

TESTING TECHNIQUES BASED ON UML

Celia Gutierrez, Universidad S. Pablo Ceu, Campus Montepincipe, Boadilla del Monte,
28668 Madrid, Spain

cgutierrez.eps@ceu.es

Universidad Antonio de Nebrija, Campus Dehesa de la Villa, Pirineos 55, 28040 Madrid,
Spain

cgutierrez@nebrija.es

Abstract

The simple generation of test cases for a software application is an ancient dream of software developers. During the history of software testing research, there have been some attempts in achieving this dream, by promoting tools and developing environments that can help in the automatic generation of tests from different types of descriptive information of the application. Nevertheless, it is common that neither the necessary descriptions to implement techniques nor the proper environment, are regularly used in development projects due to the format, frequency of use or the toughness of the applications in those developments.

In this communication another way of test validation is described, which is highly recommended due to its simplicity and trustfulness. It is based in the UML methodology and it consists that the test cases of an application can be obtained through the application activity diagrams (which can be obtained from the use case specifications) in such a manner that all possible ways must be covered by the test cases.

Keywords: Activity Diagrams, Specification, UML, Test Cases.

1 INTRODUCTION

Software testing is a classic technique for software verification and validation, alongside some other alternatives of evaluation, such as revisions, audits, metrics, etc. Software testing can be described as “the execution of a program with the objective of detecting mistakes” Myers G.J. (1979). For this reason, the design of the testing is based on the creation of test cases whose execution will detect possible symptoms of errors. A test case can be defined as “the group of entries, conditions of execution and expected results developed for a particular objective, for example, to exercise a concrete way of a program or to verify the accomplishment of a determinate requirement”, in IEEE (1990).

All the capacity of the detection of mistakes of test cases is based on the consideration of any type of discrepancy between the obtained and the expected exit (related to the instructions in the specification) such as symptoms of a problem in the software. In consequence, all these design techniques of test cases are based on taking the specifications as a reference of the application’s behavior. Logically, it is important that the specification has been validated with the greatest care using revisions to detect possible mistakes from Yourdon E. (1985):

- Completion.
- Coherence.
- Exactitude.

- Quality.

In general, it is recommendable that these and other qualities, like it is said in IEEE (1998,a), will be present in the specifications considered as references for tests.

Software test techniques assume a series of demonstrated principals:

- The two principal philosophies of designing are the white box and the black box. The best results normally happen when we combine the advantages of both, Myers G.J. (1979).
- It is inevitable to propose exhaustive tests of any application (as in Myers G.J. (1979) and Beizer B. (1990)) due to the fact that it is impossible to prove all probable entries and/ or situation of function in reasonable periods and amounts.
- In consequence, it is very important that the test design is based on the selection of some entries or situations to exercise (from all possible cases). Their behavior will be representative of other similar cases (Myers G.J. (1979), IEEE (1998,b)). The choice criterion (coverage criterion) normally endeavours that the percentage covers a representation of the possible universe of structural elements or functional situations. In fact, some authors argue that the selection of trial cases is more important than the design of those cases, Choi E.M. and von Mayrhauser A. (2000).
- Finally, nothing can stop the design and the planification of tests beginning in parallel to the specifications. Not only is this possible, but it is also recommended.(IEEE (1998,c), Paul M.C. et al. (1993), SEI Consejo Superior de Informática (2001)).

In our proposal, we wish to approximate some methodological views with the highest degree of implementation in the projects of professional and commercial development. In this way, the UML implementation Booch G., Rumbaugh G. and Jacobson I., (1992), as a notation of objects oriented development has allowed the diffusion and real implementation of techniques of software specification like the use cases.

2 UML'S SPECIFICATIONS BASED ON TEST CASES

Proposed by Jacobson Booch G., Rumbaugh G. and Jacobson I., (1992), use cases are an essential part of the UML notation for the specification of the external behavior of a system. A use case is defined as a "coherent unit of functionality [...] manifested by sequences of messages exchanged by the system and one or more external actors with the actions developed by the system" Jacobson I. (1999). It supposes a good expression of the interactions of actors with the system to obtain an observable and valuable result for them Booch G., Rumbaugh G. and Jacobson I., (1992). As specification the use cases are especially descriptive when they are complemented with activity diagrams.

An activity diagram is a special case of a state diagram. This diagram can be defined as a model that shows the union of states and object lives in an application, with the other changes that allow passing from one state to another one. In activity diagrams, almost all states are action states (identify what action is executed when we use them) and almost all the transactions are activated in the executed action final in the previous state. It allows detailing the use case behavior of an object or a method in an object. A use case description (in a standard format Object Management Group (1999)) can be expressed with an equivalent activity diagram. We show the example of *user deletion* use case.

2.1 Objective

Complete deletion of a user's information from the application database.

2.2 Normal event flow

Actor	System
1. He demands user deletion	2. It demands user's ID
3. He introduces user's ID	4. It shows user's information and asks for deletion confirmation
5. He confirms user's deletion	6. It shows confirmation for user deletion

Table 1. Event Dialog for User Deletion use case

2.3 Alternative event flow

In step 4, if ID does not exist, it shows an image and it returns to step 3. In any case, if the actor selects Cancel, no option is performed and it returns to the main menu.

2.4 Preconditions

The user we want to delete must exist.

2.5 Postconditions

User's information deletion from the application database.

From now on, we will assume a system specification that uses use cases and employs an activity diagram to better explain the behavior in each case and prove the specification. In this communication we aim to show how this type of UML specification can enable a quick generation of scenarios of tests.

3 CRITERION OF TEST GENERATION FROM USE CASES: TYPES OF SCENARIOS

The use of use cases as a base for test cases design has been studied in several conferences and publications. In fact, even in the fifth edition of the JICS, an idea for the acceptance tests Schneider G. and Winters J.P. (2001). was presented, where the use cases were complemented with the study of combinations present in Myers G.J. (1979).

In our case, we propose to adopt a choice criterion of test cases based on the exploration of the distinct scenarios of execution of each use case. In an activity diagram, we can assume that there are several types of scenarios defined by each way, that can be drawn starting from the initial state. The number of different ways in an activity diagram is normally very high. We must consider that a way can be different from another simply by the fact that it "walks" in a cycle of the activity diagram a distinct number of times. For example, a way in the previous example draws the following sequence of events:

- Way 1: It asks for the user deletion, the system requires a user identification, the identification is introduced, the system proves that it isn't correct (not correspond to any user that is in the data), a new identification is introduced, it shows the data, it confirms its deletion, the system shows the confirmation and the case ends.
- Way 2: It asks for the user deletion, the system requires the user's identification, the identification is introduced, the system proves that isn't correct (not correspond to any user that is in the data),

the system proves once again it isn't correct (not correspond to any user that is in the data), a new identification is introduced, it shows the data, it confirms its deletion, the system shows the confirmation and the use case ends.

Obviously, the actor has failed twice on giving to the user's identification. Not only can the number of different ways of a diagram be very elevated, but also inside each way (type of scenario) can we distinguish multiple scenarios of possible tests simply considering all the possibilities of data values for possible entries. For example, in the previous case, if the user's identification corresponds to a six-digit integer number, we would have at least one million options, all of them valid values (the possibilities of admissible invalid values are virtually infinity: introducing characters that aren't numbers, etc.). As indicated in the introduction, due to the fact that proving the software behavior in all situations is nonviable, we must select the test cases by electing the ones that result representative of other equivalent ones.

In our case, the first criterion to select cases will be based on the draw of ways through the activity diagram. It will be similar to the one that is proposed with other types of diagram in Booch G., Rumbaugh G. and Jacobson I., (1992). To achieve this, we will draw on the theory of graphs and we will use the analogical criterion which McCabe Fernández J.L. (2000) used with the flow graphs of the procedural code. We must remember that the criterion of structured tests demands that a minimum group of representative ways should be tested (lineally independent, due to the thought of McCabe) out of the possible total group. This group should assure that the draw of execution has been in all the arrows or graph edges and it is equivalent to the traditional coverage criterion of decisions contemplated in Myers G.J. (1979).

Thus, our first criterion of tests coverage based on a use case is expressed as follows: "A use case, through its activity diagram, will be tested enough if the used ways (types of scenarios) allow them to pass through all the arrows or transitions of the activity diagram at least once". The thought of this criterion is related to the intention that all the options of behavior contemplated in the use case, through its activity diagram, have been tested at least once: in other words, we don't only test the "happy day" flow of events, but also the alternative flows of the case. Equally as in Fernández J.L. (2000), it would be possible to put in function all the classic formulas to calculate the maximum number of ways (types of scenarios), applicable to every type of convex graph, that we need to accomplish that coverage criterion.

The formula to obtain the maximum number of independent ways of the G graph, $V(G)$, are the following:

$V(G)$ = number of regions.

$V(G)$ = $P+1$.

$V(G)$ = $A-N+2$.

P is the number of predicated nodes (nodes with more than one exit arrow), A is the number of arrows or flow edges and N is the number of nodes or circles. The regions are close areas limited by flow edges considering the area that circles the graph as a region. In the previous case, we see that $V(G)=4$, something that can be intuitively previewed when you see the graph. The four independent ways are the following:

- a. 1-2, 3-4, 5-10-1...
- b. 1-2, 3-6-8-9-10-1...
- c. 1-2, 3-6-7-9-10-1...
- d. 1-11

In the case of the activity diagram, the calculus makes us limit the number of ways (types of scenarios) to 4, that in practice we could condense in the following classes of interaction:

- Ask for "deletion", requires ID and cancel.
- Ask for "deletion", requires ID, introduces an incorrect Id, introduces a correct ID, confirms "deletion".
- Ask for "deletion", requires ID, introduces a correct ID, cancels.

- Ask for “deletion”, requires ID, introduces a correct ID, doesn’t confirm “deletion”.

4 CRITERION OF TEST GENERATION FROM USE CASES: BLACK BOX

In each pathway of the activity diagram (type of scenario) there could be too many different scenarios to assure the representation of all of them in a single test case.

Following the traditional recommendations of tests design forms for the black box Myers G.J. (1979), we should make an analysis of the domain of entry values to detect group/classes of data that have a homogenous behavior in the system. In other words, the application uses all the group values in the same way, so if the system fails when it has been tested with data, we can assume that it would fail when we introduce another value from the same class.

This technique is called equivalence classes. Some of them are composed of valid entry data and other invalid data. For example, if a data specifies a range (for example, values between 0 and 99), we could have two equivalence classes that are invalid (values out of the limit of the range, in the top, higher than 99, and in the bottom lower than 0) and another valid one (values between the range).

Once equivalence classes are found, it is recommendable to choose the values that are part of the limit of the class instead of testing any one because they have a higher possibility of detecting mistakes.

Thus, our second coverage criterion of tests based on use cases is expressed as follows Myers G.J. (1979): “For each data entry in one use case, we should accomplish that all the equivalence classes that are valid would be exercised by a test case and that each invalid class would be tested alone in a specific case, accomplishing that all the limit values would be exercised”.

The combination of both criteria in one activity diagram is easy using the elements of representation contemplated in the UML’s notation. So, in the steps of one use case that supposes data entries, we will not use the concept of action, but activity. In our case, each step of data introduction will become a state of activity whose sub diagram should express the different options of test based on equivalence classes and limit values. In the example of “deletion” of the user, the design of equivalence classes for the introduction of the user ID supposes the creation of a table as the following:

Restrictions of entry	Groups of valid data	Groups of invalid data
ID of user	Is number (1) Has 6 digits (2)	Is not number (3) Higher than 999999 (4) Lower than 000000 (5)

Table 2. Class table to identify the use

The cases that should be generated from table 2 should assure that all the classes have cases with all their limit values contemplated. An example of these cases and the classes that are contemplated are the following:

999999 (1)(2).
000000 (1) (2).
A546B6 (3).
1000000 (4).
-000001 (5).

To achieve this, it is important to transform the original activity diagram, which simply represents the structure of use flows. The new diagram should contemplate one activity in all the steps producing data entries and a sub diagram with the options structure of the corresponding cases.

5 TEST CASE GENERATION METHOD

Contemplating the two coverage criteria expressed in our proposal, the test case generation method will be done in the practice in the following order:

- Generate for each use case from the description of its interaction, one diagram of primary activity where the ways of flow of events are marked.
- Identify the introduction of data in each flow and analyze for each case the equivalence classes, limit values, and how we use the combinations (as in Schneider G. and Winters J.P. (2001). taken from Myers G.J. (1979)).
- Transform each action of entry data in a state of activity that contains as sub diagram the options of entry and of using them, generating the definitive diagram.
- Contemplating the definitive diagram, accomplish the two criteria of coverage in such a way that all the arrows and edges of the diagram can be exercised at least once.

Starting from a graph it is easy to have a program that can give us automatically both the number of existent ways and the description of them. The algorithms that exist are very varied although, as in the trials of the white box, they can be supported by the regular expressions that characterize the distinct typological structure of the flow graphs McCabe T.J. (1976), Shooman M.L. (1983). Due to the fact that these algorithms are clear and well defined, the construction of programs that can make this task is immediate. In our case, this program is already designed, even though it has to start from a specific introduction of the topology. Currently we are evaluating the possibility that the structure of the diagram can be directly imported from CASE tools of huge diffusion (Rose, Tau, etc.).

Evidently, as in the proposals of structural tests traditionally of McCabe T.J. (1982), it is important to consider the following points for a correct practice application:

- Due to the fact that the coverage must be performed by forming ways of execution through the definitive diagram, it is important to control that the formed ways can't be executed because they combine incompatible results. In this regard, the possibility of applying stereotypes to the different actions of the diagram, would make the automatic detection of these impossible ways easier (for example, to detect that two error processes in the same test case, following the recommendation of Myers G.J. (1979)).
- Each test case should incorporate both data and events entries, and the expected exit according to the specification.
- With the objective that the control of the requirements that must accomplish the application becomes more effective, we are currently analyzing the possibility that the method of generation of cases/ scenarios will be integrated in tools of requirements like IrqA, on which we have been working for some time through the agreement with TCPSI. In this tool, as recommended in the bibliography of tests and in the methodologies and process of development in general (see chapter 1), it is convenient to start tests from the establishment of the software requirements. This type of tools allows us to describe scenarios for the software tests that we are specifying.

6 CONCLUSIONS

We can learn a lot from the experience of carrying out the independent tests in management applications that, for a given environment, were supposed to have well documented specifications. The specifications of structured types have proven inadequate in controlling acceptance tests as has been shown in the above mentioned projects where significant detection of mistakes was not detected. The mistakes were detected only after new and costly analysis. The use of techniques more suited to today's developments, such as the UML use cases and specifications described in this paper, has an

added advantage, which is a very reliable description of a group of tests appropriate to the system. This generation of test cases can be even more productive through the development of automatic methods from the documentation of diagrams in CASE. We can't ignore the fact that the obtained case descriptions could be transformed, with a little development, in scripts that would feed a tool of GUI replay for tests.

The author wants to thank for the financial support by the Spanish Ministry of Education and Science (MEC) through the TSI2005-00986 Research Project.

References

- Myers G.J. 1979. 'The art of software testing'. John Wiley & sons.
- IEEE. 1990. 'IEEE Std. 610 Computer Dictionary'. IEEE.
- Yourdon E. 1985. 'Structured walkthroughs'. Prentice- Hall .
- IEEE .1998. 'IEEE Std. 830 guide to software requirement specifications'. IEEE.
- Beizer B. 1990. 'Software testing techniques'. Van Nostrand Reinhold.
- IEEE .1998. 'IEEE Std. 829 for software Test documentation'. IEEE.
- Choi E.M. and von Mayrhauser A. 2000. 'Testing object- oriented systems using extended use cases'. *PDPTA Conference*, vol.2.
- IEEE.1998. 'IEEE Std. 1012 for software Verification and Validation'. IEEE.
- Paulk M.C. et al. 1993. 'Capability Maturity Model For Software v. 11'. CMU/ SEI-93-Tr-25.
- SEI Consejo Superior de Informática.2001. 'Métrica versión 3. Guía de referencia'. MAP.
- Booch G., Rumbaugh G. and Jacobson I. 1992.'The Unified Modelling Language'. Addison- Wesley.
- Jacobson I. 1999. 'Object-oriented software engineering'. Addison- Wesley.
- Object Management Group. 1999. 'OMG unified modeling language specification. Version 1.3'.
OMG, June.
- Schneider G. and Winters J.P. 2001. 'Applying Use Cases. A Practical Guide'. Addison- Wesley.
- Fernández J.L. 2000. 'Utilización de casos de uso en las pruebas de aceptación'. *V Jornadas sobre la Calidad del Software*: 65-76.
- McCabe T.J. 1976. 'A complexity measure'. *IEEE Transactions on software engineering*, 2: 228-363.
- Shooman M.L. 1983. 'Software engineering: design, reliability and management'. McGraw- Hill.
- McCabe T.J. 1982. 'IEEE Tutorial: structured testing'. IEEE Computer Society.