An Analysis of Integration related problems in Object Oriented System

Dr. Amit Kumar Chaturvedi
Assistant Prof., MCA Deptt.
Govt. Engineering College, Ajmer, India

Rakesh Chandra Verma
Ph.D. Scholar, Department of Computer Science
Bhagwant University, Ajmer, India

Abstract- Integration of object-oriented system is a complex task. One of the major strengths of object-oriented programming is the possibility of building independently manageable blocks (i.e., classes), which can be combined to obtain the whole system. A well designed object-oriented application is composed of simple building blocks assembled together. In this paper, we have presented various integration strategies like Top-down and Bottom up , Big-bang, and Threads Integration. During analysis work, we encountered with many Integration errors and faults like Interpretation Errors, miscoded call error, Interface error, and global errors. In this paper, we have presented strategy for integrating classes.

Keywords – Integration Strategy, Integration Errors, OO Systems, Polymorphism, faults.

I. Introduction

It is well known that software testing is a time-consuming, error-prone and costly process. Therefore, techniques that support the automation of software testing will result in significant cost and time savings for the software industry. Automatic generation of the test cases is essential for the automation of software testing. Once the test cases are generated automatically, a software product can even be tested fully automatically through a test execution module, to realize an integrated automated test environment. Automatic approach for test case generation will be not so much productive, if we wait for the generation of test cases at the end of the development. That means it is simply a waste of time, if we follow the source code based testing. So, automatic test case generation using design document (or system specification or model) is more reasonable [1-6].
Integration of object-oriented system is a complex task. One of the major strengths of object-oriented programming is the possibility of building independently manageable blocks (i.e., classes), which can be combined to obtain the whole system. A well designed object-oriented application is composed of simple building blocks assembled together. Therefore, in object-oriented systems the complexity tends to move from modules to interfaces between modules, i.e., from units to interactions among them. If for traditional imperative language 40% of software errors can be traced to integration problems [8], we could expect percentage to be much higher in object-oriented software.

II. INTEGRATION STRATEGIES

The main traditional integration testing strategies can be classified as *top-down integration*, *bottom-up integration*, *big-bang integration*, *threads integration*, and *critical modules integration*. Some of these strategies can be suitably tailored for object-oriented systems, while some other might not be applicable in this context. In the following we reconsider these strategies in an object-oriented environment.

a. *Top-down and Bottom-up Strategies*

Top-down and *bottom-up* integration strategies are still applicable to object oriented systems, provided that we correctly identify dependencies among classes. Unlike traditional procedural languages, where the identification of the *use* relation among modules is straightforward, object-oriented systems are characterized by several different relations, which can hold between classes. These relations have to be taken into account, and more subtle interactions between units have to be considered, with respect to the traditional case. A class can depend on another class even if it does not "*use*" it in the traditional sense of the term. As an example, let us consider a class Queue together with its elements of type QueueElement, which represent a typical case of aggregation relation. In this case, objects of class Queue contains objects of class QueueElement, but they might never invoke any of their methods (i.e., might never "*use*" them). Nevertheless, the presence of class QueueElement is necessary for class queue to be instantiated (to be tested). The same holds for hierarchical relations. A subclass cannot be tested in the absence of its ancestors. The main problem when applying this strategies to object-oriented systems is related to the presence of cyclic dependencies among classes. In the presence of cycles, it is impossible to find an integration order for bottomup (resp., top-down) strategies which allows for avoiding the construction of stubs (resp., drivers).

In its incremental integration strategy, Overbeck [9] provides a detailed analysis of integration testing and a bottom-up integration strategy addressing the problem of cyclic dependencies. The approach is based on test patterns, defied according to relationships between client and server classes and by taking into account inheritance relationships. Two basic level of testing are identified: unit test (performed by means of *self-test* suites) and pairwise interclass test (performed by means of *contract-test* suites). Overbeck addresses the problem of cyclic dependencies within a system by means of stubs which are used to break cycles. To minimize the cost of scaffolding, attention is paid to accurately select which classes have to be replaced by stubs.

Kung [23] proposes a technique for defining the integration order when testing an object-oriented system. The technique is based on a program representation called the *Object Relation Diagram* (*ORD*). This diagram contains information about the classes composing the system and the relations between them. In order to address the problem of cycles, Kung proposes a technique based on the breaking of cycles. The author asserts that every cycle must contain at least one association, whose elimination allows for breaking that cycle. When the testing has been completed, the association may be reintroduced and tested. Even though the proposed solution is interesting, the author does not provide any evidence supporting his assertion about the inevitable presence of associations within cycles. Moreover, no explanation is provided about how association elimination can be accomplished.

b. Big-bang Strategies

Big-bang integration strategies can be straightforwardly applied by just pulling all the objects composing a system together and exercising the whole resulting system. We consider this approach to be applicable only to small systems, composed of a few classes. In accordance to Siegel's experience [17], we expect such an approach to be ruinous in the general case. Due to the complexity of the interactions among objects, integration should aim at minimizing the number of interfaces exercised at once. Localization of faults by means of debugging can become prohibitive, in the case of a whole system whose components' interactions have never been exercised on smaller subsets.

c. Threads Integration

Threads integration strategies can be adapted to object-oriented systems as follows. As long as we consider object-oriented systems as sets of cooperating entities exchanging messages, threads can be naturally identified with sequences of subsequent message invocations. Therefore, a thread can be seen as a scenario of normal usage of an object-oriented system. Testing a thread implies testing interactions between classes according to a specific sequence of method invocations. This kind of technique has been applied by several authors.

Kirani and Tsai propose a technique for generating test cases from functional specification for module and integration testing of object-oriented systems. The method aims at generating test cases that exercise specific combinations of method invocations and provides information on how to choose classes to be integrated.

A similar approach is proposed by Jorgensen and Erickson [28], which introduces the notion of *method-message path* (*MM-path*), defined as a sequence of method executions linked by messages. For each identified MM-path, integration is performed by pulling together classes involved in the path and exercising the corresponding message sequence. More precisely, Jorgensen and Erickson identify two different levels for integration testing:

Message quiescence : This level involves testing a method together with all methods it invokes, either directly or transitively.

Event quiescence : This level is analogous to the message quiescence level, with the difference that it is driven by system level events. Testing at this level means exercising message chains (threads), such that the invocation of the _rst message of the chain is generated at the system interface level (i.e., the user interface) and, analogously, the invocation of the last message results in event which can be observed at the system interface level. An end-to-end thread is called an *atomic system function* (*ASF*).

The main drawback of this method is the difficulty in the identification of *ASF*s, which requires either the understanding of the whole system or an analysis of the source code.

III. Integration Errors

When integrating object-oriented systems, we may expect to have both problems similar to the ones typical of the traditional procedural case and new failures specifically related to the object-oriented paradigm. In addition, some of the traditional errors may become less frequent or less dangerous in the case of statically typed object-oriented languages.
Following is the classification of the Integration Errors:

1. Traditional faults that can occur with the same probability and can be addressed in the same way they are addressed in traditional systems: as far as these faults are concerned, there should be no need for defining new techniques, and it should be possible to apply traditional approaches straightforwardly.
2. Traditional faults less frequent in an object-oriented context: it could be the case that this class of faults occur too seldom to justify the testing effort required to address them.
3. Traditional faults more frequent in an object-oriented context: the increasing of their probability of occurrence may justify additional testing effort aiming at revealing them. Roughly speaking, testing devoted to identify them might become cost-effective in an object-oriented context.

a. Interpretation Errors

Interpretation errors are due to the common problem of a caller misunderstanding the behavior of a callee. The misunderstanding is about either the functionality the module provides or how it provides it. An interpretation error occurs every time the interpreted specification differs from the actual behavior of the module. Interpretation errors can be further classified as follows.
Wrong function errors: The functionalities provided by the callee may not be those required by the caller, according to its specification. In this case, the developer's error is to assume that the callee actually provides the functionalities needed by the caller. The error is neither in the misunderstanding of the callee's behavior nor in a wrong use of its interface, but rather in a faulty implementation of the caller. There is no reason to believe that in object-oriented systems the probability of occurrence of such errors could differ with respect to the traditional case. The wrong interpretation of a specification can occur for a class exactly as for a module. We thus place such errors in the first category of the framework.

*Extra function errors***:** There are some functionalities provided by the callee which are not required by the caller. The developer is aware of the existence of these extra functionalities, but he wrongly assumes that no inputs to the caller will ever result in an invocation of them on the callee. In this case, the developer's error is to overlook some particular input to the caller causing it to invoke such functionalities, with unexpected results. This kind of errors can easily occur as a consequence of either modification to the caller or reuse in a different context of the pair caller-callee. This type of errors has to be considered as specific to the integration. They are not identifiable by testing single modules, since the problem involves verifying that no input to the caller will cause an input to the callee that in turn results in an invocation of these extra functionalities.

In the case of object-oriented systems, this kind of errors could occur in two different contexts, and different cases worth a separate analysis.

- The problem could occur in relation to interactions between different methods belonging to the same class. According to the hypothesis of adequately tested unit, this should never be the case while performing integration testing. During unit (i.e., intra-class) testing the developer/tester should never put any constraint on the allowed inputs to methods, since a class is a cohesive unit and its method should cooperate as specified no matter how they are invoked. Errors of this kind would imply that inadequate unit testing has been performed. We can safely consider this kind of problem as a competence of unit testing.
- The problem could be related to interactions among two methods belonging to different classes. As long as we consider classes as modules together with their operations, this problem is analogous to its traditional counterpart. Nevertheless, we consider these errors much more likely to occur in object-oriented systems

than in traditional code. Object-orientation enforces reuse. It is very common that a class designed and developed to be used in a specific system is reused in many different contexts. Moreover, if the caller is a subclass and one of its ancestors happens to be modified, this can result in a different behavior of the caller itself causing the invocation of the extra function. These kind of errors might deserve an additional testing effort in object-oriented systems, and thus we place them in the third category of our framework.

Missing function errors: There are some specific inputs from the caller to the callee that are outside the domain of the callee. Such inputs usually result in unexpected behaviors of the callee. Unit testing has no means of revealing these kinds of problems, whose identification is thus left to integration testing.
In this case, the presence of polymorphism and dynamic binding complicate the picture. The programmer might overlook that a valid input for a method in a superclass is outside the domain of some redefinition of the method in a subclass. Moreover, since type hierarchies can grow, the problem could arise even when the developer has verified a method and all of its redefinitions. We consider this type of errors to be more frequent in the object-oriented case, and thus we place it in the third category of the framework.

Miscoded Call Errors
Miscoded call errors occur when the programmer places an instruction containing an invocation in a wrong position in the code of the caller. These errors can lead to three possible faults:

- *Extra call instruction:* The instruction performing the invocation is placed on a path that should not contain such invocation.
- *Wrong call instruction placement:* The invocation is placed on the right path, but in a wrong position.
- *Missing instruction*: The invocation is missing on the path that should contain it.

In object-oriented systems, this kind of errors might occur as frequently as they do in the traditional case. Nevertheless, they are well localized errors that should be revealed by class testing and should be given little attention during integration. We can classify them as belonging to the second category of the framework.

IV. INTERFACE ERRORS

Interface errors occur when the defined interface between two modules is violated. It is possible to identify several kinds of interface errors. For example: parameters of the wrong type, parameters not in the correct order, parameter in the wrong format, violation of the parameter rules (e.g., call by value instead of call by reference), mismatching between the domains of actual and formal parameters. Beizer [9] identifies interface errors as the most frequent errors during integration testing. In the general case, most (if not all) interface errors cannot be identified during unit testing. They have to be addressed during the integration phase When using a statically typed object-oriented language, most of these errors can be caught statically, during compilation. We can safely consider this kind of errors much less frequent in the object-oriented case, and thus place them in the second category with respect to the framework.

Global errors

A *global* error is an error related to the wrong use of global variables. This kind of errors have necessarily to be addressed during integration testing.

In the object-oriented languages we are considering, there is no way of defining global variables. Nevertheless, we have to consider the case of publicly accessible static attributes. Being globally accessible and possibly modifiable, they are analogous to global variables of traditional languages. This kind of use of static attributes is an example of bad programming practice. We consider such situation as very unlikely to occur in well-designed object-oriented systems, and thus consider this class of errors as less frequent in the object oriented case than in the traditional case.

Table 4.1 summarizes the classification, with respect to our framework, of traditional errors identified by Leung and White. Three types of errors result to be less likely to occur in an object-oriented system, namely, miscoded call, interface, and global errors, while wrong function errors resulted to be as frequent in object-oriented systems as they are in traditional programs. Even though this information can provide several hints for interclass testing, we are mostly interested in errors which appear to be more frequent in object-oriented systems than in traditional programs. Such errors are more likely to require specific techniques to be adequately addressed.

	1	2	3
Wrong function error	X		
Extra function error			X
Missing function error			X
Miscoded call error		X	
Interface error		X	
Global error		X	

Table 4.1 : Classification of Integration Errors

These types of errors can be related in the third category of the framework with two specific object-oriented characteristics: Inheritance and polymorphism.

- *Inheritance* can cause problems due to the consequences of modifications in ancestors on the behavior of heirs.
- *Polymorphism* can introduce specific integration testing errors related to the impossibility of statically identifying the actual receiver of a message.

V. Proposed Strategy

Our strategy for integrating classes composing a system. In our work, we assume that:

- we are provided with the whole code of the system we are testing,
- the system is written in a language compliant with the Java-like language,
- Adequate unit testing has already been performed, and thus classes composing the system have already been exercised in isolation.

Our main goal is to choose an integration order allowing for incrementally adding classes one at a time and testing the interactions between such class and the already tested subsystem. Besides considering association, aggregation, and composition relations, we take into account hierarchical relationships. A superclass must always be tested before its subclasses for both implementation and efficiency reasons: firstly, when testing a subclass, it is in general quite complex to build a dependable mockup to replace its parent class with; secondly, testing the parent class before its heirs allows for minimizing the effort by partially reusing test documentation and test suites.

Our technique can be classified as a bottom-up integration testing strategy (it could be applied for top down integration as well). To address the problem of defining a suitable incremental integration order, we consider dependencies between classes induced by all the different relationships which can hold between the elements composing a system. The approach, partially inspired by the work on modular formal specification presented by San Pietro, Morzenti, and Morasca [71], is based on the representation of the system under test in terms of a directed graph, whose nodes are the classes composing the system. The edges of the graph represents the relations between classes within the system.

VI.CONCLUSION

The main contribution of the work presented in this thesis, can be summarized as:

- An analysis of the main problems related to object-oriented testing. This allows for identifying several issues which are still to be addressed. In particular, it allows for identifying relationships among classes, polymorphism and dynamic binding as a major problem with respect to object oriented integration testing.
- A study on how object-oriented languages influence traditional integration testing strategies, and the proposal for an integration strategy specific to the object-oriented case. The proposed strategy is based on the analysis of a graph representing the system under test. It takes into account new dependencies, introduced by relations among classes, which are more subtle than the ones occurring between traditional modules and need to be specifically addressed.
- The identification of a new class of failures occurring during integration of object-oriented systems in the presence of inclusion polymorphism.
- The definition of a technique for addressing them. The technique, based on data-flow analysis, allows for identifying critical paths and bindings to be exercised during integration. Besides allowing for defining test selection criteria, the technique can be used to define test adequacy criteria for testing of polymorphic calls between classes.

The design of a tool which allows for applying the technique in a semi-automated way.

REFERENCES

[1] Dustin, Elfriede (2002). Effective Software Testing. Addison Wesley. p. 3. ISBN 0-201-79429-2.
[2] Marchenko, Artem (November 16, 2007). "XP Practice: Continuous Integration". Retrieved 2009-11-16.
[3] Gurses, Levent (February 19, 2007). "Agile 101: What is Continuous Integration?". Retrieved 2009-11-16.
[4] Pan, Jiantao (Spring 1999). "Software Testing (18-849b Dependable Embedded Systems)". Topics in Dependable Embedded Systems. Electrical and Computer Engineering Department, Carnegie Mellon University.
[5] Rodríguez, Ismael; Llana, Luis; Rabanal, Pablo (2014). "A General Testability Theory: Classes, properties, complexity, and testing reductions". IEEE Transactions on Software Engineering 40 (9): 862–894. doi:10.1109/TSE.2014.2331690. ISSN 0098-5589.
[6] Rodríguez, Ismael (2009). "A General Testability Theory". CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1–4, 2009. Proceedings. pp. 572–586. doi:10.1007/978-3-642-04081-8_38. ISBN 978-3-642-04080-1.
[7] IEEE (1998). IEEE standard for software test documentation. New York: IEEE. ISBN 0-7381-1443-X.
[8] Kaner, Cem (2001). "NSF grant proposal to "lay a foundation for significant improvements in the quality of academic and commercial courses in software testing"" (PDF).
[9] J. McGregor and T. Korson. Integrated object-oriented testing and development processes. Communications of the ACM, 37(9):59–77, September 1994.
[10] J. McGregor and T. Korson. Testing of the polymorphic interactions of classes. Technical Report TR-94-103, Clemson University, 1994.
[11] B. Meyer. Object-oriented Software Construction. Prentice Hall, New York, N.Y., second edition, 1997.
[12] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. Lecture Notes in Computer Science, 1445:355–376, 1998.
[13] L. J. Morell. A Theory of Error-Based Testing. PhD thesis, University of Maryland, April 1984.
[14] L. J. Morell. A theory of fault-based testing. IEEE Transactions on Software Engineering, 16(8):844–857, August 1990.
[15] G. J. Myers. The Art of Software Testing. Wiley - Interscience, New York, 1979.
[16] A. J. Offutt. Automatic Test Data Generation. PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, 1988.
[17] J. Overbeck. Testing Object Oriented software - State of the art and research directions. In 1st European International Conference on Software Testing, Analysis and Review, London, UK, October 1993.
[18] H. D. Pande and B. G. Ryder. Static type determination for c++. Technical Report LCSR-TR-197-A, Rutgers University, Lab. of Computer Science Research, October 1995.
[19] A. Paradkar. Inter-Class Testing of O-O Software in the Presence of Polymorphism. In Proceedings of CASCON96, Toronto, Canada, November 1996.
[20] D. Perry and G. Kaiser. Adequate testing and object-oriented programming. Journal of Object-Oriented Programming, 2(5):13-19, January/February 1990.
[21] P. S. Pietro, A. Morzenti, and S. Morasca. Generation of Execution Sequences for Modular Time Critical Systems. to appear in IEEE Transactions on Software Engineering, 1999.
[22] J. Plevyak and A. A. Chien. Precise concrete type inference for object oriented languages. In Proceedings of the 9 ACM SIGPLAN Annual Conference on Object-Orinted Programming, Systems, Languages, and Applications (OOPSLA'94), pages 324 340, 1994. ACM SIGPLAN Notices 29, 10.
[23] S. Rapps and E. J.Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, SE-11(4):367-375, Apr. 1985.
[24] G. Rothermel and M. J. Harrold. Selecting regression tests for object oriented software. In International Conference on Software Maintenance, pages 14-25, September 1994.
[25] J. Rumbaugh, M. Blaha,W. Premerlani, F. Eddy, and W. Lorensen. Object- Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, 1991.
[26] B. Shriver and P. Wegner, editors. Research Directions in Object-Oriented Programming. The MIT Press, Cambridge, Mass., 1987.
[27] S. M. Siegel. Strategies for testing object-oriented software. Compuserve CASE Forum Library, Sept. 1992.
[28] M. Smith and D. Robson. A framework for testing object-oriented programs. Journal of Object-Oriented Programming, 5(3):45-53, June 1992.
[29] M. D. Smith and D. J. Robson. Object-oriented programming: the problems of validation. IEEE, November 1990.
[30] A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard, July 1995.