ACCEPTANCE TESTING OF SOFTWARE PRODUCTS FOR CLOUD-BASED ON-LINE DELIVERY

BOGDAN WISZNIEWSKI

Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology
Narutowicza 11/12, 80-233 Gdansk, Poland

(received: 12 June 2015; revised: 14 July 2015;
accepted: 20 July 2015; published online: 1 October 2015)

Abstract: Software products intended for on-line delivery by distributors serving an open community of subscribers are developed in a specific life-cycle model in which the roles of major stakeholders are strongly separated, unlike in any other software development model known in software engineering. Its specificity underlines the fact that a distributor of the final product, responsible for its acceptance for publication and delivery to subscribers (users), is not a member of the product development team.

Similarly, users of the product, who normally act as clients in other software development models cannot participate in the process until it is published by a distributor. In the paper a test methodology defined by the industrial IEEE standard is analyzed in the context of that on-line delivery software development model and basic recommendation for the NIWA distribution platform to be operated by the CI-TASK Academic Computer Centre at the Gdansk University of Technology are formulated.

Keywords: quality attributes, acceptance policy, test procedure, testing automation

1. Introduction

Digital distribution of software, either in the form of an executable code ready to install on a user device, or a remote service to be called from such a device, have become prominent in the past decade with the advancement of network bandwidth capabilities and cloud computing centres, serving globally large communities of users. Moreover, any member of such a community may develop an application of any kind and submit it to the centre for immediate distribution to other members. The life-cycle model of a software product developed in such a manner is shown in Figure 1; it involves *phases* and *stakeholders* who have significantly different objectives and roles, when compared to the classic software models used in software engineering today [1].

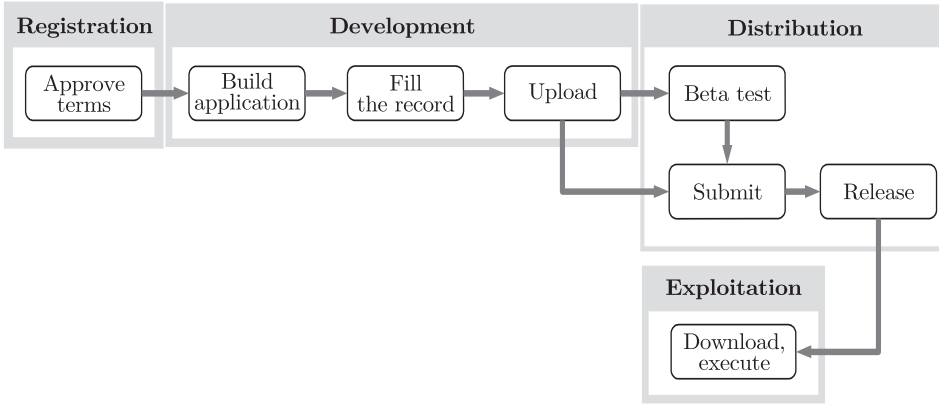


Figure 1. A distributable software product life-cycle

Phases of the distributable software products are the following:

1. *Registration* – a community member accepts the terms of use of the distribution platform and enrolls to the process;
2. *Development* – a registered community member builds his/her application with the use of software development tools just recommended or physically provided by the platform operator. In the latter case the tool may be downloaded by the programmer to work off-line or used on-line as a remote service. Upon completion of the application the programmer provides the platform with the required data and uploads it to the indicated server for further processing;
3. *Distribution* – the submitted application is tested by the platform operator for conformance with the publicly available platform standards. Some platforms may provide programmers with a Beta-test facility, making the submitted application available for a limited time to a qualified group of prospective users. Upon successful completion of tests, the submitted application is accepted and released to the public;
4. *Exploitation* – all community members served by the platform may download and install the released application code on their devices or call it remotely on selected servers of the platform. Throughout the rest of the paper these two scenarios will be referred jointly as on-line software-product delivery, since the NIWA platform is assumed to support user communities in either way.

There are three major types of stakeholders in the process specified above:

1. *developer*, the platform user who invents, builds and submits software applications for on-line delivery;
2. *distributor*, the platform operator, who controls the distribution phase and solely decides on conformance of each submitted application to the standards set up for the platform;
3. *subscriber*, the platform user who downloads the application code to run it on his/her device or calls it on the platform server for the input data he/she provides.

Each stakeholder listed above contributes to the final product quality assurance, where testing plays the key role. However, due to the specificity of the life-cycle model presented in Figure 1, overall organization of the quality assurance process is different from classic approaches [2]. Before going into more detail let us first review the basic terminology.

1.1. Testing terminology

In each phase of its life-cycle a software product is being subjected to *static* and *dynamic* analysis. The former refers to its static components, such as specifications, source code and other documentation, while the later involves experiments with its executable code.

Basic activities that may be performed during static analysis include inspections, walkthroughs and audits. *Inspection* is performed by the evaluation team to verify the consistency of the product source code and its related documentation. It incurs costs to the team, as the activity is time consuming, must be planned and properly documented – including a complete list of all shortcomings found by the team. A *walk-through* is a less formal presentation of the source code by the developer to his/her peers, who may ask questions and suggest improvements, and may be easily implemented for on-line communities with the Internet forum mechanisms. Finally, *audit* is very similar to inspections, except that the evaluation team is independent of the product developers. The activity is less costly, as analysis concentrates on a specific subset of issues compared to the scope addressed by inspections.

The objectives of experiments with an executable code during dynamic analysis are: *testing*, to check for errors of specific types, and *measurements*, to determine various characteristics of the product.

The basic notion in testing is the *test case*, defined as a single element from the enumerable set of all possible behaviors of the tested code that may be observed during experiments. A model used by testers to represent that set, along with a predefined criterion used to decide whether to conclude the test as satisfactory or to continue it to look for more errors, is called a *testing strategy*. Two basic classes of strategies may be considered when planning the test: *white-box* or *structural*, when test cases are selected based on an internal structure of the test item known to the tester beforehand (usually its source code), and *black-box* or *functional*, when the only source of information on the expected behavior of the test item is its formal requirements specification or user manual. Note that, since during experiments with the test item its source code may be used, selection of a testing strategy to evaluate the submitted item by the distributor in the distribution phase may determine the formally required content of the submission uploaded by the developer in the development phase (see Figure 1.

Testing strategies and their related sets of test cases constitute a *test scenario*, which implies a systematic observation of the expected behavior of the test item during execution of its code in the *controlled* mode [3]. By 'systematic' we mean here planning of which test cases and in what order should be exercised,

and what should be the expected result returned by the test item for each test case, whereas 'controlled' execution implies proper configuration and instrumentation of the test item execution code and its execution environment to register to the logfile all important data, including the returned results, diagnostic messages, exceptions, *etc.* A list of all required activities of the tester, including selection criteria for test cases, configuration of the execution environment, test item code instrumentation, logfile structure and test completion criteria constitute a *test procedure*.

The above nomenclature has been introduced by the IEEE standard [4], which defines formally all activities and documents required to perform testing of software products in an orderly manner. In the context of product life-cycle presented in Figure 1 some additional notions will be used further in the paper. One is the *acceptance test*, involving experiments focused on evaluating specific quality attributes of the product, important to its final users. Another is a *conformance test*, being a sort of the acceptance test for the products aspiring to a certain class of products, for which some quality standards as set high [5]. Finally, a *beta test* involves trial exploitation of a software product in its target environment by a limited group of users. During beta tests the product is still owned by its developer, *i.e.*, has not been yet accepted by the distributor.

From the point of view of the final product quality, testing is a process involving activities that are necessary to evaluate how good the product could be. The decision, whether the product tested to be good (in terms of the adopted test procedure) may be used in its target environment, is referred to as *validation*. Validation in the context of the life-cycle model shown in Figure 1 has to be performed by the distributor, acting as a third party with regard to both developers and subscribers. We will argue later in the paper that this specificity necessitates to narrow the focus of acceptance testing to the following set of quality attributes:

- *security*, describing how well the evaluated product is protected against threats external to it;
- *safety*, indicating to what extent the evaluated product can harm its environment;
- *reliability*, implying that the evaluated product functions properly:
- *functionality*, assuring that all functions serving the principal purpose of the product are present;
- *performance*, including various quantitative characteristics of the product;
- *usability*, expressing how easy it is to operate the product interface by its user.

1.2. Quality attributes

Importance of individual quality attributes depends on the assumed stakeholder's perspective. Consider Table 1, where symbols '✓' i '±' indicate 'high' and 'moderate' preferences, respectively, *i.e.*, 'must be' or just 'may be' assured.

Table 1. Quality attributes for on-line software application delivery

Attribute	Developer	Distributor	Subscriber
Security	✓	✓	±
Safety	±	✓	±
Reliability	±	✓	±
Functionality	±	±	✓
Performance	±	±	✓
Usability	±	±	✓

Certainly, a major concern of any developer should be ensuring security of his/her code after submission, therefore, each distributor is expected to handle the code properly and ensure protection of the developer's IPR during evaluation and after the code release.

The developer's preferences on other quality attributes listed in the table may vary, depending on the particular purpose and target users of the application. On the other hand, top preferences of the subscriber would be functionality, performance and reliability, over the remaining ones, depending on the application semantics. In the context of this paper and the NIWA platform, the most important attributes are security, safety and reliability attributes, as the distributor should satisfy both, developers and subscribers, and yet to be forced neither to incur expenses of the development of the submitted applications nor to bear any liability costs after their release. Let us consider the developer's perspective in more detail.

Security; A submitted application should always protect its prospective users to some reasonable degree, which would depend on its purpose and functionality. Qualification of this attribute by average subscribers may vary, especially when they are not aware of what threats could be brought to them by modern software applications working in the Internet. Therefore, distributors should exercise caution and follow the principle of "limited confidence" – their acceptance tests should always involve a standard subset of test cases, no matter what intentions are declared by the developer in the submission record and the related product documentation. On the other hand, developers may reasonably expect protection of their rights, what implies approval of the relevant terms of use of the platform services by each developer upon registration, and providing the platform with all respective security mechanisms to protect the submitted code, *e.g.* its digital signature, secure passwords, encryption, *etc.*

Safety; Both developers and subscribers must be safe from any harm that a submitted application could possibly affect them, and the responsibility to prevent that can certainly be attributed to the former. As in the case of the security attribute before, acceptance tests performed by the distributor should also involve a relevant and standard subset of test cases, no matter what intentions are declared by the developer in the submission record and the related product documentation.

Reliability; Although the submitted application code should always execute properly and return correct results, it would be unrealistic to expect the acceptance tests performed by the developer to cover a full range of functions specified in its relevant documentation, simply because the distributor is not responsible for the product development. However, acceptance tests may address some generic reliability issues by registering such events as application crashes, system error messages, unexpected shut-downs, hang-ups, *etc.* Moreover, distributors may introduce mechanisms enabling remote monitoring of downloaded applications, executed outside of the distribution platform, or implement services to collect bug reports from subscribers to make a decision on permanent or temporary withdrawal of the accepted submission.

Functionality; A developer is free to specify the full range of functions performed by the submitted application. The question is to what extent distributors would be able to test all such functions. By the same argument as before, it would be unrealistic to include a distributor in the development phase, and require him/her to provide enough resources for that, and for the costs hard to predict. Therefore the generic set of the acceptance test cases should focus on exercising each specified function with some randomly generated input data just to check whether it terminates properly, waiving verification of the returned result to the subscriber. The latter would require services for collecting bug reports. Moreover, even if no bugs are reported by subscribers, monitoring of the number of downloads (calls) of each accepted application may indicate if its functionality is of potential interest to subscribers.

Performance; Speed, memory consumption, response time and other characteristics of this kind, which determine popular opinions on the product expressed by the subscribers, are hard to standardize by the distributor. In consequence, selection of the generic acceptance test cases may be difficult, due to the profound diversity of requirements for various classes of applications, execution capabilities of personal devices and multitude configuration options that may be set to use them. One exception is when the application is to be run on one of the servers of the distribution platform – satisfying some specified minimal values of selected metrics might be one of the conditions to accept the submission by the distributor. Note that as far as the computational cloud platform is concerned, some additional effort will be required from its operator (a distributor) to assure scalability of the new service (submitted application) before it can be released. The above mentioned standardization problems would be irrelevant for commercial distributors, who are vendors of operating systems and other software dedicated only to execution devices of the particular type and make. Acceptance criteria would may include then concrete and measurable performance characteristics.

Usability; Arbitrary interaction patterns that may be implemented by developers in their applications also make it difficult to define a standard set of the acceptance test cases. In the context of the NIWA platform, usability of each submitted

application interface may take advantage of static analysis of its related screenshots to assess their legibility, graphical composition, and use of the commonly understood widgets. Experimental evaluation of the usability attribute, *e.g.* by measuring the percentage of errors made by the user when performing some standard tasks, called the *relative user efficiency* (UEFF), or an *average amount of time to learn* (ATL) [3], might be impractical, due to the additional workload that the distributor would have to assign to the group of testers evaluating usability of the submitted application interface. In the case of commercial distributors the evaluations may be reduced to checking conformance to the obligatory layout and functionality of the interface displayed on a screen. In other cases, only some limited performance characteristics of the interface might be measured, such as reaction time of the user gestures or opening and closing time of the application. The related test scenario might involve measuring time that elapses between pressing any button on the interface and observing some substantial change of its appearance.

2. Distributor policies and procedures

Each submitted application must satisfy a predefined set of rules that form the basis of the distributor's acceptance policy. Before proposing such a set for the NIWA platform let us first review policies of the key players in the global online software delivery market, both commercial distributors and free repository hosting services.

Commercial distribution platforms would require submissions to comply strictly with the standards set by the host companies operating them, as each accepted application may potentially become a part of the company's offer. For example, App Store operated by Apple Inc [6] distributes only applications that run under the Mac OS X system, have a company defined user interface layout and are dedicated to the specific execution device. Windows Store operated by the Microsoft Corporation [7] distributes applications that correctly run under the Windows 8 system, and like the former requires applications to comply with the company defined user interface layout, but allows a wider range of execution devices. Finally, Google Play operated by Google [8] distributes applications that run under the Android system, is less restrictive with regard to the user interface layout and allows a wider range of execution devices.

Policies of repository hosting service platforms are more relaxed, as their principal objective is to serve open-source communities and public projects. No particular operating systems, programming languages, execution device architecture or user interface standards are imposed for the code to be accepted, nor are any specific acceptance test of the submission performed. Roles of the developers and subscribers are interchangeable, as platform users may search for specific code items stored in the repository, incorporate them into their projects, open new ones and invite other developers to join them. They provide various revision control

and source code management functionality for that. Examples include GitHub [9] or SourceForge [10].

A short survey of the most important policies of on-line software distributors mentioned before will be used further in the paper as a point of reference for policies of the NIWA platform.

2.1. Submission

Each submitted product has to be inspected first for conformance of its content with the terms accepted by the developer upon registration. The most common features that are considered in the surveyed policy documents are: *suitability*; *user interface*; *objectionable content*; *payments*; *privacy*; and *legal issues*.

2.1.1. Application suitability

Certainly, a software product published on a distribution platform should have a clear purpose. Products that have no reasonable purpose appreciated by subscribers may affect its reputation and in consequence reduce the interest of prospective developers. The following criteria are commonly used to decide on suitability of the submitted application:

- *lasting value* – the most preferred products are applications and services that are attractive and introduce new functionality compared to the current offer of the distributor;
- *duplicates* – applications with user interfaces and functionality closely resembling other products already offered by the distributor usually are not accepted;
- *spam* – multiple publication of the same product is not possible and submissions of this kind are considered spam. One exception is a successive version of the existing product, which may be withdrawn upon acceptance of its new version;
- *demo* – submissions that are intended to only demonstrate functionality of future products are not accepted. It concerns also products that just mimic any real functionality;
- *code size* – if the submitted code is intended to be installed on mobile devices, or is otherwise considered to be too voluminous for download by subscribers, the application is not accepted;
- *advertisements* – policies usually exclude or severely limit acceptance of products which sole purpose is to present users with any advertising or marketing material. Special care is taken to check if the submitted application may require users to accept specific settings of their execution devices that enable or simplify reception of any promotional material after installation;
- *neutral to device* – applications that attempt or request modification of the execution device system or hardware settings are in general considered dangerous and are not accepted. Sometimes it may be allowed that the application asks users for permission to change some settings under the provision that it can automatically restore them to the original values prior to termination;

- *excessive content* – for the sake of inspection costs incurred by the distributor, no excessive material (descriptions or graphics) not related directly to product functioning, implies rejection of the submission.

Besides that, a common sense must be applied by distributors not to promote applications confined to any specific functionality class, technological platforms, execution device architectures, *etc.* to prevent from narrowing the group of prospective users, both subscribers and developers. All products in the offer should be available to the wide spectrum of users, with no excessive costs of upgrading the hardware or software, data transfers, effort to learn how to install and exploit, *etc.* A notable example could be Softpedia [11], which is a library of free Internet software of multiple purpose, formats, operating systems and execution devices.

2.1.2. User interface

Commercial distributors require submitted applications to provide user interfaces with strictly defined layout and structure, conforming to the company's brand awareness strategies. The NIWA platform will be relieved from such constraints, nevertheless the following criteria are worth considering as a recommended component of its policy rules:

- *screen layout* – there are some commonly agreed general rules for organizing the content of graphical displays, making it intuitive and comprehensible to human users, and easy to implement by developers with modern software development tools. Inspection of the application screen layout may then rely on reviews of screenshots included in the submitted material; applications with illegible, confusing or otherwise poorly designed interfaces should be rejected;
- *widget functionality* – action buttons and other widgets, commonly associated with such operations as "save file" "print", "close window", and so on, should rather not implement other functionality that those suggested by their relevant icons and placement in the screen. Moreover, graphical symbols should not be confusing, misleading, fuzzy or inappropriate in any reasonable sense;
- *content redirection* – commercial distributors usually do not accept applications that dynamically redirect users to some external content, not documented by the submitted material. One reason for that is the inability to assess appropriateness of the unspecified content by the means of static inspection, and another is the potential of exposing the subscriber to any undesired marketing material or introducing to the application some additional functionality hard to control by the distributor.

2.1.3. Objectionable content

Distributors should not deliver any content that is not allowed by law, inappropriate, or otherwise giving grounds for legal actions against stakeholders listed before in connection to the lifecycle model specified in Figure 1. The most common exclusions of the content include:

- *promotion of bad habits*, including consumption of alcohol, tobacco and drugs;
- *pornography*, according to its legal definition;

- *discrimination* of people on various grounds, *violence* and *cruelty*, including animals;
- *stalking* with anonymous or prank phone calls and messages;
- missing *parental control* mechanisms for applications addressed to minors.

The exclusions listed above refer in the first place to distributors, but if the application is supposed to be installed on the subscriber's device also to the platform user. Additional complication comes from the fact that downloaded applications may be used in countries with diverse legal systems. To prevent that, distributors should be able to introduce the effective embargo mechanism for selected countries, and accept increased costs of implementing a more sophisticated user management functionality. For the NIWA platform it seems more reasonable to exclude any potentially questionable content. Such a provision, however, would require a relatively deep inspection of the submission, including all texts and images that may be generated, and in the case of computer games also analysis of their story. One example is the popular game where users impersonate drug-dealing gang members, steal cars and may score points by killing policemen and innocent pedestrians. Another problem may be the proper rating of software products to conform to the local jurisdiction of the subscriber. Because of that it might be recommended to limit on-line distribution of software products by the NIWA platform only to registered subscribers, for example students of Polish higher education institutions, companies of the ICT sector and administration agencies.

2.1.4. Payments

Commercial activities of distributors considered in this paper imply them to follow specific policies concerning payments and other financial operations performed by subscribers. Typical operations of submitted applications not to be allowed by the distributor include:

- *payment asked without warning* – the application does not clearly specify its financial purpose to the user;
- *third party goods offered* – commercial distributors may not want to participate in selling goods provided by its competitors;
- *lotteries or gambling* – applicable jurisdiction of the subscriber who wish to use such an application must permit buying lottery tickets or real money gambling.

With regard to the open character of the NIWA platform, and the profile of user communities (both developers and subscribers), it is reasonable to assume that products submitted for distribution should not require users to make any payments, neither directly by calling specialized Web payment services, nor indirectly by displaying real bank account numbers, amounts and instructions how to make payment to the indicated accounts. Inspection of the submitted source code and screen shots should be able to find all messages of that kind.

2.1.5. Privacy

The meaning of the content displayed by each application may be evaluated subjectively by various subscribers and extreme care must be taken by distributors during inspection of the submitted material to avoid possible misinterpretation of the content and potential rejection of the application by its users as violating their privacy or personal freedom. Most important aspects in that regard are the following:

- *personal data* – besides such clearly defined by law "sensitive" data as names, surnames, birth-dates, tax identification numbers, *etc.*, other data may also be considered "sensitive", depending on the context they are used by the application. Examples include registration plates of cars filmed on the road or faces of people caught in specific situations that the application may store or disseminate;
- *tracing and localization* – automatic collection of such data has become a common practice with the advent of mobile devices. One example is automatic georeferencing of photographs by personal devices and cameras. Applications distributing photographs may contribute implicitly to tracing its users, not mentioning those that do it on purpose;
- *offensive language* – clearly, any application displaying content that is likely to disgust its users should not be accepted. Interpretation context, however, may require thorough analysis of local laws and cultural norms of each targeted group of subscribers;
- *religious texts* – translation, interpretation and comments of religious texts may often cause confusion or irritation of various groups of users. Resolving of a possible controversies may require inspection by specialists, employed for that purpose by the developer.

Given the aspects listed above it seems reasonable for the NIWA platform to limit the scope of submissions to the educational and scientific applications only, and in the case of any potential controversy, to use a common sense approach of an inspection team including reasonable academic persons.

2.1.6. External services

Any application or service available on-line may itself rely on other on-line applications or services. Policies of commercial distributors mentioned before are quite restrictive in that regard, and submissions that intend to take advantage of applications and services not specifically indicated in the respective policy document are normally not accepted. The following classes of external services are usually considered:

- *beta testing* – relevant services may be supported by distributors (see Table 1) to improve quality of the product before its final release;
- *push notification* – if any application wants to forward notifications to subscribers on updates, new applications, to fetch user data, *etc.*, it is supposed to use the service specifically provided for that by their host platforms;

- *game management* – a distributor may provide a whole range of services to support on-line gaming with applications installed on personal devices of the platform subscribers;
- *payments* – only specific and trusted payment services may be allowed to be called by the submitted application, either recommended or implemented by the distributor;
- *download and installation* – commercial distributors may not accept applications that intend to download third party products distributed by other distributors;
- *automatic upgrades* – if the developer stipulates any improvement of the submitted product after its release the application code may implement automatic download of the new version upon acceptance of the latter in the future. This mechanism will be combined with the push notification service provided by the developer.

The most valuable provision of the NIWA platform would certainly be the Beta testing facility, which may help developers to improve their code in the target environment before submitting the application for the final approval. The underlying "test flight" service should enable trial exploitation of the application by a limited group of registered subscribers, who may perform their own acceptance tests and relieve the distributor's staff of testing the application's functionality. The level of interest generated by thus Beta tested application may help the distributor to assess its potential for becoming of any lasting value, whereas subscribers may contribute to that goal by indicating possible improvements to the developer. Since the principal objective of the NIWA platform is to serve the open community of independent developers, applications requiring external services that support commercial activities should not be accepted, nor should services of that kind be offered by the platform.

2.1.7. Legal issues

Extreme care has to be exercised by distributors to ensure that the purpose, functionality and content of applications submitted for publication comply with the national and local legal systems of the subscriber. This applies in particular to medical applications, which usually are not allowed for distribution unless authorized by the relevant certification agency. Consequences of using an unauthorized medical application may reveal a long time after its publication in the form of claim suits and other legal actions against the distributor for damages caused to subscribers by a faulty medical application, *e.g.* a smartwatch application incorrectly measuring blood pressure of its user. Another threat of this kind is potential infringement by developers of some third party copyrights, either directly or indirectly. Direct infringement involves a developer distributing somebody's else code, what should be detected by the distributor inspecting the submission. Indirect infringement is more subtle and may be harder to detect in the acceptance phase, if the application functionality can make subscribers to unconsciously perform unlawful activities; they may later take legal actions

against the distributor, *e.g.* claiming reimbursement for a penalty imposed by the subscriber's taxation authority for distributing music of video files using the BitTorrent protocol, or trading virtual objects for real money in computer games [12].The following legal aspects regarding the submitted applications are usually considered by commercial distributors:

- *human health* related purposes of the application, *e.g.* for performing alcohol or drug sobriety tests, collecting various health parameters, controlling dosage of medicines, buying or just advertising them may require certification of some authority independent of the application distributor;
- *copyrights* of any third party logos, music, photos and video clips must not be violated;
- *similarity to other applications*, posing a threat of suing the distributor for plagiarism, is normally not accepted;
- *file sharing*, usually related to the use of some external service not provided by the developer, is not advised, as it puts more workload on distributors to check for possible violations of copyrights, privacy or security of subscribers;
- *impersonation* of others has a potential of laying the grounds for possible wrongdoing and is generally not accepted.

Special consideration must be given to the form of licensing the published code of accepted applications and the content they can deliver to subscribers. Based on that it will be decided what legal aspects of the submissions will have to be inspected.

2.2. Submission reception

Based on the policies of the key commercial distributors, outlined in Section 2, the following components of the submission should normally be required by the distributor:

- *application source code* and files with all text and graphics used by the application during run-time for inspection – to exclude objectionable content listed in p. 2.1.3;
- *screen shots*, made by the developer at some key moments of the application run-time, to enable initial screening of submissions, and rejection of interfaces with quality below the predefined threshold of acceptance – before the actual testing of the submitted execution code can take place;
- *installation package* with the executable code, installation scripts, configuration files, *etc.*, enabling developers to reproduce the generation and installation path of the submitted product.

By assuming that the average NIWA subscriber has basic computer programming and administration skills, acceptance testing of each submission may be reduced to a reasonable minimum: to check if installation scripts execute correctly and the executable code is free of any common malware. If, however, submitted applications are to be delivered as services by the platform, dynamic analysis of

the executable code will be more exhaustive. Further in the paper a more comprehensive approach will be proposed, based on the software industry standards [4].

2.3. Submission acceptance

Inspection (static analysis) of the submission content is not sufficient for accepting it for publication, for the reasons given before, and certainly should precede testing (dynamic analysis) of its executable code. Similarly to the static analysis, its dynamic counterpart also requires careful selection of features to be analyzed (tested). It is worthwhile to refer again to acceptance policies of the key commercial distributors mentioned before and define such features for applications intended for the NIWA platform. They should not put too much workload on testers analyzing each submission, and at the same time should guarantee a reasonable acceptance rate of published applications by the community of subscribers. By accepting a submission its distributor gives subscribers a sort of warranty that specific features listed in the relevant acceptance policy have been checked. Such a warranty is for a limited time only, since during the exploitation phase subscribers may discover defects not revealed earlier in the acceptance phase or their interest in the application may gradually decrease below a reasonable level.

A generic set of features to be tested during the acceptance phase of applications submitted for distribution by the NIWA platform is proposed below. Further in the paper several concrete testing strategies will be recommended and illustrated with a realistic case study of a specialized software library – one of the candidate products planned to be published there.

2.3.1. Acceptance criteria

According to the terminology introduced in Section 1.1 acceptance testing requires testers to select a set of quality attributes that are used to assess a software product of interest. By determining threshold values, or setting various intervals to differentiate them, acceptance criteria are defined and used to formally accept or reject the analyzed product [3]. Below we consider in more detail the relevant features of software products submitted for distribution by NIWA that will be tested to access attributes listed in Table 1.

Security; During each operation supported by the available functionality of the application its users must be protected against any threat that may be posed by its execution environment. Typically, features that may be tested in that regard include: presence of viruses and malware in the code, collection of sensitive (personal) user data by the application, dynamic modification of its code, logging user activities or location, as well as handling of payments by it. Some of these features might be analyzed already during inspection, *e.g.* by checking what user data are collected or what services are used to handle payments, but only systematic monitoring of the properly instrumented code during runtime may allow testers of the application to verify clarity of intentions of its developer.

Safety; Execution of the application should also pose no threat to its operational environment or user. Features to be tested in that regard usually include: performing I/O outside of the designated memory space, inadvertent use of execution device hardware, distraction of users operating various types of vehicles (drivers or pilots) and system integrity. These features should be tested for each product submitted for publication whenever possible. In particular no application will be allowed to compromise integrity of its underlying operating system.

Reliability; A subscriber may reasonably expect each publication published by the distributor to reliably perform its functions, specified in its respective manual. The application should correctly respond to all legitimate user actions and reject or diagnose the illegitimate ones. Assessment of this attribute during the acceptance phase may be a demanding and costly task for the testing staff, as applications may implement arbitrary functionality. Therefore, the only reasonable support that distributors may provide to the open community of subscribers is assuring *stability* of the published application code, implying that during its execution it will not unexpectedly shut down, hang-up, fail, or otherwise exhibit any visible and obviously erroneous behavior. For non-commercial distributors some attention during acceptance testing may be given by testers to *portability* of the code. This feature should be addressed by developers in a submission record (see Figure 1), by specifying alternative execution devices the submitted application may run on, operating systems, I/O devices, and so on. During acceptance testing the developer's staff should then be able to verify that. If the range of possible execution devices is particularly rich, workload for testing portability of the application may be significant. Distributors may cope with that by providing the Beta testing facility mentioned before. An interesting solution in that regard has been provided by Microsoft to Windows Store developers, who may remotely access any device currently in the offer by Nokia by calling a dedicated Web service, before actually submitting their applications for publication [13].

Functionality; Besides reliable execution, the functionality of the application should be complete and implemented in full, as specified in the manual or other documentation, usually distributed with it. Testing the application functionality is the sole responsibility of its developer, and should be completed before its actual submission for acceptance and publication. Distributors rarely participate in the development process of submitted applications, so acceptance tests that they are able to design can focus only on some general features. One is the added *value* that the application's functionality can bring to the community of subscribers – the application must perform some real computation rather then mimic it (demos are not accepted), must not create a false impression that it solves a real computational problem, should not mostly advertise other products instead of performing any useful computation, and so on. Another general functionality feature of submitted applications that may be tested by the

distributor is conformance of the graphical user interface and menu components of submitted applications to the commonly understood functionality of standard widgets used by them, like action buttons, sliders, combo boxes, *etc.* If the distributor decides to perform yet more exhaustive testing, the next candidate feature worth considering is correctness or precision of diagnostic messages displayed by the application. Testing application functionality may also focus on checking whether the observed behavior of the tested application is consistent with the functionality class declared by the application's developer, *e.g.*, a text editor really edits texts, a navigation tool displays locations on maps, *etc.*

Because of a very wide range of functionality classes of applications to be accepted for distribution by the NIWA platform, testing of their functionality will not have to be performed often – probably only when the NIWA staff is somehow involved in the development phase. It may be expected that the community of subscribers will be able to quickly verify the value of each published product; applications that are not able to attract attention of the community, or loose it after some time, may be simply withdrawn from the publication as not functional enough.

Performance; Most popular applications distributed today are intended for mobile devices, therefore performance related features that have to be tested during the acceptance phase may be quite specific and related to certain operational limitations of such devices. Typically they concern the size of the executable code that must be downloaded via WiFi or cellular connection and installed on the device, the bandwidth required for using the installed application, its time of reaction to the user stimulus on the touch-up screen, battery load consumption, quality of media playing, *etc.*

Performance testing of products submitted to NIWA for publication, may focus on checking how the specific values of metrics declared by the developer upon submission compare to values measured by the NIWA staff during acceptance testing. The experiment would require configuring the execution environment for the application as specified in its submission record, and next measuring the indicated metrics. If the measured values are not worse than the declared ones, the application may pass the test. Applications intended to run on a distributor's server would require more work on planning and executing performance tests, especially if the distributor (in this case the NIWA platform) declares a specific level of scalability.

Usability; Distributors prefer publishing a product that is usable, *i.e.*, a level of difficulty of using it by subscribers should not exceed their average level of competency. Typical features that may be tested with regard to this attribute include: *understandability*, reflecting how fast users can learn to use the application, or what the error rate is when they use it, *conformance* of the application interface to the standard (if any) for the class it belongs to, *complexity* of its interface, its *aesthetics*, as well as support the developer can provide to users of

the application. In the case of commercial distributors, designing usability tests is straightforward: each distributor requires submissions to conform to standards set-up for all products with the distributor's logo. Any product deviating from the standard is considered by the distributor to be of lesser usability. For the NIWA platform no specific interface standard can be defined, except for some general guidance and commonly agreed good practices, namely aesthetics of the interface and its minimal functionality, *e.g.*, the *undo* and *help* buttons. Proper evaluation of understandability and complexity of the interface may be left to subscribers (application users), who may express their opinions and suggest improvements on the product forum. Based on that the distributor may decide to withdraw publication of the product from the NIWA platform.

3. Test scenarios

The specificity of the life-cycle model of the software product intended for on-line delivery may affect the test scenarios to be considered by developers in several ways. Firstly, communication between stakeholders is determined by different distribution of their roles when compared to other life-cycle models. Note that developers, distributors and subscribers do not form a single team and their objectives are different: developers design and implement their applications without any formal participation of users (who are clients in other models), whereas distributors act as independent auditors, who normally do not suggest improvements to the submitted software product and perform acceptance testing without its future users. Secondly, acceptance testing of the same applications is performed several times: by the developer, who wants the application to be accepted, by the distributor, who checks the submitted code against the acceptance criteria currently in force, and by the subscriber, who checks if the published application can satisfy his/her needs. The developer usually cannot take advantage of test results obtained by the distributor, on the other hand, crash reports recorded by the distributor may be returned to the developer, who can use them to improve the product. Crash reports may also be returned to developers by subscribers, if only the application code enables that. Finally, the distributor may provide the developer with meaningful data from the market, concerning various statistics on the use of the published application. Gathering such data directly by developers would incur costs and requires expanded infrastructures, usually operated by big companies. One example of such support may be Google Analytics, the Internet statistics service available to developers subscribing to Google Play [14].

The life-cycle model of products to be published by the NIWA platform includes four activities that involve static or dynamic analysis methods explained earlier in the paper: application *building* during the development phase, and *beta test*, *submission* and *release* in the distribution phase. Below we characterize shortly test scenarios within the context of each activity.

3.1. Build application activity

Test scenarios considered by developers during the development phase are beyond the scope of this paper, since they are practically free to choose any software design patterns, development tools, implementation techniques and languages, and of course acceptance testing strategies, when building their applications. Commercial distributors, however, may provide developers with specialized IDE tools to ease the process of application building and testing, as well as request conformance to specific coding standards; they may also deliver additional services to advice developers on various test design and implementation issues. Although delivery of specific development tools is not planned by the NIWA platform, support and advice may be provided by enabling a forum service for exchanging opinions between developers and subscribers.

3.2. Beta test activity

A distributor may grant some resources (computation nodes, memory space) to the developer willing to beta test his/her application with the help of its future users. Releasing the application before submitting it for publication has obvious advantages – to the developer, who can access a wider group of subscribers at the distributor's expense, and to the distributor, by reducing the workload of its testing staff. Commercial distributors often take advantage of this provision [6]. Clear distinction of the *beta test* activity in the distribution phase from the *execute* activity in the exploitation phase may improve communication between the developer and its subscribers, contribute to the level of maturity the submitted product may get (mostly because of the potentially rich set of testing strategies used by the community of subscribers), and reduce a time to wait for acceptance of thus tested submission. In the initial version of the NIWA platform the beta test facility enabling direct interaction between developers and subscribers is not enabled, however, in a longer run such a service will have to be provided.

3.3. Submit activity

As indicated earlier in the paper, a proper completion of the *submit* activity requires a distributor to statically analyze the submitted material with regard to the textual and graphical content, interface language, structure of the source code and size of the binary code. Depending on the assumed depth level, the content may be inspected, reviewed or audited to check if the submitted content is appropriate and the application is written well (not necessarily correct). The policy of the NIWA platform has not been yet formally set up, when writing this paper – the results of the analysis of policies of the selected distributors, presented in Section 2 should provide a base for that.

3.4. Release activity

Positive results of the static analysis performed during the *submit* activity imply performing *release* activity as the final activity of the distribution phase. Its

completion requires making a decision on whether the execution code satisfies the acceptance criteria set up in the relevant policy. If so, the product is released to the exploitation phase with a warranty to subscribers that it conforms to the policy rules published by the distributor. During the latter phase the product is still being evaluated by subscribers, interacting with themselves and the distributor using communication forum services, provided by the latter, to exchange opinions, suggest improvements, indicate errors, even participating in the development of somebody else's code [11]. A distributor moderating the forum may decide on withdrawing the publication from its offer and advise the developer to improve it. Also developers may use such a feedback from subscribers to decide on submitting improved versions of their applications. Subscribers may also directly interact with developers, if the latter are included in their applications contact data.

The executable code may be exercised by subscribers in arbitrary ways on their devices, but when executed on the distributor's servers, additional services for logging user actions may be provided, certainly more informative than crash reports mentioned before.

If the underlying policy of the distributor requires performing acceptance tests by its staff more effort is required to complete the *release* activity. Depending on whether the submission included the design documentation and the code, or just the code (source and executable), the distributor may choose to use *black-box* or *white-box* strategies [3]. However, if the distributor's staff did not participate in the development of the submitted application, white-box testing may require too much effort to analyze the semantics of the submitted source code. This is because selection of test cases based on the source code is much more efficient when testers can directly interact with the its programmers or designers, as anomalies found in the code structure may have to be clarified before actually running the test experiments; this kind of interaction is not provided by the life-cycle model presented in Figure 1.

The prospect of providing developers with white-box test case selection mechanisms by the NIWA platform, in the context of the *beta test* activity, remains open. It will not imply participation of the distributor's staff in the product development process, as in the case of other life-cycle models [1]. One important argument against white-box testing of a submitted application by the NIWA staff, is its strong dependence of the required analysis on a language in which the application code is written. Since no specific requirements on implementation languages or development tools in the related acceptance policy will be given by NIWA (quite the opposite to policies of commercial distributors), automation of test case selection may be a demanding task, even for such strategies straightforward to implement as *branch* or *mutation* testing [3]. For each submission the NIWA staff would have to plan a separate test team or provide expert support to developers.

Black-box strategies could be much more flexible in optimizing the distributor's costs of acceptance testing: selection of test cases would not require source

code analysis, and its automation is fairly easy to implement, *e.g.* the *Monte-Carlo* testing strategy [3]. In Section 5 examples of such strategies will be given for one of the first submissions to NIWA. Another argument in favor of black-box strategies for acceptance testing of applications submitted to NIWA is that their features to be tested, explained in p.2.3.1, do not require analysis of the application code semantics.

In summary, acceptance testing of each NIWA submission should be able to check if its code:

- does not do anything not allowed by the platform's policy (criteria of safety and security);
- does not shut down unexpectedly, crash or hang-up (criterion of reliability);
- reacts to the user's stimuli in a predictable time (criterion of performance);
- performs reasonable computations (criterion of functionality);
- delivers an understandable and logically structured user interface (criterion of usability);
- does not require for its installation and execution any services or applications not generally available, especially not published by NIWA (criterion of usability).

4. Methodology

Features of the application code submitted to NIWA will be tested in accordance to the methodology defined by the IEEE 829-2008 standard [4], which specifies a set and structure of documents necessary to systematically plan, execute and evaluate experiments. Implementation of this standard will involve a predefined sequence of standard steps, implementing the NIWA acceptance policy. Basic facets of the above mentioned methodology will include: test item transmittal, test plan, test design, specification of test cases, specification of the test procedure, test logging and incident reporting, and finally the test summary and conclusions. Since submissions may consist of multiple components, *e.g.* a library of functions [15], different testing models, test case selection strategies, and test scenarios, may have to be considered and implemented for the same software product; throughout the rest of this section we will refer to each tested component briefly as the 'item'.

4.1. Test item transmittal

In order to be tested, items should be made available to responsible testers at some specified location on the distributor's server. They will be in a form ready to use by testers, specified formally by the document called the *test item transmittal report*. The report provides complete information enabling testers to perform all the steps mentioned before. The NIWA platform may support automation of these steps, based on such formal specification, with its workflow management service; it would, however, require additional work to integrate some third-party test automation tool with that service. The advantage might be

reduced effort spent on performing the test procedure, orderly execution of its steps and repeatability of the entire process [16]; so far, all activities following the test item transmittal must be performed manually by NIWA testers.

4.2. Test plan

Selection of features to be tested for each item, and their proper justification, provides a basis for the test plan. In the model shown in Figure 1 the features are already implied by the respective distributor's policy and related acceptance criteria. Therefore, the NIWA tester will start the test planning activity by selecting the most relevant (or all) features from the list specified in the previous section, and decide how the correctness of the test results should be verified. In many cases the rule will be fairly simple, *e.g.*, the tested item must launch properly on a given device. For more complex features, *e.g.*, a service reacts to the user's stimuli in a reasonable time, specific quality characteristics must be defined, *e.g.*, the service scales-up linearly [2].

In the case of implementing test automation by the NIWA staff it may be useful to consider a standard catalog of recommended methods for checking whether the obtained results are in agreement with the ones required (by the distributor) or declared (by the developer). Such standardization would be possible and realistic, since the set of features to be tested for all submissions is fixed. If NIWA testers are required to test also the functionality of applications being submitted, in particular to verify the correctness of their computations, a specialized CAST tool may have to be acquired or developed; one example is the Rational Test Workbench tool, well integrated with many operating systems and IDE tools [17]. However, introducing such a facility to the NIWA platform would not be economically viable. Instead, for its open character, the major role of NIWA functionality testers should be attributed to the community of developers and subscribers, by keeping responsibility only for the acceptance criteria underlying the remaining quality attributes discussed in detail in the previous section.

4.3. Test design

Test items listed in the transmittal report – intended for testing according to the assumed test plan – require testers to carefully design the test. This step involves determining the features to be tested for each single item, defining a relevant test case selection strategy for each feature and a method for evaluating the results obtained for each case. Selection of test cases should be justified, based on a comprehensive analysis of the product documentation and in a way enabling evaluation of the results obtained for each case. The list of test cases must indicate relevance to each feature to be tested and the respective decision rule for accepting the result.

Four specific techniques are planned for the NIWA platform to exercise test cases specified in the test designed for each item: *virtualization* of the application's target execution environment (sandboxing), *exploration* of functionality and

dynamic content of its user interface, and *measurements* of its selected physical metrics.

4.3.1. Execution environment

The virtualization technology will enable effective separation of running applications from the NIWA platform hardware and system. Such separation is needed to protect subscribers from the yet untrusted code and to enable its controlled mode of execution. As explained in p.1.1, this mode involves instrumentation of the run-time code and its execution environment, which enables monitoring and logging of various operations concerning access to the CPU, RAM, system registers and communication ports of the application's target execution device, as well as input and output streams of the application. The application is placed in a sort of a sandbox, and cannot affect the original system of the distribution platform.

The underlying distributor's system running the sandbox remains transparent to the virtualized execution environment, where the tested item is exercised. Any event of interest generated by that item occurs only inside the sandbox and may be easily registered to the test log without affecting the behavior of the item, in particular without causing a *probe effect* – an unwanted phenomenon occurring during execution of the instrumented code [3]. The test log content may later be analyzed by testers to assess whether the tested item can really pose any threat to its target execution environment. In extreme cases execution of the code inside the sandbox may be interrupted and the item rejected.

Another advantage of using the virtualization technology for acceptance testing of submitted items is the possibility to configure its target execution environment exactly as specified in its transmittal report, and run experiments with many items in parallel.

4.3.2. Exploration of interface functionality

Applications published by NIWA are supposed to communicate with users, devices and systems in arbitrary ways, from textual or graphical user interfaces up to software APIs and Web services. Their communication may involve events, generated asynchronously and in any order, as well as single data or streams of data. As argued before, the black-box approach will be used to provide a basis for the test case selection, and product functionality will not be considered a primary set of features to be tested. In consequence, the Monte-Carlo strategy seems to be the best candidate for automatic selection of test cases – drawn at random from the set of equally probable cases. Such selection is justified, given the fact that events and data related to various features determining evaluation of quality attributes considered in Subsection 1.2 are equally important from the acceptance policy point of view. Moreover, their uniform distribution over the set of possible cases to be exercised guarantees their systematic selection.

The Monte-Carlo strategy applied to test a graphical interface would imply generation by testers (or a testing tool) of arbitrary sequences of events (gestures) captured by the widgets present in the application's window, as simple as mouse

clicks on action buttons, ticks of radio buttons, shifts of horizontal or vertical sliders, or more complex, as selection of specific entries in pull-down or pop-up menus. For each combination of such events the tested item should react reliably, by performing some computation or returning a diagnostic message. CAST tools, like the Rational Functional Tester [16], provide mechanisms for logging events and data returned by the application under test in response to such test scenarios, often implemented with specialized test scripts written or recorded by testers before. A similar mechanism may be introduced to the NIWA platform: sequences of user events could be generated randomly based on the initial sequence recorded by a tester, and next provided to the application running in its sandbox to be monitored for unreliable behavior. This class of test scenarios is fairly easy to implement for application interfaces based on the popular *Model-View-Controller* (MVC) [18] or *Model-View-ViewModel* (MVVM) [19] design patterns, which distinguish objects responsible for intercepting user generated events on the screen from objects responsible for handling them. By monitoring the communication between these two classes of objects, the logging mechanism would be able to register all exceptions indicating any wrongdoing of the item under test. This technique of a *test pilot*, as called in the literature for its resemblance to performing various maneuvers of the prototype plane to check its structural integrity, enables exercising practically any possible combination of user gestures on the analyzed interface; only the time available for the experiments could be a limit here.

A variant of this technique for exploring the textual interface is also possible: for user commands typed in terminals or streams of arbitrarily encoded data input to Web service ports, as well as software APIs. In each case the underlying principle is to generate a random sequence of, user commands, Web service calls or APIs, respectively, and the subsequent execution of tested items in the controlled mode in their respective sandboxes.

Besides dynamic exploration of an interface of each tested item, automation of static analysis of its content may also be implemented by NIWA. First of all, the source code and other related text files of submitted applications may be searched for certain phrases of keywords. It may be assumed that developers will not intentionally include any unlawful or inappropriate content in their submissions, however, some analysis in that regard is recommended – to aid developers who not aware of various cultural or linguistic nuances. Related texts could also be spell-checked or even verified for their originality with the existing SowiDoc anti-plagiarism tool to be integrated with NIWA in the near future [20].

A bigger challenge for NIWA testers could be analysis of the graphical content of submitted applications. The principal source of information in that regard would be files with graphics used by the application and its screen-shots included in the submission package. Verification of their content may require making screenshots when exploring the application functionality described before, and logging them along with other data for comparison with the screenshots provided by the

developer. They may also provide material for assessing the appearance of the user interface and conformance to the standards or recommendations for user interfaces, if planned to be published by NIWA in the future.

4.3.3. Measurements

Dynamic analysis of tested items usually involves measuring their selected physical parameters, and determining, if necessary, various characteristics of interest [2]. They will enable testers to verify performances of the tested application declared by its developer in the submission record and contribute to the overall assessment of its reliability. Black-box strategies to be recommended for that are [3]:

- *load tests*, which call for data that are close to or exceed their type range declared by the developer in the related user or installation manual. For the former case the tested item should still be able to complete its operations, whereas for the latter, it should reject the input;
- *volume tests*, which involve data of sizes from moderate to extreme. The results enable testers to determine the scalability of the tested item and identify its operational limitations;
- *stress tests*, which involve high intensity of input data or events. The tested item should be able to handle them in the orderly manner and not to hang-up or crash.

A properly instrumented sandbox, where the running item is tested using the strategies listed above, could provide log data to analyze performance of the tested item by testers and to generate automatically crash reports for the developers. If the load, volume and stress tests are to be performed in the context of the beta test activity supported by the NIWA platform, the crash reports should be comprehensive enough to enable developers to identify, localize and eliminate defects from their applications. Some distributors, *e.g.* AppStore [6], enable developers to remotely configure sandboxes to individually control the monitoring level of each application. This would be a realistic form of a direct "costless" support to developers by NIWA.

4.4. Test case selection

The test design described in Subsection 4.3 relates test strategies with features of the item to be tested. Based on that test cases may be systematically defined and selected, including input data, expected results, pre- and post-conditions describing the proper relation of the tested item with its execution environment, events, interrupts, exceptions, triggers and other asynchronous stimuli that the item should react to, and what these reactions should be. Because the criterion of functionality would be of less importance to the NIWA distributor that the other ones considered in Subsection 1.2, specification of test cases may be much less formal than required by the mentioned IEEE standard. In particular, no in-depth analysis of semantics of submitted applications will be needed, described more or less precisely by developers in various pieces of

product documentation: specification requirements, architectural design, reference and user manuals, even the source code. In fact, a developer and subscriber of the NIWA platform would rather not be willing to prepare or submit such a rich set of the product documentation for publication. Therefore, instead of formally specifying test cases the NIWA testers will mainly be configuring the sandbox and setting its selected parameters according to the following:

- values in a specific range or set, from which elements are drawn at random;
- time period in which the item is expected to return results and get ready to perform another computation;
- minimal configuration settings declared by the developer for the given item in its user manual or submission record, concerning CPU, RAM, disc space, bandwidth, version of the operating system, other software installed, *etc.* Actual settings should be recorded in the test log before starting the experiment.

Developers may also indicate when testers should step in the experiment and perform some additional activity, *e.g.*, switch off a physical device, or do some additional processing of data recorded in the log after the experiment. For the classes of applications considered by NIWA, however, such extras are expected to be rare. Another issue is exercising test cases in some predefined order – CAST tools can support that by providing scripting mechanisms to implement test scenarios. For the NIWA platform such a scripting mechanism in not planned in the near future, so implementation of more complex scenarios would require increased workload of its testers. For example, acceptance testing of the KOALA library submitted to NIWA for publication and discussed in the next section would require considering interdependency of test cases.

4.5. Test procedure specification

Since the list of acceptance criteria for software products submitted to NIWA for publication would be the same for practically all submissions, the test procedures could be to a large extent automated. Owing to that, the workload of testers could be optimized, the procedure made repeatable and assessment of products of different developers unified. The steps of the procedure specified by the IEEE standard will be:

1. Set-up the execution environment by configuring the sandbox, installing the test item in it and generating a relevant set of test cases;
2. Start (launch) the test item;
3. Execute the first (next) test case from the generated set;
4. Log selected data in the test log (input data, results, metrics);
5. Suspend execution if unexpected events occur (shut-down or crash) and log any other useful data, *e.g.*, make a core dump. Depending on consequences the procedure may have to be started again (step 2) or even the execution environment may have to be reset (step 1);
6. Resume execution (step 3) if in the set generated in step 1 there are test cases not yet exercised;

7. Close the test log file;
8. Conclude the test procedure and remove the sandbox.

The procedure listed above produces the test log file.

4.6. Test logging

Records of the test log file represent in a chronological order all events registered during the experiment. Since the functionality criterion is of a secondary importance to the distributor considering acceptance of submissions, information collected in each respective record of the log file may be much less detailed as required by the IEEE standard, unless the log content is planned to be analyzed further by the developer in the beta test mode. A general recommendation to NIWA is to provide the test log record structure with just as much information as would be sufficient to properly justify possible rejection of the submission:

- *Results*, including the accepted range of values and the actual value of each measured parameter, input data for the test case, relevant system messages, exceptions, and user actions that were registered, *etc.*;
- *Configuration* specified by the respective settings of the execution environment when recording the results;
- *Incidents* (unexpected events) observed by the tester, not described in the relevant test scenario, which disrupted or even prevented proper conclusion of the test procedure, along with other events observed immediately before and after each event, and description of activities attempted to resume the procedure, *e.g.* attempts to reset sandbox parameters, unknown error occurring in the tested item, *etc.*

4.7. Test incident reporting

If incidents were recorded in the test log a comprehensive report based on the recorded data must be prepared by testers. Each anomaly should be briefly described, including the relevant step of a test procedure, the respective test case exercised and operations attempted by testers to counteract it. Owing to the role of the distributor in the life-cycle model shown in Figure 1 – who is not responsible for the evaluated product development, suggestions on what the origins of the reported problem could be, or what additional activities might be performed by the developer to localize it in the code, as recommended by the IEEE standard, will not be needed. Just for the internal use, the distributor may include in the incident report information on who the testers were and what their recommendation concerning the product approval was.

4.8. Test summary

A distributor makes its final decision on acceptance, rejection or resubmission of the software product based on the test log and incident report, in the test summary document. The summary must be returned to the developer with proper justification and include the following elements:

- *Final conclusion* including general assessment of the test items with references to the transmittal report (list of items, versions, declared settings of the environment), a test log and incident report;
- *Completeness assessment* of the acceptance test procedure indicating all tested features and their relationships;
- *General assessment* including all anomalies observed and investigated further, as well anomalies just noted down, with explanations, if necessary;
- *Detailed assessment* indicating specific test cases and all key results recorded for them to characterize each respective quality attribute of the product with its related criterion. If a modification and resubmission is to be requested, the developer must specify what changes will be made, and possibly until when. The risk of accepting the submission may also be assessed, if the test results are not complete or reliable enough. In such a case the distributor may require the developer to perform more testing before resubmission. The number of resubmissions should be limited, to prevent developers from abusing resources of the distributor, if the latter does not provide any beta test facility;
- *Test wrap-up* providing qualitative assessment of all test related activities performed by the distributor, including time and effort of its testing staff, the total volume of test logs, CPU time, *etc.*

If the submission is decided to be rejected or resubmitted the test summary document should be returned to the developer with the information from what location the latter may download files with test cases and a test log file.

5. Case study: the KOALA library

A methodology of acceptance testing of software products submitted for on-line delivery by cloud-based distribution platforms considered before may be illustrated with one of the first submissions to NIWA, the KOALA library [15]. This submission has all the features considered before: it may be used by the community of NIWA subscribers as a library of classes to download and integrate with a third party code to develop new applications, or as a Web service with a high performance computing capability provided by the CI-TASK Academic Computer Centre at the Gdansk University of Technology. It was developed by an independent team of developers. Moreover, NIWA testers do not have time and budget resources to perform in-depth functionality tests of it (as they would have to if they were members of the KOALA development team), nor do they have access to a complete documentation of that product.

Below we will review all facets of the proposed methodology, and we will consider assessment of the functionality attribute of KOALA, as less important than other attributes listed in Subsection 1.2.

Submission analysis; The KOALA library provides over 75 methods coded in C++ and implementing most of the known algorithms for graph-theoretic problems defined in discrete mathematics, listed in Figure 2.

```
avg,                    createWattStrog,        isDAG,
bellmanFord,            createWheel,            isInterval,
bfsComponents,          criticalPath,           K,
bfsTree,                deg,                    lexbfsComponents,
bfsVisit,               deselect,               lexbfsTree,
blocks,                 dfspostComponents,      lexbfsVisit,
cartesianProduct,       dfspostTree,            lexProduct,
createBarAlb,           dfspostVisit,           map,
createCaterpillar,      dfspreComponents,       max,
createClique,           dfspreTree,             maxFlow,
createCompBipartite,    dfspreVisit,            maxMatching,
createCompKPartite,     dijkstra,               min,
createCycle,            edgeEnd,                modules,
createEdgeIds,          edgeStart,              outdeg,
createEmpty,            eulerCycle,             print,
createErdRen1,          eulerPath,              randint,
createErdRen2,          finde,                  random,
createFan,              findv,                  readG6,
createHorizPath,        get,                    sccComponents,
createLineGraph,        haskey,                 select,
createLineGraphDir,     hasseDiagram,           spanningForest,
createPetersen,         indeg,                  strongProduct,
createRegTree,          isChordal,              tensorProduct,
createVertexIds,        isCochordal,            topologicalOrder,
createVertPath,         isComparability,        typeof.
```

Figure 2. The set of KOALA methods

Systematic testing of procedures from the above list requires a test plan for exercising all respective methods in the library, and when required, also taking into account their internal relationships.

Test plan; Each method in the submitted code will be tested with regard to all the acceptance criteria listed in p. 2.3.1. For the version intended for downloads by subscribers it will be reasonable to compile and exercise the code on all the operating systems currently used by NIWA subscribers, or at least on target systems explicitly indicated by developers, like Microsoft Windows 7.0 and 8.1, Linux Ubuntu 14.10 and Fedora 21, and Apple OS X 10.10. For the Web service version of the library, the operating system may be selected by the distributor, however, more effort may be required to assure scalability, if high performance computing capability of the NIWA platform is made available to subscribers. The submitted source code of the library will be compiled to generate the executable code with customary compilers for each instance of the operating system installed in its respective sandbox. Each method will be repeatedly executed for automatically generated input data, and the results and associated events recorded in the test log (results) and each respective operating system log (events)

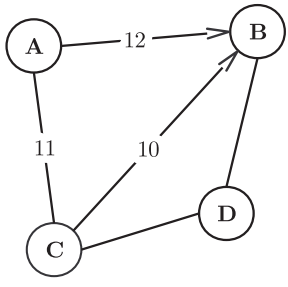
Testing strategies; Automatic generation of input data will use the Monte-Carlo strategy. For each thus generated single input the results produced by all

operating system instances will be compared. All results produced for the same input are expected to be identical, also behaviors (completion of each computation in a realistic time, no hang-ups or crashes) should be similar. Any difference will indicate potential reliability and portability problems.

Test cases; As indicated by KOALA developers in their submission, the principal input data for its methods are graphs, specified in two formats: pure textual (internal) format or marked-up (GraphML) format. The internal format was described in the submission informally as a file of records, each one specifying one node of the graph:

```
<no_vertices>
<vertex_id>'('<vertex_data')'<no_edges><direction>
   <neighbor_id>'('edge_data')'
...
<vertex_id>'('<vertex_data')'<no_edges><direction>
   <neighbor_id>'('edge_data')'
```

where `<direction>` specifies edges as indirected ('−'), outgoing ('>') and incoming ('<'). For example, a graph in Figure 3 can be specified in the internal KOALA format as shown in Figure 3.




```
4
0(x:260.0,y:101.0,label:A) 2 -1(label:11) >2(label:12)
1(x:335.0,y:232.0,label:C) 2 >2(label:10) -3
2(x:376.0,y:100.0,label:B) 1 <3
3(x:428.0,y:226.0,label:D) 0
```

Figure 3. Example graph representation in KOALA

Random generation of input data (syntactically correct textual specifications) would require testers to implement a procedure for that, what may cost additional workload of the NIWA staff. Fortunately, in the submission the following information could be found:

> *Class* `Creator` *implements methods*
> `erdRen1` *and* `erdRen2` *for the Erdos-Renyi* $G(n,p)$
> *and* $G(n,M)$ *model,*
> `barAlb` *for the Barabasi-Albert model,*
> *and*
> `wattStrog1` *and* `wattStrog2` *for the Watts-Strogatz model.*

Based to that, no detailed knowledge on graph theory, and in particular on the models for random graph generation, would be required from NIWA testers to properly implement automatic generation of test cases using the Monte-Carlo strategy. Otherwise, developers would have to provide additional training, or at least some consultancy service, for the NIWA testers – and contradict the

underlying rule of the life-cycle shown in Figure 1, separating the roles of developers and distributors.

Another issue is planning and execution of volume tests, which as indicated in p. 4.3.3, would require generation of graphs with an excessive number of vertices and edges. Despite of the declaration found in the KOALA submission on the maximum allowed size of graphs, such a limitation may be determined experimentally with the random graph generation methods mentioned above – by successively increasing the required number of vertices and edges, until rejection or failure of a generation method is recorded. Volume tests will be needed to access performance (in particular scalability) of all methods for processing graphs as their inputs, when run as NIWA Web services.

Test procedure and scenarios; The test procedure for KOALA should take into account possible interrelationships between its methods. In order to relieve testers of studying semantics of each method when doing that, a simple search in the set indicated in Figure 2 for methods that process graphs, *i.e.* their inputs accept graphs and outputs produce graphs. Next, random generation of sequences (scripts) consisting only of such methods may provide automatic generation of test scenarios in a quite straightforward way. A more elaborate implementation of this method of automatic generating of test scenarios may consider classification of the KOALA library methods with regard to their outputs and inputs, and automatically generate only sequences in which types of outputs and inputs match one another.

Logging results; Despite of the intended deployment of the KOALA library (a downloadable application or a Web service), its developers should be granted access to the log files at the level of detail sufficient to assess the reliability of implemented methods. The preferred mode for generating these files should be the beta-testing facility, run as a provisional Web service for graph processing.

Incidents; The incident report returned to the developers should comply to the IEEE standard, *i.e.* provide customary information on the observed anomalies (hand-ups, crashes, *etc.*) and describe activities performed by testers to counteract them, and above that, also some non-standard stuff with suggestions on how to improve the product. It may also be considered to recruit to the team of NIWA testers a representative of the KOALA development team, to speed up the release of the final product. Such a deviation from the model specified in Figure 1 would be reasonable and cost-effective, given the fact that NIWA testers and KOALA developers are employed by the same organization.

6. Conclusions

The analysis of the IEEE standard for software testing – presented in the paper in the context of a new software model for development of applications intended for on-line delivery – indicates that all cornerstone components of the IEEE standard may be adopted at a relatively low cost. And it does not

matter much, whether the distributor acts on a commercial basis, or serves open communities. The NIWA platform will serve its community in two ways – as a repository of free software published for download by subscribers, and scalable high-performance services installed on its servers and supplied by developers – what makes it unique when compared to commercial distributors and repository hosting service platforms. Quality attributes, used in the paper to formulate the acceptance criteria, namely security, safety, reliability, functionality, performance and usability, are sufficient to assess the product before its acceptance, and the respective test procedures indicated by the standard may be implemented with common black-box testing strategies, such as Monte-Carlo, fairly easy to automate.

References

[1] Drozdowski K, Jarzemski J, Krawczyk H, Melzer M, Smółka M and Wiszniewski B 2005 A Cooperative Model for Implementing Complex Virtual Enterprises, *Foundations of Computing, Decision Sciences* **30** (1) 39

[2] Krawczyk H and Wiszniewski B 2001 Chapter 9: Quality issues of parallel programs, *Parallel Program Development For Cluster Computing – Methodology, Tools and Integrated Environments, Huntington, New York* Cunha J C, Kacsuk P and Winter S Eds, Nova Science Publishers, Inc.

[3] Krawczyk H and Wiszniewski B 1998 *Analysis and Testing of Distributed Software Applications*, Research Studies Press, Wiley

[4] *IEEE Standard for Software and System Test Documentation, IEEE Std. 829-2008 (Revision of IEEE Std 829-1998)*, http://standards.ieee.org/findstds/standard/829-2008.html (Accessed: 2015-01-31)

[5] Garstecki Ł, Kaczmarek P, Chassin de Kergommeaux J, Krawczyk H and Wiszniewski B 2001 Testing for conformance of parallel programming pattern languages, *Lecture Notes in Computer Science* **2328** 323

[6] Apple Inc. 2014 *App Distribution Guide*, https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AppDistributionGuide.pdf (Accessed: 2015-01-31)

[7] Microsoft Corporation 2014 *Windows and Windows Phone Store Policies*, http://msdn.microsoft.com/en-us/library/windows/apps/dn764944 (Accessed: 2015-01-31)

[8] Google Inc. 2015 *Core App Quality, Essentials for a Successful App, Android Developers*, http://developer.android.com/distribute/essentials (Accessed: 2015-01-31)

[9] GitHub Inc. 2015 *GitHub Terms of Service*, https://help.github.com/articles/github-terms-of-service (Accessed: 2015-01-31)

[10] Dice Holdings Inc. 2015 *SourceForge Terms of Use Agreement*, http://slashdotmedia.com/terms-of-use (Accessed: 2015-01-31)

[11] SoftNews NET SRL 2015 *Softpedia terms and conditions of use*, http://www.softpedia.com/user/terms.shtml (Accessed: 2015-01-31)

[12] Portela I M and Cruz-Cunha M M 2010 *Information Communication Technology Law, Protection and Access Rights: Global Approaches and Issues*, Idea Group Inc.

[13] Nokia 2015 *Remote Device Access service (RDA)*, http://developer.nokia.com/resources/remote-device-access (Accessed: 2015-01-31)

[14] Google Inc. 2015 *Google Analytics*, http://www.google.com/intl/pl_ALL/analytics/index.html (Accessed: 2015-01-31)

[15] Giaro K, Ocetkiewicz K, Jastrzębski A, Turowski K, Janczewski R, Obszarski P, Goluch T and Jurkiewicz M 2015 *The KOALA Library*, http://kaims.pl/koala (Accessed: 2015-01-31)
[16] IBM 2015 *Rational Functional Tester 8.6.0*, http://www-01.ibm.com/support/knowledgecenter/SSJMXE/ (Accessed: 2015-01-31)
[17] IBM 2015 *Rational Test Workbench 8.6.0.3*, http://www-03.ibm.com/software/products/en/rtw (Accessed: 2015-01-31)
[18] Hall G 1992 *Applications programming in Smalltalk-80 – How to use model-view-controller (MVC)*, ParcPlace, Palo Alto, CA, USA
[19] Burbeck S 2010 *Pro WPF and Silverlight MVVM – Effective Application Development with model-view-viewmodel*, Apress, Berkely, CA, USA
[20] Sobecki A 2015 *SowiDocs*, https://sowi.pg.gda.pl/index.php/en/ (Accessed: 2015-06-30)