Java Idioms: Exception Handling

Arno Haase
Arno.Haase@acm.org
Arno.Haase@Haase-Consulting.com

Abstract

Exceptions are a powerful feature for the propagation and handling of failure of operations. Naive use of exceptions however introduces subtle complexity into a system due to the non-local nature of throwing an exception.

This paper presents a pattern language of eleven Java idioms which serve as a framework for using exceptions. The patterns help to balance the forces that the overall complexity must be limited, that the code must be modifiable and extensible, the source code must be clear and the overall system must be robust.

Introduction

In procedural programming languages like C, the usual way to communicate failure of a function call is to use the return value. A special value - often `null` or `-1` - indicates failure of the operation.

In order to handle every such potential failure, the calling code has to check the return value of every single function call, resulting in heaps of `if` statements throughout the code. This spreads the handling of failures and errors throughout the entire application, making it difficult to understand and test as well as easy to get wrong in the first place.

Especially the propagation of failure up the call stack is rather painful to implement because it must be carefully done in every single function. As a result, failure propagation is often conveniently forgotten; developers implement local handling of the failure as nothing more than a logged message, hoping against hope that everything will work out fine.

But what does this have to do with Java programming? After all, Java is a modern language which has exceptions as a language feature. Using the built-in exception mechanism, error and failure handling is a piece of cake, the problem is solved once and for all. Using exceptions, error handling is easy. But is it?

Well, exception handling certainly allows a better separation of failure handling from the main flow of the program, making the source code both shorter and easier to read. But throwing an exception is a powerful way to affect control flow with high inherent complexity. It has a lot in common with a non-local `goto` statement jumping to an unknown target in another method.

This raises two issues. Firstly, there is *exception safety*: If an exception is thrown, the program must make sure that all necessary clean-up operations are performed, particularly freeing any resources that were allocated for the operation. This important aspect is treated separately in the next chapter.

Secondly, there is *exception organization*. The mechanism that is used for notification of failure couples the caller to the called method. Exception throwing is non-local in nature, so this coupling is not very strong. But then it is not very straightforward either.

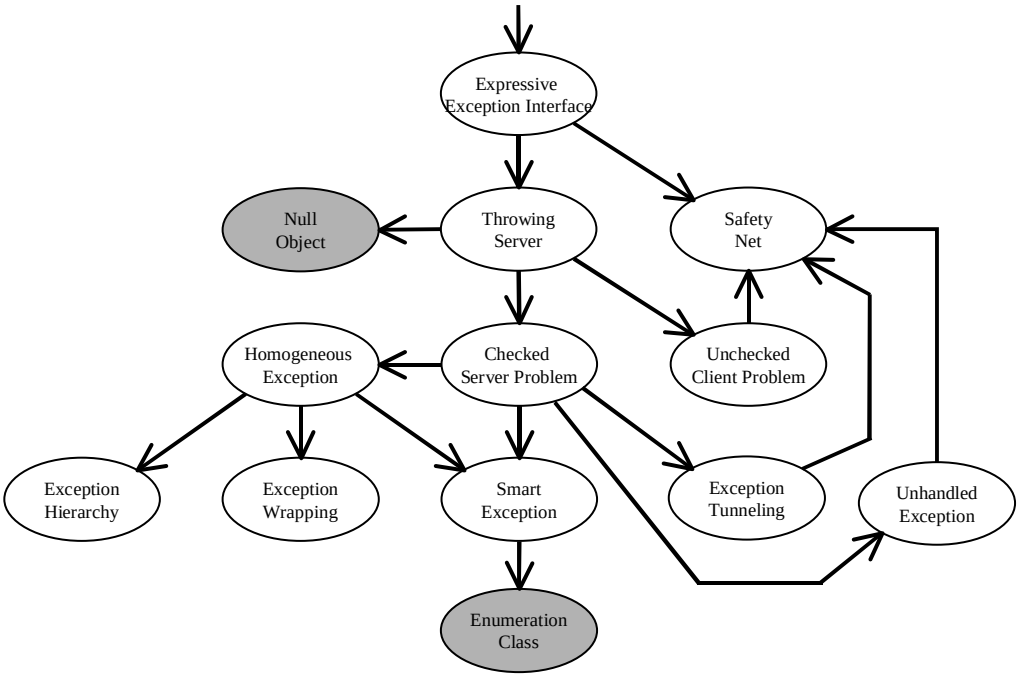
There are two conflicting forces at work that make this issue difficult. On the one hand, if clients know about the details of the exceptions that are thrown then a big system can become difficult to change. If for example the persistence layer of a business system uses a relational database and the `SQLExceptions` are propagated all the way up and caught in the presentation layer then the presentation layer directly depends on details of the relational database that is used. In such a system, changing an implementation in one place results in subtle changes in many other places.

But on the other hand, clients often need to handle exceptions that are thrown somewhere further down the call stack, so they need to know something about the exceptions that are thrown by code that they use.

This chapter is about building systems where exceptions pull their weight separating the normal flow from the handling of failures without introducing subtle non-local coupling that gets in the way of evolving and maintaining the system.

The Pattern Language

The pattern language in this chapter consists of eleven patterns; the figure below gives an overview of the patterns. The arrows between the patterns express how the resulting context of some patterns are part of the problem that other patterns solve. Grey ovals stand for patterns from other chapters that are referenced but not included in this chapter.



The starting point of this pattern language is EXPRESSIVE EXCEPTION INTERFACE which introduces the concept of exceptions as an explicit part of the interface of a class or package.

THROWING SERVER describes when a method should throw an exception as part of an expressive exception interface and when it should attempt to handle failure locally.

The cause for an exception that is thrown by a method can either be an UNCHECKED CLIENT PROBLEM if the failure is caused by faulty client behavior or a CHECKED SERVER PROBLEM if the problem lies inside the method.

If CHECKED SERVER PROBLEM is used extensively, many checked exceptions tend to accumulate in method signatures far up the call stack, creating unwieldy interfaces that expose implementation details. HOMOGENEOUS EXCEPTION addresses this problem by throwing only a single checked exception, and EXCEPTION WRAPPING shows how to maintain the details of the original exception.

EXCEPTION HIERARCHY and SMART EXCEPTION are a pattern pair showing ways to equip exceptions with information so that handling code can handle different exceptions differently. EXCEPTION HIERARCHY uses specialization through inheritance and provides a loose coupling between different types of failure whereas SMART EXCEPTION introduces an ENUMERATION CLASS (53) as a field so that all cases are defined in a single class.

A TUNNELING EXCEPTION is a way to propagate a CHECKED SERVER PROBLEM through framework code that does not contain the checked exception in its method signature.

UNHANDLED EXCEPTION shows a way to postpone actually implementing exception handling for a CHECKED SERVER PROBLEM without the full risk of creating empty catch blocks even temporarily.

And finally, SAFETY NET serves as a counterweight to THROWING SERVER by describing how a default handler can be installed that catches and handles unchecked exceptions that slip through regular handling code.

Each of the patterns solves a distinct problem, but they are most useful if they are applied together. As a whole, the pattern language describes a proven and useful way to organize exception handling in large Java systems.

EXPRESSIVE EXCEPTION INTERFACE

Handling exceptional and error conditions is inherently complex. Therefore make exception handling an explicit part of the design at the method, class, package and system level.

Problem

Naive ad hoc decisions about how failure is handled lead to tangled code that is difficult to maintain or extend. Clients are tightly coupled to the implementation of the methods they call, and changes of a method implementation force changes of the method signature, causing other changes throughout the system.

Worse still, the non-local nature of exception throwing means that some of the necessary changes in seemingly unrelated parts of the system tend to go unnoticed. At best, the result is the surprising appearance of subtle bugs during system testing, at worst the bugs make it into production and crash the system at a very inconvenient moment.

There are several aspects of the problem that make it more difficult to solve.

Firstly, the handling and propagation of failure and error conditions is usually inherently non-local so that any local approach has only limited usefulness.

Then, errors that occur can typically not simply be ignored - some sort of handling is usually called for. But often such handling is not sensibly possible where the error is noticed, so some sort of notification is necessary.

Clients need to know what can go wrong in code they are calling so they can handle errors appropriately, but the naive use of exceptions creates a dependency on the implementation rather than the interface.

And exceptions that methods can throw become part of the interface of a class; for checked exceptions this is explicit whereas for unchecked exceptions it happens implicitly. The actual throwing of the exceptions however is dispersed throughout the implementing code. Therefore changes of the implementation have a tendency to cause different exceptions to be thrown, resulting in a changed interface.

Error handling and propagation is so deeply engraved in the source code of a system that it takes very significant effort to change it after the initial stages of implementation. Refactoring often does not work well here. Firstly the changes are quite onerous, requiring thought rather than mechanistic repetition. And secondly, test coverage for error handling is typically not quite as good as for functionality, adding risk to effort.

Solution

Treat exceptions as full members of the interface - not only at the method level but also for classes and particularly for packages and subsystems.

Since exceptions create subtle coupling, actively manage this coupling. During design planning at all levels, take the propagation of failure into account. Decide on an exception strategy at the architectural level, and make sure there is understanding and consensus on it in the entire project team.

The coupling caused by exception throwing should be taken as seriously as coupling caused in any other way. Therefore, exception handling should be included in the architectural planning.

This has the benefit that laying down a system-wide policy for the use of exceptions makes it possible to address the non-local issues of exception throwing. The throwing, propagation and handling of exceptions can be adjusted so that they fit together well.

Planning and documenting high-level use of exceptions also help to avoid situations where a programmer is confronted with an error condition without seeing how to either handle it or propagate it. Since there are rules and examples for the use of exceptions, developers have help integrating their exception concerns with the rest of the system.

Then, dealing with exception handling at the global level makes it possible to strike informed trade-offs between the need to notify clients and the resulting coupling.

Explicitly designing exceptions at the class and package level allow the “exception interface” which is visible to clients to be isolated from implementation details, making it stable even when significant changes are made to the implementation.

And thinking about exception handling right from the beginning allows an informed choice to be made as to how much up-front planning is warranted for a particular system and to what degree this aspect can be refactored later.

Any form of error propagation and error handling introduces a coupling between source and handler of the error, but several other patterns in this pattern language help to reduce or manage the coupling introduced through exceptions.

THROWING SERVER addresses the question which failures to propagate to the caller at all. CHECKED SERVER PROBLEM and UNCHECKED CLIENT PROBLEM discuss the trade-off between checked and unchecked exceptions and the different kinds of coupling between a method and its callers they create.

The question of how to uphold encapsulation for the exception interface is addressed by HOMOGENEOUS EXCEPTION; this is especially relevant for packages and subsystems.

Exceptions for which handling code does more than print a message string in some way can be implemented with SMART EXCEPTION in order to reduce coupling by convention between caller and handler.

And SAFETY NET shows how a default handler for unchecked exceptions that would otherwise not be handled can be installed, reducing the risk of loose coupling between throwing and handling code.

These patterns help to strike the right balance in the strength of coupling, on the one hand allowing encapsulation so that different parts of the system can be changed independently of each other and on the other hand upholding a sufficiently strong coupling to allow reasonable handling of exceptions.

THROWING SERVER

In order to create an EXPRESSIVE EXCEPTION INTERFACE, let the caller of a method deal with exceptions that can not be locally handled in a satisfactory way.

Problem

It is a fact of life that things go wrong.

Sometimes a method can address a problem locally, for example by substituting default values if a configuration file is not found. But more often than not, a problem cannot be completely resolved where it first occurs, simply because there is not enough information available.

For example, consider a business component that is asked to update a specific record. If it cannot do so because constraints are violated, it has no way of knowing how to handle the problem. That is a natural consequence of good separation of concerns - a customer management component does only one thing and does that well, and its interface is not cluttered with issues which are unrelated to its primary purpose.

It is obviously evil to silently consume an exception that indicates failure of the operation:

```
// BAD CODE - the exception is ignored and consumed
public void updateCustomer (String id, CustomerData newRecord)
{
  try {
    .... // perform the actual update in the database
  }
  // the exception is silently ignored, and calling code
  // has no way of knowing if the operation was successful
  catch (SQLException exc) {}
}
```

Another more subtle but no less common way of achieving the same misbehavior is to first perform a validity check and not notify calling code if that fails:

```
// BAD CODE - silently ignores invalid data
public void updateCustomer (String id, CustomerData newRecord)
{
  if (! isValid (newRecord)) {
    // silently abort the operation without notifying the
    // caller
    return;
  }

  .... // perform the actual update operation
}
```

Both these scenarios share the fundamental flaw of attempting to hide failure from the caller without being able to completely handle it locally. Calling code is presented with the fiction that every operation completes successfully, and in the rare but important case of failure there is no one to handle it.

So how should failures be addressed that can not be handled locally?

Solution

Notify the calling code by throwing an exception. Use UNCHECKED CLIENT PROBLEM if the exception was caused by the client, for example because erroneous data was passed in as a parameter. Use CHECKED SERVER PROBLEM in all other cases.

Make sure that only those exceptions are thrown which can not be handled locally in a meaningful way.

If there is default behavior associated with the failure, consider returning a NULL OBJECT (???) as an alternative to throwing the exception.

Throwing an exception to notify calling code of a failure has several benefits. There is no danger of the failure going unnoticed, even if the immediate caller does not handle it. Exceptions can hold data regarding the details of the underlying failure, but this data is encapsulated in the exception class, and changes to the exception class concern only throwing and handling code and are transparent to code between them.

And exceptions are automatically propagated by the JVM as opposed to the cumbersome cascades of `if` statements required for error codes as return values.

Using exceptions to notify callers of failure has no liabilities in and of itself. But there are some aspects that can become problematic because throwing exceptions seems to be so easy.

One such aspect is that it is convenient to ignore exception handling entirely and just throw an exception and let callers worry about handling the problem. It is often appropriate to notify callers of a failure rather than trying to hush it up, but there are cases where some handling is appropriate before declaring an operation to be failed. Network code for example should often retry an operation or wait for a timeout before throwing an exception. And if there is an I/O problem when saving a file, the operation could prompt the user for another file name before throwing an exception.

For these reasons it is important to strike a balance. It is necessary to notify callers of failure, but it is equally important to implement a “caller-friendly” definition of failure by handling locally what can be handled locally. This is the mark of a good interface that encapsulates an abstraction well.

It is also easy to forget the complexity of the non-local goto caused by exceptions. Again, this complexity is not per se a liability of using exceptions because non-local control flow is inherent in handling failure. But exceptions are easy to use in Java, and that can result in careless use of them. The end result can be trouble, especially for large systems.

A perceived obstacle to the use of exceptions is the notion that they introduce significant performance overhead. This is not true for the normal case: If no exception is thrown, there is no performance overhead at all. In the case of failure, performance is typically much less of an issue because failure occurs much less frequently; as usual, the best approach to performance optimization is to put clarity and simplicity first and optimize only if necessary.

CHECKED SERVER PROBLEM

Use checked exceptions to express possible failures in a THROWING SERVER interface that have their root cause in the inner workings of the operation rather than invalid input by the caller.

Problem

In order to use a class it is necessary to know and understand what it does.

This is obviously true for the functionality, but it also holds for the question of how failure is handled and propagated. In order to write code using a class, it is necessary to know what can go wrong and which exception is thrown.

How can this be expressed in the source code?

Source code comments tend to get out of date. So while they some help in documenting and illustrating failure modes, it is desirable to express them in a way that is more directly coupled to the source code and its changes, and preferably supported by the compiler.

Further, every client needs to address the handling of failure of a method it calls. If it does not, subtle errors tend to result that have a nasty tendency to slip through testing unnoticed and become apparent only after the system has been released. Preferrably a way of expressing possible failures of a method should let the compiler force clients to address them.

Solution

Use checked exceptions (exceptions that are not subclasses of `RuntimeException` and must therefore be declared in the `throws` clause of a method) to notify callers that the operation failed. That way, it is explicit and obvious both in the source code and in the generated Javadoc comments which exceptions a method can thrown.

In the example of the customer management component which was introduced for THROWING SERVER, the `updateCustomer` method should be declared to throw a checked exception, making it explicit in the method signature how invalid data is handled. The following source

code presupposes the existence of a checked `ConstraintViolationException` which is used for this purpose:[1]

```
// signature comment explicitly expresses how invalid data is
// handled, but without a javadoc comment the details are
// left to the reader's imagination
public void updateCustomer (String id, CustomerData newRecord)
  throws ConstraintViolationException
{
  if (! isValid (newRecord)) {
    // notify the caller
    throw new ConstraintViolationException ();
  }

  .... // perform the actual update operation
}
```

Declaring to throw checked exceptions forces every client to take into account that an operation can fail, either handling it themselves or again explicitly declaring to throw it. Because of this, the compiler ensures that there is a handling `catch` statement for every checked exception.

Having checked exceptions in a method signature also provides documentation about which exceptions can be thrown. Sometimes the type of the exceptions even suggests when and in what ways they are thrown - for example, having an `SQLException` in the signature of a `readCustomer` method strongly suggests that all `SQLExceptions` occurring inside the method are propagated.

But usually additional information should be supplied about the details of when the exception is thrown, and the `@throws` JavaDoc tag is useful for just that; the following listing shows the method signature of the `updateCustomer` method, but this time with an explaining JavaDoc comment:

```
// JavaDoc comment using "@throws" to provide details about
// an exception
/** ....
  @throws ConstraintViolationException if the data in the
    newRecord parameter is not valid.
  */
public void updateCustomer (String id, CustomerData newRecord)
  throws ConstraintViolationException
{
  ....
```

If a method declares to throw a checked exception under certain circumstances, it is important that the checked exception is thrown consistently. The code must for example be written in such a way that no `RuntimeException` is accidentally thrown instead of the checked exception.

1. This source code example does not perform industry strength checking; it is simplified to illustrate the underlying idea. The separate check approach works for simple sanity checks or authorization checks, but it is insufficient for ensuring that a subsequent update of the database is going to be valid. The main issue is that the checking must be performed in the same transaction as the update.

It is particularly easy to accidentally throw a `NullPointerException` because of uninitialized references; the following listing gives an example how explicit checks can be used to avoid this:

```
// this implementation must take care that invalid input
// does not accidentally cause a NullPointerException to be
// thrown
public boolean isValid (CustomerData newRecord)
{
  // without the first check, the second check could
  // throw a NullPointerException instead of causing
  // 'false' to be
  if (newRecord.getName() == null ||
      newRecord.getName().length() > MAX_NAME_LENGTH) {
    return false;
  }
  .... // further checks and the actual update
}
```

The benefits of checked exceptions can be summarized by saying that their use provides documentation and ensures that exceptions are handled. There is however a downside to this, namely that using checked exceptions reduces flexibility. Other patterns of this pattern language help to address this.

If all methods simply pass through all unhandled exceptions, there tends to be a proliferation of checked exceptions near the top of the call stack, and many method signatures must be changed if the implementation of a method far down the call stack is changed to throw an additional checked exception. HOMOGENEOUS EXCEPTION addresses this problem.

Sometimes it is desirable to postpone implementing the handler for a checked exception until the throwing method is more mature, but the compiler prevents this. UNHANDLED EXCEPTION shows a way out of this dilemma.

Java does not allow an overriding method in a subclass to throw checked exceptions that are not declared in the superclass; that can lead to problems if the superclass is part of a library and can not be changed, especially if the subclass method is called from a framework. TUNNELING EXCEPTION shows a way how checked exceptions can be thrown "through" unknowing code.

And finally, SMART EXCEPTION and EXCEPTION HIERARCHY discuss how handling code can preform different operations based on detailed information about why an exception was thrown.

HOMOGENEOUS EXCEPTION

If CHECKED SERVER PROBLEM is extensively applied, method signatures have a tendency to be cluttered with many different checked exceptions, making exception handling complex and counterintuitive. To resolve this, introduce a new exception class and have all methods of a class or package declare only this.

Problem

Using CHECKED SERVER PROBLEM shows explicitly which exceptions are thrown in a method, making the code communicative and helping to prevent forgotten exception handling.

As a system grows, however, the lists of thrown exceptions tend to become long and unintuitive as exceptions that are thrown further down the call stack are propagated. This effect occurs at the method, class and package level.

If a single method naively passes all checked exceptions which it does not handle on to its caller, a proliferation of exceptions tends to occur on the way to the root of the call stack, and in the highest layer of a layered architecture, methods would tend to have just about every checked exception in the whole system in their list of thrown exceptions. The following artificial example shows how such an accumulation of exceptions occurs unless countermeasures are taken.

```
// BAD CODE: all checked exceptions are directly propagated
// to the caller, making the list of thrown exceptions very
// unwieldy
public void writeName (String name)
  throws IOException, ClassNotFoundException,
  NoSuchMethodException, NoSuchFieldException, SQLException
{
  Logger.log ("doSomething"); // throws IOException

  // uses reflection and throws ClassNotFoundException,
  // NoSuchMethodException, NoSuchFieldException
  PersistenceFramework.init ();

  // accesses the database and throws SQLException
  updateName (name);
}
```

This confronts calling code with a whole lot of possible exceptions, all of which must be handled or propagated. The only way to handle them in a single `catch` block is to catch `Exception` itself, catching all other exceptions as well.

The exceptions that are thrown are also at another level of abstraction than the rest of the method: While they give detailed information about what went wrong, they are not very helpful for understanding the significance of the failure. For example, if a failed initialization is signalled by throwing a `ClassNotFoundException` and client code handles it in this way, that may lead to surprising bugs if another method that is by some chance called close to the initialization also throws a `ClassNotFoundException` but for an entirely different reason.

Having the internally thrown exceptions in the `throws` clause of the method also breaks encapsulation because the list of exceptions publicly announces information about the kind of method calls that are made in the implementation. So changes to the implementation potentially affect the method signatures of all calling method, making the code much harder to modify or refactor.

At the class level, an additional effect further reduces comfort. Even if all methods in a class can throw only one checked exception each, using the class is painful if these exceptions are different. The reason is that client code often calls not only one method on an object but several, and adding a method call on an object which is already used is one of the more frequent changes performed on code. If each additional call means reworking exception handling, potentially far up the call stack, this adds significant effort to otherwise simple changes to the code and stands in the way of effortless refactoring.

The net effect is that the interface of a class is not cohesive if checked exceptions are growing rampant:

```
//BAD CODE: Different methods throw different exceptions,
// making usage of the class awkward
class NameManager {
  public String readName () throws IOException {
    ....
  }
  public void writeName (String name) throws SQLException {
    ....
  }
  public boolean isValidName (String name)
    throws ClassNotFoundException
  {
    ....
  }
}
```

A client using such a class is forced to catch several seemingly unrelated exceptions and duplicate exception handling code for each of the `catch` blocks. The exception classes are unrelated to each other as well as to the abstraction of the class interface, making the source code difficult to read.

```
// Code using NameManager. Writing and changing the code is
// awkward because different methods throw different
// exceptions
NameManager nameMgr = new NameManager ();
try {
  if (nameMgr.isValidName (name)) {
    nameMgr.writeName (name);
  }
}
catch (SQLException exc) {
  ....
}
catch (ClassNotFoundException exc) {
  .... // duplicated exception handling code
}
```

At the package level, this effect is even more pronounced. Clients using a package usually are not interested in its implementation details. They do not want to spend much effort understanding a wide variety of exceptions thrown under varying conditions.

So how can a simple and stable exception interface be achieved if a variety of exceptions occur that cannot be handled locally and therefore need to be passed to the caller?

Solution

Create a single checked exception and have all methods declare to throw only this. Internally, translate all checked exceptions to this new, homogeneous exception, for example using EXCEPTION WRAPPING to maintain the original information.

This makes the class or package simpler and more intuitive to use because clients need not worry about handling different kinds of exceptions. Instead, there is a single checked exception which is at the same level of abstraction as the interface which declares to throw it.

It is often handy to use a single homogeneous exception across several packages that are in some way related, for example because they belong to the same layer in a layered architecture. That makes the group of packages simpler to use for clients and underscores that they together make up a bigger something.

Often clients are not interested in details about a failure; it is sufficient for them to just know that an exception occurred. If clients need more detailed information about the kind of failure, SMART EXCEPTION or EXCEPTION HIERARCHY can be used to give clients additional details without giving up the single homogeneous exception.

The following source code illustrates the use of a single homogeneous exception for the `NameManager` example of the problem section; all checked exceptions are wrapped in a single `FailedBusinessOperation` (the definition for this checked exception is straightforward and was left out):

```
// Homogeneous Exception was applied: Now all methods
// throw a single checked exception which can contain
// the original exception
class NameManager {
  public String readName () throws FailedBusinessOperation {
    try {
      ....
    }
    catch (IOException exc) {
      throw new FailedBusinessOperation (exc);
    }
  }

  public void writeName (String name)
    throws FailedBusinessOperation
  {
    try {
      ....
    }
    catch (SQLException exc) {
      throw new FailedBusinessOperation (exc);
    }
  }

  public boolean isValidName (String name)
    throws FailedBusinessOperation
  {
    try {
      ....
    }
    catch (ClassNotFoundException exc) {
      throw new FailedBusinessOperation (exc);
    }
  }
}
```

This example illustrates the general way to implement HOMOGENEOUS EXCEPTION in a method. The body is enclosed in a `try` block which is followed by `catch` blocks for all checked exceptions that can occur. All other exceptions are handled by throwing an instance of the homogeneous exception. Of course the homogeneous exception can be thrown for other reasons than an exception further down the call stack.

Exception handling in code using the `NameManager` now is much simpler, and the code is easier to understand because the exception is at the same conceptual level as the rest of the interface of `NameManager`.

```
// Code using NameManager. The homogeneous exception makes
// the usage simpler and clearer
NameManager nameMgr = new NameManager ();
try {
  if (nameMgr.isValidName (name)) {
    nameMgr.writeName (name);
  }
}
catch (FailedBusinessOperation exc) {
  ....
}
```

Wrapping all checked exceptions of an interface in a single, homogeneous exception has several benefits. First of all, there is only a single exceptions which clients need to address.

The interface is also more homogeneous as a whole because the exception is at the same conceptual level as the method calls; calls to the methods and exception handling in client code fit together and make sense together.

Using a single homogeneous exception also encapsulates implementation details: If the implementation of a method is changed so that an additional checked exception can occur inside the implementation, this additional exception is simply also wrapped in the homogeneous exception, and the method interface remains unchanged.

These benefits however come at a price which is typical of encapsulation in general. Encapsulation makes the interface simpler to use and more robust against implementation changes, but the interface becomes less transparent. Clients have less insight and control over what is actually happening inside the method.

If an interface is very mature and well designed, this is often not a problem - it is for example not necessary to understand the implementation details of the `java.io` package in order to use it. But if the interface is less mature, encapsulation can come in the way of development, forcing at least an ongoing refactoring of the interface.

Therefore applying this pattern poses an additional responsibility on the specification of interfaces, especially if they are published in some way so that they become difficult or impossible to change.

SMART EXCEPTION

If clients need details in order to handle an exception which is caused by a CHECKED SERVER PROBLEM and there is a fixed number of different cases that need to be handled differently, create an ENUMERATION CLASS (53) for these cases and give the exception a field of the enum type.

Problem

Clients catch and handle exceptions based on the exception class; the parameter of the `catch` clause specifies an exception class, and it catches only those exceptions which are assignable to this exception class. This allows client code to handle exceptions selectively. Exceptions with different classes can be handled in different places, using different handlers.

But sometimes a client handling an exception needs additional information to handle the exception.

For example, a login may be denied for several reasons - the id/password combination may be wrong, the authentication server may be down, or the user account may be temporarily disabled. And if login failure is propagated using an exception, a client handling the exception would need to handle these cases differently.

In this case, the following forces apply. All different failure modes are always caught and handled in the same place. There is no situation in which only a subset of the cases needs to be handled and other cases need to be propagated further up the call stack.

All different failure modes also originate in roughly the same place. The complete list of cases is under the control of the same part of the system, and it is not necessary that different subsystems can add new cases independently.

And client code handling the exception actually performs different operations based on the different cases. So there is conditional logic basing decisions on the differences between the states, and a simple message string in the exception is insufficient.

Solution

Create an ENUMERATION CLASS (53) with the different cases, and make the exception "smart" by giving it the enum class as a field. This allows client code to handle the exception differently based on the value of this field.

Some of the handling logic and the different behavior can be moved into the enum class, improving testability and making the actual handling code simpler.

For the example of the failed login, the enum class would look something like the following:

```
// Enum Class with possible reasons for a failed login
public class LoginFailedReason {
  // contains information if an immediate retry makes sense
  private final boolean _mayRetry;

  private LoginFailedReason (boolean mayRetry) {
    _mayRetry = mayRetry;
  }

  public boolean mayRetry () {
    return _mayRetry;
  }

  // complete list of all instances of this class
  public static final LoginFailedReason INVALID_PASSWORD =
    new LoginFailedReason (true);
  public static final LoginFailedReason SERVER_DOWN =
    new LoginFailedReason (true);
  public static final LoginFailedReason USER_DISABLED =
    new LoginFailedReason (false);
}
```

Some of the logic of handling, namely the information if an immediate retry of the login has any chance of success, is contained in the enum class.

The constructor is private, so the only instances of the class are those declared as `public static final` fields in the class itself. This provides a complete list of instances in one single place and makes these instances accessible by name[2].

The exception class itself holds a field with the reason the exception was thrown:

```
public class LoginFailedException extends Exception {
  private final LoginFailedReason _reason;

  /** Constructor takes a reason as a parameter to ensure
    * that handling clients can later base decisions on
    * this reason.*/
  public LoginFailedException (LoginFailedReason reason) {
    _reason = reason;
  }

  public LoginFailedReason getReason () {
    return _reason;
  }
}
```

The constructor of `LoginFailedException` takes a `LoginFailedReason` as a parameter, so whenever such an exception is created and thrown, a reason must be given.

This reason can then be queried by client code handling the exception, and decisions can be based on this reason.

```
// client code handling a failed login
try {
  .... // try the actual login
}
catch (LoginFailedException exc) {
  // all login failures are handled in the same catch block

  if (exc.getReason() == LoginFailedReason.SERVER_DOWN) {
    // different reasons can be handled differently in
    // a way which makes the conditional logic easy to read
    ....
  }
  ....

  // LoginFailedReason contains logic which now simplifies
  // handling
  if (exc.getReason().mayRetry()) {
    ....
  }
  else {
    ....
  }
}
```

The conditional logic in the exception handling code is comparatively easy to read and understand because all instances of the enum class have descriptive names. The logic which was moved into the `LoginFailedReason` class also helps to keep the exception handling code clean.

The only liability that a Smart Exception incurs is the effort of creating - and later maintaining and understanding - the additional enum class. This overhead must be weighed against the benefit of simpler exception handling code and code which is clearer and structured better.

---

2. In order to make this implementation serializable, a readResolve method must be added, otherwise object identity will not be guaranteed. For details, see the description of Enum Class.

EXCEPTION HIERARCHY

If only a HOMOGENEOUS EXCEPTION is declared to be thrown but clients need to handle some cases specifically, create an inheritance hierarchy of subclasses of the homogeneous exception and throw subclasses. This allows clients to treat all exceptions homogeneously as instances of a single exception class but also gives clients the option to treat specific exceptions in a distinct way.

Problem

One key benefit of using HOMOGENEOUS EXCEPTION is that clients are not forced to deal with details of which exception is thrown when, they need to address only a single type of exception and can ignore all further details. That is a big benefit which most clients will want to enjoy most of the time.

But there are situations in which a class or even a method can fail in different ways which at least some clients need to handle differently. For example a read operation from a file can fail either because the end of the file was reached or because of a more general I/O problem. For some clients the end of the file is not a failure but rather the sign that they are finished, so they need to differentiate between the two cases.

So on the one hand, some clients want to treat all exceptions indiscriminately. For such clients it is important that as far as they are concerned, the single homogeneous exception is the only exception which is thrown.

But other clients need to handle only some of the possible failures and let others propagate up the call stack. For example, code reading from a file might want to handle the case that the end of the file was reached but let failure due to a general I/O problem propagate up the call stack. In such a situation, SMART EXCEPTION is not a natural fit although it supports clients in executing different code based on the details of the exception.

Knowledge about different ways in which methods can fail are often spread across many classes and potentially many developers or even teams, especially if a single homogeneous exception is used for several packages in a big system. If all the special failure cases were coded into a single class, this class would easily become harder and harder to change and extend because every change would affect many different parts of the system. In such a situation, SMART EXCEPTION is not a good solution either.

On the other hand, some clients handle different failure modes differently, so a field with a message string containing the details is not sufficient.

Solution

Create an inheritance hierarchy of exceptions with the HOMOGENEOUS EXCEPTION as a common superclass. This allows all methods to declare just the homogeneous exception, and disinterested clients can choose to treat all exceptions uniformly.

However it allows every operation to provide optional additional information about the cause of failure by throwing one or several subclasses of the homogeneous exception. In the example of the `read` operation from a file, this might mean throwing an instance of `IOException` itself for general I/O problems but creating a subclass `EOFException` of `IOException` for the case that the end of the file was reached.

This additional information is purely optional - the method declares to throw only `IOException`, so clients can entirely ignore the inheritance hierarchy if they choose to - but it is possi-

ble for a client to catch just one special case and let the rest propagate up the call stack. The following sample code shows this for a method reading from a `DataInputStream`.

```
// This method reads doubles from a DataInputStream until
// the end of the file is reached. Only the subclass
// EOFException of IOException is caught and handled, all
// other IOExceptions are propagated.
// The read methods of DataInputStream have IOException as
// a homogeneous exception, and this method shows how special
// failures can be handled selectively
public Collection readDoubles (DataInputStream dis)
  throws IOException
{
  final Collection result = new ArrayList ();

  try {
    while (true) {
      // this loop is terminated by an EOFException
      result.add (new Double (dis.readDouble ()));
    }
  }
  // EOFException is a subclass of IOException. Only this
  // subclass is caught, all other IOExceptions are
  // propagated up the call stack
  catch (EOFException exc) {
    // do nothing: it is just the end of the file
  }

  return result;
}
```

It is possible for a class or package to provide a new subclass of the homogeneous exception which is specific for this class of package. That makes it possible to add special failure modes which can be handled separately to the single homogeneous exception in a decentralized fashion; in order to add a new subclass, it is not necessary to make any change whatsoever to existing classes, and the new exception can be located in the same package that throws it so that not even a different package is affected.

If a method declares to throw just a homogeneous exception but also throws a subclass to express a special kind of failure, it is a good idea to document this behavior. A good way to do that is to use several `@exception` tags in the javadoc comment of the method, one for each possible class that is thrown. That ensures that the information is available in the generated HTML documentation in a standardized place.

This shows the main liability of an EXCEPTION HIERARCHY, though: It does not allow the compiler to perform static checking. Javadoc comments can get outdated just as all kind of comments can, and the method interface itself does not express that a subclass is thrown for a special case. That is the price one must pay for having a homogeneous exception which allows clients to ignore the differentiation if they choose to.

TUNNELING EXCEPTION

If a method signature can not declare checked exceptions - for example in a callback in a framework - but the implementation needs to throw checked exceptions because of a CHECKED SERVER PROBLEM, then create a subclass of `RuntimeException` which can hold a reference to another exception and throw instances of this unchecked exception.

Problem

Libraries are intended for use in a variety of contexts and therefore their interfaces need to be generic; the same is true for infrastructure code that is widely used throughout an application. This causes a problem if the library classes[3] are subclassed and methods are overridden and the subclass can fail in a way that would usually warrant throwing a checked exception.

The overriding method can not throw a checked exception because the subclass method can not declare to throw more exceptions than declared in the superclass as shown in the following code sample.

```
// DOES NOT COMPILE because the run method in Runnable does
// not declare any exceptions so neither must the overriding
//method
class SpecificRunnable implements Runnable {
  public void run () throws IOException {
    .... // code that performs IO
  }
}
```

This situation often occurs in callbacks where an interface or an abstract class is implemented and passed to a library method in order to be "executed" there. If a checked exception occurs in the overriding method, it can not be handled locally in a meaningful way.

But the checked exception which is thrown in a callback can not be simply propagated "through" the library to surrounding code either, and since the method signature is part of library code it can not be extended to declare to throw specific checked exceptions.

Solution

Create a class `TunnelingException` as a subclass of `RuntimeException` which can hold a checked exception; if an exception occurs, create an instance of `TunnelingException` around it and throw it instead. Since it is an unchecked exception, it can "tunnel" through the limited method signature in the superclass or even surrounding library code[4].

Surrounding client code can then catch the `TunnelingException`, extract the contained original exception, and handle or throw that. If in a specific situation client code can be sure that no exception is tunneled out, the corresponding `catch` block can of course be left out.

3. or interfaces. Both classes and interfaces impose the limitation that an overriding method in an inheriting class may not throw more checked exceptions than originally declared, and for the scope of this pattern, "class" and class specific terminology is used to include interfaces as well unless otherwise stated.
4. The name is borrowed from quantum mechanics where the tunnel effect denotes the phenomenon that particles can under certain circumstances move through walls which would be impenetrable to them according to classical physics.

The class `TunnelingException` can be declared as part of the library and for all callbacks. The following code shows a typical implementation.

```
public class TunnelingException extends RuntimeException {
  private final Exception _inner;

  public TunnelingException (Exception inner) {
    _inner = inner;
  }

  public Exception getInner () {
    return inner;
  }
}
```

The following source code illustrates how a checked exception can be tunneled through framework code. It assumes that there is a `CommandExecuter` that takes an implementation of `Runnable` as a parameter and calls the `run` method of it:

```
// define the callback to be executed
Runnable cmd = new Runnable () {
  public void run () {
    ....
    // The IOException does not appear in the signature
    // of the run method, so throw a TunnelingException
    // instead
    throw new TunnelingException (new IOException ());
  }
};

try {
  // the command is executed by a library class which is
  // generic and therefore does not have a checked exception
  // in the signature of the command it takes
  CommandExecuter.execute (cmd);
}
catch (TunnelingException exc) {
  // A TunnelingException means that a checked exception
  // was thrown in the callback implementation

  // We know that IOException is the only checked exception
  // which was can be thrown in this callback, therefore the
  // type cast is safe
  IOException ioExc = (IOException) exc.getInner ();
  .... // handle the IOException
}
```

The implementation of `Runnable` wraps occurring exceptions in the `TunnelingException` and throws it. Surrounding code can catch this and extract and handle the original exception.

Using TUNNELING EXCEPTION yields several benefits. First of all, it allows the overriding class to notify clients of failure rather than silently ignoring it and faking success. This is of course the key reason for using a tunneling exception in the first place.

The pattern even allows an object to pass all details about the failure to surrounding code, no information is lost. The actual exception object that represents the failure is preserved and made available to clients, including its message text and stack trace. That is true even if the

exception did not originate in the overridden method itself but in another method which is called from it.

All this is achieved without the necessity for the method signature to declare specific exceptions. And exception propagation is optional: calling code can catch and handle exceptions if that is necessary, but other clients are not forced to do so in the absence of exceptions.

There are however some liabilities and limitations as well.

The main liability of the pattern is the loss of compile time checking which is a direct result of the use of an unchecked exception. It is possible for a client to forget to handle a tunneling exception, especially as code evolves.

This is usually an advantage because it does not force all clients to implement senseless empty `catch` clauses - which would introduce potential for bugs of their own - but it does introduce some potential for careless mistakes.

This sort of bug can only be caught at runtime, although thorough testing with JUnit can help detect it. A SAFETY NET can at least ensure that an uncaught `TunnelingException` does not go unnoticed at runtime.

The pattern also relies on at least some cooperation of library code through which the exception is supposed to tunnel insofar as there must not be any indiscriminate `catch` clauses for all `RuntimeExceptions`. If the library is designed with the tunneling of exceptions in mind, that is not a problem, but with third-party libraries there is no guarantee that it will work.

There is a variant of this pattern if the library operation which performs the callback can itself fail with a CHECKED SERVER PROBLEM. In such a situation, exceptions can be tunneled in this checked exception rather than an unchecked dedicated `TunnelingException`. Client code must deal with the checked exception anyway so there is no additional handling overhead involved, but the danger of forgetting to handle the tunneling exception is removed.

This approach is for example taken by the SAX standard for XML handling. The `parse` method of `org.xml.sax.XMLReader` can throw the checked `SAXException` which can optionally act as a wrapper of another exception. The callback interface `ContentHandler` declares `SAXException` for its methods, allowing implementations to tunnel exceptions through the `XMLReader`.

UNHANDLED EXCEPTION

If a method has a CHECKED SERVER PROBLEM and needs to throw a checked exception but you would rather finish implementing the method first instead of immediately implementing comprehensive handling code for the exception, do not implement an empty `catch` clause even temporarily but rather wrap the exception in a generic unchecked `UnhandledException`.

Problem

Writing code which handles exceptions is both necessary and distracting.

If a method is called which can throw a checked exception, a corresponding `catch` clause must be implemented somewhere or the compiler will reject the code. This is usually an advantage since it helps prevent careless mistakes, but it has a tendency to distract the focus from the train of thought the developer was following. If you are implementing new functionality, you often want to finish that before seriously switching your attention to handling the exceptions.

But the compiler will reject the class until the exception is either declared in the `throws` clause or caught.

If the exception is added to the `throws` clause of the method that may work for the class itself, but it will usually make the compiler reject other methods which call this one; they suddenly need to handle an additional exception. So propagating the exception in general requires significant work in other classes.

Implementing code that handles the failure which caused the exception also often requires thought, luring developers into making ad hoc implementations of `catch` blocks, sometimes even empty, just so they can finish implementing the functionality they were working on.

Of course they make a resolution to come back later and clean the mess up, but the human weakness of forgetting good resolutions means that this does not always happen. The result is at best exception handling which is less than optimal, at worst some exceptions simply disappear and make the system behave in subtly unexpected ways.

How can this dilemma be resolved?

Solution

Declare an unchecked `UnhandledException` that acts as a wrapper for an arbitrary exception.

The following listing shows a typical implementation of an `UnhandledException`; such an implementation can be stored in a library and reused without change across systems.

```
import java.io.*;

/** intended for ad hoc handling of exceptions to help
  * postpone implementing proper handling without the
  * danger of it being forgotten.
  */
public class UnhandledException extends RuntimeException {
  private final String _cause;

  public UnhandledException (String message, Throwable cause) {
    super (message);

    // store the stacktrace of the cause in a string field
    final StringWriter sw = new StringWriter ();
    cause.printStackTrace (new PrintWriter (sw));
    _cause = sw.toString();
  }

  public String getMessage () {
    return "Unhandled Exception: " + super.getMessage () +
      "\n  " + _cause;
  }
}
```

The constructor takes a message string in addition to the `Throwable` which is being wrapped so that every `UnhandledException` contains a short explanation.

The `getMessage` implementation prints both the message and the stacktrace of the original `Throwable` so that all details about the cause are presented when either `getMessage` or `printStackTrace` are called on the `UnhandledException`.

With this class readily available, when you want to postpone implementing handling of an exception you wrap the exception in an `UnhandledException` and throw it instead. That is

particularly useful for checked exceptions but can serve as a mnemonic in cases where unchecked exceptions need to be handled as well.

That makes it possible to first finish a train of thought before giving full attention to exception handling. It also significantly reduces the risk of forgetting to handle the exception at all because any usage of UnhandledException serves as a reminder that the code is unfinished. Such usages are easys to spot and identify both by humans, for example in a code review, and by tools.

The following example shows how the `UnhandledException` can be used to postpone addressing `IOExceptions` while writing file handling code.

```
// intermediate stage of the program: UnhandledException
// is used to postpone handling IOExceptions but needs to
// be removed before the code is finished
try {
  // file access can throw IOException
  FileReader file = new FileReader ("config.ini");

  .... // actually read and process the file
}
catch (IOException exc) {
  // throws an unchecked exception instead of IOException
  throw new UnhandledException ("reading config.ini", exc);
}
```

Using an instance of UNHANDLED EXCEPTION has several benefits.

It makes the source code immediately compilable, and when an exception is actually thrown, there is no danger of it silently disappearing - the `UnhandledException` is bound to be handled somewhere further up the call stack. So the details of the failure are available for debugging even while the final exception handling has not been implemented yet. To have control over exception handling in such situations, a SAFETY NET is useful.

Usages of `UnhandledExceptions` are also easy to find both by humans and by tools. So although the `UnhandledException` is an unchecked exception and therefore easy to forget - which is why it was introduced in the first place - there is little danger that it will be forgotten in the long run and accidentally left in release code.

To make sure that all usages of `UnhandledException` are replaced by meaningful exception handling in due time, one can use simple “find in files” functionality to monitor its use in the whole system.

There is however the danger that these possibilities are not used and the temporary solution remains in the code. After all, there is no immediate pressure to remove it, and other things are prioritized by management... It is important to keep this potential danger in mind; in a context where such things tend to be the rule rather than the exception, it is better to avoid this pattern.

The other main liability of the pattern is that means writing some code which will be thrown away for the final system. Often this is more than made up for by the benefit of being able to address one problem at a time, but sometimes it is less effort to just immediately implement the real handling code.

UNCHECKED CLIENT PROBLEM

Let a THROWING SERVER throw an instance of a subclass of `RuntimeException` for failures that are directly or indirectly due to bad input data. This ensures that the failure will not go unnoticed but does not force every client to explicitly handle the exception.

Problem

Many methods depend on client cooperation for their successful completion, either through passing in parameters or initializing the object. As a result, they differentiate between "permitted" states and parameter values and "forbidden" ones that prevent successful completion.

An example of such an assumption is the constructor of a `Name` class that takes a first and last name as two strings, neither of which may be `null` or empty. If one or both parameters are invalid, the constructor cannot perform its operation successfully.

The following code shows a client trying to create a `Name` instance, passing invalid parameters to the constructor.

```
// How should the Name constructor react to invalid input
// which prevents it from completing successfully?
Name name = new Name (null, null);
```

How should this sort of failure be treated? Obviously, it is not an option to silently fail and fake success.

If a checked exception is thrown, that forces every client to explicitly handle it, adding effort to the usage of the class. The benefit of making the mode of failure explicit which comes with checked exceptions does not pull its full weight here because it is in the clients' hands to avoid the exceptions in the first place by keeping their side of the contract. If indeed this sort of failure occurs, it is more in the nature of a bug appearing at runtime than of a normal runtime failure.

But if neither covering failures with the veil of mercy nor making the innocent suffer with the guilty is a desirable solution, how should such client-induced failures be treated?

Solution

Throw an instance of a subclass of `RuntimeException` if an operation fails due to incorrect client behavior.

This ensures that the failure neither goes unnoticed nor forces immediate clients to explicitly address the problem. Clients which find it more convenient to handle the exception than to check the validity before the call can do so, but others are not forced to bother.

In the following example, the constructor of the `Name` class throws an `IllegalArgumentException` if one of the parameters is `null` or empty.

```
// The Name constructor throws an unchecked exception if it
// cannot complete successfully so that clients are notified
// but are not forced to explicitly address failure
class Name {
  private final String _firstName;
  private final String _lastName;

  public Name (String first, String last) {
    if (first == null || first.length() == 0)
      throw new IllegalArgumentException ("first name");
    if (last == null || last.length() == 0)
      throw new IllegalArgumentException ("last name");

    _firstName = first;
    _lastName = last;
  }

  .... // implementation of accessors and logic
}
```

This ensures that invalid parameters are rejected but does not force clients to address potential failure directly if they are confident it can not happen.

Although immediate client code should not be forced to handle such exceptions - after all, that is the reason why they are unchecked - they must be handled somewhere, at least so that their details are written to a log file to trigger a bug report and help fix the bug. A SAFETY NET provides confidence in ignoring `RuntimeException`s locally.

The package `java.lang` contains a wide variety of subclasses of `RuntimeException`, and often it is sufficient to use one of them instead of creating a new one. Especially `IllegalArgumentException` and `IllegalStateException` with an expressive message string are often a good first choice.

If on the other hand handling code is to be enabled to handle different kinds of `RuntimeException` differently, it is of course possible to provide specific subclasses of `RuntimeException` and throw them. Since more technical problems are more or less covered by the exceptions from `java.lang`, additional `RuntimeException` classes will typically be domain specific.

But introducing specific subclasses of `RuntimeException` makes code throwing them harder to read while the benefit is often more perceived than real - more often than not all `RuntimeException`s are indiscriminately handled anyway.

EXCEPTION WRAPPING

If an exception is converted into a HOMOGENEOUS EXCEPTION, wrap the original exception inside the new one so that all details about the original exception are preserved.

Problem

Catching one exception and throwing another instead, for example because HOMOGENEOUS EXCEPTION is applied, hides implementation details - which is one reason why it is done. For

debugging purposes, however, these details about the root cause of the exception are desirable because they help understand the problem.

The following example illustrates this. The `executeOperation` method performs a complicated operation in the course of which a number of different exceptions can occur. These were united into a single exception so that exception handling for clients of the method is easier[5].

```
// BAD CODE: the method throws only one exception which is
// at the semantic level of the operation, but all details
// about the root causes of the exceptions are lost.

void executeOperation () throws OperationFailedException {
  try {
    .... // complicated code which can throw a variety of
         // exceptions
  }
  catch (Exception exc) {
    throw new OperationFailedException ();
  }
}
```

However, it discards all details about the root cause of the exception. And while these are not necessary for handling the exception it makes debugging harder. The type, stacktrace and message of the exception that is passed further up the call stack do not directly correspond to the context in which the problem originated.

On the one hand there are forces working towards hiding the details of the original exception so that the interface is stable even in the face of implementation changes and implementations can be exchanged; on the other hand there is the need to make all details of the original exception available for debugging.

Solution

Allow an exception to optionally take a reference to the exception that caused it as a constructor parameter. Override the `getMessage` method so that it yields the type, message and stacktrace of the causing exception. This presents exception handling code a homogeneous view of the exception but gives humans access to all details of the root cause of the exception.

Overriding the `getMessage` method has two big benefits. Firstly, the wrapped information is logged almost automatically since the message string is printed by both the `toString` and the `printStackTrace` methods of `Throwable`, so there is no additional effort to make the information available.

Secondly, using the `getMessage` method in this way ensures that the information about the root cause is kept even if the exception is wrapped several times. Every wrapping exception includes the message string of the wrapped exception in its own message string, so no information is lost.

The wrapping exception should keep no reference to the original exception but rather store its details in a string. This allows serializing and deserializing of the exception across process boundaries as is for example used by RMI.

It is possible that the JVM which is deserializing the exception does not have the class definition of the wrapped exception in its classpath so that a `ClassNotFoundException` is

5. The definition of OperationFailedException was left out because it is arbitrary at this point.

thrown at runtime. This runtime dependency can be avoided by storing only a string representation of the causing exception rather than the exception itself.

The following source code shows a typical definition of a wrapping exception.

```
// Implementation of OperationFailedException which takes
// a causing exception as a parameter
import java.io.*;

class OperationFailedException extends Exception {
  private final String _cause;

  public OperationFailedException () {
    _cause = "";
  }

  public OperationFailedException (Throwable cause) {
    // store the stacktrace of the cause in a string field
    final StringWriter sw = new StringWriter ();
    cause.printStackTrace (new PrintWriter (sw));
    _cause = sw.toString();
  }

  public String getMessage () {
    return "Operation failed:\n  " + _cause;
  }
}
```

A method which throws this exception can now pass an exception which causes the `OperationFailedException` to the constructor of the new exception so that all details are preserved for debugging.

```
// executeOperation revisited: All details about the
// root cause are contained in the thrown exception

void executeOperation () throws OperationFailedException {
  try {
    .... // complicated code which can throw a variety of
         // exceptions
  }
  catch (Exception exc) {
    // the causing exception is now passed on to the new one
    throw new OperationFailedException (exc);
  }
}
```

JDK 1.4 introduces direct support for this pattern. The class `Throwable` and many of its subclasses (including `Exception`, `RuntimeException` and `Error`) have constructors that take a causing exception as an argument. The data of this stored exception is used for both `getMessage` and `printStackTrace`.

There is one major difference between the implementation in the JDK and the one suggested here: The class `Throwable` stores a reference to the actual causing object. That gives handling code access to this object but can lead to problems if exceptions are serialized and deserialized in different address spaces.

This feature is called “chained exception” in the JDK documentation, but that name is misleading because SUN uses the same term for a slightly different concept in the description of `java.sql.SQLException`[6].

Wrapping the causing exception in this way has the benefit of presenting client code with a cleanly encapsulated view of the called class, effectively hiding all implementation details. It also makes all details about the root cause of the exception available for debugging, even if several such conversions take place between the throwing and the handling.

But there are some minor liabilities as well because the messages of the exceptions can become quite long, especially if the root exception is wrapped several times[7].

Firstly, if the exceptions are logged to a file, the information about the wrapping significantly increases disk consumption, potentially causing trouble during development when many exceptions occur in a short time.

Secondly, the long combined stacktraces of the exceptions can make log files hard to navigate.

SAFETY NET

Install a default handler for `Throwables` that accidentally are not handled regularly. This rounds off the EXPRESSIVE EXCEPTION INTERFACE of an entire system by making explicit how unexpected exceptions are treated, for example forgotten instances of TUNNELING EXCEPTION and UNHANDLED EXCEPTION.

Problem

No matter how much attention is paid to exception handling during the development of a system, there can always be places where it is forgotten. For checked exceptions the compiler makes sure that this does not happen, but `RuntimeExceptions` can slip through - including those introduced by TUNNELING EXCEPTION and UNHANDLED EXCEPTION. The same is true of `Errors`.

The default behavior of the JVM is to print the stack trace of an otherwise unhandled exception[8] to `System.err` and end the thread[9] which is better than crashing the system or aborting the application with a core dump but nonetheless is not very helpful for real-world applications.

Exceptions that are simply ignored potentially result in corrupted internal structures, leaving the system in an undefined state - especially since an application thread is terminated by the JVM.

---

6. java.sql.SQLException also supports linking several exceptions, but there the idea is to propagate the original exception and append other exceptions that occurred later, whereas here the idea is to hide the original exception and propagate a wrapper around it.
7. The memory overhead introduced by EXCEPTION WRAPPING are usually not an issue because the memory is used only temporarily. An upper limit for the amount of memory taken up is *<Cumulative size of wrapped exceptions>*<Max. number of concurrent tasks>* which even for very large systems is only in the order of a couple of MB.
8. For the rest of this pattern, “exception” is used to denote all throwables to make the text easier to read.
9. Many frameworks provide some default handling, alleviating some of these consequences. Swing and RMI for example ensure that no thread is accidentally terminated, and many application servers additionally log the exception.

There is also the problem that an exception just written to `System.err` easily goes unnoticed, creating the impression that an operation terminated normally without however performing its designated task. This can lead developers on a wild-goose chase for bugs that are actually in a different part of the system.

How can you make sure that unhandled exceptions do not go unnoticed and trigger a default handler?

Solution

Install a default handler as a safety net which is called for all exceptions that have slipped through the application's regular handling mechanisms.

The system should be built in such a way that there is no known way an exception could possibly reach the safety net; all expected exceptions should be handled regularly.

This introduces a distinction between *expected* and *unexpected* exceptions.

Expected exceptions are those which the developer thought of and which are therefore handled in some way. For them, the system ensures that all necessary clean-up is performed, and they leave the system in a well-defined state. These exceptions are typically handled based on their type, and when a system is released to the customer, all exceptions should be in this category.

Unexpected exceptions are all those that are not in the "expected" category. For them there is by definition no well-defined and specific handling, so they potentially leave the system in an undefined state. Those are the exceptions for which a default handler is necessary as a safety net.

Due to the unspecific nature of these exceptions and the potential severity of their cause, the safety net can and should not do much beyond logging the exception. In a critical server system, triggering a message to the system administrators would be a typical action.

But how do you implement and register such a safety net?

If the creation of all threads is under direct control of the application, the best way is to create a subclass of `java.lang.ThreadGroup` and override the `uncaughtException` method.

```
// This ThreadGroup acts as a default handler for exceptions
// which are not otherwise caught
class SafeThreadGroup extends ThreadGroup {
  // every thread group needs a name
  public SafeThreadGroup (String name) {
    super (name);
  }

  public void uncaughtException (Thread t, Throwable e) {
    if (! (e instanceof ThreadDeath)) {
      .... // handling code goes here
    }
  }
}
```

Then all threads are created with an instance of this thread group.

```
// a single instance of SafeThreadGroup is enough for
// the purposes of the Safety Net
private final SafeThreadGroup globalThreadGroup =
  new SafeThreadGroup ("global safe ThreadGroup");

....

// all threads are created with the instance
// of SafeThreadGroup
Thread thread = new Thread (globalThreadGroup,
  new Runnable () {
    public void run () {
      .... // the thread implementation goes here.

      // all exceptions thrown by this method trigger
      // a call to uncaughtException in the thread group.
    }
  });
```

This causes a call to `uncaughtException` whenever the `run` method of a thread throws an exception.

Implementing a SAFETY NET using a customized `ThreadGroup` with an overridden `uncaughtException` method provides full control over the actual handling of the exception. There is however the severe limitation that this approach only works if the application has direct control over the creation of all threads. Since this is not the case if frameworks like Swing or RMI are used, most systems need an alternative.

As noted above, the default way that Java handles an exception is to print it to `System.err`, and most frameworks leave this behavior unchanged so that another way of providing a SAFETY NET is possible.

It is based on the assumption that all output to `System.err` means that something went terribly wrong. If that is so, why not implement the SAFETY NET as an `OutputStream` and register it as `System.err`?

```
class SafetyNetOutputStream extends OutputStream {
  public void write (byte b[]) throws IOException {
    ..../// log the message and take additional action
  }

  public void write (byte b[], int off, int len) throws IOException {
    ..../// log the message and take additional action
  }

  public void write (int b) throws IOException {
    ..../// log the message and take additional action
  }

  .... // override flush() and close() as needed
}
```

Then `System.err` must be set to this class.

```
// An instance of SafetyNetOutputStream is registered as
// System.err so that all output to to System.err triggers
// a global failure handler
public static void main (String[] args) {
  ....

  System.setErr (new PrintWriter (
    new SafetyNetOutputStream ()));

  ....
}
```

Whenever the `printStackTrace` method is called on an exception - or any other output to `System.err` is done - one or several of the `write` methods of the `SafetyNetOutputStream` are called, making sure that the output is logged and triggering any additional emergency behavior.

The main advantage of this approach is that it works in the presence of many frameworks where using a special `ThreadGroup` is not an option. This comes at a price, however:

- `System.err` is occupied. Each and every output to `System.err` is treated like an unexpected exception, triggering serious consequences.
- Failures are not atomic. A single unexpected exception may result in several calls to several of the write methods of the `SafetyNetOutputStream`. That means that all handling code is executed several times for a single unexpected exception unless that is heuristically prevented, for example through a time-out.
- Information about the exception itself is not available. The exception which triggered the handler is reduced to one - or several - sequences of bytes containing a string representation. Specific handling based on the exception type is not possible.

Despite these limitations, registering a safety net as `System.err` ensures that all exceptions are at least logged and trigger additional handling.

Conclusion

Exceptions are a powerful tool, and like many powerful tools they can be used to disadvantage. But if the pattern language presented in this paper is applied to organize exception use in the large then exceptions will prove to be a natural and helpful part of the system, providing a clean and potent way to propagate knowledge of failure of operations.

Acknowledgements

I would like to thank Frank Buschmann who shepherded this paper for his many suggestions and questions; he helped to improve this paper a great deal. Further thanks for fruitful discussions go to Pascal Costanza, Kevlin Henney and Jan Hermanns and the participants of the Writers' Workshop at EuroPloP 2002. Special thanks are due to Jan Leßner for coming up with the stream-based implementation of SAFETY NET and helping design robust exception handling into the first large Java system I ever worked on.