Exception Handling in an Event-Driven System

Jan Ploski
Business Information Management
OFFIS Institute for Information Technology
26121 Oldenburg, Germany
jan.ploski@offis.de

Wilhelm Hasselbring
Software Engineering Group
University of Oldenburg
26111 Oldenburg, Germany
hasselbring@informatik.uni-oldenburg.de

Abstract

Exception handling mechanisms were invented in 1970s to support structured programming methods for hierarchically organised software systems. The need to increase reusability and flexibility led to the development of new programming paradigms that do not emphasise hierarchical design. Event-driven systems—in which objects communicate using notifications about changed states—are a prime example. Unfortunately, this style of communication makes exception handling more difficult than in hierarchical systems.

We contribute an analysis of the factors which influence exception handling in event-driven systems. The main focus of our discussion lies on the challenge of appropriate exception propagation. We provide results from an empirical case study performed on the source code of the Eclipse IDE that support our analysis.

1. Introduction

When exception handling mechanisms were introduced into programming languages in 1970s software systems were often designed and implemented from scratch according to the principles of structured programming. Today, system builders are much more concerned with integrating the available pre-built components and installed products. Event-driven systems are a popular method of software integration.

Unlike software architectures, exception handling mechanisms offered by mainstream programming languages have not changed much since 1970s. Additionally, there are few rules for discerning between the "good" and "bad" practices of exception handling with respect to software architecture and empirical evaluation is rarely attempted by researchers.

In this paper we observe that the traditional approach to exception handling, which relies on exception propagation to a method's invoker, does not apply to event-driven systems. The data collected in the case study serves to support our analysis. When considered separately, our results speak against an event-driven architecture. If such an architecture is preferred for other reasons, developers must take into account that the language mechanisms will likely be insufficient for exception handling and devote special attention to language-neutral alternatives.

In Section 2 we discuss how early work on exception handling relates to structured programming and why current exception handling mechanisms are appropriate for hierarchically structured software. Section 3 presents a rationale for event-driven systems, in which objects react to new information rather than to commands issued by other objects. We provide several examples showing that this style of communication is independent from the issues of concurrency. In Section 4 we focus on the exception propagation from event handlers. Section 5 contains results of our empirical study. We present our results in Section 6 and conclude with a discussion of related work.

2. The origins of exception handling

Exception handling was first proposed as a syntactic extension to languages which supported structured programming. Unsurprisingly, exception handling mechanisms are strongly related to structured programming.

We wish to stress that some design assumptions in exception handling mechanisms do not fit the features of modern event-driven systems. We first explain why they are correct with respect to the original target systems.

2.1. Structured programming

Structured programming is concerned with decomposing software systems into modules and defining desirable relationships among these modules.

A *module* consists of (optional) storage space along with operations which present an abstraction of the managed

state to its clients. In that respect, a traditional structured programming module resembles an object. Only a single instance of a module exists in the system at run-time. Hence, modules can be thought of as singleton classes [7]. The services provided by a module are either computations or input/output operations on hardware.

Structured programming recommends dividing the knowledge captured in a software system into modules to increase the locality of anticipated implementation changes. Local changes are easier to incorporate and less error-prone. Operations provided by modules should be organised in a strict hierarchy defined by their "use" relationships, with higher-level operations invoking lower-level ones [17].

Higher-level operations also need to be protected from "knowing" the secrets about data structures and algorithms encapsulated by modules enclosing the lower-level operations. Again, the reason is to constrain the effects of a system's modification (information hiding, cf. [16]). Moreover, the lower-level operations should be kept unaware of their clients to foster reuse.

Overall, structured programming improves software maintenance by making changes easier to manage and developer responsibilities easier to assign.

2.2. Structured exception handling

Structured programming did not originally include language support for run-time error handling. Language designers soon recognised this omission and invented the mechanisms that shape our present way of thinking about exception handling. For example, Goodenough's work [8] laid foundations for control flow constructs present in modern languages.

A less referenced, yet equally insightful publication from the same period is by Parnas and Würges [18]. Rather than focus on syntax, the authors point out the potential goal conflicts between information hiding and exception handling. These conflicts have become increasingly relevant with the adoption of methods such as event-driven programming.

Parnas and Würges observe that even verifiably correct programs have to account for "undesired events" (which we now call "exceptions"). Examples are bugs, hardware failures, but also inconsistent data inputs, beyond the system's control. Some exceptions can be anticipated during design. However, many are only revealed during the initial period of software use. Therefore, it is practical to design software flexibly enough to introduce exception handlers later, after we have learned about the most common or critical faults.

Importantly, while we need to design programs to enable exception handling, we do not need (and perhaps even cannot) implement the recovery algorithms early. First, anticipating faults is difficult. Second, overeager error handling encumbers software reuse. Specific recovery algorithms hard-wired into modules represent strong assumptions about their clients. To improve reuse, structured programming avoids such assumptions. Therefore, the exception handling technique proposed by Parnas and Würges supports deferral of exception *handling* decisions, but encourages early and preferably automated *detection* of errors.

The choice of where to delegate the exception handling responsibility is obvious in structured programming: as a consequence of hierarchical design, the higher-level operations maintain more information about the execution context. This fact makes them likely candidates for performing recovery or diagnostic reporting. For example, a high level program may "know" whether it is run in interactive mode with a human user capable of reacting to console error messages or in batch mode where logging or automatic retry might be preferable. In short, hierarchical systems support exception handling by *escalation* of problem information in the hierarchy.

2.3. Exception handling vs. information hiding

Parnas and Würges [18] explain the implicit limitations of exception handling concepts based on their ties to structured programming. They recognise that the possibility of exceptions clashes with the goals of information hiding: "Unless we abandon the idea of abstraction completely, no design always presents all of the information usable for recovery". Propagating an exception upwards the call chain in a lower level operation's execution may leave the enclosing module in an unspecified state. However, the client can only rely on specified states of the used module. As a remedy, Parnas and Würges recommend making the module's interface more granular. It should always be detailed enough to support reasoning about potential run-time problems. Consequently, some states that could remain hidden if no exceptions were possible may need to be exposed for the sake of enabling recovery. A similar approach can be found again in Cristian's later formalisation of exception handling in sequential programs [5]. However, Cristian treats exceptions as extensions to specifications of individual operations. He does not reflect on module-scope design issues. The difference is subtle, yet important. We shall return to it when discussing the results of our case study (Section 6).

Information hiding and recovery requirements conflict even more visibly in object-oriented systems [14]. These systems are more dynamic and apply abstraction on a greater scale than traditional structured programming. In structured systems, the correspondence between the implementation of a module and its specification can be checked statically at compile time. Object-orientation encourages compile-time implementation independence through late binding. An object's response to a received message depends on the object's run-time type, which is set earlier in

the control flow. Late binding is a welcome feature, as it allows great flexibility in replacing and reconfiguring parts of an object-oriented system.

Unfortunately, the traditional proposition that to enable recovery, the granularity of a module (or class) interface should reflect all possible run-time exceptions becomes less plausible. Underspecification and overspecification of possible exceptions are common in real systems. The former may lead to disastrous results when an unexpected exception occurs at run-time. The latter tends to increase careless handling of exceptions that in reality never occur in a particular system configuration. It makes software more expensive to develop and less robust by diluting attention devoted to exception handlers and their testing [11, 6].

3. Event-driven systems

The popularity of object-oriented programming led to the development of systems that are no longer strictly hierarchical. *Message passing*, a core concept in object-orientation, does not demand hierarchical "use" relationships among objects. Low coupling can be achieved by considering at least some objects as highly autonomous entities capable of reacting to notifications about changes in their environment and broadcasting such notifications to indicate their computational progress.

Systems based on the concept of a control flow hierarchy require their replaceable parts to fit predefined utility criteria. For example, a module's interface describes its role in the system based on its offered abstract operations. In event-driven systems, on the other hand, the focus lies on specifying communication protocols. The exact purpose of each communicating part remains unknown until late implementation or deployment. This feature is particularly advantageous in environments where only a small subset of requirements can be captured initially. Some examples include

- object-oriented frameworks, whose success depends on their ability to become customised for particular usage contexts and to accept third-party contributions,
- enterprise information systems, in which business requirements may change frequently after the initial release and which must integrate with pre-existing legacy software,
- inter-organisational systems, which have to be administered and extended decentrally due to the constrained availability of processed (business-critical) information.

Apart from the improved extensibility, event-driven systems may also offer better scalability than their hierarchically designed counterparts. For example, empirical research demonstrates that event-driven transaction processing systems better support graceful degradation under heavy load [23].

We next describe sample architectures of event-driven systems to emphasise that the general concept of objects acting upon availability of information rather than upon commands may be implemented in multiple ways. Specifically, a notifier can deliver events either synchronously or asynchronously. In our opinion, it is important to generalise from the implementation details to notice that the issues with exception handling are not caused by the limitations of a particular programming language.

3.1. Synchronous event delivery in graphical user interfaces

In synchronous event delivery the notifier blocks execution until the dispatched notification for an event is processed by all interested parties. Notifications are dispatched sequentially; the control flow is transferred to each of the recipients in turn and finally returns to the notifier.

To provide an example of synchronous event delivery, we describe the Model-View-Controller (MVC) pattern [4] in desktop GUI programming. We frequently encountered this pattern in our empirical study.

The MVC pattern specialises the more general Observer [7] design pattern and is found in most GUI frameworks. Two sources of events exist in GUI programming:

- user actions, such as key strokes or mouse movements, and
- application-generated events, such as requests to repaint a screen region.

Typically, a GUI framework translates primitive input events into semantically richer events, such as selection changes, enablement/disablement of widgets, button presses, scroll bar movements, and many more. The processing of these events is, in turn, application-specific.

Events are delivered synchronously. It is natural given a single interacting user with a single set of input/output devices. Events created by the windowing system and those created by an application are dispatched and processed in the same execution thread. Therefore, an application must handle them without delay and delegate any long-running operations to another thread.

Furthermore, the MVC pattern encourages application programmers to define their own event types based on the needs of an application. Classes in the *model* role represent data objects and dispatch notifications to *controller* classes whenever a change occurs in the data. A controller reacts to such notifications by updating one or more *view* objects.

Moreover, it processes notifications about user actions reported by the view. It possibly reacts to such actions by modifying data in the model. A developer who uses the MVC pattern can easily exchange views while keeping the same model or manipulate the model directly and rely on notifications to keep a group of views synchronised. Events are thus used to maintain desired state invariants across a system without sacrificing extensibility.

3.2. Asynchronous event delivery in enterprise information systems

Asynchronous event delivery means that an event source may proceed with execution immediately after dispatching an event without waiting for the completion of its processing. Asynchronous schemes are supported by multithreading and explicit event queues. Implementations often also support event persistence, routing and prioritisation.

Exemplarily, we consider an enterprise message bus implemented using Java Message Service [10] and its relationship to service-oriented architectures (SOA). SOA are a modern approach to structuring enterprise applications based on the following principles:

- isolate business functions from execution contexts,
- publish information about these functions using directory services, and
- provide access with standardised, implementation-neutral protocols.

The main benefit of adhering to SOA is the ability to quickly compose enterprise information systems (EIS) of fine-grained building blocks of business logic. It is an attractive alternative to the integration of earlier, shrink-wrapped, coarse-grained products of ERP software vendors.

Basic SOA focuses on decomposition of applications into services usable according to a request/response model. However, more advanced SOA concepts provide for event-based communication among services. They promise easier integration of disparate (and distributed) legacy systems, better performance (when compared to polling), traceability, and load balancing. Meanwhile, production-quality open source messaging frameworks are available that support event-driven programming in the EIS context [9]. Ongoing standardisation activities (e.g., WS-Notification [15]) also confirm its broad relevance.

The Java Message Service [10] is a vendor-specific standard by Sun Microsystems related to event-driven architectures. It specifies a programming interface for server-side Java applications which allows creating, sending, receiving and reading messages. Implementations with functionality dictated by JMS are provided by various vendors of enterprise middleware.

JMS provides two models for sending messages (including event notifications): the point-to-point model and the publish/subscribe model. The former requires the JMS client to address a message to a specific queue available in the system. The latter allows to establish a named communication channel used concurrently by multiple senders and receivers. In both cases asynchronous message delivery may be supported by a JMS implementation. Sending of messages is asynchronous by default, but synchronous invocations can be emulated by explicitly waiting for a reply after a message is sent.

4. Exception propagation during event handling

An exception which occurs in a hierarchically structured system in response to a requested operation is naturally propagated to the operation's invoker (Section 2.2). This is not true for exceptions raised during event processing in an event-driven system. Apart from implementation difficulties, such as the possible lack of a back-reference from an event handler to an event issuer, more general problems emerge.

We noted earlier that the flexibility of an event-driven system comes from decoupling the event producers and consumers. However, the premise of any exception propagation is that the recipient of a forwarded exception can better decide about the required actions than the forwarder. No general argument can be made in favour of propagating an exception from an event handler to its issuer. Unfortunately, this is the only kind of propagation supported by mainstream programming languages.

Existing advice available to developers of event-driven systems concerns whether or not to propagate exceptions from event handlers to their invokers. Two basic approaches exist, which we discuss in context of the Observer pattern. The notified objects must provide a no-throw guarantee [22]. Alternatively, they may only supply limited failure information by throwing some generic exception.

4.1. Providing a no-throw guarantee

The implementor of a notifier may decide that the notified objects must not throw any exceptions. This approach appears attractive as it frees her from any exception handling responsibilities. However, it requires that the full information necessary for exception handling be available to the notified object. It also assumes that any constraints on the behaviour of the notified object do allow appropriate exception handling. While the first requirement is self-evident, the latter needs further elaboration.

As an example, consider the notification mechanism in event-driven GUI programming explained in Section 3.1.

An application programmer provides concrete implementations of event handlers specified by the framework. To maintain a responsive GUI, event handlers must satisfy soft real-time constraints during their execution. Moreover, because of the sequential nature of event handling, they must not rely on any service provided by other event handlers. Such constraints make it impossible to report an exception to the user through the GUI or to directly solicit user input in an event handler.

To work around the problem while still maintaining the no-throw guarantee, we must handle exceptions asynchronously. That is, we delegate the processing of exceptions to another thread and postpone it until the remaining registered event handlers have seen the current event. In short, we must use custom propagation. However, the basic no-throw approach does not include any provisions nor recommendations about such alternative propagation, and it is not syntactically supported by the programming language. The necessity to deal with custom propagation partly defeats the point for using exceptions, as we no longer have a uniform, explicit way of dealing with run-time errors.

4.2. Propagation to the invoker

If the requirement of a no-throw guarantee turns out difficult to satisfy in practice, we might hope that the second approach, propagating a generic exception to the notifier, has better prospects. Alas, this is not true.

Consider a scenario in which a component manages data stored in a database shared with other components. The components are unaware of each other and coordinated by an implementation framework. When a component updates the shared database, the remaining components are notified by the framework. They react by executing component-specific code, in which an internal exception might occur. In this case, the exception is propagated to the framework, which in turn propagates it to the component responsible for the update.

While this approach seems reasonable at first, the updating component is, by design, unable to handle any exception originating from another (notified) component. In fact, the reason for such an exception might not at all be related to the update event. Obviously, exception propagation to the notifier (the framework) and consequently to the event originator (the updater) is undesirable.

5. Case study (Eclipse IDE)

We performed an empirical study to evaluate claims about the relationship of event-driven communication and exception handling made in the preceding section and in [19]. A secondary goal of the study was to test the feasibility of assessing exception handling's quality using source code inspections.

We chose an exploratory case study as the preferred research method for several reasons. First, the described propagation rules are heuristics and thus inherently difficult to evaluate analytically. Second, participants of real software development projects have to rely on similar heuristics for making exception handling decisions. Third, there is no accepted theory which would define the attributes of "good" exception handling. Due to the lack of established quality metrics, we did not attempt to formulate statistically testable hypotheses. We describe our surrogate metrics in Section 5.4.

5.1. Research questions

Our case study focused on a single system which we qualified as event-driven based on prior experience (see below). We posed the following research questions about the examined system:

- What is the incidence of event-based communication in comparison to the usual imperative control flow constructs?
- Does event-based communication affect exception handling?
- What happens with exceptions that occur during the processing of event notifications? In particular, what propagation paths are selected for such exceptions?

5.2. Context

We studied the source code of the Eclipse SDK [1], version 3.1.1. We have been using this open source Java programming environment for more than 3 years. We also implemented custom extensions (plug-ins) during that time, resulting in a familiarity with the code base and tools necessary to perform the analysis. Another reason for analysing Eclipse was its representative character: a non-trivial desktop application with a graphical user interface. Finally, we decided that it qualified as an event-driven system because its flexibility relies on indirect communication among independently contributed extensions.

5.3. Setup and execution

Our analysis covered contents of all Java packages from the standard distribution of Eclipse SDK, version 3.1.1, whose names begin with `org.eclipse`. These packages comprise the Eclipse workbench, a general platform for implementing programming tools, as well as the Java IDE

Entity	Found
Listeners	227
Types	11839
Listener classes	2368
Event handlers	3644
Event handlers with try/catch	106

Table 1. Summary of data processed in the case study, showing counts of entities found by our search algorithm. *Listeners* **are Java interfaces containing one or more event handlers.** *Types* **are all classes and interfaces over which the search was performed.** *Listener classes* **are implementations of the listeners.** *Event handlers* **are individual methods specified in listeners. The last row contains exception-sensitive event handlers.**

functionality implemented by the Java Development Tools subproject.

We used a customised version of Eclipse's own syntactic search feature to locate a subset of all methods that qualify as exception-sensitive event handlers (Table 1). Our search relied on a standard Java naming convention for applying the Observer pattern. We searched for implementations of all methods declared in Java interfaces with the name suffix `Listener`.

From all identified event handlers, we selected for further evaluation only those that contained a `try/catch` block in their source text. This filtering step may have eliminated some of the potentially interesting cases. Concerned are those cases in which exception handling (or non-standard propagation) are embedded deeper in the call chain than in the event handler's text itself. A more precise method would require static analysis [6] of such call chains. However, even then the results would have been limited by Eclipse's extensive use of Java interfaces.

After the filtering steps, we analysed the handlers individually by studying their source code. We reflected upon similarities and differences in exception handling, and incorporated them into an incrementally developed classification scheme (Table 2). After the first pass, we returned to handlers examined earlier in order to apply to them all classification criteria that were added during this pass. We analysed approximately one half of all qualifying event handlers. However, we observed that the set of classification criteria ceased to grow early during this analysis. We therefore assume that the examination of remaining handlers would contribute little to the qualitative results.

5.4. Classification scheme

We analysed each of the 50 handlers to obtain answers to the following tests from our classification scheme (Table 2):

Evaluated criteria	Sample values	
Event handler name	package.method	50
GUI event?	No	10
Print?	No	0
Log?	Yes	31
Report via GUI?	No	8
Discard?	No	5
Handle impossible?	Yes	19
Evaluate exception object?	No	0
Handle non-exceptions?	No	7

Table 2. Our classification scheme for exception-sensitive event handlers (see Section 5.4). The left column contains criteria analysed for each handler, the middle column some sample values, the right column the number of handlers which positively passed each test in our study.

1. Is the handled event a GUI event? We wanted to estimate how much of the event-based communication is triggered by the GUI framework.
2. Is the exception handling limited to writing a stack trace? This question served to assess the maturity level of the analysed code.
3. Does the exception handling consist of logging?
4. Does the exception handling consist of displaying an error dialog or otherwise indicating that the displayed result is wrong?
5. Are possible exceptions discarded? We found out whether an exception could actually occur by code inspection, then checked for empty catch blocks.
6. Are impossible exceptions caught? The opposite of the previous test, it concrened non-empty catch blocks for exceptions that could never occur. We only concluded that an exception could not occur when the invoked implementation did not throw it and was under control of the event handler's author.
7. Does the exception handling alter control flow based on the contents of an exception? Examples would be providing a degraded level of service or undertaking fault removal.

8. Are "harmless" exceptions (non-exceptions) handled? Some event handlers deal with exceptions that are not caused by error, but serve to implement required functionality. For example, Java's standard method for checking whether a string represents an integer value throws an exception.

5.5. Additional analysis

Apart from the above described classification, we performed a more thorough code inspection in several cases to evaluate how exception handling could be improved. We were especially interested in finding out whether exceptions identify faults (causes of errors [2]) and carry information relevant for recovery. Finally, we investigated whether a degraded level of service could be offered after the exception's occurrence.

6. Conclusions

Confirming our expectations, Eclipse employs a substantial amount of event-based communication. Roughly every fifth class contains an implementation of one or more event handlers (Table 1). A minority of the examined event handlers (10 of 50) process GUI events. Notification mechanisms are thus not used due to the programming style imposed by the windowing toolkit (SWT), but rather as a general method of maintaining consistency across system components.

We were surprised by the number of cases in which impossible exceptions were handled (19 of 50 analysed event handlers). On the one hand, this observation means that we had overestimated the incidence of exception handling during notifications. On the other hand, the numerous cases in which impossible exceptions are caught and logged without comments suggest that exception handling had not been considered in detail by developers.

We found 7 cases in which exceptions did not indicate a run-time error. They were either used instead of normal return values or were specified in catch blocks, but could not in fact occur and were safely ignored. Consequently, only a half (24) of the analysed exception-sensitive event handlers dealt with genuine exceptions.

The prevalent way of handling exceptions was logging them and continuing execution. The event handlers thus provided a no-throw guarantee to their invokers (cf. Section 4.1). The logged messages typically consisted of a context description in natural language together with other textual information carried by the exception object. The latter information was produced at throw sites rather than collected during propagation.

Eclipse's developers had made liberal use of mapping low level exceptions to higher level ones. This approach is often advocated in the literature [3, 12, 20]. From the viewpoint of understanding exception handling, we consider it a poor alternative to the solution proposed by Parnas (see Section 2.3). Catch blocks examined in our study referred to very abstract exceptions (e.g., the `CoreException`, "a checked exception representing a failure"). High level exceptions carried integer status codes to describe the underlying faults. A possible reason for introducing such (documented) status codes was to limit the overall number of exception classes. However, in consequence we could not determine the relevant faults simply by simply examining method signatures. In many cases, method documentation described these faults, with varying levels of accuracy. Although the error codes were recorded at throw sites, we found no single case of their evaluation in catch blocks. This is not surprising because subclasses of caught exception types can add their own status codes. However, an invoker cannot assume anything about the existence of such subclasses.

The most important result of the study is that a system's design—including its hierarchy of exceptions and their processing during propagation—severely impacts the later possibilities for the analysis and improvement of exception handling. The study demonstrates that exception handling in our system fulfils the role of reporting and propagation of *failure* and contributes little to explaining and removing *faults* or to maintaining an acceptable level of service under unusual conditions. Exception handling mechanisms offered by modern object-oriented programming languages do not help to solve this problem, which is related to system design, not to implementation.

7. Related work

Past research focused on support for the evolution of exceptional specifications in object-oriented software development [13]. However, it seems unlikely that providing type-safe ways to alter method specifications can solve the overall exception-related problem of interface and implementation mismatch. The argument provided by Parnas [18] demonstrates that exceptions may induce a complete redesign of a module's interface. Our empirical study provides evidence that this argument is not followed in practice. We speculate that the need for extensibility outweighs concerns related to exception handling.

Other researchers have criticised deficiencies of mainstream exception handling approaches for specific classes of systems. For example, Romanovsky and Kienzle treat concurrent systems [21]. Our work differs in two aspects: the type of examined systems and the research approach. Romanovsky and Kienzle postulate an appropriate system structuring for satisfactory exception handling. In contrast, we point out that architectural decisions may be dominated

by factors not related to reliability, such as extensibility and the availability of tools.

Howell and Vecellio [11] identify patterns of inappropriate exception handling based on an analysis of a range of critical systems. Our work is similar to theirs in the approach of deriving practical advice based on the empirical evaluation of existing systems.

We are aware that the amount of support for handling and analysing exceptions present in a system is a design decision—one of the many that need to be made during software development. We observed that exception handling in our case study was reduced to the reporting of run-time errors. Assuming that it is a common practice in other systems, a question remains: is it bad? Such questions can be satisfactorily answered only by taking into consideration the requirements and criticality of each system. While there is little hope for development of absolute metrics for the quality of exception handling, there is much to be learnt in the process.

References

[1] Eclipse SDK, version 3.1.1 (`http://www.eclipse.org`), retrieved 2005-12-17.

[2] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput., 1(1):11–33, 2004.

[3] J. Bloch. Effective Java Programming Language Guide. Addison-Wesley Professional, 1st edition, June 2001.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. S. and. Pattern-Oriented Software Architecture: A System of Patterns, volume 1. John Wiley & Sons, 1st edition, August 1996.

[5] F. Cristian. Exception handling and tolerance of software faults. In M. R. Lyu, editor, Software Fault Tolerance, pages 81–108. John Wiley & Sons Ltd., 1995.

[6] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of Java server applications. IEEE Trans. Softw. Eng., 31(4):292–311, 2005.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[8] J. B. Goodenough. Exception handling: issues and a proposed notation. Commun. ACM, 18(12):683–696, 1975.

[9] J. Hanson. Event-driven services in SOA. JavaWorld, `http://www.javaworld.com/javaworld/jw-01-2005/jw-0131-soa.html`, January 2005.

[10] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service. `http://java.sun.com/products/jms/docs.html`, April 2002.

[11] C. Howell and G. Vecellio. Experiences with error handling in critical systems. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, Advances in exception handling techniques, volume 2022 of LNCS, pages 181–188, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[12] J. D. Litke. A systematic approach for implementing fault tolerant software designs in ada. In TRI-Ada '90: Proceedings of the conference on TRI-ADA '90, pages 403–408, New York, NY, USA, 1990. ACM Press.

[13] A. Mikhailova and A. Romanovsky. Supporting evolution of interface exceptions. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, Advances in exception handling techniques, volume 2022 of LNCS, pages 94–110, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[14] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In M. Aksit and S. Matsuoka, editors, ECOOP '97 - Object-Oriented Programming. Proceedings of the 11th European Conference, volume 1241 of LNCS, pages 85–103. Springer-Verlag New York, Inc., 1997.

[15] P. Niblett and S. Graham. Events and service-oriented architecture: The OASIS web services notification specifications. IBM Systems Journal, 44(4):869–886, 2005.

[16] D. L. Parnas. Information distribution aspects of design methodology. In IFIP Congress (1), pages 339–344, 1971.

[17] D. L. Parnas. Designing software for ease of extension and contraction. In ICSE '78: Proceedings of the 3rd international conference on Software engineering, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[18] D. L. Parnas and H. Würges. Response to undesired events in software systems. In ICSE '76: Proceedings of the 2nd international conference on software engineering, pages 437–446, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[19] J. Ploski and W. Hasselbring. The callback problem in exception handling. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, Developing Systems that Handle Exceptions. Proceedings of ECOOP'05 Workshop on Exception Handling in Object-Oriented Systems, pages 39–62. Department of Computer Science, LIRMM, University of Montpellier II, France, July 2005.

[20] M. P. Robillard and G. C. Murphy. Designing robust Java programs with exceptions. In SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on foundations of software engineering, pages 2–10, New York, NY, USA, 2000. ACM Press.

[21] A. Romanovsky and J. Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, Advances in exception handling techniques, volume 2022 of LNCS, pages 147–164, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[22] B. Stroustrup. Exception safety: concepts and techniques. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, Advances in exception handling techniques, volume 2022 of LNCS, pages 60–76, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[23] M. D. Welsh. An Architecture for Highly Concurrent,Well-Conditioned Internet Services. PhD thesis, University of California, Berkeley, CA, USA, Fall 2002.