A Scenario-Based Approach for Modeling Abnormal Behaviors of Dependable Software Systems

Chien-Tsun Chen, Yu Chin Cheng, and Chin-Yun Hsieh
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, Taiwan 106
{s1669021, yccheng, hsieh}@ntut.edu.tw

Abstract

Traditionally, requirement capturing primary concerns about the normal behaviors of a system and leaves the abnormal behaviors as an afterthought. However, without the requirement document for the abnormal behaviors, a system's exception handling code tends to be poorly designed and error-prone and consequently decreases its robustness. In this paper, through the use of a scenario-based approach, we show that the abnormal behaviors can be modeled in a traceable and testable way in all life-cycle phases. Software developers, including the analyst, the architect, the designer, the programmer, and the tester can write or use different levels of scenarios to fulfill the robustness needs from the most relevant different perspectives. An example is given to illustrate its use.

1. Introduction

With software now being an integral part of many products, the increasing demand on better reliability in products has made software reliability an important issue. Reliability consists of two different but close related attributes: *correctness*—the ability for a system to deliver correct services according to its specification—and *robustness*—the ability to handle exceptional conditions [8]. From the behavioral aspect, correctness is embodied by a system's normal (functional) part and the robustness is represented by the abnormal part. To build reliable systems, software developers should consider both types of behaviors [8].

Regarding the construction of normal behaviors, a rich set of software methods and notations have been developed throughout the software development process from requirement, analysis, and design to implementation and testing. In contrast, relatively little is devoted to abnormal behaviors. While it is generally agreed that exception handling should be considered at the early stage of development, unfortunately, in most software development methodologies exception handling or robustness design is usually not the first-class concern but is often added as an afterthought. The decision of how an application handles exceptions is commonly postponed until implementation; the responsibility falls on individual programmer, who is usually ill-informed for making the right call. Without enough guidance for dealing with abnormal behaviors, the resulting implementation can be fragile.

Methods have been proposed to capture abnormal behaviors from use cases [9][11]. They are suitable for capturing failure scenarios and identifying failure points in which exceptions should be handled. However, many developers still do not know how to design a proper exception handler to cope with the exceptional condition. In most cases, exceptions are merely logged and without further treatment. Similar to the development of normal behaviors, there must be some sorts of methods to guide developers in developing abnormal behaviors from the requirement to the implementation in a streamlined manner.

To bridge the gap, this paper adopts the quality attribute scenarios (QASs) [2] and introduces *robustness QAS*s (RQASs) as a means to gather robustness-specific requirements and guide their implementation through the development life cycle states in a *traceable* and *testable* way. By establishing three capability levels, developers can have a well-defined goal that can be followed to design the exception handlers. Our approach is intended to complement current development methods with respect to robustness.

The rest of this paper is organized as follows. In Section 2, the QAS method is briefly reviewed and the proposed RQAS is introduced. Section 3 describes the steps of applying the RQAS approach. An example is presented in Section 4 to demonstrate its usage. Section 5 gives a conclusion and indicates possible future research directions.

2. Quality attribute scenarios and robustness requirements

2.1 Quality attribute scenario

The quality attribute scenario is a means of characterizing quality attributes for software architecture. It provides an *operational definition* for gathering quality-attribute-specific requirements such as availability, modifiability, and security. It consists of six parts [2]:

- *Source of stimulus*. The entity (a human or software) that generated the stimulus.
- *Stimulus*. An event to be considered when arriving at a system.
- *Environment*. The context where the stimulus is arrived.
- *Artifact*. This is the target which receives the stimulus.
- *Response*. The performed activity after the arrival of the stimulus.
- *Response measure*. The response measurement criteria which make the requirement testable.

Six types of QAS templates (i.e., general scenario generation) have been established to generate the requirements of availability, modifiability, performance, security, testability, and usability. To capture robustness requirements, a QAS template for robustness is proposed.

2.2 Robustness QAS

The most difficult part of defining a QAS template for robustness is to specify the *response* part of it. In this paper, robustness is achieved by exception handling. Thus, specifying the response implies that we have to enumerate all possible strategies for exception handling, which is arguably impractical. Even if we could do that, the result can still be too general to be useful. In our pervious work [4], we have proposed three evolvable exception handling capability levels as a foundation for specifying a system's exception handling strategy in a domain independent manner.

- Level 1: error-reporting. All errors must be reported but no other actions are made to cope with errors.
- Level 2: state-recovery. Error-reporting is practiced and the system must exist in a correct state in the face of exceptions by performing error handling [1] and cleanup.
- Level 3: behavior-recovery. State-recovery is practiced and the system takes alternative actions to deliver services before admitting its inability (performing fault handling [1]).

We adopt the capability levels in specifying the response. The proposed RQAS template is presented in Table 1. The stimulus is the exception that binds the source of stimulus (callee) and the affected artifact (caller). The source of stimulus is a software entity which fires the stimulus by raising an exception. The artifact, which could be a software entity or a human actor, is the receiver of the stimulus. The environment is the context where the stimulus is raised. Finally, the response measure provides one with some tangible criteria to verify the response. It enables one to observe a number of things, e.g., is a specific exception propagated? Are all planned exceptions caught? And is the system in a correct state in the face of exceptions?

The RQAS template will be used to generate specific RQASs which act as robustness requirement documents to guide the development of abnormal behaviors.

3. Applying robustness QASs

The workflow of applying the RQASs is illustrated in Figure 1. There are two development tracks in our model, one for normal behaviors and the other for abnormal behaviors. Note that adding the track for tackling robustness concerns should not and does not change the main flow of development activities; otherwise, it will not be easily adopted. The development track for abnormal behaviors complements the normal track where robustness is concerned.

3.1 End-to-end RQASs

Ideally, robustness and exception handling design should be considered in the requirement capturing phase. For example, Stelting [11] suggests exploring more fine-grained behaviors from a use case to capture *failure points* so that developers can plan suitable exception handling strategies based on them. In our approach, the system is viewed as a black-box. In this way, it is sufficient to write *end-to-end* RQASs at requirement phase. Specifically, we identify exception conditions (failure points) from *system events* [7] and rank these conditions based on severity of the consequence of failures, which are specified by the customers. Those with a high severity consequence are identified as *sensitive failure points* (SFPs) of the use case on hand. Since the SFPs are extracted from two interacting parties, namely the external actor and the system, the weaker the trust between the two, the higher the possibility that an exception will become a SFP.

Table 1. Robustness general scenario generation

Portion of Scenario	Possible Values
Source of Stimulus	Callee, server, system, a lower level component of a call chain, runtime environment (OS, VM)
Stimulus	Checked or unchecked exceptions
Artifact	Caller, user, client (another system), top level component of a system (root thread, main program), body components in a call chain (excluding tail and head), boundary components (e. g., facade), code
Environment	Analysis time, design time, coding time, and runtime
Response	— Report fault — Recover state — Recover behavior
Response Measure	— Exception propagation — Exception coverage — Deviation: the degree the system deviates from its correct state or normal behavior — Design diversity: the version of alternative implementation for each individual exception to be tolerated — Data diversity: the number of default values — Correctness and robustness of implementation

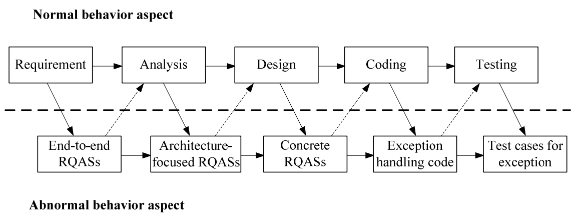


Figure 1. The development workflows

3.2 Architecture-focused RQASs

As soon as architecture takes shape in the development process, (e.g., in an architecture-centric process like the unified process (UP) [6][7]) architecture level constraints for robustness can be specified by writing *architecture-focused* RQASs. These architecture-focused scenarios are important because they outline the global exception handling policies at the architecture level, which will influence the subsequent exception handling design. For example, in a layered architecture, the architect may specify that "the semantics of all exceptions propagated over a layer boundary must be transformed in a form that is understood by its upper layer." Such a policy influences the exception handling design of all boundary components (e.g., a Facade). As another example, the architect may require a particular *application controller* [5] to implement the behavior-recovery capability of exception handling because it has enough application context information to do so.

3.3 Concrete RQASs

In the design phase, a designer constructs design classes and assigns robustness responsibilities to these classes. To do so, the designer combines the end-to-end RQASs and the architecture-focused QASs to write specific RQASs. The designer then realizes the robustness requirements by devising *concrete exceptions types* and choosing suitable *exception handling strategies* for the design model classes. For example, for a component with the state-recovery capability, an error-handling (restoring the component from error state) and a cleanup (releasing allocated resources) have to be done. Commonly used methods for error-handling include transaction, backward and forward recovery techniques [1]. You can put the cleanup code in the *finally* clause in Java and C#, or in the destructor in C++.

3.4 Coding and Testing

Once the concrete RQASs are ready, the implementation and testing become clear. Although exception handling involves non-trivial coding, programmers now have a specification for exception handling to follow and their goals are now explicitly stated. Moreover, by using exception handling utilities [4] and IDE tool support [10], the implementation effort can be significantly reduced.

Finally, the testers refer the response measure of the RQASs to design test cases and test the system's abnormal behaviors.

3.5 Traceability

Figure 2 depicts a traceability model with respect to normal behaviors (the left side) and abnormal behaviors (the right side). There are two new types of traceability: the end-to-end RQASs, the architecture-focused RQASs, and the concrete RQASs are traced to the use cases, the architecture, and the design models, respectively; the concrete RQAS is traced to the architecture-focused RQAS, and then is traced to the end-to-end RQAS. With the help of the traceability model, any change request can be reviewed regarding the normal and abnormal behavioral aspects.

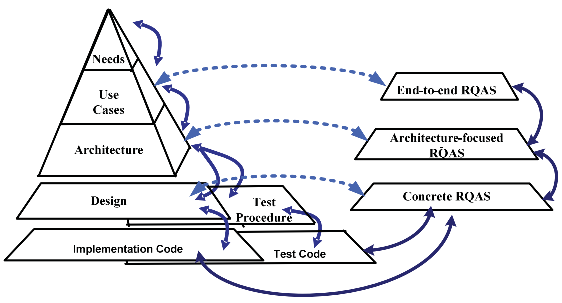


Figure 2. The traceability model for normal and abnormal behaviors

4. RQASs in action

The use of the RQAS is illustrated in the following development scenario derived from an open source software project: a Java continuous integration system (JCIS).

4.1 The example project background

JCIS supports continuous integration, a software development practice that is used in agile methods. Figure 3 shows its primary use case model. In the JCIS system, exceptions arise from failures in responding to the following system events: creating a project, checking out code from a repository, completing an integration task, and displaying integration reports. To keep the discussion concise, we consider the checkout failure SFP for the Checkout use case, which checks out project code from a CVS repository for integration.

The user issues a checkout command, which is received by the presentation layer component, PIC (project information center). The PIC then delegates the command to the CVSCmdManager, a controller for managing a variety of CVS commands. The CVSCmdManager then creates a CVSCheckout object and controls it to perform the checkout operation. The implementation of the CVSCheckout is realized by an open source library JavaCVS (a NetBean CVS module), which actually communicates with a CVS repository and handles low level CVS commands.

Now, considering the failure scenario:

Network connection break. The JavaCVS is connected to and is downloading code from a CVS repository. Network connection break causes the JavaCVS to throw a checked exception, ResponseException.

The application architecture regarding the checkout use case and the exception flow is depicted in Figure 4.

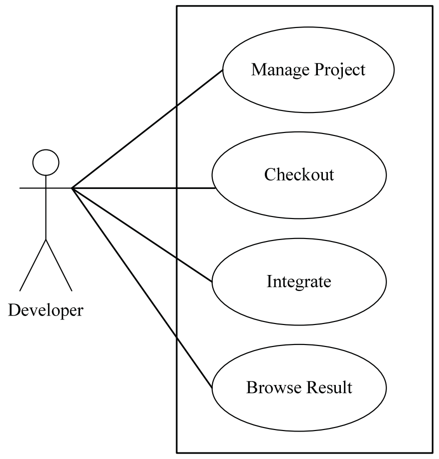


Figure 3. JCIS use case diagram

4.2 Writing RQASs

Now, suppose that we will implement the checkout use case and the SFP under consideration is the checkout failure in the current development iteration. Because the subsequent processing tasks depend on this use case, it is required that the failure should not mess up the local workspace (code storage). To specify this after completing the Checkout use case in requirement capturing phase, the analyst writes up an end-to-end RQAS as shown in Figure 5 (a), where the response is state-recovery.

In the analysis phase, according to the project's functionalities and quality requirements, the layered system architecture has emerged. For the layered architecture, the architect specified two architecture-focused RQAS as shown in Figure 5 (b) and (c). Figure 5(b) indicates that the boundary components have to report exceptions that are meaningful for their upper layer's callers. Note that this is a common exception handling best practice for layered applications [3], which will be followed in the project. Figure 5(c) shows the allocation of the state-recovery responsibility to an application layer component, which is expected to have the enough application contexts to implement the state-recovery strategy.

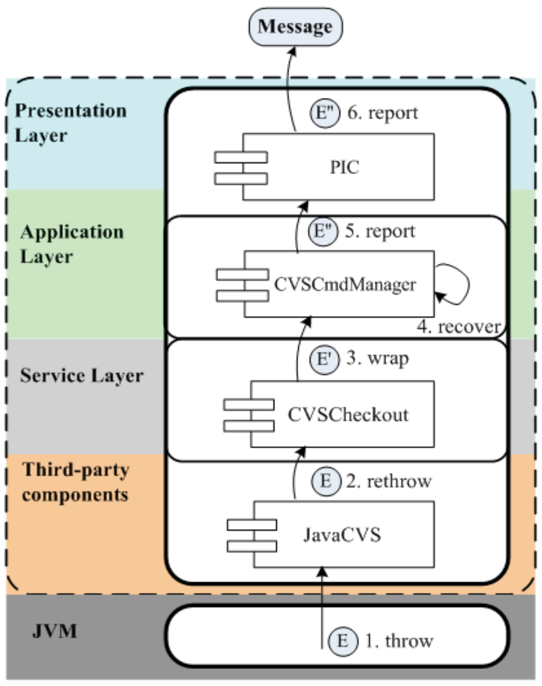


Figure 4. The JCIS layer architecture and exception flow for the Checkout use case

In design phase, the designer first constructs the design class model according to the use case and the system architecture. To assign exception handling responsibilities to these design classes, the designer then writes two concrete RQASs as shown in Figure 5(d) and (e), respectively. The RQAS in Figure 5(d) specifies that "A checked exception ResponseException is received by the CVSCheckout component while it is downloading code from the repository via the JavaCVS component. The CVSCheckout, which is a command object, does not have enough context information to cope with the exception. So, it wraps the exception into a CheckoutException, which is propagated to its caller, the CVSCmdManager." In the concrete RQAS of Figure 5(e), the CheckoutException is captured by the CVSCmdManager (an application layer controller), which is responsible for restoring the workspace.

With both the design class model and the concrete RQASs in place, the programmers can now implement the exception handling requirements. First, two exception classes, CheckoutException and CVSException, are created and bound to the CVSCheckout and the CVSCmdManager, respectively. Then, in the CVSCheckout component an exception handler is written, which catches the ResponseException and wraps it into a CheckoutException. In order to be prepared for recovering from an exception, the CVSCmdManager makes a backup copy of the workspace before invoking CVSCheckout. It then restores the workspace if a CheckoutException is caught. Figure 6 is the code snippets of the CVSCmdManager.

Figure 7 illustrates the test case for testing the checkout function of the CVSCmdManager. It is easy to design a test case for such an abnormal behavior from the concrete RQAS.

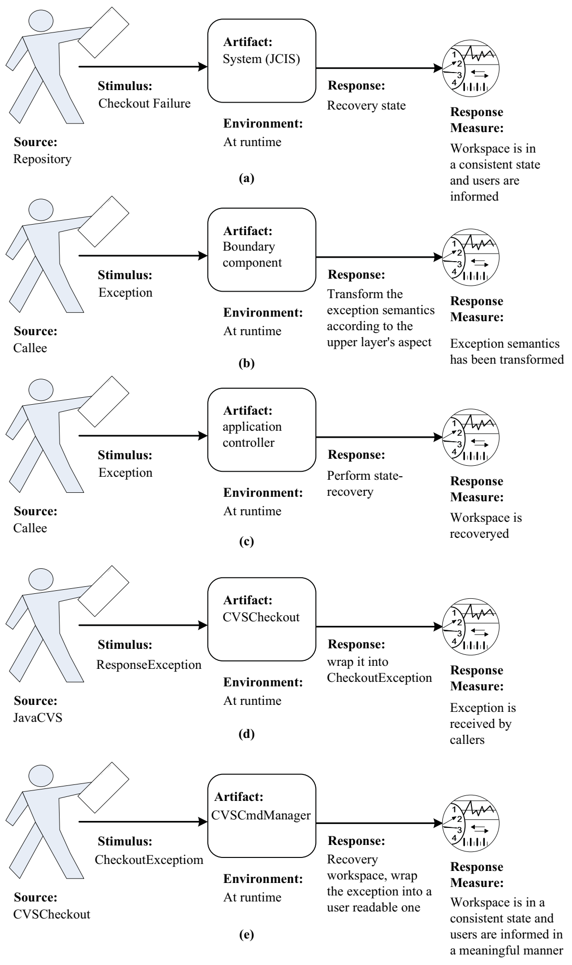
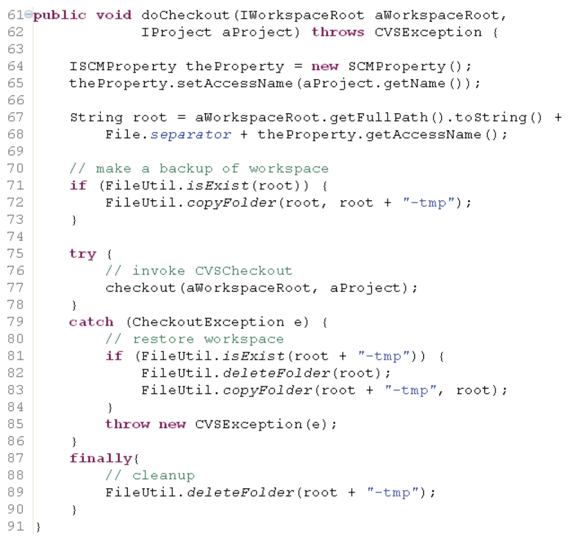

Artifact: System (JCIS)
Stimulus: Checkout Failure
Response: Recovery state
Source: Repository
Environment: At runtime
Response Measure: Workspace is in a consistent state and users are informed
(a)
Artifact: Boundary component
Stimulus: Exception
Response: Transform the exception semantics according to the upper layer's aspect
Source: Callee
Environment: At runtime
Response Measure: Exception semantics has been transformed
(b)
Artifact: application controller
Stimulus: Exception
Response: Perform state-recovery
Source: Callee
Environment: At runtime
Response Measure: Workspace is recoveryed
(c)
Artifact: CVSCheckout
Stimulus: ResponseException
Response: wrap it into CheckoutException
Source: JavaCVS
Environment: At runtime
Response Measure: Exception is received by callers
(d)
Artifact: CVSCmdManager
Stimulus: CheckoutException
Response: Recovery workspace, wrap the exception into a user readable one
Source: CVSCheckout
Environment: At runtime
Response Measure: Workspace is in a consistent state and users are informed in a meaningful manner
(e)

Figure 5. The RQAS examples

5. Conclusion and Future Work

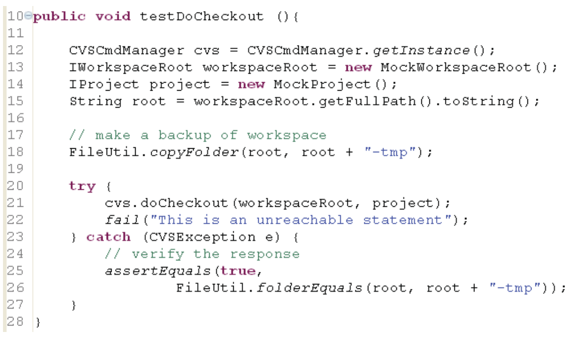
In this paper, we have proposed the robustness quality attribute scenarios for capturing the abnormal behavioral requirements of a system in a traceable and testable manner. We introduced three levels of scenarios—the end-to-end, architecture-focused, and concrete RQASs—which complements the traditional software development life-cycle artifacts for increasing a system's robustness by exception handling. The exception handling of a real-world application demonstrated to show the usage of RQASs.

Currently, we are applying this method on evaluating the robustness of a verity of software architectures. By writing parallel sets of RQASs, a tradeoff could be made to judge the suitability of the candidate software architectures for a particular application domain.



```java
public void doCheckout(IWorkspaceRoot aWorkspaceRoot,
        IProject aProject) throws CVSException {

    ISCMProperty theProperty = new SCMProperty();
    theProperty.setAccessName(aProject.getName());

    String root = aWorkspaceRoot.getFullPath().toString() +
        File.separator + theProperty.getAccessName();

    // make a backup of workspace
    if (FileUtil.isExist(root)) {
        FileUtil.copyFolder(root, root + "-tmp");
    }

    try {
        // invoke CVSCheckout
        checkout(aWorkspaceRoot, aProject);
    }
    catch (CheckoutException e) {
        // restore workspace
        if (FileUtil.isExist(root + "-tmp")) {
            FileUtil.deleteFolder(root);
            FileUtil.copyFolder(root + "-tmp", root);
        }
        throw new CVSException(e);
    }
    finally{
        // cleanup
        FileUtil.deleteFolder(root + "-tmp");
    }
}
```

Figure 6. Code snippets of the CVSCmdManager



```java
public void testDoCheckout (){

    CVSCmdManager cvs = CVSCmdManager.getInstance();
    IWorkspaceRoot workspaceRoot = new MockWorkspaceRoot();
    IProject project = new MockProject();
    String root = workspaceRoot.getFullPath().toString();

    // make a backup of workspace
    FileUtil.copyFolder(root, root + "-tmp");

    try {
        cvs.doCheckout(workspaceRoot, project);
        fail("This is an unreachable statement");
    } catch (CVSException e) {
        // verify the response
        assertEquals(true,
                FileUtil.folderEquals(root, root + "-tmp"));
    }
}
```

Figure 7. The test case for the CVSCmdManager

6. Acknowledgement

This work is partially funded by the National Science Council of Taiwan under the grant NSC95-2221-E-027-046-MY3.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, Jan-Mar 2004, pp. 11-33.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, 2nd*, Addison Wesley, 2003.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*, 1996, John Wiley & Sons.

[4] Y. C. Cheng, C.-T. Chen, and J.-S. Jwo, "Exception Handling: An Architecture Model and Utility Support," *apsec*, 12th Asia-Pacific Software Engineering Conference (APSEC'05), 2005, pp. 359-366.

[5] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002.

[6] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison Wesley, 1999.

[7] C. Larman, *Applying MUL and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd*, Prentice Hall PTR, 2005.

[8] B. Myer, *Object-Oriented Software Construction, 2nd*, Prentice Hall PTR, 1997.

[9] C. M. Rubira, R. de Lemos, G. R. M. Ferreira, and F. Castor Filho, “Exception Handling in the Development of Dependable Component-Based Systems,” *Softw. Pract. Exper.*, John Wiley & Sons, 2005, pp. 195-236.

[10] T.-C. She, *An Exception Handling Architecture and Utility Support for Java Language*, Master Thesis, National Taipei University of Technology, Taipei, Taiwan, 2006.

[11] S. Stelting, *Robust Java: Exception Handling, Testing and Debugging*, Prentice Hall PTR, 2005.