Computer Languages, Systems & Structures 36 (2010) 123 – 141

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Extending object-oriented languages with backward error recovery integrated support

Daniel Fernández Lanvin*, Raúl Izquierdo Castanedo, Aquilino Adolfo Juan Fuente, Alberto Manuel Fernández Álvarez

University of Oviedo, Dept. Informática, Oviedo 33007, Spain

A R T I C L E I N F O

Article history:
Received 26 June 2008
Received in revised form
16 March 2009
Accepted 16 May 2009

Keywords:
Recoverability
Object-oriented
Error recovery
Reconstructor
Software robustness

A B S T R A C T

One of the requirements of software robustness is controlling and managing runtime errors that might arise at certain points of application execution. In most object-oriented programming languages, this requirement is commonly implemented by means of exception handling. Although exception handling is a powerful tool to avoid system failure arising, there are still many situations where it is not sufficient to restore the system to a consistent state. Exception handling allows the developer to detect and locate errors, but it gives no information or tools to cover the error recovering task. Therefore, we propose an extension of the semantics of common object-oriented languages to restore the previous consistent state of the system in the presence of runtime errors, avoiding some of the tasks that exception-handling mechanisms delegate to developers. Our proposed solution is centered in the concept of "reconstructor", a declarative component oriented to automatically return the system to its last stable state. Based on this concept, we develop a non-intrusive code enrichment tool for Java, and we apply it to a real application in order to check the feasibility of the proposal. We evaluated the performance of the resulting code, obtaining reasonable and viable rates and overload.

1. Introduction

Ever since the NATO conference on *software development* in 1968, this task was viewed as an engineering process, and consequently a new objective emerged in every software project: every product needed to reach minimum levels of *software quality*.

Thirty years later, a great number of research lines are still devoted to obtaining new techniques and technologies to increase software quality. This work considers a specific factor of software quality: software robustness.

Software robustness can be defined as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [1,15]. It depends on two complementary factors: technological support and the developer's professional capacities. The wider the technological coverage given by the platform, the easier the developer's task of implementing robust software. The logic needed to implement recovery mechanisms in low-level languages was therefore more complicated and harder to maintain than the logic needed nowadays when developed with more mature technologies like modern object-oriented platforms.

The current exception-handling mechanism, present in almost all object-oriented languages, covers the tasks of *detection* and *localization* of the error source, but does not help to solve the third task that must be carried out to maintain consistency in the presence of any critical error: *error recovering*. This task is delegated to the developer who, as it will be seen later in this

* Corresponding author. Tel.: +34 985 105 094.
E-mail addresses: dflanvin@uniovi.es (D. Fernández Lanvin), raul@uniovi.es (R. Izquierdo Castanedo), aajuan@uniovi.es (A.A. Juan Fuente), alb@uniovi.es (A.M. Fernández Álvarez).

1477-8424/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2009.05.001

document, sometimes does not have enough information to reach his/her objective without developing complementary support processes.

In our opinion, some of these tasks can be implemented automatically by means of the information contained in the source code, so it is possible to avoid *some of the work* a developer must do to implement the necessary mechanism for maintaining consistency in the presence of errors. That is the motivation of this paper, which can be summarized thus:

> The modern object-oriented languages can be extended with a new semantic layer that complements current exception-handling mechanisms, simplifying the implementation of these processes purely oriented to consistency maintenance in the presence of exceptional scenarios.

What we propose in this document is the implementation of a mechanism to make consistency restoration easier for the developer. Since this is a very compressed expression of the idea, the next section justifies this work, starting from the analysis of some of the scenarios where the current exception-handling mechanism does not cover all the requirements of consistency maintenance, and describing the solution proposed as an extension of the modern object-oriented software development platforms. This extension is based on the concept of the *reconstructor*, and it is integrated with modern object-oriented platforms as a new semantic layer by the decoration of the source code.

2. Analyzing the problem: exception-handling limitations

As we said before, exception-handling is not always powerful enough to reach its objective: to *return the system to a consistent state and continue program execution*, especially when the only way to recover the consistency is to recover the last consistent state the system was in. It works right over the first two tasks of *detecting* and *localizing* the error source, but it does not cover *error recovering* or *correction*. This task is usually delegated to the developer, who must implement the *catch* block to reach the consistent state, but what should be coded inside? The presence of a catch block does not provide any warranty of model consistency restoration. We know the type of error, and where it happened, but there is no information about *how* to restore the system's consistency inside the catch block. Let us examine a set of common scenarios where recovery of the last consistent state of the system becomes difficult if we are using only the exception-handling mechanism.

In Fig. 1, the developer does not know how to restore the previous state of the model, which of the operations inside the block raised the exception and neither the changes produced in the model by the operations invoked before the failing one. Each operation that can fire an exception can be isolated to avoid some of these problems, but with consequent code obfuscation.

Consider the code in Fig. 2. Even in this new scenario, where the error source is isolated, we still have several problems in restoring the model. In this example, developers ignore the figures contained in the *paint* container which were rotated before the exception was fired. A good design could solve the last example, but sometimes that is impossible without determining application design.

In the example shown in Fig. 3, the developer knows what to restore, but s/he cannot do it. We could solve this problem by implementing a new method to decrease the counter (*counter.dec*( )), but we would be determining the final design on the basis of something that is not an original requirement of the system. Finally, sometimes we know what we must restore, and we can even restore it, but there is not enough information to do it, as in the example shown in Fig. 4.

```
try {
        doAnything ( );
        . . .
        doTheLastThing ( );
}
catch ( AnyException e ){

         // what should we do here?
}
```

Fig. 1. The exception-handling mechanism does not give any information on how to restore consistency.

```
//The paint object is a container for all the figures in the Canvas.
try {
        paint.rotate();
}
catch( AnyException e ){

        // which of the figures in the paint have been moved?
}
```

Fig. 2. Operation isolation helps to localize the exception source.