented with the values *{a-, c+}* where *a-* is a value just below the minimum acceptable value *a* and *c+* is a value just above the maximum acceptable value *c*. [BVT] This is shown on figure bellow.

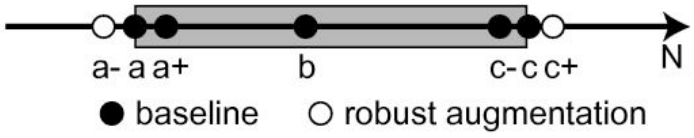

Figure 4 - BVT - single variable, single range, robust value set

In case input consists of multiple data subranges then also boundary and boundary-adjacent values on the subrange boundary are added to value set - for example see figure bellow:

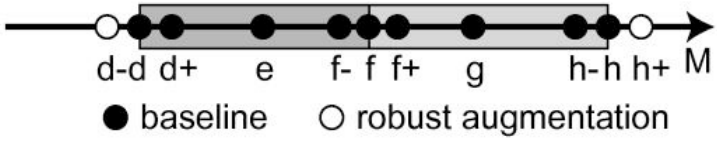

Figure 5 - single variable, multi range, robust value set

3.2.2 Multi variable BVT

In real world we almost never meet system with just single variable. Systems consists of multiple input variables. In this case then we construct value set for each variable (e.g. *N* and *M*) and then to gain final value set we create *Cartesian product (M x N)*. Example of such an situation can be clearly seen on figure bellow:

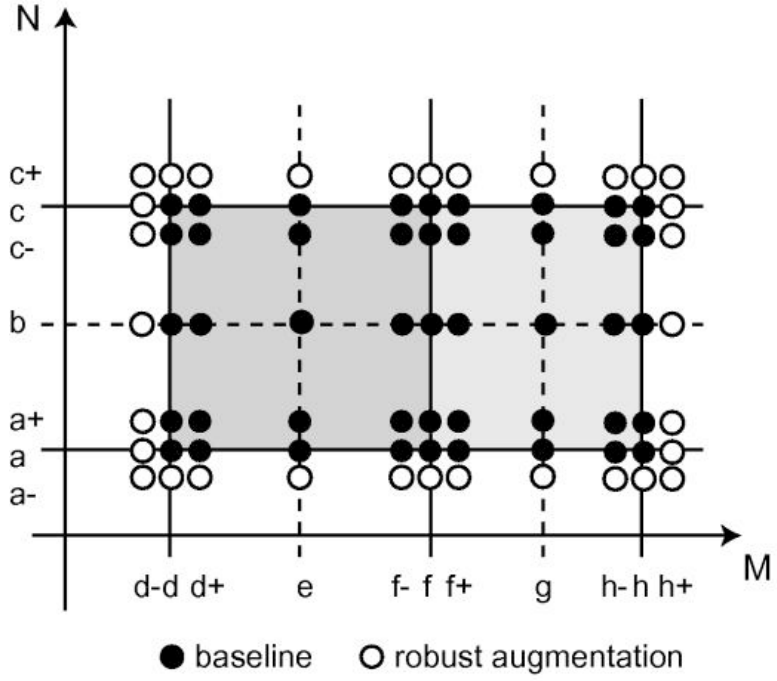

Figure 6 - multi variable, multi range, robust data set

3.2.3 Modification for string variables

BVT can be easily applied on systems expecting numerical inputs but with small modification it may be also used for some types of string inputs. Modification lies in describing each string input into several numerical attributes. E.g. string input field expecting email field may be described using these numerical attributes and their range values:

- Total length (5-100)
- Length of TLD (2-5)
- Contains "@" (0, 1)
- Position of "@" (1-96)
- Contains "dangerous" characters like `, ', " ; (0,1 for each character)
- Number of characters with diacritics (0 - 100)

For each value set of these attributes, constructed with algorithm described above, can be then generated random string (or multiple strings) which will server as test input.

3.2.4 Method suitability

The mechanical nature of the procedure and the symmetry of the value sets identified make the BVT easy to remember and use. Number of test cases can be easily quantified and resources properly

allocated and planned. Method works best on systems expecting only or mostly numerical inputs.

Method should be used in combination with other (preferably white-box) methods. If used alone then to verify robustness against black-box tests (or attacks) or to verify expected boundaries for input variables.

Note that BVT doesn't deal with test result evaluation - how to detect if test failed or passed - at all and leaves this issue to be solved independently. Possible ways how to deal with it range from manual verification (suitable only for small test sets) through modeling system behavior (not suitable for complex systems) to running tests on legacy system (if available).

4 Summary

In this essay I tried to present and compare commonly methods for automated software testing. I tried to focus on how they deal with construction test cases, test data, execute tests and evaluates results.

As may be expected there is no "one size fits all" solution. Selection of proper method depends on many external factors like type of project (greenfield development vs. maintenance release of legacy system), time, budget and skills available. Also organizational structure and company "style" has to be considered - larger organizations with conservative waterfall development approach need different methods then agile start-up company with focus on rapid release delivery.

By selection automated testing method is whole process just initiated. Proper processes must be put in place, tools to support them selected and installed, staff educated.

5 Bibliography

[RR] "Agile Regression Testing Using Record & Playback", Gerard Meszaros , Ralph Bohnet Jennitta Andrea, 2003

[KDTW] "Keyword-driven testing", Wikipedia, http://en.wikipedia.org/wiki/Keyword-driven_testing

[TDD] "Test driven development", http://c2.com/cgi/wiki?TestDrivenDevelopment

[BDDI] "Introduction BDD", Dan North, http://blog.dannorth.net/introducing-bdd/

[SEQ] "Sequitur - inferring hiearchies from sequences", Craig Nevill-Manning, Ian Witten, http://www.sequitur.info

[DEL] "delta - Heuristically minimizes interesting files", Daniel S. Wilkerson, Scott McPeak, http://delta.tigris.org/

[BVT] " A Review of Boundary Value Analysis Techniques", Dr. David J. Coe, CrossTal, April 2008