The effectiveness of testing models

Janis Bicevskis

University of Latvia
Rainis Blvd. 19, Riga LV-1459, Latvia
e-mail: bics@lanet.lv

This paper presents analysis of statistics concerning concrete, object-oriented program system testing procedures, with the goal of evaluating the effectiveness of various testing technologies. Specific testing models are presented as an alternative to traditional models: data base management, system object management, user interface, and calculation checking. The results of different kinds of tests are compared: in one case they are tests which are performed "intuitively", i.e., without a concrete testing scenario; and in the other case they are tests which are performed systematically, i.e., according to previously fixed testing models. The statistics clearly prove the advantages of systematic testing, and they suggest various proposals on improving the testing models

1. History

The traditional approach to the testing of program systems has involved the following scheme: The programmer chooses the values of input data – tests on which the program is executed. The programmer estimates the correctness of the program on the basis of the operational results. By repeatedly choosing different tests and controlling the results, the programmer can ensure the correctness of the program. The essence of the problem lies in choosing a series of tests which:

- Discover errors properly (i.e., which allow the researcher to spot weaknesses in the system with just a few tests, pointing out the places where the system does not meet specifications, if any);
- Involve criteria that report on the adequacy of testing when all errors are discovered at a high level of trust.

Over the course of time, various methods have been offered to solve this problem. One of the first criteria to be developed was called C1, and all feasible branches of a program can be executed on this test set. This, as well as another structural criterion [Beiz 95], in which information about the structure of the program or its data are utilized, are usually used for the testing of individual modules. Ideas about structural testing which have been worked out by theoretical researchers have been applied in practice since the early 1990s. Most of the tools which are used for this purpose allow for the automated establishment of a control flow graph from the

text of the program and thus to check the completeness of the testing procedure – whether all branches on the graph have been covered. Unfortunately, these structural methods face significant difficulties in testing object-oriented systems:

- The structural methods usually utilize information from the text of the program, which characterizes the implementation of the task and is not essential from the perspective of the task itself;
- Structural testing methods are laborious and may not be appropriate for many situations;
- Structural methods are oriented toward the testing of individual modules, while in an object-oriented system it is much more important to test the cooperation among class methods ([Over 95]).

It should be added that as far back as the early 1980s there were some methods which were based on more sensitive testing criteria – i.e., testing not only by performing all edges of the control flow graph, but performing them from various states of the program ([Bic 79]), where "state" is understood to be the minimal information which determines the further execution of the program. Because they are quite complicated, these criteria were not widely used.

A radically different approach has been taken to functional testing, when the functions of the system that is being developed are checked without use of information about the structure of the program. This approach mirrors what is needed in practice, because the testing can be done without the presence of developers and in accordance with the specification of the program system or the user guide.

Functional testing is well-supported by the tools of testing automation, which allow for the accumulation and repetition of tests. The majority of testing support tools which have been developed contain various features such as:

- An accumulation of tests, both through writing test scripts and through executing programs and recording user input;
- Repeated execution of updated programs through the use of tests accumulated in libraries;
- Forming of testing models from the text of the program or its specifications;
- Checking the completeness of the test on the basis of previously determined testing models.

Practice shows that the methods of structural testing are good for the testing of modules, while the methods of functional testing are appropriate when it comes to module integration and the acceptance testing stage. The comparison [Beiz 97] offers the use of structural and functional testing methods, pointing out that structural methods are used in 80% of cases involving module testing, while functional methods are used in up to 80% of cases involving acceptance testing. It must be said, however, that in the theory of testing, one fundamental question about the choice of testing criteria has not yet been answered.

It is the contention of this paper that it is impossible to use one universal testing criterion to check different components of a system. The developed system must be tested in the basis of the appropriate type of qualities: the correctness of the data base, response of user interface to appropriate requirements, correctness of object processing and calculations, etc. Besides, it is not only all of the possible operations of the object that must be tested, but also all of the operational pairs. This principle is supported by considerations which have already been reviewed [Auz 91]: If an operation is applied to an object only once, the object without any question enters a

new "state". This means that it is necessary to apply the operation repeatedly to the same object and from the new "status", hoping that the repeated application of the operation will not create new conditions. For example, when the editing window of a concrete object is opened, the system enters a new status, but if a second editing window is opened in the same object (provided that the system permits this), no new status is created. This principle has been tested through functional testing of a specific, fairly complicated system.

On the basis of the specifications of the system that was being developed, several testing models were chosen, in concert with which the programs were tested within the group which did the developing. The selected testing models allowed for a test of only some of the system's functions, albeit those which the project managers thought were the most important ones. The testing of the rest of the system functions was done by the programmers, and they did not use testing models. The tested programs were later turned over for independent repeat testing by a quality control group and by system users. The repeated testing showed that:

- The discovered errors mostly applied to those system functions which were tested by programmers without testing models, and this affirms the need for systematic testing instead of intuitive testing;
- Testing done according to the "depth 2" principle was, at least in this case, sufficient to reveal errors properly.

This means that the main result of the project was the accumulation of experience in the testing of object-oriented program systems with specific testing models.

2. A brief description of the system that was tested

The effectiveness of the testing models was reviewed on the basis of statistics about a specific system – Mosaik 5.0. This is a static calculation system with which information is stored about the various elements and costs involved in a major project. Users of the system enter the various parts of a large project, along with numerical calculations about their duration, costs, participants, etc. Graphs are designed to show the sequence of tasks that are to be performed, providing alternative ways of doing the job (OR branching), or parallel constructions for carrying out the tasks (AND branching). The system allows users to calculate various versions of the project execution which differ in terms of the sequence of tasks and in terms of the numerical assessments of the tasks. The system, depending on the stated goal, can work with the following objects: the project, the type of task, the tasks, the characteristics of the tasks (duration, costs, etc.), the value of the characteristics, the type of calculation etc. Project objects are stored in a relational data base (repository).

The testing of the developed programs was divided into two parts – testing of the complex components and testing of the simple ones. The complex components were the following:

- Capture of projects in the data base (including the establishment of the data base, entry of project objects, and reading from the data base);
- Object management, including such standard operations as the addition of new object types and objects to the project, as well as updating or deleting of objects from the project;

- Cooperation among windows, which includes information exchange among windows that are open simultaneously;
- Correctness of calculations.

Other components of the system, such as depiction of the project in graph form and the operations of individual window elements, were seen as simpler, and special testing models were not elaborated for their testing.

In other words, in place of a universal testing criterion such as C1, functional testing of the system was performed in concert with several specific criteria. What's more, systematic testing was carried out at various stages of development.

3. Organization of the testing

The programs were tested in several stages, collecting statistics about the errors that were discovered, about authors, etc. The following main stages of testing where elaborated (see Figure 1).

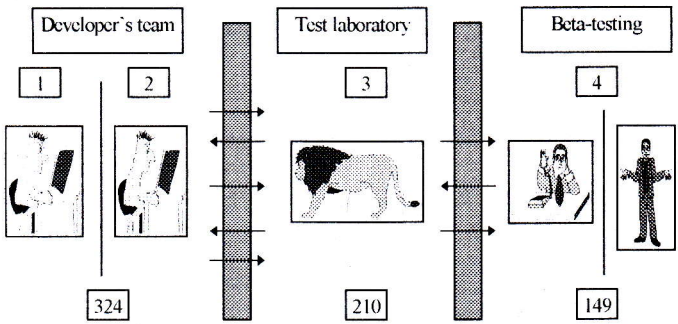

Figure 1. ***Organization of the testing***

1) Autonomous testing of modules done by the developer of the module. Some programmers tested their modules systematically on the basis of previously fixed test models, while others did the work intuitively. Because autonomous testing was individual work, statistics were not collected on the errors that were discovered in this stage of the process.

2) Testing within the software development team. The programmers tested their own modules, as well as those of their colleagues, noting all discovered errors in a log. The log served as a means of communication among programmers, but it did not, as far as possible, affect the salaries, work status, etc., of the programmers. 324 reports were logged.

3) Testing within the quality control group of the company. Before each new version of the system was turned over to the client, the company's quality control group tested it. The work was done by an independent tester. Error reports were prepared for each discovered error, and these were turned over to the developers of the programs. The independent tester did not know the internal architecture of the system and conducted functional testing. The quality control group made 210 reports which, like those in the earlier stage, allowed for an analysis of the nature of the errors that were discovered. It should be noted in particular that the statistics testify to a surprisingly large volume of errors which were noted by the developers during their testing stage, but which were also found later by the independent tester. This may be because of the time frame for the overall process, which meant that programmers were not able to repair all known errors on time.

4) Testing by the client's company. The client noted the defects discovered by system users (beta testing) and did an independent test to review the operations of the system as a whole. All errors and inadequacies were fixed in requests for changes which were sent to the developer. The company filed 149 reports. Statistics show that the greatest number of undiscovered errors were found by users and by the client precisely in those modules which were not tested systematically. In those modules which were tested on the basis of specific criteria and testing models, there were few errors discovered later.

4. Statistics about errors

The summary of statistics, which is reflected in the following table, included 10 types of errors:

	Developer group		Quality control		Client	
	No.	%	No.	%	No.	%
Project capture in the data base	30	9%	10	5%	15	10%
Object management in the project	98	30%	28	13%	13	9%
Cooperation among windows	53	17%	28	13%	5	3%
Correctness of calculations	7	2%	1	1%	3	2%
Project reflection in a graph	11	3%	58	28%	13	9%
User interface	51	16%	32	15%	25	17%
Correctness of report texts	34	11%	21	10%	20	13%
Other defects	35	11%	19	9%	16	11%
User recommendations	4	1%	10	5%	25	17%
Unjustified complaints	1	0%	3	1%	14	9%
Total	324	100%	210	100%	149	100%

1. Capture of projects in the data base. This group included errors in the opening of new data bases, the creation of new projects, changes in the format in existing projects, and deletion of projects. According to statistics, the percent share of

these errors in all phases of testing remained virtually unchanged, which can be explained through changes in the specifications of the task.

2. Object management in the project. The objects of the Mosaik 5.0 project were characterized by the fact that more than 16 types of objects were specified, and each type of object could freely be changed by the user through an addition of new attributes (fields) or by an updating of existing attributes. The project permitted many identical objects which reflect the tasks that are to be performed and which are characterized by attribute values. The individual objects are linked by performance sequence relations. According to statistics, the percent share of these errors declined in later testing phases. This is because systematic tests were implemented after a large number of errors was discovered in the first testing phase.

3. Cooperation among windows. In the Mosaik 5.0 project, the depiction of project objects is particularly complicated, because the same object can be displayed in several windows and in several formats. The system must be able to display objects correctly in several windows after they are changed in one window. Additional difficulty is caused by the fact that changes in the object type lead to changes in the object image in other windows. As was the case with the previous instance, the percent share of errors declined in later testing phases because of systematic testing after the first testing phase.

4. Correctness of calculations. In this group those errors were included which led to erroneous calculation results. It should be noted that Mosaik 5.0 is meant for the calculation of various versions of a project's costs, duration, material resource necessity, and other aspects, basing the calculations of the sequence and parallel extent of the tasks that are to be performed. The correctness of the calculations was tested by the programmer with tests that were prepared on the basis of a previously chosen testing model. The statistics show that the check of the correctness of calculations was tested at the best level, even though this is one of the most complicated parts of Mosaik 5.0.

5. Project reflection in a graph. In Mosaik 5.0, the sequence of tasks to be performed is displayed in the form of a graph. Fairly complicated operations can be done with the graph: sub-graphs can be separated from the overall graph, sections of the graph could be collapsed into one vertex, one vertex could be replaced with a sub-graphs, etc. Systematic testing was not done on the basis of a chosen model. Statistics show that the programmer was not himself able to test the complicate algorithms, and independent testing did not provide the desired results either. Apparently this component of the system needed systematic testing.

6. User interface. This group included errors that were seemingly easy to discover – insufficient review of data entry, defects in the appearance of the display screen, improper cursor movements, etc. Systematic testing was not done. The statistics show that there was an unexpectedly large number of errors.

7. Correctness of report texts. Changes in textual information caused by spelling or pronunciation errors in the texts were registered separately. Statistics show that there was an unexpectedly large number of errors in this category throughout the testing process.

8. Other defects. This group included errors not covered by the previous groups. The number of such errors was relatively low, and this allows us to conclude that most system errors fell into one of the previous seven groups.

9. User recommendations. This group included proposals on changes in the operations of the system vis-a-vis the initial specifications. Naturally these proposals were made mostly by the system users.

10. Unjustified complaints. This group includes reports on errors which were caused by user misunderstanding or carelessness.

Given the significance of the choice of testing models in systematic testing, the paper will now turn to a simplified description of the testing models that were applied.

5. The testing models

The functionality of the system was tested on the basis of the system specification. Some of the requirements were tested on the basis of the common sense and responsibility of the programmers, but in the case of other functions, specific testing models were developed:

- A model for testing the correctness of data base management;
- A model for testing the correctness of object management;
- A model for testing cooperation among windows;
- A model for testing calculations.

The four models secured the testing of the most complicated aspects of the developed system, at least according to the views of the development project management team; other elements were seen as being easier to test. As we see in statistics about the errors that were discovered, in those components which were tested in accordance to testing models, later testing phases revealed fewer errors than was the case in other components.

Next we will take a simplified look at the models which were used in the practical testing procedure.

5.1 The data base management testing model

The data base management testing model was used to review the ability of Mosaik 5.0 to open a new data base in one of two formats:

- In the first format Mosaik 5.0 projects were saved in separate files with a specific inner structure which allowed for the rapid opening and saving of projects;
- In the second format, Mosaik 5.0 projects were saved in a relational data base, which allowed access to specific project objects through SQL commands. Unfortunately, this did not allow for the rapid opening and saving of projects.

Irrespective of the format that was applied, several projects had to be kept in the data base, and there had to be opportunities to establish new projects, to update, delete and read projects, to save them in various formats, to transfer from one format to another, etc. The state transitions diagram used in the construction of the formal testing model is shown in Figure 2, where the vertices of the graph characterize the possible states of the data base: uninitiated, in format one or in format 2. The branches of the graph characterize operations with the data base objects, i.e., the

projects. Operations with data base objects were tested in depth 2, which means that during the test, the data base was first initialized, then a format was chosen, and then a project was entered into the data base, edited, deleted, saved, or read from the data base with a transfer from one format to the other. The test was considered sufficient when all operation couples were tested. It should be added that the correctness of changes in the project objects was tested with another model, which is reviewed in the next section of this paper.

Because tools of test capture and repeated usage were not used, the testing of the data base required a lot of time, as a result of which the testing was not always conducted in full. Secondly, this testing model was developed by the programmer group and applied only after many errors had already been revealed in further testing phases. Therefore, the statistics do not truly reflect the effectiveness of the testing model; rather, what are shown are the consequences of failure to use the testing model. It is only after the implementation of the systematic testing procedure in the development group that the number of errors revealed in further testing phases declined significantly.

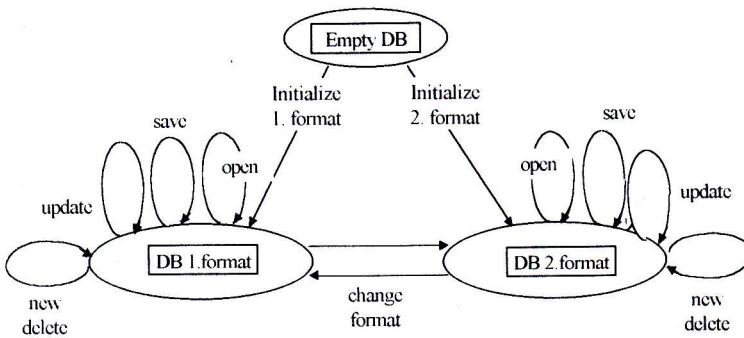

Figure 2. ***Data base management testing model***

5.2 The project object management testing model

Mosaik 5.0 project objects include many types of objects, where every type can be changed freely by the user – adding new attributes (fields) or changing existing ones. Many single-type objects can be used in the project, and they basically have to reflect the tasks that are to be done and they must be characterized by values of their attributes. Separate objects are mutually connected through sequence relations.

The testing of project object management includes a check of the correctness of object types, as well as of changes in the value of object attributes. The testing model is depicted in Figure 3, where the vertices of the graph who one type of the object and objects corresponding to it, while the branches represent operations with object types and objects. During the test, operations with object types and the corresponding objects were tested at depth 2. That means that the testing was considered to be sufficient when type changes were checked along with the correspondence of changes in corresponding objects to all possible operation couples.

As was the case in the data base testing, the object management test also did not use tools for test capture and retrieval. This meant that much time was needed, and the testing model was developed and applied by the programmer group only after problems had been encountered in the testing of object management. As was the case previously, after the implementation of systematic testing in the development group, the number of errors discovered in further phases declined significantly.

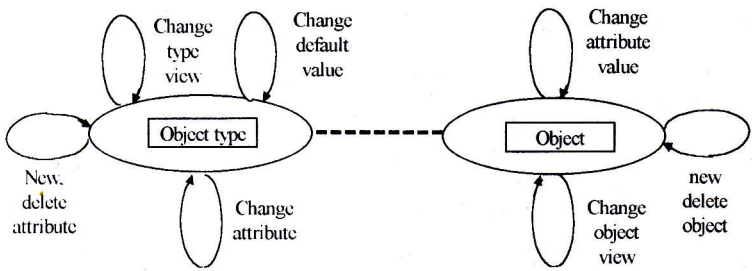

Figure 3. ***The project object management testing model***

5.3 The model for testing of window cooperation

In the Mosaik 5.0 project, the depiction of project objects is particularly complicated, because one and the same object can be displayed in several windows and in several formats. As a result of this, the programmers developed a window cooperation testing model, which is illustrated in Figure 4. The vertices of the graph represent the windows which are open during the operations of the system, while the branches represent operations with the windows.

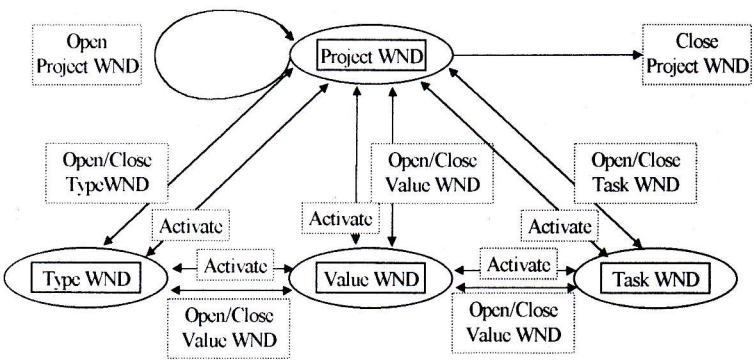

Figure 4. ***The model for testing of window cooperation***

The testing of window cooperation involves the opening and closing of many copies of a single type of window, as well as a check of the correctness of transfers among windows of various types. The test was considered to be complete when cooperation between two copies of all types of windows had been tested.

Statistics affirm that the timely elaboration of a model and the implementation of systematic testing in the development group significantly reduced the number of errors that were discovered in the last phase of the testing.

5.4 The model for testing of calculations

The test of the correctness of calculations was performed as follows: Several projects were elaborated in the Mosaik 5.0 system where each project served as the basis for one set of tests. In order to check the correctness of the operation of the developed programs, calculations with different options and object values were executed, and the results were compared to standard values as calculated by the tester. This well-known method for checking calculations proved to be most effective. After the tests were drafted and the standard values were calculated, the test itself took little time (meaning the execution of the program and a comparison of the result to the standard values). The test was easy to repeat, and it was very sensitive to program errors. The testing model is represented in Figure 5, where the vertices of the graph show the constructions which are permitted in the project graph: linear branch, OR construction, AND construction, and segment. The branches show their applicability in the corresponding constructions. The tests were elaborated so that they would include all possible combinations of constructions at depth 2 (meaning 12 in total). This means that a test was developed in which the linear branch included the OR construction, the AND construction, and a segment, where each of the constructions for its part included a linear branch, an OR construction, an AND construction and a segment, etc.

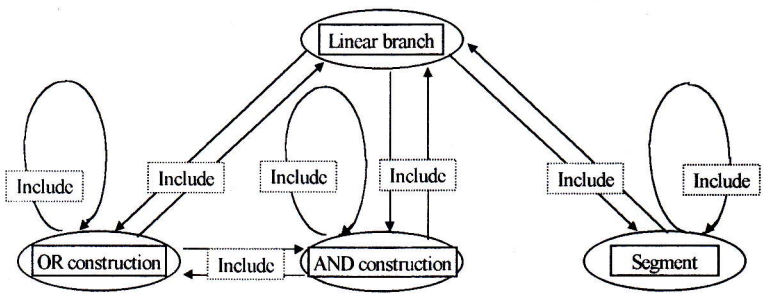

Figure 5. The model for testing of calculations

The model for testing of calculations proved to be so sensitive toward errors in the program that after the development of the calculation testing project, only a few insignificant errors were found in the next testing phases.

6. Main conclusions

Even though the statistics collected from the testing of one specific system do not allow us to draw generalized conclusions, the following can be said safely:

- Systematic testing has distinct advantages over intuitive testing;
- The functional testing of specific systems must use testing models which are appropriate for the specific instance and which are set out by the specifications of the system that is being developed;
- Testing on the principle of "depth 2" was, at least in the testing of this system, sufficient to reveal errors properly;
- This paper provides a brief look at models for testing of data base management, object management, user interface and calculation testing, and these are typical for the testing of many different kinds of systems.

References

[Beiz 95] Boris Beizer. Black-Box Testing Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc, USA, 1995, 294 pp.

[Over 95] Jan Overbeck. Testing Object-Oriented Software and Reusability - Contradiction or Key to Success. *Conference Proceedings, Eighth International Software Quality Week 1995*, Software Research Institute, USA, 1995.

[Bic 79] J.Bicevskis, J.Borzovs, U.Straujums, A.Zarins, and E.F.Miller. SMOTL- a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, SE-5, No. 1,1979,pp.60-66.

[Auz 91] A.Auzins, J.Barzdins, J.Bicevskis, K.Cerans, A.Kalnins. Automatic construction of test sets: a theoretical approach. *Lecture Notes in Computer Science*. Vol. 502, Springer - Verlag, 1991.

[Beiz 97] B.Beizer An overview of testing. Quality Week Europe 1997. *Tutorial Notes.* Software Research Institute, USA, 1997.