Functional Case Test Design to Optimize the Software Development in Italian Tax Processes (Part I): Methodology Definition

*Fernando Rolli**, *Chiara Parretti*, *Paolo Citti*, and *Massimo Rinaldi*

Department of Innovation and Information Engineering, Guglielmo Marconi University, Via Plinio 44 - 00193 Rome, Italy

Abstract. In Europe's general context of economic integration, the National States have preserved a few competences. Among them, the most important competence is taxation management, which has now become an important lever to stabilize the State's budget and to meet the economic parameters set by the European Agreements. From this perspective, it is crucial to identify a software development mode to reduce the time spent for the implementation/adjustment of the tax payment software application and, at the same time, to minimize the overall risk level. In software development, approximately 40% of time is spent in a series of testing activities: this stage of the development process is mostly placed at the end of the implementation activities. Consequently, since many testing activities have to be waived in order to meet the deadlines for software delivery, applications that are not entirely in compliance with the user's needs and/or entailing non-compliance are more likely to be introduced. This paper focuses on improving the testing process in tax procedures. The proposed method aims to improve the process, by introducing an integrated procedure based on Axiomatic Design. The approach developed will facilitate a reduction both in the testing preparation time and in performing the test cases. In this scenario, it will be possible to optimize the data compilation process, to verify the compliance within the technical specifications as provided by the Italian Revenue Agency, to identify possible critical scenarios with a proactive approach, and to avoid classes of non-conformities.

1 Introduction

The applications used for the management of tax obligations must comply with the regulations introduced by the budgetary laws and their implicit regulatory mechanisms, and have, therefore, an annual feature. In Europe's general context of economic integration, the National States have preserved a few competences, and the management of taxation the most important of them. The Revenue Agencies have become an important lever to stabilize the budget and meet the economic parameters set by the European Agreements. It is crucial, in this scenario, to identify a software development mode that can effectively reduce the time spent for implementations/adjustments to be applied to the tax payment software applications, as well as minimize the overall risk level. In the software development's life-cycle, around 40% of the time is spent for a various number of testing activities [1]. This phase of the development process is mostly placed at the end of the implementation activities, thus implying that, in order to meet the software delivery deadlines, many testing activities are being neglected. This situation can lead to consequences such as the release of applications that do not correspond exactly to the customer attributes and/or with a high level of non-compliance. The use of the

selection system of the functional tests with an AD approach can, instead, reduce the testing time -because many activities can be anticipated- and, at the same time, ensure the coverage of the functional requirements. As of literature, there are indeed several procedures based on AD that have successfully applied AD to optimize processes in several fields [2-4].

2 Building an integrated testing process

The current scientific literature has widely shown that it is impossible to produce a completely error-free software. Nonetheless, this issue should not prevent programmers from seeking new ways of improving the testing process, to increase software's quality level and to optimize both the time and the resources used. The aim of this article is to define an integrated process in the test case preparation for verifications of a functional-type on the applications used to manage the fiscal deadlines. Each test case (CT) consists of a set of required instructions to run a test on a defined object: it is formalized by the input data for the test, by the expected result and by the conditions of test application. This article focuses on optimizing the process of selection and execution of functional tests. In particular, this paper proposes an integrated methodology for the

* Corresponding author: f.rolli@unimarconi.it

selection of test cases, starting from the most abstract logical levels of functional requirements decomposition and proceeding to test the elementary code objects produced. The final objective of this integrated process is to build a complete test case matrix, which will allow to associate a specific test case to each decomposed functional requisite. This mechanism guarantees the coverage of each functional requirement with a specifically selected test case. From this perspective, the choice of test cases is no longer merely tied to the experience of designers –or, even worse, to chance – rather it follows a logically consistent path.

3 Testing process architecture

3.1 Testing process

The testing phase has the task of verifying the adherence to the functional and performance requirements and the correctness of design and coding. Unfortunately, a major part of this phase is placed at the end of the software implementation process. Therefore, it is affected by any problem related to the urgency of its release. Moreover, software companies are often unequipped in terms of cross-reference teams that could help identify the test cases and allow the implementation, which should be performed by highly trained personnel – and which, above all, should not be directly involved in the design and coding phases [5, 6]. First of all, it is part of a good practice to run a thorough estimate of the time and resources devoted to testing. Secondly, considering these estimates, a plan must be defined for the quality of testing. This plan has to include the resources to be used, the time needed, the levels of accuracy required to produce the software, and the functional coverage of the test cases (test case matrix) [7].

3.2 V model testing process

In scientific literature, there are various testing models, each with their own significance in specific contexts. In the present article, we will refer to the so-called V Model, which better adapts to the axiomatic decomposition process of the functional requisites. The figure below illustrates the operating methods of the testing process, according to the V-model [6-8]. On the left, Figure 1 shows the test preparation activities, which are built starting from the customer attributes up to the coding phase of software objects. At each logic level, the test cases deemed necessary to verify the implementation phase are created. The problem that arises at this stage of the process is to verify the adequacy and the functional coverage of the various test cases. The problem is solved with the so-called ‘software static testings’ – in other words, with checks that do not involve the execution of a program or parts of it [7]. The present article focuses in particular on this side of the testing process, proposing an AD approach to test case designing. The second part of the process, instead, concerns the execution of test cases on the software produced. In this case, we speak of dynamic testing. The process starts with tests on the

single software components and continues with integration testings and a system testing, to finally reach testing for the procedure acceptance performed by the client. This process may involve recycling. At each stage, testing results must be recorded and any updated documents related to the project must be saved in special repositories and made available online at any time. A corporate Database can be possible for this purpose: it should be accessed by staff involved in the project.

Fig. 1. V model testing process.

3.3 Non-conformity management

In case of more complex projects, it is useful to keep a record of the detected and corrected malfunctions encountered during testing, as well as any non-conformities occurred after the release of the application. A specific supporting service can analyse and categorize all non-conformities (NC) using a multi-level database with a HNCR approach (Holistic Non-Conformity Reduction) [9-11]. A Database of this type allows recording any malfunctioning through the assignment of unique and structured elementary data. This process triggers a proactive intervention, performing an analytical insight carrying an impact on several NCs at a time. In this way, we avoid an unnecessary and time-consuming activity of addressing them one by one. In this context, value is given to the experience gained over the years, on a more general level, with a ‘lesson-learned’ approach that can be used to solve problems, starting to troubleshoot from their deepest roots. The adoption of the HNCR approach mentioned above, allows to define, starting from every single non-conformity, the clusters of problems occurring at a major level of abstraction. This abstraction allows to solve more non-conformities at the same time, and to implement any additional preventive policies [11-13]. Similarly, appropriate test cases on specific issues can be set. In this way, we can avoid solving problems one by one, as soon as they occur. In addition, the recording and cataloguing of the problems arisen at any stage of the software’s life cycle allows producing valuable statistics that will be useful in future projects.

4 Design of an integrated testing system for fiscal procedures

The AD design methodology can be used to optimize the design process [14, 15] and the execution of functional tests both in case of a custom implementation of a fiscal procedure and in case of an annual adjustment of the tax management software. The final objective of this integrated process is the construction of a complete test case matrix. The matrix must allow the coverage of all functional requisites. In this way, we can avoid duplicating the activities or, considering a worse scenario, skipping the verification of the important functionalities of the software produced. The whole testing process is described in Figure 2.

4.1 Fiscal requirements decomposition

4.1.1 Functional requirements decomposition

The documentary management described in subsection 3.3 allows to predefine the possible areas for intervention quite in advance, thus improving the method application's procedures. In article [11], Rolli et al. proposed a specific AD methodology to detect the robust decomposition of the functional requirements. In the present article, the same decomposition methodology is used to associate each identified functional requirement to its own specific test case. This strategy will allow us to define the minimum test-case set needed to guarantee the coverage of the functional requirements to be tested, and the logical consistency of the configuration that will be obtained. The whole process of decomposition will be now described, as per Figure 2 below.

Fig. 2. Testing process architecture. Source [11].

4.1.2 Definitions and correspondences

As far as the taxation management processes are concerned, the customer attributes (CA) consist in the technical specifications and the compiling instructions provided extensively by the Revenue Agency. Likewise, the design constraints (C) are the restrictions imposed by the regulations, such as tax deadlines and the modalities of electronic transmission of tax statements. (NC) are the non-conformity clusters defined in the process of

cataloguing and reclassification of historical non-conformities detected by the supporting service. They have a proactive value and represent warnings for specific areas of concern, and must be taken into account during the annual preparation of test cases. The system's functional requirements (FR) are the design formalization of the fiscal procedures to be implemented or adjusted, under the conditions imposed by the design constraints and in respect to the indications given by the analysis of non-conformity clusters. Design parameters (DP) are the personal data, the income data and the fiscal data to be processed and compiled. Nonetheless, we consider as process variables (PV) the software components that implement the objects defined at the designing stage.

4.1.3 Correspondence between functional decomposition and software testing levels

The decomposition of the functional requirements follows a zigzagging process in the set of data, producing the logical design of the system at its greatest level of detail [11, 16, 17]. At each level, it is possible to define a particular type of test. For example, at a logic design level, the coverage matrix of functional tests corresponds to the single tests execution plan (Unit Test). At an intermediate level of abstraction, the testing coverage matrix corresponds to the system tests execution plan (Integration Test). Consequently, at the highest level of abstraction we have the definition of the acceptance test plan. This correspondence is made possible by using the correlation matrix between functional requirements and data sets. For each level of abstraction, this matrix indicates the relationship between decomposed functional requirements and the set of reference data (DP_{ij}), in respect to the implemented methods (A_{ij}()). This means that the decomposition of the system's functional requirements (FR) does not only lead to the identification of the data files (DP) –which are the object of a logical treatment–, but also of the test cases to be performed in order to detect any malfunctions.

	DP ₁	DP ₂	DP ₃
FR ₁	A ₁₁ 0		
FR ₂		A ₂₂ 0	
FR ₃			A ₃₃ 0

Fig. 3. Correlation matrix between functional requirements and data sets.

4.1.4 Compatibility between Object-oriented Design and Axiomatic Design

The proposed decomposition approach is particularly effective when combined with object-oriented software implementation techniques. These techniques allow using simplifications and correspondences, which are extremely useful during the setting of test cases. In

particular, we used the Axiomatic design of object-oriented Software Systems (ADO-OSS) methodology to optimize the entire software's life cycle [16, 17]. To illustrate this approach, the following correspondence is introduced. Considering the mapping between Functional domain and Physical domain, the zigzagging process allows to decompose the functional requirements in parallel with the datasets for which such requirements have meaning. The zigzagging process has produced the decomposition of functional requirements, as represented in the figure below:

Fig.4. Decomposition process.

The requirements' decomposition moves to a level of detail deemed sufficient to define all the elements necessary for the design. Then, the ADO-oSS methodology allows to translate this decomposition into UML (Unified Modeling Language) diagrams, which are the essential tool to design and implement a software system [16-19]. This new representation is a consequence of the properties of "legacy", "polymorphism" and "encapsulation" of the object-oriented programming techniques. In this case, the FR_i functional requirement is defined as the object that is implementable by the methods on the i-th row in the decomposition table (A_{ij}) that have access to the DP_{ij} data files, according to the mapping in the corresponding table. At the most detailed level of decomposition, objects it will be represented by elementary methods and data files [11, 16].

Fig. 5. Class diagram. Source [16].

Therefore, the decomposition of functional requirements can be represented by the following simplified UML diagram:

Fig. 6. Simplified UML diagram.

4.2 Selecting the minimum set of test cases

4.2.1 Model selection based on the Function point analysis

The present article proposes an integrated system for the arrangement of functional test cases, using a decomposition approach based on the verification of the axioms of independence and information [11, 16]. The axiom of independence ensures consistency in the coverage of the functional requirements - for which, at least one test case will be performed. The axiom of information allows to select the minor requirement in case of more functional requirement configurations. Function Points [11] are a tool to estimate the information content of the system. Therefore, a robust configuration of the functional requirements is the one that has a lower number of function points, considering the same level of functionalities provided. This is the solution that carries the minor effort, in terms of timing and design costs, but also -and especially- in terms of preparing and executing the tests. In fact, there is an empirical relationship between the capacity of the software to be tested, and the effort needed to test the software expressed in man/days. Capers Jones [20] estimates that the number of test cases can be determined by the function-point estimate for the corresponding effort. The formula is Number of Test Cases = (Function Point Effort)^{1.2}. The actual effort in person/days is then calculated with a conversion factor obtained from previous project data. In this case, cataloging and archiving project data in a structured multi-level database, as described in paragraph 3.3, also allows obtaining data on productivity. This information can be obtained from the overall implementation data from the previous software versions of the tax management program to be adapted, or, in their absence, from similar previous developing experiences. In addition, the use of function point analysis (FPA) allows to define a testing plan from an end-user perspective, simplifying many steps and taking advantage of many correspondences, as we will discuss in the following paragraphs.

4.2.2 Effort estimation

The relationship between decomposed functional requirements and data files can be represented in equivalent form with UML diagrams. UML diagrams allow an estimate of the software development project's effort, by using the Function Point measurement technique [21-24]. In this case, the data structures (DP_{ij}) represent data files (ILF and EIF); while the methods (A_{ij}) represent the transactional functions (EI, EO, EQ) [11, 22, 23]. By counting the function points of the project, it is possible to estimate the cost in terms of man/days. Indeed, techniques such as COCOMO II allow to determine an estimate of the time required to complete the project [25, 26]. Then, the knowledge of a particular type of project allows to categorize the estimated man/resources by their professional profile, and match them to the various phases of the software's life cycle. In testing, we often refer to the golden rule of Frederick P. Brooks [1], according to which 40% of the estimated time for a software project should be used for the testing stages. This rule is still valid. However, in the case of projects to be developed in familiar contexts, experience helps to obtain a lower percentage of commitment. In this case, the background series of similar projects can provide useful information.

4.2.3 Alternative effort-estimation methodologies

Several authors criticize the use of FPA to calculate the effort of the various testing activities, on the grounds that, it is difficult to make an estimate in the absence of detailed functional requirements, [27]. Therefore, various alternative effort-estimation methodologies of the test cases were set, such as the Use Case Point (UCP) and the Test Case Point (TCP). These are measuring techniques borrowed from the FPA, of which they keep some simplified elements. However, in this article we must stress the importance of the FPA, since it is generally considered as an accepted standard for software estimations. Other methodologies are not standard, yet their diffusion is very limited. Instead, the context of study of this article concerns the software systems for fiscal management, for which the technical and regulatory specifics are well known. In addition, the transition from any present software version to one related to the following year can be easily managed by using the FPA technique.

4.3 Test Case Attribution by level of functional decomposition

The proposed methodology consists in associating, to each functional decomposition level, the minimum set of test cases in order to verify each decomposed functional requirement. Thus, the logical design of the system not only does follow its corresponding functional test plan, but the same selection of the tests can affect the implementation phase, directing designers towards easier testable solutions - in other words, with a lower risk level. In fact, this organic approach to the selection of

test cases allows to match the functional architecture of the system with a network of test cases already defined in the system designing phase. In this way, we can avoid neglecting parts of the system, especially during the integration process of elementary modules and sub-systems. For the small and medium enterprises and organizations, this method constitutes a major advantage: in fact, it allows to define the complete organization of functional tests to be performed already during the software designing phase. In the present article, for simplicity's sake, we set three decomposition levels of the functional requirements. In this case, each level of decomposition corresponds to a specific type of test. The highest abstract logic level corresponds to the system test: this type of test has, as its objective, the verification of the application in its entirety, in an environment that is similar to production. The second level concerns, instead, an integration test among self-contained program modules. At this stage, the interface among the various parts of the system is verified. Finally, the third level concerns the verification of self-consistent elementary objects. As shown in Figure 8, the test configuration is performed starting from the most abstract logic level to the most basic level, following the decomposition of the functional requirements with an axiomatic approach. Alternately, the testing execution starts with a verification of the individual elementary components. Then, the execution of the integration between the various elementary modules and/or sub-systems is verified. This testing phase can be anticipated: it is not necessary for all the sub-systems to be available, some can be simulated by specially developed software applications [5-7]. This allows to split the development activities among different teams that can work in parallel. Finally, a system test execution is performed, and it is carried out in an environment similar to production. Instead, the acceptance test is carried out by the software client, and it is performed in order to verify the compliance of the product developed by the supplier in terms of technical specifics. Basically, the acceptance test is identical to the system test.

4.4 Building the Test Case Coverage Matrix

The decomposition of the functional requirements with an AD approach allows building a correlation matrix with some important features. It represents the robust decomposition of the functional requirements [11, 24]. For example, for each A_{ij} element of the matrix of Figure 4, we need to build a specific test case (CT_{ij}). A set of d data input is defined, where $d \in DP_{ij}$ so that A_{ij} (d) is correct when it meets the specifications of Fr_{ij}, or it is incorrect if it is different from the result expected from the functional requirement [28, 29]. For example, the A_{11,11} () method is verified with the execution of the CT_{11,11} functional case test. Therefore, CT_{11,11} includes the input data to be used for the test and for the application rules provided by the program. For this reason, the A_{11,11} (d) result obtained by performing A_{11,11} () on the d data input with $d \in DP_{11}$, is correct if it meets the specifications of FR₁₁. It is incorrect if it is different

from the expected result from the functional requirement. The correctness of the $A_{11,11}$ () method in respect to a d data input is indicated with “ok” ($A_{11,11}$, d). The $A_{11,11}$ () function is correct, i.e., $ok(A_{11,11}) \Leftrightarrow \forall d \in DP_{11}, ok(A_{11,11}, d)$ [28]. Therefore, the test case $CT_{11,11}$ may be considered ideal, if the absence of malfunctions detected implies the absence of a total malfunction, i.e. $ok(A_{11,11}, CT_{11,11}) \Rightarrow ok(A_{11,11})$. In other words, this occurs if further testings do not detect any malfunctions that have not already been discovered by $CT_{11,11}$. The aforementioned process determines the following test case coverage matrix:

		CT ₁₁	CT ₁₂	CT ₁₃	CT ₂₁	CT ₃₁
FR ₁	FR ₁₁	CT _{11,11}	CT _{11,12}	CT _{11,13}		
	FR ₁₂		CT _{12,12}			
	FR ₁₃			CT _{13,13}		
FR ₂	FR ₂₁				CT _{21,21}	
FR ₃	FR ₃₁					CT _{31,31}

Fig. 7. Test case coverage matrix

This matrix ensures the complete coverage of the functional requirements. This is the minimum configuration that allows to verify that the $A_{ij}()$ methods comply with the functional specifications required by the Revenue Agency instructions.

5 Conclusions

The methodology proposed in this article helps setting functional tests already at a software designing stage, reducing the overall time required for software development. In addition, the axiomatic decomposition of functional requirements allows to identify the suite of test cases that have a full functional coverage. At least one test case is set for every single software object. It is acknowledged that it is impossible to build tests that can demonstrate the absence of malfunctions in a software program, but we can arrange a detect & debug dynamic mechanism for software errors that results in a ‘lesson learned’ approach for any following version of the program. Such mechanism, based on a methodical cataloguing of the non-conformities detected both during the development phase and during the application’s execution phase, allows to define a specific test case for each class of non-conformity. Moreover, the reuse of test cases related to any previous software versions allows to focus on the effort of the system components affected by the annual regulatory changes that are intrinsically typical of the taxation management systems.

Fig. 8. Test Case Design Process.

This paper is dedicated to the memory of our wonderful colleague and co-author of this work, Dr. Massimo Rinaldi, who recently passed away.

References

1. F. P. Brooks Jr, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, 1995);
2. G. Arcidiacono, K. Yang, J. Trewin, L. Bucciarelli, *Application of Axiomatic Design for Project-Based Learning Methodology*, Proc. CIRP, **53**, 166-172 (2016);
3. G. Arcidiacono, A. Giorgetti, M. Pugliese, *Axiomatic Design to improve PRM airport assistance*, Proc. CIRP, **34**, 106-111 (2015);
4. G. Arcidiacono, D. T. Matt, E. Rauch, *Axiomatic Design of a Framework for the Comprehensive Optimization of Patient Flows in Hospitals*, JHE, vol. **2017** (2017);
5. I. Sommerville, *Software Engineering* (Addison-Wesley, 2009);
6. G. J. Myers, *The Art of Software Testing*, (John Wiley & Sons, 2004);
7. A. Avellone, M. Cislighi, E. Colanese, *Collaudo e qualità del Software*, (AICQ-CI, Uni Service, 2010);
8. D. G. Firesmith, *Common System and Software Testing Pitfalls: How to Prevent and Mitigate Them: Descriptions, Symptoms, Consequences, Causes, and Recommendations*, (SEI Series in Software Engineering, 2013);
9. C. Cavallini, A. Giorgetti, P. Citti, A. Meneghin, *Sviluppo di un approccio olistico per l'analisi e la risoluzione delle non conformità*, AISS, **3**, 1 (2012);
10. F. Rolli, A. Giorgetti, P. Citti, *Integration of Holistic Non-Conformities Management and Axiomatic Design: a case study in Italian Income Tax Returns Management*, Proc. CIRP **34**, 256-262 (2015);
11. F. Rolli, A. Giorgetti, P. Citti, M. Rinaldi, *Improvement of the compilation process of the Italian income certifications: a methodology based on the evaluation of the information content (Part I)*, Proc. CIRP **53**, 56-62 (2016);
12. A. Giorgetti, C. Cavallini, A. Ciappi, G. Arcidiacono and P. Citti, *A holistic model for the proactive reduction of non-conformities within new industrial technologies*, IJMERR, **6**, 4 (2017);
13. A. Giorgetti, G. Arcidiacono, A. Ciappi, R. Barbieri, and P. Citti, *HNCR model following robust approach*, QREI (forthcoming);
14. G. Arcidiacono, P. Placidoli, *Reality and illusion in Virtual Studios: Axiomatic Design applied to television recording*, Proc. CIRP, **34**, 137-142 (2015);
15. G. Arcidiacono, A. Giorgetti, A. Ciappi, *An Axiomatic Design framework for reliability improvement*, ACM International Conference Proceeding Series, pp. 214-217;
16. NP Suh, *Axiomatic Design - Advances and Applications*, (Oxford University Press, 2001);
17. SH Do, NP Suh, *Object-oriented software design with axiomatic design*, Proc. CIRP **49**, 278-84, (2000);
18. C Togay, AH Dogru, JU Tanik, *Systematic component-oriented development with axiomatic design*, JSS, **81**, 115 (2008);
19. P.J. Clapis PJ, J.D. Hinterstienner, *Enhancing object-oriented software development through axiomatic design*, Proc. ICAD **1**, 271-77 (2000);
20. C. Jones, *Applied software measurement*, (McGraw-Hill, 2008);
21. A.J. Albrecht, *Measuring Application Development Productivity*, Proc. SHARE/GUIDE IBM Appl. Development Symposium 83-92 (1979);
22. IFPUG, *Function Point Counting Practices Manual*, (IFPUG, 1999);
23. R. Meli, L. Santillo, *"Function Point Estimation Methods: a Comparative Overview"*, Proc. FESMA'99, 271-286 (1999);
24. F. Rolli, A. Giorgetti, P. Citti, M. Rinaldi, *Information content evaluation to obtain robustness of the management in Italian fiscal process (Part 2): optimization of Italian income certification process of the year 2016*, Proc. CIRP **53**, 63-69 (2016);
25. A.L. Rollo, *Functional Size measurement and COCOMO – A synergistic Approach*, Proc. SMEF 259-67 (2006);
26. L. De Rore, M. Snoeck, G. Poels, and G. Dedene, *Cocomo II as productivity measurement: a case study at KBC*, (SSRN Electronic Journal, 2008);
27. S. Nageswaran, *Test Effort Estimation Using Use Case Points*, QW2001 (2001);
28. J.B. Goodenough, S.L. Gerhart, *Toward a theory of test data selection*, SIGPLAN Notices, **6**, 10 (1975);
29. L. Bougè, *A contribution to the theory of program testing*, TCS **37**, 151-181, (1985).