Particularities of Verification Processes for Distributed Informatics Applications

Ion IVAN[1], Cristian CIUREA[1], Bogdan VINTILĂ[2], Gheorghe NOȘCA[3]
[1]Department of Economic Informatics and Cybernetics, Academy of Economic Studies, Bucharest, Romania,
[2]Ixia, Bucharest, Romania
[3]Association for Development through Science and Education, Bucharest, Romania
ionivan@ase.ro, cristian.ciurea@ie.ase.ro, vintilabc@gmail.com, r_g_nosca@yahoo.com

This paper presents distributed informatics applications and characteristics of their development cycle. It defines the concept of verification and there are identified the differences from software testing. Particularities of the software testing and software verification processes are described. The verification steps and necessary conditions are presented and there are established influence factors of quality verification. Software optimality verification is analyzed and some metrics are defined for the verification process.
Keywords: *Distributed Informatics Applications, Software Testing, Software Verification, Verification Process, Software Optimality*

1 Characteristics of Distributed Informatics Applications

Distributed informatics applications are software constructions that are based on architectures whose components, through interaction, realize allocations of resources in real time. Distributed informatics applications include:

- a heterogeneous group of users, having many elements that, through interaction, solve their well-defined problems, as data entry volume, sequence operations that activates and with concrete results that marks the success of performing the interaction or the need to retake some components from the operations chain specifying the cause and the manner of disposal: after a few replays each user successfully completes the interaction getting the message meaning that his problem has been solved correctly and completely;
- dynamic definition of computer network through which is realized the messages transfer of user exempted in the ground indefinitely, the only restrictions being those related to hardware resources that ensure compatibility with data acquisition system and connection performance;
- conducting an achievements cycle that includes steps, such as: defining the target group and setting its size; defining the set of distinct problems, which are subject to processing; in this respect for each problem are used appropriate definition tools to reduce the risk of sub-definitions or supra-definitions, situations attracting reversals of prior steps when minuses of information in the case of sub-definitions, or the excess of information, in the case of supra-definition of the problem, generate effects that lead to discontinuation of the development cycle with the impossibility to pass to the next stage; the stage of clear specifications, consistent, the development stage of informatics solutions variants accompanied by performance estimation models requires choosing the suitable variant against the criterion with which the multiplication effects is managed at the moment of implementation; the code elaboration stage as optimal resource allocation process, knowing that the instructions, data structures defining mechanism and building of sequences procedures must be understood as infinite resource use, but which differ from each other in terms of performance criteria of informatics applications, taken as a whole; the testing stage [1] play a very special role for the distributed informatics applications

because these operates independently of the developer; the user has limited possibilities to manage uncontrolled situations resulting from existing errors in procedures that allocate resources wrongly or generates random processing behavior, which ultimately creates users discomfort (elaboration, documentation, implementation).

Modern distributed informatics applications are investments, so they include:

- the investor who pays the development of the distributed informatics application;
- staff providing application development;
- staff ensuring exploitation management;
- users that solves the problem with the distributed informatics application, which becomes service provider;
- from the amounts transferred by the users to other destinations, a part returns to the investor, a part to the management of the application and the investment is recovered, and those who ensure the management have their profit.

Modern distributed informatics applications have into the user a beneficiary of the options, database, there are investors who recover the investment by the fact that users beneficiates of services: for example booking.com. It is considered that there is an investor. In the database the hotels place details regarding the number of rooms, prices, photos, etc. In the database are loyal users and new users after their first transaction. The hotel owners and the customers will beneficiate by the services of the website. The customer pays at the hotel and an amount x% go to the website or to the investor, the hotelier pays only to appear there and to be hosted on the website.

The informatics system from a bank is collaborative because it has a large number of components, a large variety of links between them and requires a high level of connectivity and integrability [2].

The components of banking informatics system are distributed applications that communicate with each other and are integrated into a whole. Over the time, banks have improved their informatics systems by increasing the integrability degree of their components applications.

Another indicator that banks are seeking is the portability degree of informatics applications from the bank, according to a bank can migrate its informatics system from one work environment to another, especially to fulfill the disaster recovery procedures.

A distributed informatics application that is used in a bank is the Collaborative Servicedesk application, which allows analyzing the types of problems reported by internet banking users. Having the database with all customer requests, the bank determines the strategies to address each client, depending on the history of problems that he encountered.

The Collaborative Servicedesk application adapts to input data and modifies its components so as to provide maximum utility and customer support regardless of category they belong.

The verification process of Collaborative Servicedesk application is different by the testing process, because it requires understanding the problem, discussions with analysts, according with the objectives established at the application implementation.

The application testing involves performing a battery of tests to ensure the accuracy of the information recorded, and validate the operation manner of the application. Verification testing process involves:

- verification that all tests proposed have been run;
- verification that the tests realized cover all or a part of the problem;
- verification that the test report correctly reflects what happened with the application.

Figure 1 shows the stages of verification testing process for the distributed informatics application:

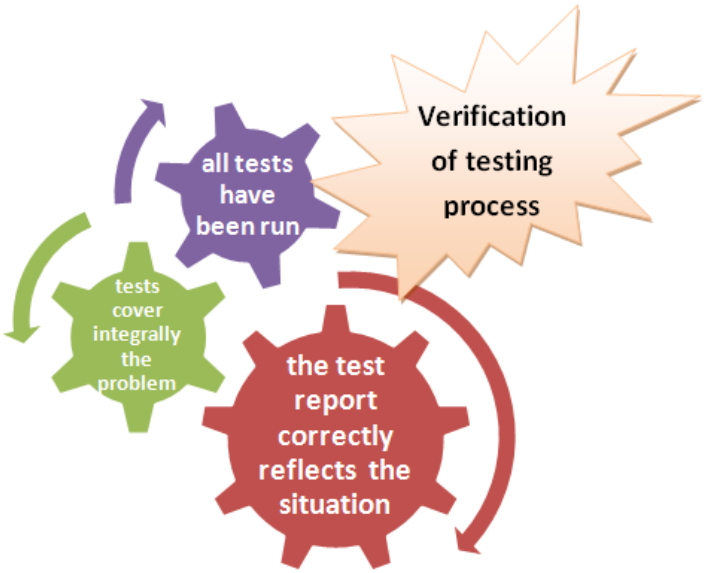

Fig. 1. Verification stages of distributed informatics applications

In the current use stage of the application, the verification process includes the following elements:

- verification whether the access to resources was defined for all enrolled users; if there are n real users and m enrolled users in the application, then if n=m is fine, if n>m means that not all users were enrolled, and if n<m means that and other users were enrolled, although they had no access right.
- verification of errors types by application users; each user, whether is customer or analyst, check the types of errors and if the messages received from the application are consistent with the errors.

The Collaborative Servicedesk distributed informatics application integrates the following features:

- automatic change of links position in the knowledge base, according to the number of requests registered on each category;
- automatic creation of a new category of requests;
- automatic creation of vocabulary of specific terms to the problems recorded;
- automatic distribution of customer requests to the analysts enrolled;
- reclassification of a request on the category suggested using a genetic algorithm.

All these facilities require the verification of correctness and completeness with which they are made. In this case, the testing process does not cover everything, requiring additional verification of the fulfillment of all facilities implemented. The verification process follows what happens with a request from its registration moment until it is solved. There are identified routines and repetitive activities in order to eliminate them and increase the application performance.

2 Particularities of the Software Testing

Software testing is the process through which, on the basis of input data sets, execution and result evaluation, software is evaluated. Software testing uses input data sets [3]. These consist of some or all settings one user can possibly do on a user interface, regarding configuration, and different values for data entry fields that statistically cover all the real-world situations. Figure 2 shows the spine of the software testing process.

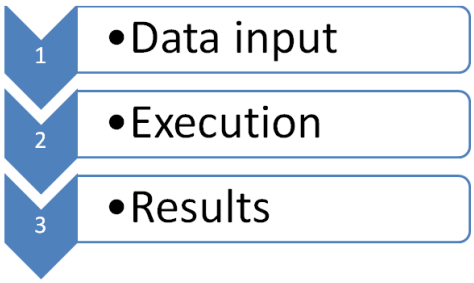

Fig. 2. Software testing process

Due to a vast set of factors, the software testing process must adapt and shorten its duration. Should all possible tests be run on a software application, even for a very small one, the possibilities are almost unlimited and, thus, the duration of the process is extremely large. Budget and time are the main factors of limiting the software testing to only a very small part of the total executable test cases.

The quality of the tested software product is given on a large basis by the experience and expertise of the QA team that creates, executes and evaluates the test cases. Test cases are supposed to cover all execution paths and for this they must cover all GUI options as well as supplying different values in different fields as these influence the calculations underneath. Should an inexperienced QA team create test cases that cover only at a small degree the functionality of the tested software, the problems will arise at the customer site and the experience will be bad.

The execution of the test cases is almost as important as the test cases themselves. Even if the test cases cover the largest part of the application's functionality and the sample values are good enough to catch most of the problems, if run by inexperienced people the results of the process will be unreliable. In the execution phase the QA member must also pay attention to other factors that might influence the execution of the application such as the Internet connectivity and previous actions. Some issues in software programs occur only after a very long and complex set of actions and the QA member executing the test cases must be attentive enough to remember all actions he did when a problem arises in order to be able to successfully and quickly reproduce and document it.

Test cases and their execution lead to results. These are the data provided by the software program under test after the processing of the data input. In order for an application to have high quality the results must be correct, complete and consistent.

The correctness of the results is given by the fact that actual results for some data input matches or not the expected results. As QA team knows the application's functions, they also can predict the results for a certain data input set. If the actual and expected results don't match there are some situations that might have caused this:

- the test case was poorly designed and the input data set is not valid for the expected results; this might be due to modifying the input data set after manually computing the results, or an error in the manual computation;
- the test case was executed poorly meaning that not all settings were done or not all values were inserted correctly; this is the result of inexperienced team members or environment issues such as stress;
- the functionality of the application in the tested area changed and the QA team is not aware of it; this might happen because the changes in functionality are very new and there was not enough time for the information to propagate towards the QA team or because of poor communication between the development and QA teams;
- there is a problem in the application's functionality; in this case the QA team must raise the problem and document it well so that the development team can reproduce it easily and be able to test the fix once it is done.

The completeness of the results means that exactly the expected results will be supplied, no more, no less. If more results than the expected ones are supplied, there might be a problem, a setting might not have been done or a value of the test case might have been inserted wrongly. The case of missing results is similar, but adds something more. When incomplete results appear something might have happened that halted the application's execution at some point and there is likely a warning or exception thrown about the fact. When an warning is shown to the user it notifies him that some of the input data might cause problems, but the execution will continue. An exception notifies the user that

some data input is wrong and the computation can't continue.
Software testing is done in different manners: on procedures, on modules, on the whole product, by developers, by QA team members, by marketing team at the client site, by clients.
The testing process is present in all stages of the development cycle and the person that tests is not the same. Developers test their code during and after the implementation by supplying test input values, observing intermediate results and comparing the final results with the expected ones. QA team tests the software once the developers state it reached a state of stability. Managers and directors test new features to see how they work before reaching to clients.

3 Software Verification Processes

Software verification, as opposed to software testing does not imply the execution phase. Verification of something means checking if it complies with the rules it was built by.
Software verification is essential in all development stages as it leaves room for no mistakes in design and improper implementation of applications. In each stage of the development cycle different things are verified and they all subject common rules, but a keen eye must be kept open for any possible issue.
As very large applications are not designed and implemented by one team and in a very small amount of time, the stages of the development cycle repeat for every single feature that is developed and we will focus on the development cycle of features (or very small applications) instead of focusing on the development cycle for huge applications.
The ***problem defining stage*** is the one in which the data input, models and algorithms to be used and results are established. For this stage the verification consists in checking if:

- the problem statement includes all possible particular issues and cases; for this an analyst must analyze the problem and the problem statement and see if they are consistent;
- all data input fields are mentioned; if this is not the case, later on in the development cycle major issues will appear as data needed for computation is not available, thus halting the entire process to a stop, causing additional expenses of financial and time resources;
- all used models can be supplied with data and they are the simplest that can do the job; supplying the data falls in the above category; using simple models that can do the job well and efficient causes less defects in the software program as the members of the development team are able to easily understand, implement and work with them;
- all algorithms can be supplied with required data, are efficient, have a very large or complete coverage rate; the effectiveness of an algorithm is give by the amount of time it needs to solve a problem when the dimension of the data input increases greatly; for small data sets (few hundred entries at max) almost all algorithms are efficient; even a brute force algorithm will deliver results very quickly if the input data set is small, but increasing the data set leads gives birth to the need of optimization; as computers become more and more complex, so does software and small input data sets are quite a rarity; thus, verifying an algorithm is efficient means checking if the performance level is linear regardless the size of the input data set; depending on the nature of the algorithm and the computation it makes not all of them have linear behavior, but this is the desired and best case; verifying if an algorithm has a very large or complete coverage rate means to find values that are needed and the algorithm can't compute; the important thing here is to focus on values that are needed, as, often, not all the values in a range are needed; if such values are found and they count a significant part of the total than the algorithm doesn't have a high enough coverage rate; this means it must either be

changed or supplemented by an algorithm that can compute those values.

When performing verification at this stage of the development cycle, a keen eye must be kept open to see if the defined problem is really the one the users experience. A poor defining of the problem leads only to waste of resources.

The ***target group definition stage*** of the development cycle identifies the generic set of user that the software product addresses. Defining the target group means considering criteria such as territory, age, education level, access frequency, etc. Some of these criteria are general regardless the software product that is developed, such as the access frequency, but most of them depend on the nature of the software. For this stage verification consists in:

- checking that no significant part of the future users have been left aside; small omissions of user categories are acceptable as long as their share in the total is under few percents, but larger omissions means that the software is not designed to fit all the users and, thus, is prone to losing market share;
- checking that groups with a very low chance of becoming actual users have not been included in the target group; as citizen oriented informatics applications must be designed so that all users from the target, regardless their education and experience can use the application without prior training, including groups with a very small chance of becoming real users in the target group just adds extra weight to the process of design and development due to extra restrictions these groups add; also, the benefits brought in by such small groups are outnumbered by the costs of the extra resources needed in the development cycle;
- check that all significant criteria has been introduced in the characterization of the target group; the criteria that are used to characterize the target group determine the characteristics and particularities of the software application as, through these criteria, behavior patterns are determined; ignoring one or more of the important criteria might lead to the loss of a particular part of the target group as the software product doesn't correspond to their requirements;
- check that all criteria used for the characterization of the target group has an impact on the behavior patters of the users; if criteria that have no impact on the behavior patterns are considered, the application will be overloaded with features and particularities that don't improve the user experience, but increase the complexity of the software and the necessary of resources during the development cycle.

For online applications success is given not by the offered service, not by the clean interface, not by the neat features but by the number of users. The larger the number of active users, the larger the number of potential users and the higher the perception of the application's quality are. Social networks have clients for mobile devices and for desktops. Not always the interface is straightforward intuitive, clean or easy to use, but the very large number of users attracts more and more users every day.

The ***specifications definition stage*** of the development cycle is the one that gives the first insight of the real functionality of the software through the eyes of the user. Messing up this stage means developing for no one as you will not respond to any needs of the users. The specifications must be exact, complete and correct. To ensure exactness one must verify that all measurable input, output or process has limits defined for values. In the case of a variable, from the user's point of view, any wrong values must be highlighted in GUI along with a clear and easy to read error. For string variables the maximum and minimum length must be specified, for decimal numbers, the maximum number of decimals is important. In the case of algorithms the execution time and memory consumption are the concerns. In the case of online applications the memory consumption never occurs to the user as most of these run within a browser, but for the

standalone applications memory is a real concern. More important than the memory consumption is the execution time. No user wants an algorithm that solves a relatively simple problem to run over a long period of time. Users don't actually know which is the complexity of the problem that the algorithm solves, but they have a slight perception of it. The higher the perceived complexity, the higher the time the user is willing to wait. If time consuming algorithms can't be avoided, one must verify that the GUI includes visual cues that give the user information about the progress of the task and, if possible, of the estimated remaining time. This makes the users stop the processing less often as they see clearly that there is progress, the computations have not stopped and the remaining time decreases over time. In the case of trivial problems for which there aren't algorithms that have a complete coverage, it is preferable to limit the input interval and use a simple and efficient algorithm, that use a time consuming one. Computing some values through one efficient algorithm and others through a time consuming one that can return a value, without providing visual cues in GUI is not a good solution either. Without visual indicators the users will start to ask why for some values the application's feedback is instant and for others it takes forever, or at least observable time. Without understanding the process, they will soon doubt its correctness and start looking for alternatives. Providing some visual indicators, such as messages stating that the input value requires some special processing that will take some time, informs the users of the special situation they find themselves in and makes them expect something to be different, in this case, the processing time.

In the ***project building stage*** of the development cycle the data structures, functions and procedures, modules and interdependencies are established. The definition of the modules and interdependencies is very important for the modularity of the application and the order of development for different modules and functionality. For this, one must verify that a module that depends on another is not scheduled for development before that one or at the same time. Data structures are entities that are used by functions and procedures to perform tasks. Problems can be solved in many ways. The difference between a poor piece of code ad a good and efficient one is made by data structures and algorithms. Using the right data structure and the right algorithm enables the increase of the dimension of the input data. Given a simple problem, such as determining all combinations of numbers from a set that summed up give zero, one can easily solve it by iterating through the set and creating all possible combinations. If the resulting sum of the combination is zero the combination will be added to the solution list. This approach is simple and easy to implement, but the time needed for large data sets is very large. For a set of 10 elements, there are about 10^3 combinations to be done, and that's not that large, if not done frequently, but for a set only 10 times larger, the needed time is 10^6 and its about one thousand times larger than that of the previous data set. The increase of the input data set dimension by a factor of 10 leads to the increase of the needed time by a factor of 10^3. It is clear that this algorithm is not suitable for solving this problem for large input data sets. An easy improvement of the algorithm is to calculate sums of 2 elements and see if there is a third element that has the same value as the sum, but negated. This reduces the factor by which the execution time multiplies.

At this point, one must verify that no data structures that were defined are unused. If this is the case, those must be removed. The algorithms must be verified to see if all the data structures they need have been defined. If there are data structures that have not been defined, these must be defined. After these two verifications, only the required data structures will be defined.

For functions and procedures one must verify the signature, meaning return type and parameters, if all parameters are used within the computations and if a result is returned.

Table 1. Input data \ Results table

Results / Input data	R1	R2	R3	R4	R5	R6
D1		X				X
D2	X			X		
D3		X	X			
D4				X		
D5				X		X
D6						
D7			X			

For functions and procedures it is very important that all data input is part of the results. In Table 1 one can easily identify the useless data input (D6) and the results that are not computed (R5). If rows where no X has been placed are identified that means that the corresponding component of the data input set is not used for any results. When columns where no X has been placed are identified this means that the corresponding result uses no input data for computations. When useless data is identified one must verify it isn't needed and then remove it from the function/procedure's signature. When results that use no input data are identified, these either are calculated from constants, and in this case they should be cached, either there is a problem and input data has not been considered for them. For the last case, one must identify the necessary input data and include it.

Table 2. Input data \ Formulae table

Formulae / Input data	F1	F2	F3	F4	F5	F6
D1	X					
D2						
D3					X	
D4	X		X			
D5		X			X	
D6				X		
D7	X					

These considerations are valid for the input data – formulae relation. All input data must be used in at least one formula, and all complex formulae must use at least one component of the data input set.

Table 3. Results \ Formulae table

Formulae / Results	F1	F2	F3	F4	F5	F6
R1		X			X	
R2						
R3		X				
R4			X			
R5				X		
R6		X				
R7						X

In Table 3 the correlation between results and the formulae used is presented. As before, no Xs on one row means that the result uses no formulae for the computation and no Xs on

the columns means that the corresponding formula is used for the computation of no result.

For a given procedure:

Public Return_type Name(Type$_1$ P$_1$, Type$_2$ P$_2$, …, Type$_k$ P$_k$)

one has to verify that:

- k is the necessary one; no unnecessary parameters have been insert and no necessary parameters have been omitted;
- the parameters are in the correct order; as function overloading is based on the order and type of parameters, the correct order is essential;
- all types (return type and parameters types) are the correct ones; assuming at least one type is not the correct one, the function is useless.

For a *for* loop:

```
for (int i=0; i<N; i++)
{
      //do something
}
```

one must verify that:

- the statement's syntax is correct and complete; this one is automatically done by most IDEs nowadays and the developer can't quite get it wrong, but at this stage, where portions of generic code in algorithms can be defined, is important to check the correctness of it;
- the counter is initialized with the right value; initializing the counter variable with the wrong value leads to incorrect results due to elements that are not considered, to inefficient times, memory corruption, infinite loops;
- the stop condition is valid and it will be reached after a finite time; placing a wrong stop condition leads to infinite runtime and stack overflow;
- the step of the counter variable is correct; if the step of the counter variable is incorrect, the results are wrong due to omitted elements and stack overflow might occur due to not meeting the stop condition.

Mistakes in this stage of the development cycle have long lasting consequences in development time and resource consumption but also in maintenance and updating processes.

The ***code writing stage*** of the development cycle is usually characterized by the parallel work of multiple programmers that write different parts of the software module or application. Most applications strive for a unique style and language across all modules, but this is not always possible. There are applications of very large dimensions that have been initially developed using one language, then, as technologies evolved, some modules were developed using different programming languages. When developers use more than one language to write code in, the coding standards for the used programming language must be obeyed. If only one language is used, the developers make a habit out of the coding standards after a while and they don't even pay attention to how they do it. When using more than one language, one must be careful in order to be able to respect the coding standards for each of them. Verification, at this stage, means aside checks the correctness of the written code, making sure that the coding standards for the used language have been respected.

In the case of the distributed applications, the ***server loading stage*** is the point in time after which the testing team starts working. At this point, verification means making sure that the server has all components needed for the application to run installed, has enough free space, enough memory, Internet connectivity, the test database has users that can access the application.

The ***technical testing stage*** of the development cycle assumes that the quality assurance team runs tests to see if the application works as defined by the specifications or not. The QA testing assumes the execution of the application using test datasets and the comparison of actual results with expected ones. When these don't match, an issue has been discovered and it's recorded for further investigation. Verification, at this stage, assumes checking if all tests have been executed, all results have been compared to

the expected ones, if the execution has been correct or not.

The ***sample testing stage*** of the development cycle assumes the testing of the application with real data samples. Regardless the experience of the QA team, the number of tests they can run is limited and can't cover all the functionality of the application with all possible values. The sample testing, on the other hand, should used random samples of data input that users utilize. These can be obtained by recording them from the users, but without saving sensitive date and by asking their agreement. By testing using samples, issues can be found that were passed by the QA team. Verification at this stage means checking that the pool from which the samples are extracted is large enough and covers a very large part of the new implemented functionality. Also one must verify that the tested samples are correctly executed and they cover the newly implemented code.

The ***documenting stage*** of the development cycle assumes the documentation of code through comments. At this point one must verify that there are no complex code segments that lack documentation and that the existing documentation is clear, specific and exact. If documentation lacks it must be added and if it is not clear or precise enough, it must be reformulated.

The ***implementation stage*** assumes the distributed application is installed and configured on the client's server. This is one of the final stages of the development cycle and of great importance as the real users will start using the application after this step. Verification, at this point, means that all application's components are installed at the right place, that the connection to the database is correct, that the server has all components needed for the application to run, that the server has Internet connectivity and other requirements specific to the application.

The ***maintenance stage*** of the development cycle lasts between the implementation and the removal from use. Its purpose is to correct any problems that were not identified and corrected during the development cycle and implement new features as per users' requests. Verification, at this stage has the following aspects:

- verification of the issues that are visible now but they weren't identified during the development cycle;
- verification of new features implemented as per users' requests.

The verification of the issues that are discovered during the users' use assumes checking which are the conditions under which the problem appears, checking if the analyze of the root causes has been complete, checking if the solution covers all situations and provides the users with the desired results.

The verification of new features means, first of all, to check if the feature is really required by so many users as to be worth implementing it. If this is the case, additional verifications must be done: if the needed input is available or additional changes must be done, if the extra feature can cause problems with the existing application, if the extra feature's entry point is where the users' requested it, if the planned functionality if the one the users' demanded.

The ***software reengineering stage*** of the development cycle happens when the application in cause is so hard to maintain as to justify a complete refactoring. Not all applications pass through the software reengineering stage as not all of them last so long as to cause maintenance across a few years to cost as much or more as implementing the application again using newer technologies, but when it happens one must verify that:

- the newly chosen technology is fully compatible with the existing functionality of the application;
- the planned development process does not use more resources that currently allocated;
- the process is transparent to users;
- the maintenance costs after the reengineering process will be significantly lower than the current ones;

- the users' data won't be affected in any way by the process;
- the chosen technology is not at the end of its lifecycle.

Software reengineering is a powerful tool that allows good applications to benefit of new technologies and improve even further the user experience.

Remove from use is the last stage of the development cycle and it assumes the application is closed down and users' data is archived or dispatched of. At this stage one must verify that all application's components have been removed from the server, that the database has been backed up and deleted, that the additional components needed only by this application have been uninstalled, that the removal of the application doesn't affect other applications that depended on it.

4 Software Optimality Verification

Software optimality [4] assumes that given some criteria, the piece of software complies with them all and no improvement in one area can be done without affecting another. In the software industry some criteria are classical, such as cost, time needed to run, needed memory, storage and communication needs, number of simultaneous users. Before one submitting his work, he must verify if it is optimal or can support further optimization.

In the case of software products optimization can have many forms:

- removal of dead code assumes the analysis of code and removal of the portions that will never execute; validations are done in numerous locations and it is not quite surprisingly that some areas of code never get executed as all needed conditions are never fulfilled; removal of code makes the remaining code clearer and eases the effort of maintaining and updating due to the reduced complexity;
- redundant computing is a time consuming easily identifiable and removable issue; in algorithms that compute complex problems often long expressions are composed of many smaller ones; by computing the most common expressions, storing their values and reusing them when necessary precious time is saved;
  let $e=(a^2+b^2+c^2)/(a^2+b^2+c^2-1)$ be a relatively simple expression that is to be computed; from a glace the number of operations can be easily identified:
  - six multiplications
  - four additions
  - one subtraction
  - one division
  - one assignment

  to a total of thirteen; by simply rewriting the expression as $e=x/(x-1)$ where $x=(a^2+b^2+c^2)$ we can reduce the operations to:
  - three multiplications
  - two additions
  - one subtraction
  - one division
  - one assignment

  to a total of eight thus saving five operations, almost forty percent of the original number; this is just a simple example and a simple optimization improves the result by a significant percent; in complex cases the optimization makes the difference between the success and the failure of an algorithm, procedure, module or even software product [5];
- unneeded conversions are another problem that causes code to slow down; many ignore the importance of using the right data type from the start and abuse the conversion mechanism thus adding unnecessary latency to the execution; assuming conversions can't be totally avoided the verification of the optimization assumes checking if the minimum number of conversions is done; for long sequences of code where a variable must be converted to a certain format each time it is used, it is recommended to use a temporary variable to store the converted form and use it in the whole sequence and setting back the final value on the original variable in the end; this saves all but two conversion for the entire sequence;
- discarded results assume that the value of a variable is computed and then the

variable is assigned another value without the prior one to be used in any way; this happens as the developer starts with one idea that he drops after half implementing it and starts developing based on another idea, but reusing the initial variables; the verification in this case assumes that the developer follows the entire code sequence once it is written and ensure no variables are computed and discarded before using the computed value; if such happen, the computation must be removed from the sequence, as it is not only useless, but also a burden for the efficiency of the code;

- invariance management assumes the discovery and elimination of operations that happen multiple times when they should only happen once; it's not uncommon to have repetitive structures in code when iteration through the elements of a collection is needed, a certain task must be executed for many input data; repetitive structures are places where disasters can occur if the developer's focus is not maintained during the whole process; let us assume we have a function to be computed for one billion elements; each line code that forms that function will be executed one billion times for our data; each line of code we remove will not be executed; even small improvements in such a function that is executed very often leads to spectacular increases in performance and system responsiveness;
- common code grouping assumes to include as much code in a block as possible and avoid repetitive atomic operations; let us consider the following code sequence that computes the sum of a vector's elements, the sum of the positive elements and the sum of the negative ones:

```
s = 0;
sn = 0;
sp = 0;
for (int i=0; i<n; i++)
    s+=x[i]
for (int i=0; i<n; i++)
    if (x[i] < 0)
          sn+=x[i]
for (int i=0; i<n; i++)
    if (x[i] > 0)
sp+=x[i]
```

for this sequence a number of operations are made:

- ~5*n additions if there are no null elements;
- ~5*n comparisons;
- ~5*n assignments

after grouping the common code and discarding unnecessary for loops we have the following sequence:

```
s = 0;
sn = 0;
sp = 0;
for (int i=0; i<n; i++)
    if (x[i] < 0)
          sn+=x[i]
else
sp+=x[i]
s = sn + sp
```

that has the following number of operations:

- ~2*n additions
- ~2*n comparisons
- ~2*n assignments

it is clear that through such a small and simple optimization the final time of execution for very large vectors is around two and a half times better for the second code sequence;

- file reading optimization depends greatly on the storage technology the machine the software is running on uses; storage technology evolved greatly in the last decade, but the growth has been more in capacity than performance and it hasn't got even close the growth the processors and graphics knew; considering a machine that uses a conventional HDD that spins and has mechanical parts the optimization must consider the natural limitations of such devices and address them; for classical HDDs the speed, number of operations per second and delay are the highest bottlenecks; the delay can be tricked only by accessing the HDD as seldom as possible; the number of operation per second is a factor that must

be taken into consideration when many files must be read or written in a short amount of time; even if their size is not large, the whole operation will last some time because the HDD can't initiate read/write operations at a very large rate; to overcome this issue one must design the storage so that fewer files are accessed; with the information stored in fewer logical bags, the number of read/write operations is reduced and thus the total operating time; another issue here might be caused by the reading/writing of very small data segments; let us consider a file that has one million integers meaning around four million bytes and a code sequence that has to compute different statistical operations on the data; as one million integers don't use a lot of memory, once these are read from the HDD, they will be stored in RAM and used for computations; reading the values from the file is another story; if one reads the integers one by one, the HDD will make one million read operations; divided by the number of operations per second we can approximate the time needed for the operations to complete; in a simple test run with one hundred million chars, the data from Table 4 was obtained:

Table 4. Times needed for different operations on same dataset

	Run		
Operation	1st	2nd	3rd
Write all	1123	873	996
Write each	2480	2277	2415
Read all	904	702	846
Read each	4914	5085	4989

one can easily see that the operations that are done in bulk need much less time for completion than the atomic operations;

- code duplication is something that all developers avoid due to issues that appear at maintenance and updating processes; sometimes, though, the duplication of code can save lots of time; functions and procedures calls can be time consuming and even outlast the time needed for the procedure to execute; when this is the case and the procedure's code is short and simple, it is better to duplicate code than to have terrible performance; duplicating code should not be a habit of any developer and when such extreme situations appear, the duplication must be back-up-ed by serious documentation;
- cost optimization assumes to obtain a set of predefined results with the minimum costs; for this, one must verify that the chosen solution is the one that involves the smallest costs and he must take into account the team's training, the known technologies, available licenses, etc; forgetting about an important cost factor might lead to choosing the not so cheap solution and once the project starts few have the guts to step back and restart the whole development machine.

Through optimization of software machines that seem obsolete are used again, resources are saved and efforts are directed through continuous development and optimization.

5 Verification Processes of Distributed Informatics Applications and Influence Factors of Verification

It must be established a very clear relation between testing, validation and verification. It is considered a distributed informatics application currently in use, obtained by covering the full development cycle phases.

At some point a result R is desired. For this purpose there are selected the options $<O_1, O_2, ..., O_n>$. There is a procedure defined that is executed by an operator for many times and the success rate is very high.

Verification in this case means to process the result R and to see if:

- the structure at qualitative level, but also at quantitative level, is the expected one;

- the volume of processed items is the necessary one, there being some control keys;
- the indicators making up the result structure at its overall belong to the established fields and whether the correlations identified are respected either by mathematical formulas or identified experimentally.

Verification is a routine operation following the execution of some routine procedures, and the product verified is used to achieve a well-defined objective.

The verification result leads to the idea that the procedure which was executed and what has been achieved meet the requirements and the product obtained has quality and is used successfully. In statistics is supported a verified product with a specific test that proves the product is good. The hypothesis is accepted and result that it is true.

There are situations when verification concludes that the procedure was executed and what has been achieved is of good quality, but when is going to use output it shows that it is not. The hypothesis is accepted, but in reality it is false.

There are situations where the hypothesis is rejected at verification, although is good and means that the hypothesis was rejected when it was valid.

The verification has an associated procedure that shows how this process looks like:

- the entries are established as $I_1, I_2, ..., I_n$;
- the operations are established, which define the verification $op_1, op_2,..., op_m$, the operations are in finite number and in the sequence imposed, being specified the situations where interactions are permitted or indicating which are the optional operations;
- is established when the verification result concludes that the product is good, is accepted and when it must be rejected.

Verification is a routine activity, usual, which does not bring new elements, focused on compliance or not with the actions, entries or outputs from the procedure. The procedure is a rigorous construction that involves inputs, actions, activities, outputs (results).

The procedure has the following characteristics:

- determination;
- finite number of steps;
- applicable to repeat the objective achievement.

The procedure is defined and constructed to achieve a goal. It is verified that the procedure was applied or executed correctly and completely.

In the case of the replacing operation of a broken mirror of a car, there is a procedure to remove the old mirror and a procedure for new mirror installation. The objective achievement results from the existence of the new mirror mounted and functional. It is verified that the new mirror has all the right components and do the right thing.

There is a difference between verification and control. For a procedure to be performed is verified that the procedure was well done. Verification is done by one who has executed the procedure or by someone else. The control is done explicitly by someone else and is designed to see if the product or operation is well done. In the case of control, there is not about a procedure to follow.

The following actions are specific to the control, but not to verification:

- control the quality of a product;
- control how moneys were spent.

The action to verify the amount of money spent, meaning that the sum was correct, is specific to verification process.

Testing means that for a program or software product already made someone want to see if it does what he needs. It is verified a routine thing. In the case of mounting 1000 mirrors, it is verified to all of them if the mounting operation was well done.

In the case of testing, there is a unique program and is tested to see if it works. For another program, other tests are carried out. There exists also the notion of knowledge testing, which means testing more individuals having a unique feature.

In the case of web applications, the verification is essential. For an electronic payments application:

- the IBAN of the beneficiary is selected;

- the payment amount is inserted;
- the payment details are filled.

Verification involves validating the accuracy of the IBAN and the correlation between the amount paid and the amount entered by the user. In the case of a bill payment of 30 RON, if the user enters the amount of 300 RON, the verification involves comparing the amount of the invoice with the one submitted in the electronic payment application. For this, there must be a clear procedure for verification.

In the case of the production process, verification is placed between the production operation and the product use operation.

It is considered the procedure:

$P = <I, O, E>$,

where:

I – set of inputs;

A – set of operations;

E – set of results.

The procedure is repetitive and involves effective elements.

$P^{(mine)} = <I_{mine}, O_{mine}, E_{mine}>$,

In this case, verification is a routine matter to see if the theoretical $I_1, I_2, ..., I_k$ are the same with $I_1^{mine}, I_2^{mine}, ..., I_k^{mine}$. The same thing must be realized for operations and outputs. Verification means that $P^{theoretical}$ and P^{mine} are identical.

Verification for outputs is actually exploiting the results (outputs) to see if the user really uses them correctly, or the buttons work properly in the case of car mirrors.

A relationship must be established between the concepts of validation, control, testing, verification, in order to clearly distinguish the difference between verification and others.

In the case of production process, the testing operation appears to the end to see that the product is made according to the specifications established. Testing also serves to quantify the percentage of the specifications that were met. If we set a threshold alpha for product acceptance, then all tests that have results above alpha are validated and all that fall below the alpha are rejected.

Verification in audit processes [6] involves activities throughout the whole period when the team works to get a better result. Upon receipt of the software product, the audit team checks for the following entries:

- specific documentation;
- test datasets;
- source texts;
- executable to be used by customers.

The audit is based on these inputs, the quality of the final result being influenced by the outcome of the verification process of them [7].

During the audit process, verifications are made in order to give assurances that:

- reports were built in compliance with all requirements;
- the indicators underpinning the decision of acceptance or rejection are calculated using all representative data and the chosen indicators are appropriate to the specific of the application that is subject to audit.

The final audit report should be verified to:

- contain all the standard structure elements;
- include all the arguments underlying the final decision;
- provide a clear conclusion, so that the developer know what to do, based on solid arguments;
- eliminate redundant elements;
- manage the quality level;
- provide a logical approach, gradual and rigorous.

The verification of software applications at client level must show that:

- performs basic functions (if the application is on mobile phone and requires a GPS localization, on must see if it make localization correctly);
- the options are working and whether they perform directions according to the keywords generating alternatives;
- validate data entered from the keyboard and how the re-input process is done (with introduction only of the inaccurate data or of all);
- erroneous data are marked with correct messages;

- between what the application require and the data from invoices there is consistence (the number of invoices is different from one utilities provider to another, each utility provider having its own encoding, missing a standardization, so that for the mobile phone, for the energy, for the gas the invoices look different, the field position is given by a so-called absurd and unnecessary custom design;
- the IBAN account position on the invoice and the lack of contracts between utilities providers and banks make the data difficult to enter and the pre-filled payment orders does not exist;
- in addition, the lack of transparency and integration of databases (banks does not read the databases of utilities providers) make also difficult the data entry on payment orders.

We must verify the consistence between the data from the invoice and the ones on the payment order or verification of any other data from the documents, and only after this step we can accept and validate the payment.
Even if the software product indicates errors and it returns to the initial state with error messages or with incorrect fields that are colored in red, verification saves us the reintroduction and validation of data.
It is worse when we select a resource or if the payment amount introduced is higher (wrong), because the allocation of resources is already made and costs are incurred and also is necessary time consuming to fix or spend (for hotel reservation, if we realize the day before the accommodation is paid that we want to cancel, then the accommodation must be paid because the reservation cannot be canceled).
We can say that it is verified the ease to identify a product or a service, knowing that the free search function, where the user enters a string, if it is not accompanied by searches using flexible algorithms based on similarity, will never lead to find the product or street or town.

6 Conclusions

Software testing proved a vital stage of the development cycle from the first pieces of software ever realized. There is no such thing as software without bugs and without extensive testing their number would be by far larger in any software product. Even as the technologies evolve and numerous automated testing products appear, as the applications become more and more complex, the number of bugs in software decreases very slowly. Due to the increased complexity of the software, the number of test cases increases much more than the ability of automated tools and thus many cases remain uncovered.
In order to ease the strain on the testing process, the software verification is done mainly by those designing and developing the software products. After the verification there are small chances that major errors will appear further on, thus saving important resources and limiting the amount of strain in the bug-fixing period. Not only the strain is reduced during bug-fixing, but also during development as correct specifications and code sequences lead to a lower rate of issues between developers. Even if the developers allocate around twenty percent of the development time to designing and executing dev-tests, the verification is a vital process as it can eliminate most of the issues even before they cause any trouble.
The verification of the software optimality assumes the consideration of criteria and areas to work on. Optimizations are done in order to decrease the time needed for execution, to decrease the memory footprint, to lower costs. In areas where there are plenty of technological and performance limitations the optimization of software makes things possible.
Distributed informatics applications should be standardized. Verification involves seeing how easy the information can be accessed, how flexible is the application, in order to verify that the application is user friendly.

References

[1] I. Ivan, B. Vintilă, C. Ciurea, D. Palaghita, S. Pavel, "Autotesting of the Citizen Oriented Informatics Applications," *Ekonomika, statistika i informatika. Vestnik UMO*, MESI, Russia, No. 4, 2009, ISSN 1994-7844.

[2] P. Pocatilu, C. Ciurea, "Collaborative Systems Testing," *Journal of Applied Quantitative Methods*, Vol. 4, No. 3, 2009, pp. 394-405, ISSN 1842–4562.

[3] I. Ivan, C. Boja, A. Zamfiroiu, "Procese de Emulare pentru Testarea Aplicațiilor Mobile," *Revista Română de Informatică și Automatică*, Vol. 22, No. 1, 2012, pp. 5-16, ISSN 1220-1758.

[4] H. Eto, T. Dohi, "Optimality of Control-Limit Type of Software Rejuvenation Policy," Proceedings of 11th International Conference on Parallel and Distributed Systems, Vol. 2, pp. 483-487, 22-22 July 2005.

[5] C. Boja, M. Popa, I. Nițescu, "Characteristics for Software Optimization Projects," *Informatica Economică*, Vol. 12, No. 1(45), 2008, pp. 46-51, ISSN 1453-1305.

[6] M. Popa, "Techniques and Methods to Improve the Audit Process of the Distributed Informatics Systems Based on Metric System," *Informatica Economică*, Vol. 15, No. 2, 2011, pp. 69-78, ISSN 1453-1305.

[7] M. Popa, C. Toma, C. Amancei, "Characteristics of the Audit Processes for Distributed Informatics Systems," *Informatica Economică*, Vol. 13, No. 3, 2009, pp. 165-178, ISSN 1453-1305.

Ion IVAN has graduated the Faculty of Economic Computation and Economic Cybernetics in 1970. He holds a PhD diploma in Economics from 1978 and he had gone through all didactic positions since 1970 when he joined the staff of the Bucharest Academy of Economic Studies. He is the author of more than 25 books and over 75 journal articles in the field of software quality management, software metrics and informatics audit. His work focuses on the analysis of quality of software applications. He has participated in the scientific committee of more than 20 Conferences on Informatics and he has coordinated the appearance of 3 proceedings volumes for International Conferences. From 1994 he is PhD coordinator in the field of Economic Informatics. His main interest fields are: software metrics, optimization of informatics applications, developments and assessment of the text entities, efficiency implementation analysis of the ethical codes in informatics field, software quality management and data quality management.

Cristian CIUREA has a background in computer science and is interested in collaborative systems related issues. He has graduated the Faculty of Economic Cybernetics, Statistics and Informatics from the Bucharest Academy of Economic Studies in 2007. He has a master in Informatics Project Management (2010) and a PhD in Economic Informatics (2011) from the Academy of Economic Studies. Other fields of interest include software metrics, data structures, object oriented programming in C++, windows applications programming in C# and mobile devices programming in Java.

Bogdan VINTILĂ graduated the Bucharest University of Economics, the Faculty of Cybernetics, Statistics and Economic Informatics. He finished the PhD in the field of Economic Informatics at University of Economics in 2011. He is interested in citizen oriented informatics applications, developing applications with large number of users and large data volumes, e-government, e-business, project management, applications' security and applications' quality characteristics.

Gheorghe NOȘCA graduated Mechanical Faculty at Military Technical Academy in 1981, and Cybernetics, Statistics and Informatics Economics Faculty at Academy of Economics Studies in 1992. He obtained his PhD degree in Economics, Cybernetics and Statistics Economics specialty in 2003. He is currently researcher at Association for Development through Science and Education. He has published (in co-operation) 3 books, 16 articles in informatics journals. He has taken part in about 20 national and international conferences and symposiums. His research interests include data quality, data quality management, software quality cost, informatics audit, and competitive intelligence.