The Callback Problem in Exception Handling

Jan Ploski[1] and Wilhelm Hasselbring[2]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany,
Jan.Ploski@offis.de,
[2] Carl von Ossietzky University of Oldenburg, Software Engineering Group, 26111 Oldenburg, Germany,
Hasselbring@informatik.uni-oldenburg.de

Abstract. Procedures invoked across program modules to notify about event occurences rather than to request concrete services are termed *callbacks*. They are frequently used to decouple program modules. However, their use complicates exception handling, turning default exception propagation with the established termination model unsatisfactory.
Existing advice on handling exceptions in callbacks either requires implementors to provide a no-throw guarantee or to propagate a generic exception from a callback to its invoker. We demonstrate that neither approach supports adequate exception handling in practice.
To aid programmers, we propose a tactic which mimics default propagation in the callback-less context: If possible, handle an exception locally; otherwise, propagate it to the invoked callback's *clients*, that is, modules affected by the callback's failure. We suggest early explicit specification as a means to locate such modules.

1 Introduction

The Observer design pattern [1] decouples two classes through a notification mechanism based on a specified interface known to both classes. An instance in the *subject* role notifies one or more instances in the *observer* role by sending them a message at each occurence of an agreed-upon event, typically a state change in the subject itself. The subject does not need to know the concrete type of each notified observer.[3] Also, it does not need to know the behaviour implied by the notification. Observers register with the subject prior to notifications and may unregister later on.

Instances of the classic Observer pattern exist in almost every GUI framework, often to implement the more specific Model-View-Controller pattern. More generally, one can employ callback mechanisms to facilitate communication between independent objects by connecting them through a bus which can support synchronous or asynchronous messaging. This solution can be found in various message brokers, for example in implementations of the JMS specification [2].

[3] We refer to classes "knowing" things in the sense of information being available to their programmer at design time.

The actual terms used to describe the overall concept vary considerably: callbacks, observers, listeners, slots, subscribers, views.

The relationship among communicating objects as defined by the Observer pattern gives rise to the problem of dealing with exceptions which might occur during the notification of an observer. Specifically, default exception propagation supported by imperative programming languages is inappropriate in such context, while it suffices in the simpler case of a direct invocation dependency:

In a traditional synchronous invocation scenario, it is useful to propagate failure information in form of an exception object back from an invoked method to its invoker to support adequate exception handling. This is true because the invoker is fully aware of the invoked method's contribution to satisfying the invoker's postcondition. The programmer may therefore either find an alternative way of satisfying the postcondition or broaden the recovery context by restoring local invariants and reporting failure farther up the call chain.

2 Exception Propagation from Callbacks

In contrast with the above scenario, exception propagation from callbacks is challenging. In this section we explain the problem and present two basic approaches which fail to provide a solution to the general case. We illustrate their drawbacks on examples and then briefly discuss practical consequences of their failure.

2.1 Callbacks and Information Hiding

Callbacks enforce information hiding. A callback's invoker is, by design, aware of neither the concrete implementation nor the abstract goals accomplished by the executed notification. More formally, the complete postcondition of a notification is not known at development time of the notifier.

We assume for simplicity that the precondition is empty. In reality, the notifier may be obliged to satisfy certain preconditions, such as executing the notification on a particular thread. Such preconditions can be easily specified as part of the notification interface.

Even if the information about the nature of work performed by the notified object is available, it should be treated as secret for the sake of reusability and independent maintainability of the invoker implementation.

2.2 Basic Approaches

The above considerations motivate the basic approaches to exception propagation from callbacks presented next: The notified objects must either provide a no-throw guarantee [3], or they may only supply limited information about their failure by throwing some generic exception.

Providing a No-throw Guarantee Imposing a no-throw restriction on each notified object is attractive from the point of view of a notifier's implementor, 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. Furthermore, it requires that the constraints on the behaviour of the notified object allow appropriate exception handling. While the first requirement is self-evident, the latter needs further elaboration.

As an example, consider the notification mechanism typical in event-driven programming utilizing GUI frameworks. An application programmer provides concrete implementations of abstract event handler methods specified by a GUI framework, invoked to notify about key strokes and mouse movements, and to request repainting of widgets. 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. These constraints make it impossible to report an exception to the user through the GUI or solicit user input directly in an event handler.

To work around the problem while still maintaining the no-throw guarantee, one must resort to asynchronous exception handling, that is, delegate the processing of exceptions to another thread and postpone it until the remaining registered event handlers have seen the current event.

Propagation to the Invoker If the requirement of a no-throw guarantee turns out difficult to satisfy in practice, one might hope that the second approach mentioned, propagating a generic exception to the notifier, has better prospects. Alas, this is not true. To illustrate why, we depart from the domain of GUI programming and consider a scenario in which components react to a database update triggered by another component.

In this scenario, the updating component manages its own data stored in a database table which is also shared with other components. The components are unaware of each other and are coordinated by an implementation framework. After an update of the shared table, each component is notified by the framework to execute some component-specific action, during 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 glance, the updating component is, by design, unable to handle any exception originating from another (notified) component. In fact, the occurence of such an exception might not at all be related to the update event, provided that the updater only uses the shared table to manage its own private data. Obviously, exception propagation to the notifier (the framework) and consequently to the event originator (the updater) is a design flaw.

While the simplified scenario described above is constructed for the sake of our discussion, it is inspired by a similar problem which occurs in the Eclipse Workbench [4] when a plug-in updates a project description.

2.3 Practical Considerations

At this point, we would like to reflect on how shortcomings in the described approaches affect the quality of exception handling in software. While implementation of GUI event handlers does not exceed the level of training normally expected from a client-side application programmer, asynchronous exception handling and related multi-threading concerns might well do. However, we are not aware of any GUI framework with explicit support for handling exceptions asynchronously. This task is left as a challenge for GUI programmers.

Unfortunately, the transition from the expected (easy) to the actual (difficult) nature of event-driven GUI programming caused by exceptions rapidly leads to inadequate exception handling in real code. Rather than expend a considerable amount of effort, it is more likely that implementors will skip the issue of exceptions altogether. A quick-and-dirty solution to the problem involves logging exceptions, optionally followed by propagation to the GUI framework's default handling mechanism. The latter can obviously do nothing more than display a cryptic error message or terminate the program. Such shortcuts are often taken, because leaving out exception handling seems natural when prototyping GUIs, and its importance continues to be neglected while a prototype evolves into a final product.

3 Proposed Solution

We have described two unsatisfactory approaches to exception handling in callbacks. In this section we present a more sophisticated approach.

Our solution is based on the already mentioned observation that exception propagation in the sequential call chain without callbacks proves sufficient and effective. In short, exceptions should be propagated towards computational contexts which fulfil at least one of the following requirements:

1. The context is (negatively) affected by the exception's occurence ("do not withhold bad news").
2. The context is capable of compensating for the negative consequences of the exception ("do not carry your problems to a place where no help is available").
3. The context is capable of removing the exception's cause to prevent its future occurences ("treat the disease, not the symptoms").

Note that with these rules we neither specifically prescribe nor forbid propagation of an exception to the immediate invoker of a method, as the previous approaches do. After introductory definitions we explain the above recommendations in greater detail.

3.1 Definitions

We adapt the terminology defined by Cristian [5]. Sequential programs (e.g., object methods) are invoked to cause transitions from an initial storage state

to an intended final storage state. A *storage state* is a mapping from object identifiers to object values; these values may model states of real-world entities. A *standard specification*, or *postcondition*, of a program is a predicate over the required initial and intended final storage states. Storage states in the domain of a postcondition form the corresponding *precondition*.

Exceptions are run-time events that occur when a program's execution diverges from its standard specification, regardless of whether the divergence is detected or not. The divergence occurs as soon as a storage state is entered such that the continued execution of the program cannot result in fulfilment of the standard specification.

In another sense, exceptions are named entities which may be used by a software developer to signal detection of divergences described above. Exceptions should be raised to indicate the impossibility of providing a program's specified standard service, in other words, to report a violation of its standard specification, or the program's *failure*. *Exceptional specifications* may be written to describe the intended final storage state reached by a program upon occurence of a named exception.

We would also like to stress that exceptions are *undesired* events. Their signaling should not be used for implementing functional requirements of software, but rather for addressing unavoidable threats that endanger the correctness of a specific implementation. This definition supplements the former one in that it discourages some uses of exception handling mechanisms (cf. [6], [7]).

3.2 Impact of Exceptions

As background information for our first rule of propagating exceptions from callbacks towards affected computational contexts, we now consider the possible consequences of such exceptions.

A callback's standard specification has to be understated at invoker's design time, such that concrete callback implementations may become proper behavioural subtypes [8]. The standard postcondition of a callback, as specified in the callback interface, may be simply *true*, or it may reflect restrictions imposed on its behaviour. In any case, a concrete callback may (and will) only strenghten the standard postcondition.

If an exception occurs during a callback's execution, its standard specification is violated. Thus, the entered final storage state differs from the intended one. Additionally, the entered state may be internally inconsistent, even if exception safety is guaranteed at the level of individual objects. The global consistency requirements for a system can be specified as a predicate over allowable storage states at well-defined points of execution. The predicate can be also viewed as a conjunction of *system invariants*, of which *object invariants* are a special case restricted to specifying relationships between a single object's representation variables. The traditional notion of exception safety is restricted to preserving object invariants.

We next provide a realistic example of a case in which an exception leads to an unintended yet consistent storage state, as well as one in which an inconsistency occurs.

Exception With Preserved System Invariants A preferences dialog in a desktop application lets the user modify some preferences and finally click the "Apply" button to persist them. The "Apply" button then becomes disabled to indicate no outstanding changes. If an exception occurs in the "Apply" event handler, the system will not advance to the next correct state, even though invariants of all objects are preserved (i.e., the "Apply" button remains enabled). The final storage state, even though internally consistent, is unacceptable for the user, who expects that preferences have been saved.

Exception With Broken System Invariants In an application using the MVC pattern, different state aspects of a model object are visualised by separate views. After an exception occurs during state change notification, a view might no longer reflect the model state and no longer show content synchronised with other, successfully updated views. Two system invariants are broken in this case: one over the model and view state, and another one over the state of all views.

3.3 Determining Affected Code

We have shown how an exception occurence in a callback leads to entering an unintended storage state as its immediate consequence. In an object-oriented context we are interested in determining methods with preconditions that become impossible to satisfy through normal execution continued from the entered exceptional storage state. Knowing these methods, one can notify their enclosing objects before their next regular invocation. The rationale is that the objects may then switch over to an alternative implementation, or at least prepare to fail gracefully. This is analogous to propagating an exception to the invoker of a service method.

More formally, we need to find elements of two object sets:

1. M, the set of objects modified during the callback's invocation
2. D, the set of callback's *clients*, defined as objects dependent on the updated state of elements from M

For a failed method invocation in a sequential program without callbacks, M typically consists of objects known to the invoker and D contains the invoker itself or objects controlled by the invoker. Members of these sets can be determined by following the next execution steps that would be taken after the failed invocation. Accordingly, it makes sense to refer to the invoker as the failed method's *client* [6]. However, when callbacks are employed, the method's invoker steps out of its usual client role.

To avoid the algorithmically difficult computation of M and D from the source code alone, we suggest that both sets are specified by the programmer

when the callback's purpose becomes known. The specification should be written regardless of whether exceptions are expected to occur during callback invocations or not. In programming languages which support specification of postconditions, M does not require to be explicitly enumerated, because it follows from a postcondition. However, D still has to be specified explicitly. With this preparation, when later implementations of the invoked methods signal exceptions, the exceptions can be propagated to callback's clients as an early warning that the state on which they depend has become inconsistent or outdated.

To illustrate why the up-front specification of M is worthwhile, we describe an informal, labour-intensive procedure for reconstructing it from source code:

Let M be an initially empty set of affected objects. For every statement s_i of the callback implementation:

1. Let C be the set of method invocations from s_i.
2. For every method invocation c_j from C:
    1. Let o_j be the object on which c_j is invoked and P the set of objects passed in as formal parameters.
    2. If c_j does neither modify the state of o_j nor of any object from P, proceed to c_{j+1}, otherwise:
    3. Add the object(s) with modified state to M.

Finding out whether a method modifies an object's state, lacking specification, requires analysis of all transitively invoked methods. This task becomes non-trivial and error-prone when the invoked methods are themselves callbacks, configured and registered at some other point of execution. On the other hand, if an accurate specification is available for a (callback) method, the analysis of its implementation is not required, speeding up the process.

Updated objects from M could theoretically be examined for their participation in system invariants affected by the callback's invocation in order to construct the callback's clients set D. However, we are unaware of practical approaches for reconstructing system invariants based on static analysis of source code alone (for a brief description of a dynamic approach, see section 4.1). We must therefore assume that informal understanding of the intended role of each element from M and the relationships between them and other objects is utilised in this process. However, such understanding is only likely to exist at the time when the callback is originally added to the system.

These observations highlight the need for early specification of callback effects. However, two related problems remain as topics for further research:

1. The proposed declarative specifications concern the callback's clients, which could in principle remain hidden if we disregarded exception handling. The trade-off for achieved explicitness is a decrease in modularity.
2. The proposed declarative specifications may be difficult to maintain as programs evolve.

3.4 Exception Handlers

According to our second proposed rule ("do not carry your problems to a place where no help is available"), not every exception originating from a callback should be directly propagated to its clients defined in the previous section. Before such propagation occurs, an attempt should be undertaken to find exception handlers that can reduce or even completely eliminate the signaled exception's consequences.

In practice every exception-throwing callback should be wrapped into one that catches all exceptions and decides about their propagation. The wrapper object communicates exception occurences to handlers which can be either specified statically or registered at run-time. If required, notification of handlers may be performed asynchronously, like in the GUI example from section 2.2.

There are two basic courses of action for handlers: restoration of system invariants and enforcing the original callback's postcondition (cf. section 3.2).

The effort expended on the restoration of system invariants is related to the amount of damage present when the exception is signaled. If available, a description of the damage should be encapsulated in the exception itself. Both roll-back and roll-forward recovery can be performed to restore the consistent storage state. It is even more preferable for the callback to perform the necessary clean-up actions *before* signaling the exception, however, it may not be always possible nor sufficient.

The original postcondition can only be enforced by retrying the callback's invocation if the exception's cause was removed by a handler. Otherwise, a less satisfactory exceptional postcondition may still be enforced. The exceptional postcondition may be functionally inferior to the original one, which calls for a redesign by weakening of system invariants, or it may offer lower quality of service (e.g., use a more robust, but slower algorithm).

Depending on the handlers' achievements, (some of) the callback's clients may still need to be notified about the exception's occurence. The communicated information should include relevant countermeasures undertaken by the handlers.

Deciding whether and which clients should be informed, and to what extent, depends on a specific application. For example, if a maskable disk failure occurs in a RAID array, clients which depend on the immediate success of the IO operation do not need to be notified about the exception, however, it is prudent to notify a monitoring client so that the faulty disk is replaced (cf. [9]).

3.5 Removing Causes of Exceptions

Exception handlers can provide two kinds of service: temporary workarounds and permanent fixes.

The first category implies a degradation of service level, which may or may not be noticeable to the clients, like in the above RAID array example.

The "permanent fix" category is more interesting. It requires both determining and removing the fault which causes an exception—two tasks which may be

sometimes economically infeasible. In simple cases, easy configuration changes may be all that is required. In complex cases, significant investments in hardware redundancy and improved system design may be necessary. Execution of permanent fix handlers may require human intervention, and a system may enter prolonged exceptional storage states while the permanent fix is being prepared. To reason about exception handling in this context, explicit specification of persistent exceptions [10] becomes important. Such exceptions can be caused by faults in system configuration or data that exist independently of any executed program. They do not fit well into the traditional programming-language-centric view of exception handling.

Unfortunately, we cannot at present offer sound advice on techniques for removing faults. One step in the right direction would be constructing their taxonomy based on examination of real-world systems. After faults are classified into well-defined categories aligned with strategies applicable to their removal, design patterns and components might be conceived to not just handle resulting exceptions, but often to avoid the need for exception handling altogether. It is worth pointing out that even if this succeeds largely, exception handling mechanisms will still remain useful for addressing unpredictable, dormant faults.

4 Related Work

In this section, we briefly point out related work from the research community and articles published by software developers.

4.1 Academic Research

The concept of uniform programmatic exception handling was popularised by Goodenough [7]. Goodenough informally describes a syntax and a proposed notation for both the termination and the resumption model of exception handling. Our suggested way of handling exceptions in callbacks relies only on the more popular termination semantics, but it replaces the default exception propagation when necessary.

Cristian [5] provides a formal semantics for the termination model. He focuses on an examplary direct invocation scenario and does not specifically consider callbacks, though he briefly discusses exception propagation from an indirectly invoked method.

Meyer [6] defines guidelines for using exceptions according to the termination model under the label “organised panic” in his textbook on object-orientation. Like Cristian, he does not address propagation of exceptions from callbacks, nor does he elaborate on how (and when) contracts should be specified for them in order to successfully apply his proposed basic rules of exception handling.

Our described problem can be viewed as a specific instance of the general conflict between exception handling and object-oriented abstraction discussed by Miller and Tripathi [9]. However, their paper does not provide guidance with respect to balancing information hiding for modularity and exposure for exception handling.

Buhr [11] points out the problems resulting from premature specification of exception lists for routines invoked through pointers (i.e., callbacks) and polymorphic methods. His observations support our view that specification of exceptions is not practical at the level of callback interfaces and must be delayed until implementation.

Awareness of system invariants aids determining code affected by a callback's failure. Ernst [12] proposes a dynamic approach for reconstructing invariants from executed code and provides a related tool, Daikon. Only relatively low-level invariants are recognised automatically, however, they may provide valuable hints for the programmer about more abstract invariants.

Barnett and Naumann [13] propose a methodology for specifying invariants across object ownership boundaries, which could be potentially adopted as a formalism for expressing specifications suggested in our paper.

Borgida [10] proposes a language for describing and handling persistent exceptions in object-oriented databases and points out that a relationship exists between run-time exceptions and flaws in data maintained by information systems. Our argument that exception handling in a broad sense should ultimately aim at removing the causes of exceptions is based on similar observations.

4.2 Software Industry

The Observer pattern is one of the very basic patterns commonly taught to software engineering students and widely applied in practice. Accordingly, one would assume that the problems described above were already addressed by researchers and practitioners. A survey of articles on "best practices" in exception handling reveals the opposite.

Doshi [14] does not address the issue at all, while he provides the following advice: "When a method from an API throws a checked exception, it is trying to tell you that you should take some counter action. If the checked exception does not make sense to you, do not hesitate to convert it into an unchecked exception and throw it again". This is wishful thinking, as should be obvious from our previous example.

Shenoy [15] follows the EJB specification [16] by distinguishing between application and system exceptions. The latter are handled by the EJB container in a default way (logging, aborting the current transaction). Shenoy suggests that application exceptions which have been propagated to unaware components should be caught and converted to system exceptions. He does not address the inappropriateness of propagation of such exceptions, but rather treats them as a "real-world complexity".

5 Conclusions and Future Work

Important practical problems remain that make the proper application of established exception handling concepts difficult for programmers using today's object-oriented programming languages.

We have presented one such problem—exception handling in callbacks—and provided a set of relevant examples based on the Observer design pattern. We have also shown that existing simple guidelines available to programmers for solving this problem are either overly restrictive or leave the most challenging decisions open. Finally, we have described a more sophisticated alternative approach based on the early specification of callback effects combined with the well-established concept of exception propagation.

The remaining issues to be addressed by future research include:

1. Syntactic and semantic description of our suggested additional specifications in the context of a chosen object-oriented programming language: An approach similar to [13] could be taken for specifying system invariants which cross object boundaries. We also plan to evaluate the support offered by existing tools for static and dynamic control and data flow analysis [17].
2. Empirical evaluation of the proposed additional specifications' usefulness: Metrics have to be defined that measure the amount of work expended on exception handling and its quality in software products. Such metrics should build upon prior research on misuse of exception handling.

In a broader view, we feel that exception handling needs tools and techniques beyond simple syntactic checks such as determining missing or empty handlers. In addition, discovery of failure modes, intent, and stakeholders of a software application should be made possible. Offered solutions should fit into a bigger framework for building dependable software by recognising the role of other quality-of-service attributes and multiple programming languages.

The problem described in this paper is not at all restricted to small object-oriented programs employing specific design patterns, as could be subsumed from our simple examples and definitions. It prominently figures in component-based system development (CBSD), where it is a basic assumption that independently developed software artefacts are combined and configured at a later time in flexible ways to obtain desired functionality.

Deployed components may invoke each other either as services or to deliver notifications. All invocation relationships cannot be determined until integration time. For these reasons, approaches which position exception handling in the programming domain or request that exceptions are explicitly specified by developers as part of component interfaces [18] appear too limited.

Instead, we feel that the specification of exceptions and implementation of handlers need to be delayed until all required information about possible exception sources and propagation paths becomes available. It could be shortly before deployment or (re-)configuration of a component-based system. Developing methods and tools for realising this vision represents our long-term research goal.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts (1995)
2. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service. `http://java.sun.com/products/jms/docs.html` (April 2002)
3. Stroustrup, B.: Exception safety: concepts and techniques. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in exception handling techniques. Volume 2022 of LNCS., New York, NY, USA, Springer-Verlag New York, Inc. (2001) 60–76
4. Arthorne, J.: Project builders and natures. `http://www.eclipse.org/articles/Article-Builders/builders.html` (January 2003)
5. Cristian, F.: Exception handling. Technical Report RJ5724 (57703), IBM Almaden Research Center (1987)
6. Meyer, B.: Object-Oriented Software Construction. second edn. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)
7. Goodenough, J.B.: Exception handling: issues and a proposed notation. Commun. ACM **18** (1975) 683–696
8. Liskov, B.H., Wing, J.M.: Behavioural subtyping using invariants and constraints. In: Formal methods for distributed processing: a survey of object-oriented approaches. Cambridge University Press, Cambridge, UK (2001) 254–280
9. Miller, R., Tripathi, A.: Issues with exception handling in object-oriented systems. In Aksit, M., Matsuoka, S., eds.: ECOOP '97 - Object-Oriented Programming. Proceedings of the 11th European Conference. Volume 1241 of LNCS., Springer-Verlag New York, Inc. (1997) 85–103
10. Borgida, A.: Language features for flexible handling of exceptions in information systems. ACM Trans. Database Syst. **10** (1985) 565–603
11. Buhr, P.A., Mok, W.Y.R.: Advanced exception handling mechanisms. IEEE Trans. Softw. Eng. **26** (2000) 820–836
12. Ernst, M.D.: Dynamically Discovering Likely Program Invariants. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington (2000)
13. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: MPC '04: Proceedings of the Conference on Mathematics of Program Construction. (2004)
14. Doshi, G.: Best practices for exception handling. `http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html` (2003)
15. Shenoy, S.: Best practices in EJB exception handling. `http://www-128.ibm.com/developerworks/linux/library/j-ejbexcept.html` (May 2002)
16. DeMichiel, L.G., et al.: Enterprise JavaBeans specification, version 2.1. `http://java.sun.com/products/ejb/docs.html` (November 2003)
17. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, USA (1998)
18. Romanovsky, A.: Exception handling in component-based system development. In: COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference, Los Alamitos, CA, USA, IEEE Computer Society Press (2001) 580–588