**Exception Handling Strategies**

In my work as consultant I often get to look into a clients existing applications. The exception handling I see in these applications is anywhere from almost random, to almost useful. There is often no formal, coherent exception handling strategy. Or, there is a strategy, but it is insufficient, meaning it does not contain all the rules, information and precautions necessary to fully handle all exceptions. That's why I decided to write this trail on exception handling strategies.

You may have noticed my trail on [Java Exception Handling](http://tutorials.jenkov.com/java-exception-handling/index.html), which contains a list of text explaining a set of basic exception handling techniques. That trail however, does not cover how to put all these techniques into a coherent exception handling strategy.

In this trail I will explore how to put all the individual exception handling techniques into a single, coherent exception handling strategy. Don't treat this strategy as the one-and-only strategy possible. Treat it as a template for your own exception handling strategy. A suggestion, in other words. Make the adjustments you want or need in your concrete situation.

The exception handling strategy presented in this trail is language independent. You can implement this strategy in pretty much any language which has a try-catch-finally mechanism, including Java, Scala, C# etc.

Being an exploration of the subject, the suggestions inhere may be able to be improved. If you have any suggestions or comments, I will be happy to hear from you. I may also update this trail over time myself, as I get more knowlegde on the topic.

## Table of Contents

Here is a list of the topics covered in this trail. This list (menu) is also present at the top right of every page in the trail.

|  |  |  |
| --- | --- | --- |
| **Exception Handling Strategies** | | |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Introduction](http://tutorials.jenkov.com/exception-handling-strategies/index.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Overview](http://tutorials.jenkov.com/exception-handling-strategies/overview.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Requirements](http://tutorials.jenkov.com/exception-handling-strategies/requirements.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Application Survival](http://tutorials.jenkov.com/exception-handling-strategies/application-survival.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Notifying Relevant Parties](http://tutorials.jenkov.com/exception-handling-strategies/notifying-relevant-parties.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Error Diagnostics](http://tutorials.jenkov.com/exception-handling-strategies/error-diagnostics.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Separation of Abstr. Layers](http://tutorials.jenkov.com/exception-handling-strategies/separation-of-abstraction-layers.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Maintainable Code](http://tutorials.jenkov.com/exception-handling-strategies/maintainable-code.html) |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Error Location and Context](http://tutorials.jenkov.com/exception-handling-strategies/error-location-and-context.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Error Causes, Types and Reactions](http://tutorials.jenkov.com/exception-handling-strategies/error-causes-types-reactions.html) |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Strategy Elements](http://tutorials.jenkov.com/exception-handling-strategies/strategy-elements.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Error Detection](http://tutorials.jenkov.com/exception-handling-strategies/error-detection.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Error Info Gathering](http://tutorials.jenkov.com/exception-handling-strategies/error-information-gathering.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Throwing Exceptions](http://tutorials.jenkov.com/exception-handling-strategies/throwing-exceptions.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Propagating Exceptions](http://tutorials.jenkov.com/exception-handling-strategies/propagating-exceptions.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Catching Exceptions](http://tutorials.jenkov.com/exception-handling-strategies/catching-exceptions.html) |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [A Strategy Template](http://tutorials.jenkov.com/exception-handling-strategies/strategy-template.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [AppException Template](http://tutorials.jenkov.com/exception-handling-strategies/template-exception.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Throwing the AppException](http://tutorials.jenkov.com/exception-handling-strategies/template-throwing.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Propagating the AppException](http://tutorials.jenkov.com/exception-handling-strategies/template-propagating.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [The ErrorInfo List](http://tutorials.jenkov.com/exception-handling-strategies/template-error-info-list.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [Catching the AppException](http://tutorials.jenkov.com/exception-handling-strategies/template-catching.html) |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Execution Context](http://tutorials.jenkov.com/exception-handling-strategies/execution-context.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Avoid Exception Hierarchies](http://tutorials.jenkov.com/exception-handling-strategies/avoid-exception-hierarchies.html) |

# Exception Handling Strategy - Overview

An exception handling strategy consists of all actions and conventions necessary to properly handle exceptions in your application.

The following diagram shows an overview of the various parts of an exception handling strategy, from the error is detected, all the way up to where the error is handled.

|  |
| --- |
| An exception handling strategy overview. |
| **An exception handling strategy overview.** |

At the bottom of the strategy is errror detection. You detect an error either youself, or you catch a third party exception (AKA alien exception).

At the second level you try local exception handling if possible. This is where you e.g. send your request to a backup server or wait X seconds and try again, etc. You do what you can, if anything, to handle the error and continue whatever operation your application was currently carrying out.

At the third level, when local error handling did not work, you gather all the information necessary to diagnose, reproduce and report the error.

At the fourth level you convert the caught exception, or the error detected, into an application specific exception, and throw it. You add any information necessary to properly handle the exception.

At the fifth level you propagate the application specific exception up the call stack, closing any opened resources on the way up (like files, network connections, freeing allocated buffers etc.). Additionally, you add any contextual information which may be useful in determining the cause and severity of the error.

At the sixth level you catch and react to the exception. At this point local exception handling is no longer possible, so most often the only thing your application can do is to notify users, and non-users of the exception. If the exception is really, really bad your application may even shut down gracefully.

# Exception Handling Requirements

The purpose of an effective exception handling strategy can be grouped into a set of primary and secondary purposes, or �requirements� as we like to call them in software development. The requirements state what should happen if an exception (error) occurs in an application.

The primary requirements are the most important to meet with your exception handling strategy. The secondary requirements are not nearly as important, but it is still a good idea to take these into consideration, if you have the time and opportunity in your project.

## Primary Requirements

The primary requirements of an effective exception handling strategy are:

1. The application survives the exception and can continue to process events, requests, files etc. Or, the application shuts down gracefully.
2. The relevant parties are notified. These parties include:
   * End users
   * System operators and administrators
   * Developers
   * Customers / owners of the application
3. The exception / error can be diagnosed and reproduced, so it can be corrected.

## Secondary Requirements

The secondary goals of an effective exception handling strategy are:

1. Encapsulation / Separation of abstraction layers.
2. More readable, maintainable code.

Both primary and secondary requirements are explained in more detail in the following texts.

## Application Survival

When an exception is thrown in an application, the application should be able to survive this exception, and continue to process events, requests, files etc. Application survival is divided into two sub-requirements:

1. Catching and handling the exception, so the application continues running.
2. Opened resources are closed correctly.

Both of these requirements are described in more detail below:

## Catching and Handling the Exception

When an exception occurs, the application must catch the exception somewhere, handle it, and continues as if the exception had never occurred (if possible).

If the error causing the exception is so severe that the application cannot continue executing, e.g. a required configuration file is missing, the application must shut down gracefully.

## Closing Resources Correctly

Open resources which require explicit closing or freeing after use, should be closed or freed correctly, even if an exception is thrown while using them.

Typical examples of such resources are thread locks, streams, network connections (sockets / channels), database connections, JMS connections etc. In applications using Java NIO some memory buffers must be freed explicitly. In SWT applications SWT GUI components need explicit disposing after use.

You should of course not close each and every resource in the application when an exception is thrown. Only resources opened to serve the request that fails should be closed when an exception occurs. Connections in connection pools and other long term cached resources should not be affected by the exception. Of course resources obtained from pools should be returned correctly to the pool even if an exception occurs, but the pool itself should not be closed.

Only if the application is shutting down completely should all open resources be closed.

# Notifying Relevant Parties

When an exception occurs in your application the interested parties should be notified about it. The most typical interested parties are:

* The application end user, if any.
* The application operators.
* The application administrators.
* The application developers.
* The application owner, if external from your organization.

Each of these parties should receive different information. Additionally each party is also typically notified using different notification mechanisms. Here is a diagram showing an overview of notification:

|  |
| --- |
| Notifying relevant parties when an exception occurs |
| **Notifying relevant parties when an exception occurs.** |

I'll explore each of these categories of relevant parties in the sections below.

## End Users

An end user should be notified that the requested operation cannot be completed, but the user should probably not see a stack trace, or other internal details about the exception. Such information can actually contain information that can be used to hack into your application. At most the end user should see an error description and perhaps a unique error code, in case the user wants further details about the error from the organization operating the application.

Application end users are typically notified about exceptions via the application user interface, whether that user interface is a command line, a desktop GUI or a browser GUI.

## Computerized Clients of Services

Not all applications have human end users. Services that are intended for computer-to-computer communication typically do not have a user interface. In such cases the calling computer (the client) should be treated as the user. In other words, in case an exception occurs in the service, the client should be notified of the error. Yet again, it may not be a good idea to present the client with stack traces or other internal details except a unique error code and an error message.

## Application Operators

The application operators are the people who monitor the application to make sure it is functioning correctly. In case of errors that the application cannot handle itself, the operators are typically also responsible for getting the application back in operational state.

By �responsible� I don't necessarily mean that they have to do it themselves. They just have to make sure that the right people are mobilized depending on the nature of the error.

Application operators should also be notified of errors in the application. Of course operators should only be notified of errors that need their attention. An exception thrown because the user types in something invalid are often not necessary to forward to the application operators.

Application operators are typically notified via the application log.

## Application Administrators

The application administrators are the people who administer the computer the application is installed on. Hence, the application administrators are typically responsible for installing the application on these computers too. Application administrators are also responsible for starting and stopping applications.

Application administrators should be notified if an error in an application is caused by something related to the computer or application administration. For instance, a configuration file could be missing or contain invalid settings. The administrators should then check if the configuration file is present, and that it contains valid settings. Or, an application could be reporting that an external system needed by the application does not respond. The administrators should then check the state of this external system.

Application administrators are typically notified of errors via the application log, or via the application operators.

In some organizations the application operator and application administrator responsibilities are carried out by the same people. In other organizations these responsibilities are divided between two different groups of people.

## Application Developers

The application developers are the software developers who have developed or are maintaining the application.

Application developers should be notified of errors that may be caused by bugs in the software. Actually, the developers are often notified of any error that operators and administrators cannot resolve.

Application developers are typically notified via the operators who monitor the application log.

## Application Owners

If the application is being hosted for an external customer, that customer may also want to be notified of errors. Exactly what errors they are interested in depends on the application and the customer, but if an error causes their application to be unavailable you can be sure that they are very interested in knowing that.

# Error Diagnostics and Reproduction

To be able to correct an error in an application you must first be able to diagnose it. Diagnosing an error means determining the precise cause of the error. To diagnose an error from the error message alone is not always easy. Therefore it is often necessary to be able to reproduce the error, before you can determine its cause.

An effective exception handling strategy should take both diagnostics and reproduction into consideration. I will elaborate a bit on both here.

## Diagnostics

To determine the cause of an error you need to know:

* The exact location in the code where the exception occurred.
* The context in which that location in the code was reached. Some locations in your code are reachable via different execution paths. For instance, a utility method may be called from many different places in your application. You may need to know where the utility method was called from, in order to determine the cause of the exception.
* A sensible description of the error including any relevant variable values, including parameters, internal state variables (members, global variables etc.) and possibly external system state, for instance data in a database.

## Reproduction

To be able to reproduce an error you need to know what external actions lead to the error. Sometimes the error is caused by a single action, and sometimes it is caused by a series of interdependent actions. For instance, your application may fail on the third of three related requests.

To reproduce the error you may have to go through all three requests, and to do that, you need information about each of these three requests.

It is not always easy to store information for multiple dependent steps. For instance, the two first steps (e.g. pages in a wizard-like flow in a web app), may succeed. Should you still log that, even if they succeeded? Then the third step fails. Now it would have been nice to have information about step 1 and 2, in order to reproduce the error, but perhaps you didn't log that information.

You cannot log every tiny little thing your application does. It would slow the application down a lot, and make the log files grow huge. Therefore, logging the full detail needed for reproduction is not always possible.

# Separation of Abstraction Layers

When you search the internet for advice on exception handling you will often see people say things like:

�The upper layers in an application stack should not know anything about the lower layers. Hence they should not know how to handle special exceptions thrown from these layers.�

What does this mean?

Imagine you are developing a persistence layer in your application and you decide to hide it behind a DAO layer. In other words, you don't want the layers of your application using the DAO layer to know what technology you have chosen underneath. Therefore, you don't want the layers above to know about any exceptions thrown from the DAO layer that are specific to the underlying persistence technology.

For instance, in JDBC, Java's database abstraction API, several of the methods may throw an SQLException. If your DAO's methods throw an SQLException, it reveals the underlying persistence technology, JDBC. If you were to change to an object oriented database or an XML file your DAO layer might all of a sudden throw exceptions related to these API's instead. This would break your code.

Instead, convert the exception thrown by the underlying persistence technology to a DAOException or anAppException. That way you can change the underlying persistence technology, without having to change the exceptions declared thrown in the DAO methods.

Seperation of abstraction layers is a secondary goal of an effective exception handling strategy in my opinion. It is definitely nice to have, but you can write an effective exception handling strategy, even if this goal isn't entirely met.

# More Readable and Maintainable Code

All those try-catch-finally clauses spread out in your code makes the code harder to read. An effective exception handling strategy can eliminate some of all this boilerplate code. This is done by only handling the exceptions in the place where you can actually do something about them, and by using [exception handling templates](http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html). I will get into more detail about these topics later in this tutorial.

While cleaner code is definitely nice, it is not nearly as important as keeping the application alive, and notifying the right people about errors. Skeptics may claim that with cleaner code it is easier to assure application survival, and this I cannot deny. Some of the techniques discussed in this tutorial will both make application survival easier, and clean up the code. Whether one is a byproduct of the other, or vice versa really isn't that important.

# Error Location and Context

The execution flow in an application typically looks like illustrated in the diagram below.

|  |
| --- |
| An example program flow. |
| **Program flow from some boundary (root) and downwards through the modules and components.** |

The diagram is a bit simplified, but I think you get the point.

The execution starts a some root point. A root point could be:

* The main() method of a standalone application (Java, C#, Scala etc.)
* A web service entry point.
* A Servlet, JSP, ASPX, PHP page etc.
* Something else...

Exactly what the root of execution is, is not that important. Some applications may also have multiple roots. What is important to note is that the execution flows from a root and downwards into the application code. Depending on what the application is doing, the execution will take different paths down through the code. Nothing surprising here.

At some point in the application an exception may get thrown. An exception will bubble up the call stack until caught and handled. Nothing new or interesting here either.

What is interesting to notice, is that Component 1.2 is called by both Module A and Module B. If an exception occurs in Component 1.2 then it may have to be handled differently depending on whether Module A or Module B called Component 1.2.

For instance, lets say that Component 1.2 is a file loading utility. Module A is then configuration file loader, and Module B is a user file loader, as illustrated below:

|  |
| --- |
| Config Loader and User File Loader both call the FileLoader component, but handles exceptions differently. |
| **Config Loader and User File Loader both call the FileLoader component, but handles exceptions differently.** |

If the application cannot load one of its configuration files, this may be a very serious error. The application operators (if any) should be notified of such an error.

If on the other hand, a user enters an invalid file name and the file loader (Component 1.2) cannot load it, this may not actually be such a serious error. Of course the user should be notified of the error, but it may not actually be necessary to notify the application operators. The user can just correct the file name, and retry the file load action.

As you can see, Component 1.2 (FileLoader) cannot by itself determine if failing to load the file is a serious error, or just a natural possibility due to what the application does.

## Error Location and Context Defined

Here I will define the two concepts error location and error context in a bit more detail:

* **Error Location**  
  The location in the code where an error occurs, and an exception is thrown.
* **Error Context**  
  The execution path leading to the error location.

Both error location and error context influence how you should handle an exception thrown from the error location.

This is a bit at odds with the common exception handling advice, that "the code closest to where the error occurred, knows best how to handle it". You have just seen, that this is not always the case.

# Error Causes, Types and Reactions

An application is often limited in what ways it can react to an error. The reason for this is, that an application is not very good at determining what actions are needed to correct an error.

For instance, if a configuration file is missing, you as a developer may know exactly what to do, but the application doesn't. The application only knows what you can program it to know, and this is often more limited than we initially think.

## Common Error Reactions

An application typically only has these possible reactions:

1. Abort action, and notify the user. Maybe log the error.
2. Retry the action at a later time.
3. Retry the action against a different service (e.g. backup database, or web service etc.)

Some applications may have further options that are specific to their internal design, or the architecture in which they are deployed. But the above options are the most common possible reactions to errors.

## Error Types

Which of the above error reactions should be applied depends on what type of error that occurred. There are typically three kinds of errors that can occur, each of which calls for a different reaction. These three kinds of errors are:

1. Client Errors
2. Service Errors
3. Internal Errors

The diagram below illustrates these error types, and where they occur.

|  |
| --- |
| Error Types. |
| **Illustration of the various error types.** |

## Client Errors

Client errors are errors caused by the clients wrong use of the application. For instance, a user types in the wrong file name, or types in characters where the input should have been a number. Or the client sends an invalid web service request. In other words, the client is not complying with the contract for interaction with the application.

A client error is normal. It is expected to happen from time to time. A client error is not a sign that your application has bugs. It is a sign that your application correctly rejected invalid data or usage.

### Client Error Reactions

The most common reaction to a client error is to:

1. Abort the request action.
2. Close any opened resources (connections, files, streams etc.)
3. Free any allocated resources (memory buffers etc.).
4. Notify the user of the error.
5. For debug or customer service purposes you could also log the error. For instance, your application might actually classify the error incorrectly as a client error. Or, the user may contact you to ask why the application rejected his request. In that case it may be nice to have the error logged.   
   There is often no reason to raise an alarm to the system operators, though.

## Service Errors

Service errors arise if an external service used by the application malfunctions. For instance, the database is down and is thus not responding.

A service error is not a sign that your application has bugs either. Your application is functioning correctly.

You should try to develop your application so that it can survive such temporary external service failures. For instance, if your application uses a connection pool to a database which fails, the application should be able to re-initialize the connection pool, once the database is working correctly again.

### Service Error Reactions

The most common reaction to a service error is to:

1. Either
   1. Abort the requested action.
   2. Retry at a later time, or at a different service instance (e. g. a backup server).
2. Close any opened resources (connections, files, streams etc.).
3. Free any allocated resources (memory buffers etc.).
4. Notify the user of the error.
5. Log the error and raise an alarm to the application operators, so they can investigate and correct / restart the malfunctioning service.

## Internal Errors

Internal errors are errors that are caused by your application. The cause for such errors could for instance be a bug in the application or invalid configuration. Generally, any error that cannot be determined to be a client error or service error, should be categorized as an internal error.

An internal error is the most severe of the three error types. If an internal error is detected the application is not functioning correctly, and the system will not correct itself automatically. It is necessary to call in the developers to investigate the error further, and maybe deliver a patch to the application.

### Internal Error Reactions

The most common reaction to an internal error is to:

1. Abort the requested action.
2. Close any opened resources (connections, files, streams etc.).
3. Free any allocated resources (memory buffers etc.).
4. Notify the user of the error.
5. Log the error and raise an alarm to the application operators.
6. Have developers investigate the error, and possibly deliver a bug fix.

## Temporary Internal Errors

An application may have temporary internal errors. For instance, if a web application receives too many requests during a peak usage time, the web server may temporarily be unable to respond. This will, however, fix itself once the load decreases.

Such temporary internal errors are an exception to the statement, that the application will not correct itself. Temporary errors may go away by themselves.

### Temporary Internal Error Reactions

The reaction to a temporary internal errors is similar to a service error. The application is working correctly from a functional perspective, but is temporarily unavailable.

## Error Classifications by the Application

When an error occurs in your application it should do its best to categorize the error as either a client error, service error, an internal error or a temporary internal error. This is not always easy, but that does not mean the application should not try. I'll try to describe how to do this here.

Any unit in your application may experience the types of errors described earlier. A unit here meaning �a piece of code�, e. g. a larger component, a class, an object, or a method / function.

This diagram shows three units working together:

|  |
| --- |
| Three arbitrary, collaborating units of an application. |
| **Three arbitrary, collaborating units of an application.** |

These units could be three objects, or three larger grained components, or even three separate systems communicating via remoting or web services or something similar. The granularity of the units is not really important. It is the roles of the units that matter.

It is the middle unit, the executing unit (the component), we are interested in. Actually, almost any unit in your application assumes the roles as both client, executing component, and service. Most units do no function in isolation. They are called by some other unit, and they often also call other units themselves. The role of any given unit just depends on what component you choose to zoom in on as the executing unit.

If a client component does not obey the contract of the executing component, this is a client error. For instance, if the client component calls the executing component with invalid input parameters or while the executing component is in an invalid state.

If the executing component calls a service component, and this service component throws an exception (or returns invalid data etc.), this is a service error.

Any other error which cannot be classified as a client or service error, is an internal error.

This may still seem a bit abstract, so let's look at a concrete example. The diagram below shows the call sequence of a simple web application that uses a database.

|  |
| --- |
| The call sequence of a simple web application that uses a database. |
| **The call sequence of a simple web application that uses a database.** |

The user types in some data in a form in the browser, and clicks the submit button. The form data is then sent to the web application, which passes it on to the relevant web action.

The web action (JSP, Servlet, Struts Action etc.) receives the request and validates the request parameters. In case of invalid request parameters, the rest of the requested action is aborted, and the user is notified. A client error is logged, if the error is logged at all.

If the request parameters are valid, the web action ends up calling the DAO to insert / update data in the database. The DAO validates the input parameters passed to it from the web action. In case of invalid input parameters, the DAO raises a client exception. From the DAO perspective the client has broken its contract - hence the client exception. When the exception is propagated up to the web action, it is up to the web action to determine what type it perceives the error as. This is known as error re-classification.

## Error Re-classification

As an exception is propagated up the call stack, each component propagating the exception may re-classify the exception's error type. It is not always easy to either classify or reclassify errors, but here are two examples describing situations where it is possible.

### Example 1

For instance, imagine that the web action passed a user id to the DAO, which does not exist in the database. The DAO checks if the user exists, before carrying out the action. Since the user id is not found in the database, the DAO raises a client exception. The client of the DAO has broken the contract of the DAO.

When presented with an "invalid user id" exception, the web action should know whether the user id came from inside the application (e.g. the session object), or from the client (e.g. a cookie in the browser, or a JavaScript variable in an AJAX application). Thus, the web action may change the error type set the by the DAO, to e.g. internal error.

If, on the other hand, the user id was taken from the request parameters (the query string), the web action may still classify the error as a client error. The request string might have been manipulated by the user.

### Example 2

Imagine again a call from the Web Action to the DAO.

If the DAO input parameters are valid, the DAO calls the database driver to get the update through to the database. Let's imagine that the SQL passed to the driver is invalid. The database driver then returns a client error. The database driver contract has been broken by the DAO and from the database driver perspective this is a client error.

From the DAO perspective this may not be a client error, though. If the SQL was generated / injected / hard coded injected into the DAO, and the SQL is invalid, this is an internal error from the DAO perspective.

An exception to this classification would be an application where the user can type in SQL in the user interface. If the DAO knows that the invalid SQL came from the user, then the DAO should keep the classification as a client error. It was an error in the input data given to the DAO.

The web actions also knows that the SQL came from the browser, and thus keeps the error classification as a client error. The application can then react accordingly.

If on the other hand, the database connection is broken, the database driver will raise a service exception. From the database driver's perspective this is a service error. The DAO receives the service error, and keeps that classification when propagating the exception up. So does the web action. The application then notifies the user, and logs the exception.

## Error Re-classification Table

Here is a more formal listing of what error types that can be reclassified on their way up the call stack. This table is not the final truth. Your application may contain different reclassifications.

|  |  |
| --- | --- |
| Error Type | Possible Re-classifications |
| Client Error | Stays a **client error**, if the component receiving the exception knows that the breach of contract was caused by it's own client. For instance, if the DAO receives a client error signaling invalid SQL, and the DAO knows that the SQL was typed in by the user of the application, then the DAO should keep the client error classification.   Is reclassified as an **internal error**, if the component receiving the error knows that the service contract was broken because of a buggy implementation or configuration of the component calling the service. For instance, if the DAO generates SQL and the database driver raises an �invalid SQL exception� which is a client error. The DAO should then reclassify this error as an internal error.   Client errors are almost never reclassified as service errors. If the service tells a component that it used it incorrectly, then the component should have a very good case for claiming that it was really the service that malfunctioned. It's like two components blaming each other for the error.   One exception could be if you know that the service incorrectly raises a client error in cases where the error is really a service error. For instance, an SQL driver complaining incorrectly over incorrect SQL, if the database connection is broken. In such a case you could change the client error classification to service error. But these cases are rare. |
| Service Errors | Service errors are rarely reclassified. If you call a component, and that component raises a service error, then the called component often knows best if the service it uses failed or not.   Exceptions to this is if you know that a component incorrectly classifies a certain error as a service error, when it is really something else. For instance, if a database driver is requested to connect to a given database, and the database does not respond, there are two options:   1. The database address / URL / port etc. is incorrect (client error). 2. The database is down and is not responding (service error).   The database driver may report this as a client error, assuming that the database address / URL / port is wrong. The DAO should reclassify this client error to an internal error, because the database configuration might be wrong. But if the calling component knows that the database URL is configured correctly, it may change the error classification to service error. |
| Internal Errors | Internal errors are rarely reclassified. If a component states that some error is an internal error, the called component often knows this best.   Of course there can be exceptions to this rule, just like with service exceptions. |

## Summary

How an application is able to react to an error depends on what type of error occurred.

Your application should try its best to determine the type of the error. It is not always possible to determine this 100% correctly though. If in doubt, classify the error as an internal error. Once the error occurs in production, you may learn more about the circumstances in which it occurs, and be able to change the classification logic.

# Strategy Elements

An exception handling strategy typically consists of the following elements:

1. [Error detection](http://tutorials.jenkov.com/exception-handling-strategies/error-detection.html)
2. [Error information gathering](http://tutorials.jenkov.com/exception-handling-strategies/error-information-gathering.html)
3. [Throwing the exception](http://tutorials.jenkov.com/exception-handling-strategies/throwing-exceptions.html)
4. [Propagating the exception](http://tutorials.jenkov.com/exception-handling-strategies/propagating-exceptions.html), possibly adding context information to it.
5. [Catching and reacting to the exception](http://tutorials.jenkov.com/exception-handling-strategies/catching-exceptions.html):
   * Retrying the action, if possible.
   * Notifying relevant parties.

I will talk a bit about each of these elements in the following pages.

# Error Detection

Error detection is what your code does to detect an error, preferrably before the error happens. For instance, you validate input parameters in a method, before using them. Here is a simple example:

public void doSomething(int value, Employee targetEmployee){

**if(value < 20)** {

throw new IllegalArgumentException("Value too low");

}

**if(targetEmployee == null){**

throw new IllegalArgumentException("Target employee was null");

}

... do actual work.

}

Notice the if-statements in the beginning of the method. These are error detecting statements. They check the input parameters for invalid values, and throw an exception if they are invalid.

The exception throwing itself in the example above is not part of the error detection. That is part of the "Throwing the exception", which I get into later. I have just shown them in the examples for clarity.

# Error Information Gathering

Error information gathering means gathering the information needed to react to, and diagnose the error. The needed information is typically divided into two categories:

* Error Cause
* Error Location

Application end users, operators, administrators and developers typically need information about what **caused** the error. They need that to determine if and how to respond to the error, and whether the error is severe or not.

If the error is severe and e.g. need some debugging and bug fixing by the developers, then the developers also need to know the **location** of the error. Knowing the location of the error is essential to know where to start looking for the cause of the error. For instance, even if the error description said something like "Parameter XYZ was null, and it should be non-null", you still need to determine why that parameter was all of a sudden null. That is what I mean by the underlying cause.

Error cause and location information often includes:

|  |  |  |
| --- | --- | --- |
|  | **Information** | **Interested Party** |
| **Cause** | Technical error description | Developers, Operators / Administrators |
| **Cause** | End user error description | End Users |
| **Cause** | Relevant input / parameter / variable data | End Users, Developers, Operators / Administrators |
| **Cause** | Relevant system data in singletons, configurations, databases etc. | End Users, Developers, Operators / Administrators |
| **Cause** | Relevant external conditions  (is system XYZ up?). | End Users, Developers, Operators / Administrators |
| **Location** | Stack Trace  (included in exception) | Developers |
| **Location** | Unique Error ID | Developers, Operators / Administrators |

Exactly what information you need to collect depends on your application, and what error that occurred. Some errors may need information about configuration setttings, or information fetched from database etc. Others may only need a simple error message - because the error is very simple, and easy to diagnose.

Additionally, what information to collect depends on what parties need to know about the error. If only the end user should be notified, you may not need to collect information about configuration settings etc.

# Throwing Exceptions

Throwing exception is the next step after error detection and information gathering. When you detect an error, you need to determine if you need to throw an exception, and what information to include in that exception.

What information to embed in the exception is determined in the previous stage - [Error Information Gathering](http://tutorials.jenkov.com/exception-handling-strategies/error-information-gathering.html)

Here is a simple example:

String errorId = ...

String errorDescription = ...

String contextId = ....

throw new MyException(errorId, errorDescription, contextId);

Later in this tutorial trail I will show you an Exception class definition that is much better suited to address the requirements listed earlier in this trail.

# Propagating Exceptions

When an exception is thrown it is propagated up the call stack until some try-catch block finally handles it. Propagation can be either passive or active.

## Passive Propagation

Passive propagation means that you just let the exception pass up the call stack without catching it. For this to be possible the exception must of course be declared by the method propagating it, or the exception must be unchecked.

Here is an example of passive prograpation:

public void doSomething() throws SomeException {

try{

doSomethingThatCanThrowException();

} finally {

//clean up � close open resources etc.

}

}

## Active Propagation

Active propagation means that you catch the exception and rethrow it, or wrap it in a new exception. You will typically use active propagation when you need to add extra information to the exception, about what actions were attempted and with what data, when the exception occurred. This extra information is, as mentioned in an earlier text, called the exception context � the context the exception occurred in.

Here is an example of active propagation.

public void doSomething() throws SomeException{

try{

doSomethingThatCanThrowException();

} catch (SomeException e){

e.addContextInformation(�more info�);

throw e; //throw e, or wrap it � see next line.

//throw new WrappingException(e, �more information�);

} finally {

//clean up � close open resources etc.

}

}

## Adding Context Information

As I explained in the text [Error Location and Context](http://tutorials.jenkov.com/exception-handling-strategies/error-location-and-context.html), the context in which an exception occurs may be just as important as the location of the exception itself. A given location in the application may be reachable via different execution paths, and the execution path may influence the severity and cause of the error, if it occurs.

If you need to add context information to an exception as you propagate it up the call stack, you need to use active propagation. In other words, you need to catch the exception in various relevant locations on the way up the call stack, and add the relevant context information to it, before rethrowing or wrapping it.

## Rethrowing vs. Wrapping Exceptions

As mentioned ealier active propagation can be implemented by either rethrowing the original exception or wrapping it in a new exception. Personally, I prefer to rethrow the existing exception, but this is not always the smartest choice.

This illustration shows how I prefer to wrap alien exceptions, and rethrow application specific exceptions:

|  |
| --- |
| Wrapping exceptions in checked exceptions and rethrowing them. |
| **Wrapping exceptions in checked exceptions and rethrowing them.** |

The above diagram assumes that you are using checked exceptions in your application.

When an exception is caught which is an alien exception, it is wrapped in an application specific exception. This is signalled by the "wrap" arrows.

If higher layers catches an application specific exception, they just rethrow it, possibly adding information to it. Or they may just propagate it passively, without touching it. This is signalled by the "rethrow" arrow.

If your application uses unchecked exceptions, here is how the wrapping and rethrowing could look:

|  |
| --- |
| Wrapping exceptions in unchecked exceptions and rethrowing them. |
| **Wrapping exceptions in unchecked exceptions and rethrowing them.** |

All alien exceptions are again wrapped in your own application specific exception. Also, if you have any old checked "legacy" exceptions you cannot just convert directly to your unchecked application specific exception already when it is thrown, wrap those too.

Once wrapped in your application specific exception, this exception is again propagated either passively or actively up the call stack, until a point where you can react to it sensibly.

### Wrap Alien Exceptions

An "alien" exception is an exception thrown by a Java API or a third party library. In other words, an exception you do not control.

I prefer to catch all alien exceptions and wrap them in an appropriate application specific exception. Once the alien exception is converted to your own exception, you can propagate that exception any way you like.

### Rethrowing Checked Exceptions can get Messy

If your application uses checked exceptions, rethrowing the original exception means that the method rethrowing it must also declare it. The closer you get to the top of the call hierarchy, the more exceptions will be declared thrown. Unless you just declare all your methods to throw Exception. However, if you do so you might as well use unchecked exceptions, since you are not really getting any benefit from the compiler exception checking anyways.

This is why I prefer to catch non-application specific exceptions and wrap them in an application specific exception, before propagating them up the call stack.

### Rethrowing Unchecked Exceptions is Easier

Rethrowing the original exception is a bit easier when using unchecked exceptions, because you do not need to declare them all the way up the call hierarchy. However, I still prefer to catch �alien� exceptions and wrap them in application specific exceptions. In fact, you may even be forced to do so for checked �alien� exceptions, thus converting the checked alien exception into an unchecked application specific exception.

# Catching Exceptions

Somewhere in your application you need to catch and react to thrown exceptions. Typically you will do this towards the top of the call hierarchy.

For instance, in a web application you may catch exceptions in the control servlet (if you have a control servlet).

In a desktop application it may look a bit different, because each listener on the various components, acts as the "top" level of the call hierarchy.

Here is a simple Java catch clause:

try{

startTheWholeThing();

} catch(MyAppException e) {

notifyUser(lookupErrorText(e));

notifyNonUsers(e);

} catch(Throwable t) {

notifyUser(lookupErrorText(e));

notifyNonUsers(t);

}

As you can see, there are two catch clauses. One for application specific exceptions, and one for any exceptions that your application has not converted to an application exception. Of course your application should try to convert all exceptions to application specific exceptions, but in case you forget an exception conversion, it is a good idea to have the extra catch clause.

## Exception Reactions

In the example shown above, there are only two different reactions:

* Notifying users.
* Notifying non users.

In the text [Error Causes, Types and Reactions](http://tutorials.jenkov.com/exception-handling-strategies/error-causes-types-reactions.html) I described two other common reactions to exceptions:

* Aborting the action.
* Retrying the action.

However, at the time the exception reaches the top level catch clause, the action should have already been either retried, and / or aborted. Therefore these two common reactions are not visible in the catch clause above.

## Notifying Users

How you notify the user depends on what kind of application you are developing.

A web service would notify the "user" (the web service client) by sending a SOAP Fault element.   
A Servlet would send an error page back to the user, or an error message embedded in the page.   
A desktop application would possibly show an error popup.

Exactly how you notify the users of your application (if any), is up to you to decide.

## Notifying Non-users

Non-users are typically notified via the application log. Exactly what kind of severity to log the exception under, depends on the type of exception. Typically, you need to determine the severity of the error before logging it.

For instance, client errors are probably not nearly as severe as some internal, unknown error is. Additionally, a service error may be severe for the functionality of your application, but it may not be necessary to take down your application. Just reboot the external service that failed, and everything works fine again (hopefully).

# An Exception Handling Strategy Template

Now that you have seen the requirements for an exception handling strategy, and seen what elements it consists of, I will try to put together an exception handling strategy template, that you can use for your own applications.

Remember, this template may not suit your application perfectly, so make the changes you need to make to it.

The strategy template consists of the following elements:

1. [An AppException and ErrorInfo template.](http://tutorials.jenkov.com/exception-handling-strategies/template-exception.html)
2. [An explanation of how to throw the AppException.](http://tutorials.jenkov.com/exception-handling-strategies/template-throwing.html)
3. [An explanation of how to propagate the AppException.](http://tutorials.jenkov.com/exception-handling-strategies/template-propagating.html)
4. [An explanation of how to extract information from the ErrorInfo list inside the AppException.](http://tutorials.jenkov.com/exception-handling-strategies/template-error-info-list.html)
5. [An explanation of how to catch and handle the AppException.](http://tutorials.jenkov.com/exception-handling-strategies/template-catching.html)

Each of these parts are described in the following texts.

# An Exception Class Template

The first place to start is to design an exception class which can contain all the necessary information.

As mentioned in earlier texts, the error information may be extended or changed as the exception is propagated up the call stack. Therefore, the exception cannot just have this information hardcoded.

Instead, it will keep error information in a separate ErrorInfo object. As the exception is propagated up the call stack, the upper layers of the code can add additional ErrorInfo objects to the error. Thus, rather than having a single ErrorInfo object internally, the exception has an internal list of ErrorInfo objects.

## AppException

Here is an example AppException class with the ErrorInfo list:

public class AppException extends Exception {

protected List<ErrorInfo> errorInfoList = new ArrayList<ErrorInfo>();

public AppException() {

}

public ErrorInfo addInfo(ErrorInfo info){

this.errorInfoList.add(info);

return info;

}

public ErrorInfo addInfo(){

ErrorInfo info = new ErrorInfo();

this.errorInfoList.add(info);

return info;

}

public List<ErrorInfo> getErrorInfoList() {

return errorInfoList;

}

}

It's a very simple exception class. No information can be contained in it, except ErrorInfo instances. Even the root cause (Throwable) cannot be contained in this exception. This too, is kept in the ErrorInfo instances.

## ErrorInfo

Now that you have seen the AppException class, let's look at the ErrorInfo class, which contains the error information. Here is an example ErrorInfo class:

public class ErrorInfo {

protected Throwable cause = null;

protected String errorId = null;

protected String contextId = null;

protected int errorType = -1;

protected int severity = -1;

protected String userErrorDescription = null;

protected String errorDescription = null;

protected String errorCorrection = null;

protected Map<String, Object> parameters =

new HashMap<String, Object>();

}

I have left out the getter and setter methods to make the example shorter.

The following table describes the fields in the ErrorInfo class.

|  |  |
| --- | --- |
| **Field** | **Description** |
| cause | The error cause, if an alien exception is caught and wrapped. |
| errorId | A unique id that identifies this error. The errorId tells **what** went wrong, like FILE\_LOAD\_ERROR. The id only has to be unique within the same context, meaning the combination of contextId and errorId should be unique througout your application. |
| contextId | A unique id that identifies the context where the error occurred.  The contextId tells **where** the error occurred (in what class, component, layer etc.). The contextId and errorId combination used at any specific exception handling point should be unique throughout the application. |
| errorType | The errorType field tells whether the error was caused by errornous input to the application, an external service that failed, or an internal error. The idea is to use this field to indicate to the exception catching code what to do with this error. Should only the user be notified, or should the application operators and developers be notified too? |
| severity | Contains the severity of the error. E.g. WARNING, ERROR, FATAL etc. It is up to you to define the severity levels for your application. |
| userErrorDescription | Contains the error description to show to the user.   Note: In an internationalized application this field may just contain a key used to lookup an error message in a text bundle, so the user can get the error description in his or her own language.  Also keep in mind that many errors will be reported to the user with the same standard text, like "An error occurred internall. It has been logged, and the application operators has been notified". Thus, you may want to use the same user error description or error key for many different errors. |
| errorDescription | Contains a description of the error with all the necessary details needed for the application operators, and possibly the application developers, to understand what error occurred. |
| errorCorrection | Contains a description of how the error can be corrected, if you know how. For instance, if loading a configuration file fails, this text may say that the operator should check that the configuration file that failed to load is located in the correct directory. |
| parameters | A Map of any additional parameters needed to construct a meaningful error description, either for the users or the application operators and developers. For instance, if a file fails to load, the file name could be kept in this map. Or, if an operation fails which require 3 parameters to succeed, the names and values of each parameter could be kept in this Map. |

In the following texts I will show you how to use the AppException and ErrorInfo classes to throw exceptions, wrap exceptions, propagate exceptions, catch exceptions etc.

# Throwing the AppException

In this text I will show you how to use the AppException shown in the text [An Exception Class Template](http://tutorials.jenkov.com/exception-handling-strategies/template-exception.html).

The diagram below is a subsection of the diagram shown in the [Overview](http://tutorials.jenkov.com/exception-handling-strategies/overview.html) text. It shows what happens before and until an application specific exception is thrown.

|  |
| --- |
| Throwing AppException's |
| **Throwing AppException's.** |

In this text I try to show you how to do this in code.

## Throwing an AppException

Here is a method that is supposed to load a file. It gets the file path to load as a parameter, and throws anAppException if the file path is null:

public byte[] loadFile(String filePath) throws AppException {

**// Error Detection**

if(filePath == null){

AppException exception = new AppException();

ErrorInfo info = exception.addInfo();

**// Error Information Gathering**

info.setErrorId("FilePathNull");

info.setContextId("FileLoader");

info.setErrorType(ErrorInfo.ERROR\_TYPE\_CLIENT);

info.setSeverity(ErrorInfo.SEVERITY\_ERROR);

info.setErrorDescription("The file path of file to load was null");

info.setErrorCorrection("Make sure filePath parameter is not null.");

**// Throw exception**

throw exception;

}

...

}

Notice how this example has both error detection (the if-statement), information gathering and exception throwing.

No parameter values are included in ErrorInfo object in the example above. It did not make sense, since we know the value of the parameter was null (filePath).

In the next example the filePath is included in the ErrorInfo because now it makes sense. The example checks whether the file exists or not, and then it is relevant to know the concrete file path.

if(! new File(filePath).exists()){

AppException exception = new AppException();

ErrorInfo info = exception.addInfo();

info.setErrorId("FileNotFound");

info.setContextId("FileLoader");

info.setErrorType(ErrorInfo.ERROR\_TYPE\_CLIENT);

info.setSeverity(ErrorInfo.SEVERITY\_ERROR);

info.setErrorDescription("No file found at file path " + filePath);

info.setErrorCorrection("FilePath should point to existing file.");

**info.getParameters().put("filePath", filePath);**

throw exception;

}

## Wrapping Alien Exceptions in an AppException

In case an exception is caught, an alien exception that is, you can wrap it in an AppException like this:

try{

FileInputStream fileInputStream = new FileInputStream(filePath);

// ...

} catch (IOException e) {

AppException exception = new AppException();

ErrorInfo info = exception.addInfo();

**info.setCause(e);**

info.setErrorId("FileReadFound");

info.setContextId("FileLoader");

info.setErrorType(ErrorInfo.ERROR\_TYPE\_INTERNAL);

info.setSeverity(ErrorInfo.SEVERITY\_ERROR);

info.setErrorDescription("Error processing file: " + filePath);

throw exception;

}

## An ErrorInfo Factory

In the examples above in this text, the creation of the ErrorInfo objects take up a lot of lines. This may potentially clutter your code. It gets the harder to read, when so many lines are just error handling code.

To fix this, you can move all the ErrorInfo code into an ErrorInfoFactory. Here is an example:

try{

FileInputStream fileInputStream = new FileInputStream(filePath);

// ...

} catch (IOException e) {

AppException exception = new AppException();

**exception.addInfo(**

**ErrorInfoFactory.getFileReadErrorInfo(e, filePath) );**

throw exception;

}

Here is the ErrorInfoFactory method:

public class ErrorInfoFactory {

public static final ErrorInfo getFileReadErrorInfo(

IOException e, String filePath) {

ErrorInfo info = new ErrorInfo();

info.setCause(e);

info.setErrorId("FileReadFound");

info.setContextId("FileLoader");

info.setErrorType(ErrorInfo.ERROR\_TYPE\_INTERNAL);

info.setSeverity(ErrorInfo.SEVERITY\_ERROR);

info.setErrorDescription("Error processing file" + filePath);

return info;

}

}

In the next text I will show you how to propagate AppException's up the call stack, and add information to it on the way up.

# Propagating the AppException

As mentioned in the text [Propagating Exceptions](http://tutorials.jenkov.com/exception-handling-strategies/propagating-exceptions.html) there are two ways to propagate an exception:

1. Passive propagation
2. Active propagation

Passive propagation pretty much just means to let the exception bubble up, without doing anything except closing opened resources. Thus, I will not touch upon that much more in this text. If you have nothing to add to an exception that is thrown from a deeper layer of code, just let it bubble up. Active propagation should only be used, if you have actually something to add to the exception.

Active propagation means catching the exception, add information to it, and rethrow it. Active propagations is also called "exception enrichment" - since the exception is enriched with information on the way up the call stack.

Here is an example:

public void loadConfigFile(String configFile) throws AppException {

try {

**byte[] file = loadFile(configFile);**

//do something with config file...

} catch (AppException e) {

ErrorInfo errorInfo = e.addInfo();

errorInfo.setErrorId ("ConfigFileLoad");

errorInfo.setContextId("ConfigFileLoader");

errorInfo.setSeverity (ErrorInfo.SEVERITY\_FATAL);

errorInfo.setErrorType(ErrorInfo.ERROR\_TYPE\_INTERNAL);

errorInfo.setErrorDescription("Error loading config file: "

+ configFile);

errorInfo.setErrorCorrection ("Stop application, " +

"and check that the configuration file is present and valid");

errorInfo.getParameters().put("configFile", configFile);

throw e;

}

}

This example shows a loadConfigFile() method which calls the loadFile() method shown in the previous text, [Throwing the AppException](http://tutorials.jenkov.com/exception-handling-strategies/template-throwing.html).

The called loadFile() method throws an exception if the file cannot be found. However, the loadFile() method does not know what kind of file is being loaded, and thus cannot know if a load error is a serious error or not. Therefore, the loadFile() method will just report the severity of the error as SEVERITY\_ERROR.

The loadConfigFile() method however, knows that it is loading a very important configuration file. Therefore, if the underlying loadFile() throws an AppException, this exception is caught, and more information is added to it (a new ErrorInfo object is added). Here is what information is updated:

* The errorId and errorContext are updated to reflect what was being done and where, when the exception occurred (ConfigFileLoad and ConfigFileLoader)
* The severity is updated to SEVERITY\_FATAL, since the configuration file must be present for the application to work.
* The errorType is set to ERROR\_TYPE\_INTERNAL, since a missing or invalid configuration file is definately an internal application problem.
* The errorDescription and errorCorrection are updated to tell what the error is, and what can be done about it. The loadConfigFile() method knows more about that, than the loadFile() method did, that threw the AppException originally.
* The configFile parameter is added to the ErrorInfo parameters, just in case loadFile() forgets to add the name of the failed file to it's own ErrorInfo.

Finally, the original AppException is rethrown. Rethrowing the AppException does not destroy the original stack trace (at least not in Java). In Java, the stack trace is filled in when the exception is instantiated, not when it is thrown. It is probably the same in Scala, since Scala runs on the Java VM. I don't know how this works in C#.

After the AppException as been rethrown, it now contains 2 ErrorInfo objects internally. One from theloadFile() method, and one from the loadConfigFile() method. This is illustrated here:

|  |
| --- |
| An AppException with an ErrorInfo list internally. |
| **An AppException with an ErrorInfo list internally.** |

## When to Add an ErrorInfo Object

You should not add an ErrorInfo object at every single propagation layer, on the way up the call stack. Only add anErrorInfo object if you have actually something new to say. Something to add to the original exception.

For instance, you may have more information about what was attempted when the exception occurred - what parameters were used etc. If loading of a file fails, and thus results in an exception, you may add information about what kind of file was attempted loaded. Was it a fatal configuration file? Or was it a user who typed in the wrong file name for a file they wanted to load in the application?

## What to set in the ErrorInfo Object

When you actively propagate an exception, you do not need to fill in every field in the new ErrorInfo object added to the exception. It is only ncecessary to set the information that is actually new.

For instance, if you do not want to change the severity of the error, don't set it. Just leave it empty in the newErrorInfo object. Likewise, if you are throwing an exception from some utility method, like loadFile(), you may not actually know what the proper error correction is. That may depend on what part of your application that called the method. In that case, just leave it empty (null).

### Playing it Safe

Of course you can decide to play it safe, and set every field each time an exception is actively propagated. It is not wrong to do so. It just takes up some more code lines.

The advantage of setting every field every time is, that you do not depend on underlying fields to be set correctly.

For instance, if the loadFile() reported the severity as SEVERITY\_FATAL already, you may decide not to set the severity in the loadConfigFile() method. However, if the loadFile() then one day is changed to set the severity to SEVERIY\_ERROR, then the loadConfigFile() file is no longer correct. It should then set theSEVERITY\_FATAL explicitly then.

By alway setting each field explicitly, you avoid these kinds of dependency errors.

## The Common Cases

Most often you will have only a single ErrorInfo object in the list, all the way up the call stack. That is, you attempted some operation, and some part of it failed. The whole operation now has to be aborted. The point where the part operation fails, will set the severity and error type. If that point in the code can only be reached through one path in the code, you know that the error is always of the same severity and type.

On the way up the call stack you may add additional ErrorInfo objects, which only add additional parameters etc. to the exception. The error type and severity are often untouched.

Only if an exception can occur in a piece of code with multiple possible paths to the failing code, is it necessary to update the error type and severity, depending on what context the failing piece of code was called in.

For instance, a FileLoader component may be called both from a ConfigFileLoader, and a UserFileLoadercomponent. If the file loading fails, it depends on which component that called it, how severe the error is, and what type it is.

Contrarily you may have an EmployeeDao which can insert Employee records in the database. You may know that this EmployeeDao.insert() is only ever called from one other component. Thus, it is always the same error type and severity if an exception occurs during insert. There is no need to add extra information to the original ErrorInfoobject, except perhaps a few extra parameters further up the call stack, if relevant.

This may all sound a bit confusing, and it is, until you start using it in your own applications. Then, little by little it will be more and more clear to you how it works.

# The ErrorInfo List

As the AppException is propagated up the call stack, a list of ErrorInfo objects are built up inside it. In this text I will examine that list, and how to extract information from it.

The ErrorInfo list can contain more than one ErrorInfo object. So, how do you determine what type of error theAppException represents? What is the severity? What error message should you show to the user? Or write to the application log? I mean, given that you could have multiple severities and error types embedded in the differentErrorInfo objects, how do you determine the real type and severity of the error?

### A ConfigFileLoader Example

Here is an example ErrorInfo list for an exception thrown during the loading of an important configuration file:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **errorId** | **contextId** | **type** | **severity** | **cause** |
| FileLoadError | FileLoader | client | error | FileNotFoundException |
| ConfigFileLoadError | ConfigFileLoader | internal | fatal |  |

This ErrorInfo list originates from an AppException being thrown by a FileLoader component. This component only knows that the file loading error, is of type client (wrong path given), and of severity error.

The FileLoader component was called by a ComfigFileLoader component. The ConfigFileLoadercomponent knows that if configuration file is not loaded correctly, the whole application will not work correctly. Therefore the ConfigFileLoader adds that information to the ErrorInfo list. It sets the error type tointernal, since either the configuration file is missing, or the paths to the configuration file is wrong. This is an internal error. Additionally, the ConfigFileLoader sets the severity to fatal, since the application cannot run correctly without the configuration file.

So, which of the two ErrorInfo objects should we use, when notifying users and non-users?

The answer is: A combination of them all.

In the example above, it is the second ErrorInfo object (ConfigFileLoader) that contains most correct information about the severity and type of the error. However, the first ErrorInfo object contains more information about what failed (FileNotFoundException).

So, when it comes to severity and type, use the latest ErrorInfo object. When it comes to logging diagnostic information, use all the ErrorInfo objects.

### A UserFileLoader Example

Here, I will look at a similar example to the one mentioned above:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **errorId** | **contextId** | **type** | **severity** | **cause** |
| FileLoadError | FileLoader | client | error | FileNotFoundException |
| UserFileLoadError | UserFileLoader | client | warning |  |

The FileLoader component is called by the UserFileLoader, which loads files which the user of the application provides the paths for. Thus, if an error occurrs during the loading of that file, it is not an error in the application. It is most likely just the user providing an invalid file path, or the file pointed to has invalid content. As soon as the user points the application to an existing, valid file, the error will go away.

### A StreamLoader Example

Imagine that the fileLoader() method calls a streamLoader() method underneath. The streamLoader()method is given a stream to load the file from. The streamLoader() method does not know where the stream is connected to, but the calling method most likely does. If the stream loading fails, here is an example of how theErrorInfo list could look:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **errorId** | **contextId** | **type** | **severity** | **cause** | **parameters** |
| StreamLoadError | StreamLoader | client | error |  |  |
| FileLoadError | FileLoader |  |  |  |  |

Notice how the second ErrorInfo object does not update the severity field. It has no new information about the severity, so it leaves it set to whatever the first ErrorInfo object set it to.

Of course, in a real application, the fileLoader() method would probably be called from some other method, which may actually know more about the severity of the error. Like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **errorId** | **contextId** | **type** | **severity** | **cause** | **parameters** |
| StreamLoadError | StreamLoader | client | error |  |  |
| FileLoadError | FileLoader |  |  |  |  |
| UserFileLoadError | UserFileLoader |  | warning |  |  |

Notice, however, how the last ErrorInfo object does not know anything new about the errorType, so it leaves it untouched.

## Extracting Severity and Error Type from the ErrorInfo List

The severity and error type of a given AppException is used to determine how to handle the exception, and which group of relevant parties to notify. Since the ErrorInfo list can contain multiple severity and error type codes, you need to somehow filter them down to one each. Otherwise, how do you know what to do about a given exception? I mean, if it is both listed as a warning and an error, and as a service and client error?

If you follow the rule of only setting the error type and severity if you actually know something new about it, you should use the latest severity and error type. Here is a code example:

int errorType = -1;

for(ErrorInfo errorInfo : e.getErrorInfoList()){

if(errorInfo.getErrorType() != -1) {

errorType = errorInfo.getErrorType();

}

}

int severity = -1;

for(ErrorInfo errorInfo : e.getErrorInfoList()){

if(errorInfo.getSeverity() != -1) {

severity = errorInfo.getSeverity();

}

}

After the above two for-loops have executed, the errorType and severity variables will contain the latest set value of error type and severity.

You may decide to encapsulate the two loops in each their method (I probably would). You probably won't be able to measure the performance difference, since exceptions do not happen that often, and since the ErrorInfo list is probably not going to be that deep.

## Extracting Error ID from the ErrorInfo List

Extracting the error ID and error descriptions from the ErrorInfo object list is a bit different than extracting error type and severity.

For error type and severity you are only interested in one value. For error ID and error descriptions you are interested in all the values. The complete error id is thus composed of all the error id's and context id's in ErrorInfo list.

Here is a code example method that illustrates how to extract the total error ID:

public static String extractErrorId(AppException e){

StringBuilder builder = new StringBuilder();

for(int i=e.getErrorInfoList().size()-1; i>=0; i--){

ErrorInfo errorInfo = e.getErrorInfoList().get(i);

builder.append(errorInfo.getContextId());

builder.append(":");

builder.append(errorInfo.getErrorId());

if(i>0){

builder.append("/");

}

}

return builder.toString();

}

This method creates a total error ID which is made up of all the context ID's and error ID's of the ErrorInfo list.

Notice that the list is iterated backwards, from last to first ErrorInfo object. Remember, the ErrorInfo objects are added to the AppException on the way up the call stack. The path to the exception in context ID's and error ID's is thus bottom up in the list. By iterating the list in reverse order, the path is changed to top down, reflecting the context path from the top of the application down to the location of the error.

Here is an example total error ID:

UserFileLoader:FileLoadError/FileLoader:FileLoadError

## Extracting User Error Descriptions

The user should be given a simple error message when something fails in your application. If the error is caused by the users behaviour, and thus can be corrected by the user, tell the user what the error was, and how to correct it.

If, on the other hand, the error is caused by something which is out of the users hands, don't give the user too much detail. Just let the user know that the requested action failed, and that it has been logged and will be investigated. You can use pretty much the same user error message for all these types of errors, as giving the user internal error ID's, parameter information etc. is not going to help the user. On the contrary, you risk giving away information that can be used by hackers to break into your application.

### Error Type Determines User Error Descriptions

You look at the errorType to determine if the error was caused by the user, or something else. If the error type is "client", that means that the client (the user) has used the application in a wrong way. By correcting the usage, the error will go away. In this case, tell the user what went wrong, and tell them how to correct their error.

If the errorType is either service or internal, then you all you do is to show the user a standard error message, saying that the error has been logged, and will be investigated.

## Extracting Non-User Error Descriptions

The non-user error description is used by the application operators and developers to determine the cause of the error, and how to correct it. Therefore, it should be as detailed as possible. The non-user error description should therefore include all the ErrorInfo objects details, and not just one of ErrorInfo objects.

The full error description becomes quite large, and possibly messy and unstructured. Therefore, it might make sense to convert it to an XML structure, and log that full structure. Having an XML structure for the error details may also make it easier for tools to monitor the log, and extract the bits of information out of it, they need.

Here is an example XML structure for an error report:

**<error>**

**<fullErrorId>**

UserFileLoader:LoadError/FileLoader:FileLoadError

**</fullErrorId>**

**<errorType>**client**</errorType>**

**<severity>**warning**</severity>**

**<errorInfoList>**

**<errorInfo>**

**<contextId>**FileLoader**</contextId>**

**<errorId>**FileLoadError**</errorId>**

**<errorType>**client**</errorType>**

**<severity>**error**</severity>**

**<errorDescription>**

The file was found, but could not be parsed correctly.

The following error was found in the file: ...

**</errorDescription>**

**<parameters>**

**<parameter name="file">**c:\data\myFile.txt**</parameter>**

**</parameters>**

**</errorInfo>**

**<errorInfo>**

**<contextId>**UserFileLoader**</contextId>**

**<errorId>**FileLoadError**</errorId>**

**<errorType>**client**</errorType>**

**<severity>**warning**</severity>**

**<errorDescription>**

An error occurred during the processing of a file which

the user requested the application to process.

**</errorDescription>**

**<errorCorrection>**

Point to a file that actually exists.

**</errorCorrection>**

**<parameters>**

**<parameter name="file">**c:\data\myFile.txt**</parameter>**

**</parameters>**

**</errorInfo>**

**</errorInfoList>**

**</error>**

# Catching the AppException

Somewhere in your application you will need to catch your application specific exception, and react to it.

As mentioned in the text [Catching Exceptions](http://tutorials.jenkov.com/exception-handling-strategies/catching-exceptions.html), the reaction to an exception typically falls into two categories:

1. Notifying users.
2. Notifying non-users.

Remember, that at the point of catching the exception, any potential local exception handling has already been carried out earlier in the call stack. That is why no local exception handling techniques are listed here.

Let's jump right into a code example:

try{

startTheWholeThing();

} catch(Throwable e) {

notifyUser(e);

notifyNonUsers(e);

}

First, the method startTheWholeThing() is called. This method declares that it can throw AppException's, but even if it is not declared, it may throw unchecked exceptions too (in Java). That is why the catch clause catchesThrowable intead of AppException

The catch clause calls two methods. The first method is notifyUsers(). This method is supposed to send a message to the user, saying that an error occurred. The second method is notifyNonUsers(). This method is supposed to notify all other relevant parties.

I'll go into a little more detail with these two methods, in the following sections.

## notifyUsers()

The notifyUsers() method should show an error message to the user, telling that an error occurred. How this is done depends a little bit on what kind of application you are developing.

### Web Applications

If you are developing a web application, and an exception is thrown and caught on the server, your web application should show a nice error message to the user. You will do that by either showing the error message inside the page that fails, or by redirecting the user to an error page.

If your web application uses AJAX, and it is an AJAX call that fails, then somehow your AJAX protocol must be able to signal to the JavaScript on the client, that an error occurred. The error message could either be embedded in the HTML page / JavaScript, or you could send it back from the server, included in the error signal in the AJAX call.

### Desktop Applications

In a desktop application you would often show the error message to the user in a popup window. That is, thenotifyUsers() method would pop up a new window showing the error message.

### Web Services

If you are developing a web service, or some other kind of remote accessible service, your users are other programs. For instance, a web service client. If an exception occurs in such a service, the access protocol should send an error message across to the client too.

Do not give the client more information than necessary. Of course, if the error is caused by the client, give the client enough information to correct the error.

### Background Services

Background services or batch jobs, often don't have direct users. They just execute autonomously in the background. Such applications will typically only notify non-users.

## notifyNonUsers()

The notifyNonUsers() method will typically log the error to some application log. Any relevant party that needs to know about the error, will be notified via the application log.

How you log the exception depends on what language you are programming in, and what logging tools you are using.

## Method Templates for notifyUsers() and notifyNonUsers()

Exactly how the notifyUsers() and notifyNonUsers() methods will look in your application, depends on your application, and your language and platform. Nevertheless, I will try to give you a little more detail about their internals here.

Here are the two templates:

**protected void notifyUsers(Throwable e){**

**if(e instanceof AppException) {**

AppException ae = (AppException) e;

**if(ae.getErrorType() == ErrorInfo.ERROR\_TYPE\_CLIENT) {**

// The error was caused by the user / client,

// show the cause and how to correct it.

**} else {**

// The error was not caused by user / client.

// Just show a standard error message.

**}**

**} else {**

//An unknown exception occurred. Show a standard error message.

**}**

**}**

**protected void notifyNonUsers(Throwable e){**

**if(e instanceof AppException) {**

AppException ae = (AppException) e;

**if(ae.getErrorType() == ErrorInfo.ERROR\_TYPE\_CLIENT) {**

// The error was caused by the user / client.

// It may not be necessary to log it, or just log

// it as info or warning. The app is working fine.

// It is the user that is malfunctioning.

**} else if(ae.getErrorType() == ErrorInfo.ERROR\_TYPE\_SERVICE) {**

// The error was caused by a failure in an external service.

// Notify operations about the external failure (log it).

// Also tell if this appliation will survive the failure,

// and continue working once the external service works again.

**} else if(ae.getErrorType() == ErrorInfo.ERROR\_TYPE\_INTERNAL) {**

// An internal error occurred. This is serious, and may need

// both operations and developers attention, as it may be a

// sign of a bug in the application. Log the error.

**}**

**} else {**

// The exception was not an AppException. This should be fixed,

// so that exception is caught and wrapped in an AppException

// in the future. Log the error.

// This kind of error is also serious, and should get the attention

// of both operations and developers, as the cause of the exception

// is unknown.

**}**

**}**

# Execution Context

An execution context is an alternative or supplement to the AppException and ErrorInfo described in the exception handling template.

As shown in the text on the AppException you can collect context information in the AppException as it is propagated up towards the top of the application. This context information is stored internally in the ErrorInfo list.

In this text I will show you an alternative, or a supplement if you will, called an "Execution Context".

An execution context is an object attached to the current thread of execution. For instance, by storing the execution context object in a ThreadLocal variable, or by mapping it to the thread in a statically available Map.

At any level in your application you can thus write execution context information into the execution context object. You can do this as the execution flows down, as illustrated here:

|  |
| --- |
| Execution Flow - with calls to an Execution Context |
| **Execution Flow - with calls to an Execution Context.** |

You can also write to the execution context as an exception is propagated up the call stack, as illustrated here:

|  |
| --- |
| Exception Propagation Flow - with calls to an Execution Context |
| **Exception Propagation Flow - with calls to an Execution Context.** |

As you can imagine, the execution context object can contain execution context information, regardless of whether a request succeeded, or failed. You can think of the execution context object as an "execution log". Whether a request failed or succeeded, this execution log contains all information related to that request processing, provided your application writes that information to it, that is.

When a request is successfully processed, you can write the whole execution context object to a log file, as a single, coherent structure (e.g. XML). Or, you can just ignore it, and clear the context.

Similarly, when a request fails you can write the complete execution context object to a log file, along with any relevant information contained in the exception.

## Benefits

The benefit of using an execution context together with the AppException is that you can get more information about the execution, than you can with the AppException alone.

The extra information about the execution may not enable you to better determine type and severity of an exception. But, when logging the error, you could log the execution context too. It might be helpful for the developers, when trying to understand why the exception occurred.

## Drawbacks

A drawback (disadvantage) of using an execution context could be, that your code gets cluttered with loads and loads of small calls to the execution context. Additionally, there is no guarantee that the logged information actually helps developers diagnose the error. It all depends on how you use the execution context, and what information you include in your calls.

## Simple Execution Context Implementation

Let's look at how you could implement a simple execution context. It's implemented as a single class calledExecutionContext, which keeps all calls to it internally in a list. The execution path is thus flat. You cannot see the real tree-like structure the execution path follows through your code.

public class ExecutionContext {

protected String contextId = null;

protected String locationId = null;

protected Object details = null;

protected static Map<Thread, List<ExecutionContext>> executionContexts =

new ConcurrentHashMap<Thread, List<ExecutionContext>>();

**public static void log(String contextId, String locationId,**

**Object details){**

List<ExecutionContext> executionContextListForThread =

getExecutionContext();

ExecutionContext executionContext = new ExecutionContext();

executionContext.contextId = contextId;

executionContext.locationId = locationId;

executionContext.details = details;

executionContextListForThread.add(executionContext);

**}**

**public static void clearExecutionContext() {**

getExecutionContext().clear();

**}**

**public static List<ExecutionContext> getExecutionContext() {**

List<ExecutionContext> executionContextListForThread =

executionContexts.get(Thread.currentThread());

if(executionContextListForThread == null){

executionContextListForThread =

new ArrayList<ExecutionContext>();

executionContexts.put(Thread.currentThread(),

executionContextListForThread);

}

return executionContextListForThread;

**}**

}

Here is a simple example of how to use it:

public class ExecutionContextExample {

public static void main(String[] args) {

**ExecutionContext.log("ExecutionContextExample", "main", null);**

try {

level1();

} catch(AppException e){

log(e);

log(ExecutionContext.getExecutionContext());

}

}

private static void level1() throws AppException {

**ExecutionContext.log("ExecutionContextExample", "level1", null);**

level2();

}

private static void level2() throws AppException{

try {

**ExecutionContext.log("ExecutionContextExample", "level2-1", null);**

level3();

**ExecutionContext.log("ExecutionContextExample", "level2-2", null);**

} catch (Throwable t){

AppException appException = new AppException();

ErrorInfo errorInfo = appException.addInfo();

errorInfo.setCause(t);

//... fill more data into ErrorInfo object.

**ExecutionContext.log("ExecutionContextExample", "level-2-error",**

**errorInfo);**

throw appException;

}

}

private static void level3() throws Exception {

**ExecutionContext.log("ExecutionContextExample", "main", null);**

}

private static void log(List<ExecutionContext> executionContext) {

//log execution context list to file. Perhaps as an XML structure.

}

private static void log(AppException e) {

//log AppException to file. Perhaps as an XML structure.

}

}

Notice how each method ( level1() to level3() ) calls the ExecutionContext. Also notice how theExecutionContext list is now logged, in case an exception occurs.

## Advanced Execution Context Implementation

The previous implementation of the ExecutionContext only keeps a flat list of the execution context information written to it. The execution path is really a tree structure, and not a flat list. Therefore, I have developed anExecutionContextTree class, which can contain this information.

In order to collect the execution tree path correctly, you must now use two methods instead of one:

ExecutionContextTree.pre("contextId", "locationId", null);

ExecutionContextTree.post();

The pre() call creates a new node, and attaches it to the parent node (if any). Any calls to pre() after this one, will result in new nodes being attached to the newly created node.

The post() call removes the node as the current parent in the execution tree. The next call to pre() will now attach a node to the parent of the node just removed as parent node.

Here is the code:

public class ExecutionContextTree {

public static class ExecutionContextNode {

public String contextId = null;

public String locationId = null;

public Object details = null;

public ExecutionContextNode parent = null;

public List<ExecutionContextNode> children =

new ArrayList<ExecutionContextNode>();

}

protected static Map<Thread, ExecutionContextNode> roots =

new ConcurrentHashMap <Thread,ExecutionContextNode>();

protected static Map<Thread, ExecutionContextNode> currentParents =

new ConcurrentHashMap<Thread, ExecutionContextNode>();

public static void pre(String contextId, String locationId,

Object details){

Thread currentThread = Thread.currentThread();

ExecutionContextNode node = new ExecutionContextNode();

if(roots.get(currentThread) == null) {

roots.put(currentThread, node);

}

node.contextId = contextId;

node.locationId = locationId;

node.details = details;

node.parent = currentParents.get(currentThread);

if(node.parent != null){

node.parent.children.add(node);

}

currentParents.put(currentThread, node);

}

public static void post(){

Thread currentThread = Thread.currentThread();

ExecutionContextNode node = currentParents.get(currentThread);

if(node.parent != null){

currentParents.put(currentThread, node.parent);

} else {

//remove top node from currentParents and from root,

// root was removed so there are no more parents or root.

currentParents.remove(currentThread);

roots.remove(currentThread);

}

}

public static ExecutionContextNode root() {

return roots.get(Thread.currentThread());

}

public static void clear() {

roots.remove(Thread.currentThread());

currentParents.remove(Thread.currentThread());

}

}

Here is an example of how to use this ExecutionContextTree:

public class ExecutionContextTreeExample {

public static void main(String[] args) {

try {

ExecutionContextTree.pre("ExecutionContextExample", "main", null);

level1();

ExecutionContextTree.post();

ExecutionContextTree.clear();

} catch(AppException e){

log(e);

log(ExecutionContextTree.root());

}

log(ExecutionContextTree.root());

}

private static void level1() throws AppException {

ExecutionContextTree.pre("ExecutionContextExample", "level1", null);

level2();

ExecutionContextTree.post();

}

private static void level2() throws AppException{

try {

ExecutionContextTree.pre("ExecutionContextExample", "level2-1",

null);

level3();

ExecutionContextTree.post();

} catch (Throwable t){

AppException appException = new AppException();

ErrorInfo errorInfo = appException.addInfo();

errorInfo.setCause(t);

//... fill more data into ErrorInfo object.

ExecutionContextTree.pre("ExecutionContextExample", "level-2-error",

errorInfo);

ExecutionContextTree.post();

throw appException;

}

}

private static void level3() throws Exception {

ExecutionContextTree.pre("ExecutionContextExample", "Level-3", null);

ExecutionContextTree.post();

}

private static void log(AppException e) {

//log AppException to file. Perhaps as an XML structure.

}

private static void log(

ExecutionContextTree.ExecutionContextNode executionContextRoot) {

//log execution context list to file. Perhaps as an XMl structure.

ExecutionContextTree.ExecutionContextNode node =

ExecutionContextTree.root();

}

}

Notice how pre() and post() calls are paired.

Also notice how the ExecutionContextTree is cleared if no exception occurs, and logged if an exception occurs.

### Insert pre() and post() calls using AOP

As you can see, the pre() and post() calls are very often insert at the start and end of a method call. This would be a lot of work to do inside every method if you had to do it manually. Luckily, this kind of task is what we have Aspect Oriented Programming for (AOP).

Keep in mind though, that you may not need the entire context tree to be available for your debugging. Use theExecutionContextTree with consideration. Collect the information that makes sense, and leave out some of the smaller detail.

Of course it can be hard to predict exactly what detail becomes important during debugging, as you don't know exactly which errors will occur, ahead of time. As you are debugging, you may need to add more calls to theExecutionContextTree to capture all the information you need.

# Avoid Exception Hierarchies

You may have noticed that the exception handling template shown earlier in this tutorial, only uses a single exception class. There are no exception hierarchies. The reason I have avoided exception hierarchies, is because they don't really add much value to the template. Allow me to explain why I think so.

If you are not familiar with exception hierarchies, I have explained how they work in Java, in the text [Exception Hierarchies](http://tutorials.jenkov.com/java-exception-handling/exception-hierarchies.html)

## A Hierarchy is an Insufficient Categorization of Exceptions

Exception hierarchies are basically a way of categorizing exceptions. The standard Java API's typically categorize exceptions after what caused the exception. Like IOException,

However, as you have seen in this tutorial, there are several parameters that are needed to determine how to handle an exception, and not just its original cause. Thus, the categorization a hierarchy provides is insufficient to carry all the necessary information needed to handle an exception.

Proper exception handling requires both knowledge about the type of error (internal, client, service), the context in which the exception occurred, the severity etc. All that cannot be fitted into a single categorization.

Since not all information needed in an exception can be contained in an exception hierarchy, it is easier to just leave out the hierarchy completely, and just use a single exception class.

## Hierarchies Clutters Exception Propagation

Leaving out the exception hierarchy also makes exception propagation easier. If a method can throw 3 different exceptions, which all extend the same superclass, should your method declare each of the 3 different exception subclasses, or just the exception superclass?

When there is only a single exception used in your application, you will always just propagate that up the call stack, if your method propagates the exception rather than handling it locally.