

Managing Software Defects in an Object-Oriented Environment

Houman Younessi

Rensselaer Polytechnic Institute-Hartford Graduate Campus

Managing defects when developing object-oriented systems has its own challenges. In this article, the impact of adopting the object-oriented paradigm on how we manage defects at various phases of the software process is discussed. Specific issues relating to both the structural implications and the environmental and process considerations are named and discussed, with solutions provided.

The benefits of the object-oriented (OO) paradigm are well publicized. Less so are the areas where this paradigm actually creates challenges and difficulties. One such area is defect management. This article looks at why object-orientation might present such challenges, and what it is like to manage defects in an OO environment.

In general, OO systems score lower in terms of testability compared to procedural systems [1]. The reasons for such low testability can be traced to the structural composition of OO systems discussed in the following sections. Each section begins a brief definition of the issue under discussion, a short synopsis or some explanation closely relating to the issue at hand, and a description of the defect management problems that are relevant. Each problem is numbered along with its corresponding solution(s), for example, (1) Abstraction Reduces Observability. Keep in mind there may be more than one corresponding solution to each preceding problem.

Abstraction

Abstraction is that essential property that allows the selection of a logical and coherent conceptual boundary so that the object is identifiable by its essential characteristics; these are those characteristics that define what it is and what it does without heed to how such is accomplished. While abstraction is of potentially great benefit to the modeler, the designer, the user, the maintainer, and the reuser, it is often of hindrance to the defect manager.

Problems

(1) Abstraction Reduces Observability [1]. Observability, or internal state visibility, is the ability to examine the internal state of an object at any one time. Given that abstraction, in essence, masks access to a lot of this information, it dramatically reduces observability.
(2) Partial/Distributed Implementation. An abstract class is one in which

the implementation of at least one feature is deferred to another class. This creates a problem in testing because all the features are not there.

Solutions

(1) Inspector Routines. These are public routines written to examine the value of each relevant attribute that otherwise will not be accessible. This may be helpful but unfortunately it is cumbersome to

“While abstraction is of potentially great benefit to the modeler, the designer, the user, the maintainer, and the reuser, it is often of hindrance to the defect manager.”

create such classes. Even if these methods are present, they may be defective, making the job of testing that much more difficult.

(1) Memento Design Pattern. A variation of the approach above is to design each class as part of a Memento Design Pattern [2].

(1) Encapsulation Breaking Mechanisms. Friend functions in C++ belong to this category. These can be defined to cut across the encapsulation wall of an object. They can then access the internal state of an object, yet have inherent side effects that make their use inadvisable in an OO system.

(2) Inspection. Inspection has proven effective in defect management of partial or distributed implementations. Inspection techniques specialized for object technologies have been developed [3].

(2) Leaf Class Testing. This is a testing technique that evaluates the abstraction structure using its concrete implementations [1, 3].

Encapsulation

In encapsulation, an object is defined as a collection of interrelated concerns wrapped into a logically cohesive unit. In OO, applying encapsulation is not restricted to the composition of classes and objects but also applies at higher levels, for example to form packages and sub-systems.

Problems

(1) Scope Escalation. The routine no longer can be considered the logical unit for testing. That honor now must go to the class. This does not in any way mean that the routines of a class are not tested, but that unit testing in OO must be done in the context of a class.

(2) Hierarchy Integration. How can you test whether the higher encapsulation levels are communicating with each other in the expected fashion?

Solutions

(1) Inspection. Inspection techniques specialized to OO could focus on the algorithmic, initialization, and temporal characteristics of individual routines in a way that is not possible when testing.

(1) Context Testing. This is the idea of giving up individual testing of the individual routines and instead testing them in the context of their operation and collaborations. This is of course risky, as the routines will only be tested in known contexts.

(2) Inspection. Inspection allows evaluation of interaction of packages at integration level.

(2) Multi-Table Class-Responsibility-Collaboration (CRC). This is a simulation technique that allows the evaluation of the design of a system from the perspective of package integration and reduction of coupling [3]. The technique is applied during design evaluation.

Genericity

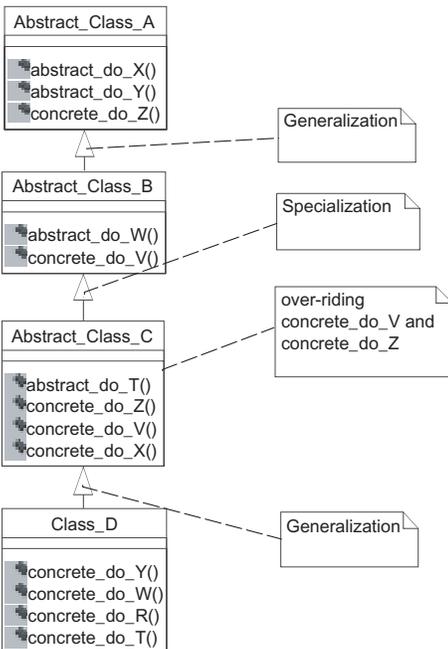
Genericity is the principle of type independence. To facilitate reuse, it is most useful to write components (e.g., classes) that work with a variety of types under a variety of situations. While not all programming languages provide for genericity at the moment, it is likely that its implementation and use would increase in the future. Eiffel [4] is one of the few OO languages that implement this concept effectively.

The rules are quite straightforward. We simply declare the type to be of some generic identifier and then use that same identifier as a placeholder or name whenever a given type is referred to. As such, generic classes are not classes in the strictest sense; they are templates for classes that hold at least one unspecified type. Only when all the unspecified types are *pinned down* does a class emerge. This means that you can write a generic class but you cannot create an object of that generic class. For that to happen, you have to first specify all the generic type placeholders using valid extant types, thus creating a true class. Only then can you create (instantiate) an object. Using genericity, it will then become possible to provide a wide range of libraries of very useful reusable classes such as container (object structure) and graphical user interface classes.

Problem

(1) Type/Behavior Variability. The varying behavior of an object based on the type or combination of types with respect to which it has been instantiated

Figure 1: Testing Class Hierarchies



creates a test case explosion.

Solution

There is no established calculus here, and problems may emerge from many unexpected corners. No good sure-fire solution exists here. Care, a good deal of anticipation of potential problem areas, and lots of testing with a wide range of potential types is best. Some guidelines (not really solutions) to this end appear in [4].

Inheritance

Inheritance is a kind of relationship between classes. It is one of the central features of object orientation. While it is not necessary to have inheritance in order to have an OO system, most such systems do incorporate inheritance. Inheritance can bring a lot of advantages; the most frequently cited is, of course, facilitating reuse.

A class should implement a particular type A sub-class, therefore, it is best to implement a corresponding sub-type. In other words, our class hierarchy should mirror our type hierarchy. This is usually called generalization. Other forms of inheritance do exist that do not follow this mirror image principle, including specialization and restriction, which do provide particular testing challenges.

Problems

(1) Substitutability Problem. Although a sub-type does satisfy and only strengthens (extends) the preconditions of its parent type, a sub-class does not necessarily do so. As such, a sub-type (generalization) can substitute for the parent class but objects built on specialization or restriction cannot. Such substitutions are, however, among the most common errors in OO.

For example, there is a case of restriction when you take a class such as SIMPLE_INTEREST_ACCOUNT and suppress the interest calculating features of it altogether to sub-class it into the new type NO_INTEREST_ACCOUNT. This new type does not have interest calculating features and thus cannot act as a sub-type of SIMPLE_INTEREST_ACCOUNT although it is a sub-class of it. This creates a problem in testing in that you do not quite know whether to test the suppressed features (as they are still part of the inheritance structure and implementation) or to ignore them (as they are not part of the type being implemented).

Under such circumstances, the testers will have a tendency to look at the con-

tract for the restricted type and then only test according to that contract, leaving behind all the potential side effects of the suppressed features.

(2) Mixing Inheritance Styles. Many designers mix different forms of inheritance in the one-class hierarchy. Although like many of the previous issues discussed, this is ultimately a design issue, it does impact the way you can effectively test a system. In other words, it can contribute to defects in the system and therefore within the scope of our interest, albeit more from a preventive aspect rather than a corrective one.

Imagine the situation depicted in Figure 1: The issue here is that class (D) is a sub-type of (C) and can be substituted for it. Class (C), which may be instantiated (or not), is however not a sub-type of (B), making (D) also not a sub-type of (B). Class (B) is a sub-type of (A), but nothing below it is, even though there might be a lot of further levels. How would you adequately test such a hierarchy?

(3) Deeply Nested Hierarchies. Even generalization, the sub-typing form of inheritance, presents challenges in defect management. In such a hierarchy, the tendency would be only for the leaf nodes to be instantiable. This, however, does not mean that all the features of all of the abstract classes are deferred, far from it. If a class cannot be instantiated, it cannot be tested directly.

Testing a class indirectly must ensure that the class is tested with respect to all possible permutations of the hierarchy down to each individual leaf level that can be instantiated. In a deep hierarchy that is also wide, this creates a combinatorial issue. There are issues, even in the case of a deep but narrow hierarchy. The complete contract of a leaf class is really the union of the contracts of all the parent types, many of them with some implemented operations; it is very possible to miss testing some of them.

(4) Multiple Inheritance. It is possible for a class to inherit from more than one super class directly. Multiple inheritance itself can be of two principal types: simple multiple inheritance and meshed inheritance also known as repeated inheritance. Meshed or repeated inheritance is the case where at least two of the super classes have a common ancestry.

The most obvious issue with testing in a multiple inheritance situation is that a sub-class may inherit a feature with the same name from more than one parent. The child class could use one or the other, but testing with respect to one may

not be adequate when the other is used. The object may interact with other objects, including a shadow or alias of itself with many strange and unexpected consequences, including method run-time clashes.

Solutions

(1) Inspect the Formal Contracts. If each type is written as a contract with its pre- and post-conditions and invariants carefully expressed, it would be possible to easily inspect the contracts of classes in a hierarchy to see if one is a sub-type of the other. There are simple rules that can be applied during an inspection session such as *the precondition of the child class must only extend or strengthen the precondition of the parent class* or *leave them unchanged*.

(1) Redesign. All hierarchies based on specialization or restriction can be rearranged into hierarchies of generalization.

(1) Use of Context Testing. This is giving up testing the individual routines individually and testing them in the context of their operation and collaborations. This is of course risky, as the routines will only be tested in known contexts.

(2) Redesign, Avoid Mixing Styles. It is a simple matter of avoiding the mixing of styles during design; you can always convert to generalization (see above).

(2) Segregate Styles. If it is not practical to convert styles, say when you have inherited the code and cannot redesign it, then you should consider each type set as a separate hierarchy and test accordingly. This means that starting from the leaf level, every time the inheritance style changes, all levels below are to be considered (logically abstracted into) one class in a current style relationship with the class above the current location.

(3) Avoid Deeply Nested Hierarchies. One solution is to avoid the problem altogether. As a rule of thumb, hierarchies of more than four to five deep are to be avoided unless they are structurally necessary (e.g., graphical user interface).

(3) Use Flattening Tools. High quality *flattening* tools – those that assist in producing a unified contract by collapsing the contracts involved in a hierarchy – can be of help but the problem is also one of logic and of testing, not of visualization alone.

(4) Avoid Multiple Inheritance. Current advice by many leading practitioners in OO is to avoid multiple inheritance if it is not absolutely necessary (very rarely is it so).

(4) Inspection. Use inspection rather

than testing to trace through the logic of multiple inheritance. Again, those inspection systems designed specifically for OO would provide facilities to deal with such issues.

Polymorphism

Polymorphism is the ability of an object to be many forms. A powerful, important, and useful mechanism available in most OO programming environments, polymorphism is considered the ability to substitute one type for another, or in other words bind a reference to multiple instances of different types, and is often closely linked to the concept of dynamic binding. Dynamic binding allows the binding of an object to be deferred to as

“It is important to realize that virtually every step in the software process is an occasion to introduce a defect that would ultimately manifest itself in the product being constructed.”

late as run time, thus permitting the use of different object types, depending on the context.

Problems

(1) Incorrect Binding in a Homogeneous Hierarchy. In homogeneous systems when various methods belonging to a polymorphic structure are closely related both conceptually and operationally, testing might not easily reveal a binding to an incorrect method.

(2) Server-Side Change. A polymorphic server might change without any regard to the client. Under such circumstances, an unchanged client may no longer be able to bind with the server.

Solutions

(1) No Real Good Solution Exists. Extreme care and extensive value testing are to be employed. Some techniques such as evaluating against an explicit post-condition might be helpful but this is not a complete solution.

(2) Inspection. Logic of the binding

between the server and the client can be clear during inspection.

(2) CRC. Anthropomorphization through the use of CRCs assists in clarifying the role of the server and its obligations. This is in essence a simulation and is employed during design or redesign. Of course, this technique is ineffective when the server is changed without knowledge of the client side.

Process Issues, Managing Defects

It is important to realize that virtually every step in the software process is an occasion to introduce a defect that would ultimately manifest itself in the product being constructed. Conversely, every step of the software process should be considered an opportunity for defect management. As software engineers, you must consider opportunities to prevent defects and opportunities to detect and therefore remove defects that have already been injected.

Another important realization, however, is that no defect management technique by and of itself is purely a preventive or a corrective one. For example, engaging in design inspection might provide the potential to identify and correct many defects that, if not resolved, would lead to defects in code. From the perspective of the design activity, this is corrective (as you are correcting the design) whereas from the perspective of implementation, it is preventive (as you are preventing the propagation of defects to implementation).

A number of such solutions and the software process stage in which they may be used are shown in Table 1 (see page 16). Software engineers must therefore select and utilize techniques that contribute to production of high quality requirements. These techniques assist in preventing the injection of defects of omission and commission into our specification document. This early preventive treatment has the potential to save you much defect management of the corrective kind later in the process.

Follow this by employing corrective techniques that attempt to identify and help remove defects already extant in the requirements document. Furthermore, you should deal with techniques that concern design, so you may generate defect-free design as much as possible. Designs, irrespective of the effort expended to generate them, will rarely be defect free. We still need to deal with design defect identification techniques such as design inspections. Program code defect identification through testing and

Software Engineering Process Stage	Preventive (P) or Corrective (C)*	Task	Technique
Specification	P	<ol style="list-style-type: none"> 1. Construct Common Dictionary 2. Set Focus/Goal 3. Build Consensus 4. Cover Model Space 5. Cover Functionality 6. Cover Non-Functional Requirements 	<ol style="list-style-type: none"> 1. Process Element Dictionary (PED) 2. Quality Matrix 3. State-Behavior Modeling (SBM) 4. UML, Formal Specification (e.g., Object Z) 5. Use Cases 6. Architecture
Specification	C	<ol style="list-style-type: none"> 1. Validate Requirements 2. Verify Requirements 	<ol style="list-style-type: none"> 1. Requirements Inspection 2. Requirements Inspection, CRC, Formal Methods
Design	P	<ol style="list-style-type: none"> 1. Ensure Traceability 2. Cover Design Space 	<ol style="list-style-type: none"> 1. Requirements Traceability Table (RTT) 2. Architectural Patterns, Design Patterns, Formal Derivation, Contracts
Design	C	<ol style="list-style-type: none"> 1. Validate Design 2. Verify Design 	<ol style="list-style-type: none"> 1. Requirements Traceability Check (RTC) 2. CRC, Formal Proofs, Design Inspection, Design Simulation
Implementation	P	<ol style="list-style-type: none"> 1. Ensure Uniformity 2. Ensure Traceability 3. Ensure Design Proximity 4. Ensure Accuracy 	<ol style="list-style-type: none"> 1. Coding Standards 2. RTT 3. RTT, Feedback 4. Pair Programming
Implementation	C	<ol style="list-style-type: none"> 1. Defect Identification 2. Failure Detection 3. Integration 	<ol style="list-style-type: none"> 1. Code Inspection, Static Analysis, Automated Analysis 2. Dynamic Testing; Specification Testing, Use Case-Based Testing 3. Integration Testing (e.g., Couple Testing, Pair-Wise Testing or Binary Testing), Regression Testing

* Indicates whether the technique in the right column has a preventive or corrective effect on the stage on the left.

Table 1: Defect Prevention Techniques

code inspection will also be needed for the same reason.

Finally you must deal with integration and defect management at the system level. Specific techniques for all these levels are available in the literature [3] and due to space limitations shall not be further discussed here.

Summary

This article presents a fault model for the OO paradigm of software development. This fault model concentrated on specific issues, whether product-based or process-based, that pertained principally to the object paradigm or resulted from its application. In doing so, however, no representations were made in terms of the absence or impossibility of other forms of faults that can arise independently of the paradigm utilized. As such, the model as presented is partial and focused.

The fault model describes the many potentials for producing defective software that might emerge as a consequence of utilizing the OO approach. It also discusses the difficulties that might possibly be encountered in managing and reducing the ultimate defect content of the

OO code.◆

References

1. Voas, J. "Object-Oriented Testability." 3rd International Conference in Achieving Quality in Software. Chapman and Hall, 1996: 270-290.
2. Gamma, E., R. Helm, R. Johnson, and

J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Design. Addison-Wesley, 1995.

3. Younessi, H. Object-Oriented Defect Management of Software. Prentice Hall, 2002.
4. Meyer, B. Eiffel: The Language. Prentice Hall, 1992.

About the Author



Houman Younessi is professor of Computer Science at Rensselaer Polytechnic Institute-Hartford Graduate Campus. He is a leading educator, practitioner, and consultant in object technology. Houman is the originator of the Single Building Model methodology and also a key member of the Object-Oriented Process, Environment, and Notation (OPEN) consortium and one of the designers of the OPEN Process. He is author of three books, including "Object-Oriented Defect Management of Software." Younessi has been instrumental in the formulation, evalu-

ation, and promulgation of the ISO 15504 Software Process Improvement and Capability Determination standard for software process capability measurement. Younessi is an international speaker, and has spoken at the International Conference on Software Engineering, the Conference on Technology of Object-Oriented Systems USA, and the Asia-Pacific Software Engineering Conference.

Department of Computer Science
 Rensselaer Polytechnic Institute
 Hartford Graduate Campus
 275 Windsor St.
 Hartford, CT 06074
 E-mail: houman@rh.edu