



## About effective Exception Handling

for distributed enterprise applications

by Dino Celovic and Nader Soukouti

### Abstract

This article attempts to explain how to effectively handle exceptions in Distributed Enterprise Applications by clearly separating exceptions related to business issues from those related to technical and infrastructure problems. We are not presenting a particular exception handling framework, other than the one provided by the Java language. Further more, we think that such frameworks are not necessary - a good layered architecture and a well defined exception handling policy suffice.

### Introduction

The intended audience should already be familiar with exception handling.

The first part of this article revisits the purpose and benefits of exception handling by clarifying exception handling *concepts* together with *responsibilities* of different actors on the project.

The second part will put those concepts in the context of a sample J2EE application, and explain on the example how to best apply them. There, we will introduce and try to justify the use of:

- ✚ **ApplicationException**-s, for business related problems,
- ✚ **SystemException**-s, for technical and infrastructure related problems,
- ✚ **Other RuntimeException**-s, due to bugs,
- ✚ **Assertions**, for defensive coding, without being defensive.

### Background

#### *What is Exception?*

Jim Cushing [ref1] wrote:

*"Exceptions in Java provide a consistent mechanism for identifying and responding to error conditions. Effective exception handling will make your programs more robust and easier to debug. Exceptions are a tremendous debugging aid because they help answer these three questions:*

- ✚ **What** went wrong?
- ✚ **Where** did it go wrong?
- ✚ **Why** did it go wrong?

*When exceptions are used effectively, **what** is answered by the type of exception thrown, **where** is answered by the exception stack trace, and **why** is answered by the exception message. If you find your exceptions aren't answering all three questions, chances are they aren't being used effectively."*

Technically speaking, "An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program.", from [ref2].

Later in this article we will see that exceptions can also be used even when the system does not 'go wrong'.

#### *Why use exceptions?*

Exception handling allows to separate the main application logic, from logic that handles exceptional cases.

"Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code.", from [ref2].

The *separation of concerns* is one of fundamental OO principles and adds clarity to programs in general [ref3]. In the case of exception handling, it frees the *application logic* from having to think too much about how to act upon and report problems. Both, application logic and problem handling logic, become simpler, cleaner and clearer.

Contrary to what we could think, exceptions are not used for debugging only, or when things go wrong with the system. An example to that would be *Application Exceptions* that we will introduce later in this article: the user interacts with the system, but the system cannot complete the request because the *request* is invalid or incomplete. The system can inform the user about the problem using exceptions, even though the system works correctly.

#### Example:

Compare the two implementations of the same logic, one using exception, one not (taken from [ref2]):

Code	
<pre> errorCodeType readFile {     initialize errorCode = 0;     open the file;     if (theFileIsOpen) {         determine the length of the file;         if (gotTheFileLength) {             allocate that much memory;             if (gotEnoughMemory) {                 read the file into memory;                 if (readFailed) {                     errorCode = -1;                 }             } else {                 errorCode = -2;             }         } else {             errorCode = -3;         }         close the file;         if (theFileDidntClose &amp;&amp; errorCode == 0) {             errorCode = -4;         } else {             errorCode = errorCode and -4;         }     } else {         errorCode = -5;     }     return errorCode; } </pre>	<pre> readFile {     try {         open the file;         determine its size;         allocate that much memory;         read the file into memory;         close the file;     } catch (fileOpenFailed) {         doSomething;     } catch (sizeDeterminationFailed) {         doSomething;     } catch (memoryAllocationFailed) {         doSomething;     } catch (readFailed) {         doSomething;     } catch (fileCloseFailed) {         doSomething;     } } </pre>

Which one do you prefer?

## The concepts

### Throwing and catching

In the exception handling jargon we talk about *throwing* and *catching* exceptions.

#### Note

An exception is *thrown* when a condition, that prevents a software component from completing the requested task is encountered.

The component then creates an object (exception) that carries information about *what*, *where* and *why* things went wrong and hands (*throws*) the exception to the exception handling mechanism. As a consequence, the application logic flow is interrupted and by *rewinding the method call stack*, the callers are offered the possibility to do something about the problem. The **caller that thinks** it can do something helpful *catches* the exception and does his helpful thing. Note that there is no short way to come back to the execution point where the exception was thrown.

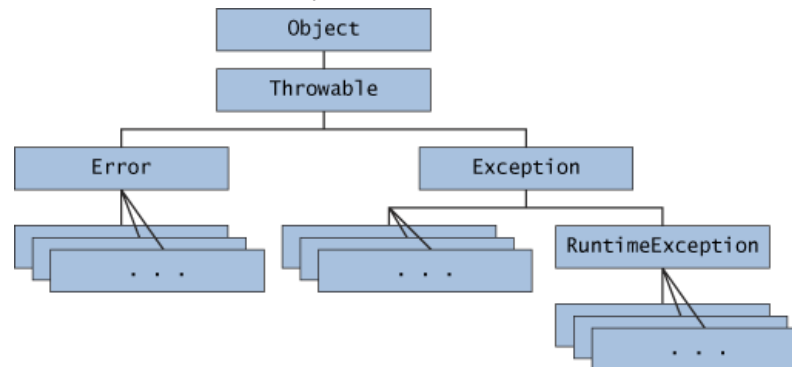
We can therefore identify two 'actors' in the exception throwing mechanism: the *thrower* and the *catcher*. They both have well defined and complimentary roles and, when used effectively, are a valuable aid in making applications more robust.

The difficulty is often in the '*caller that thinks*' bit. Exceptions are often caught where nothing helpful can be done. In the best case the catcher does something useless and re-throws the exception, in the worst case, the exception is caught and drowned - the problem is hidden by inappropriate exception handling.

## Exception types in Java

Depending on the nature of the problem which is at the origin of the exception, Java offers support for three major types of exceptions: checked exception, runtime exceptions and errors.

This is how those exceptions fit in the class hierarchy [ref2]:



The base of all exceptions is `java.lang.Throwable`. `Throwable` provides functionality common to all exceptions, such as exception nesting.

**Errors** occur due to severe runtime environment problems (virtual machine, EJB container...), that the system can do nothing about in order to recover from. A typical example is `OutOfMemoryError`: the system run out of memory. All the application can do is trigger a procedure to rescue live data, and then ask for help. Since errors are not predictable at compilation time and could occur at any point in the application, their handling is **not** enforced (checked) by the compiler. All errors inherit from `java.lang.Error`.

**Runtime exceptions** usually occur due to a bug in the code. Those problems are unpredictable as we cannot know in advance where the bugs are. Otherwise they wouldn't be bugs...

Because they are unpredictable and could be anywhere in the application code :-), the compiler does **not** enforce their handling.

From [ref2] on `java.sun.com`:

*"...Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or handle them in any way.."*

*Here's the bottom line: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception."*

All runtime exceptions inherit from `java.lang.RuntimeException`.

**Checked exceptions** are used for problems *predictable* at design time. They can be caused by the way the user interacts with the system, or by the system itself. In any case, we want the application logic to be **clearly aware** of those error possibilities and do something about it. The compiler enforces handling of those errors: software components either have to *catch* those exception, or *declare* that they are not catching it. In the latter case the calling code is aware that there is something to be done about the potential problem: catch or declare. And so on...

### Note

Checked exceptions are part of the method signature and are therefore integral part of the **contract** the class or interface has with its clients.

**AssertionError** error is part of the `java.lang.Error` hierarchy, but has a somewhat special meaning. Assertions allow programmers to add check points in their code in order to make sure that certain conditions, necessary to carry on with processing, are met. When that is not the case, the application throws an `AssertionError`.

Once the application is considered robust enough, *assertions checking can be deactivated* without altering application code: the virtual machine just ignores them. This is what makes them different from `RuntimeException`-s.

Assertions are thrown using `assert` keyword, not using `throw`. They were added to Java with version 1.4.

For example, you could use assertions to make sure that the calling code has passed all necessary and correct arguments to your method, thus avoiding exceptions such as `NullPointerException` further down in the logic, far away from the real cause of the problem:

### Code

```
public void debitAccount(AccountReference account, Amount amount)
throws AmountNotAvailableException{

    assert !Tools.isNegative(amount):"Amount cannot be negative. amount=" + amount;

    makeSureAmountIsAvailable(account, amount);
    DebitEvent event = createDebitEvent(account, amount);
    dao.saveEvent ();
}

```

In the example above we could have thrown `IllegalArgumentException` instead, but then we would not be able to deactivate it once the application is considered robust enough.

This joins the idea of throwing exceptions early (as recommended in [ref1]), without being penalised in terms of performance in a mature application.

#### Note

Assertions allow programmers to write *defensive* code in order to earlier uncover bugs, without diminishing performance once the application is considered stable.

Assertions however pollute application code and should be used only when they can be helpful in debugging.

#### Code

```
// some code ...
assert account!=null: "account cannot be null";
account.getAccountNumber ();
// more code ...

```

In the example above, assertion is not very helpful: if we removed it, *the next line* would throw `NullPointerException` anyway and it would be clear that the reason is `account==null`. The assertion above does not give us any extra information than the `NullPointerException` would.

### Grouping problems into categories

Exception handling mechanism gives you the possibility to act differently depending on the **type** of problem you are dealing with. Java API already defines few categories of exceptions: problems related to Input/Output operation derive from `IOException`, database access related problems derive from `SQLException` etc.

The mechanism used to categorise exceptions is *inheritance*. You can, and probably should, define categories of exceptions related to your application by designing the appropriate exception hierarchy. Your exception handling logic will then be able to selectively catch exceptions of different nature and take appropriate action in consequence. This is logically similar to:

#### Code

```
if(typeOfProblem instanceof FileNotFound){
    action1();
}
else if(typeOfProblem instanceof UnknownFileFormat){
    action2();
}
else if(typeOfProblem instanceof SomethingElse){
    letSomeoneElseHandleIt ();
}
etc...

```

The reason why the exception handling mechanism is based on exception type is not an accident: problems that can be encountered in a given functional domain, whether it is business related or technical, can often be spoken of in terms of their nature and they often nicely fit into a hierarchy.

Remember that the exception type carries information on *'what'* went wrong...

### Exception wrapping (nesting)

Instead of trying to deal with the problem, a component that catches an exception could only add some contextual information to the exception and re-throw it. This would help higher level catchers to better understand the problem.

The technique to use for adding contextual information to an exception is **exception wrapping**: the component creates a *new* exception that contains the original one, plus some extra info, and then re-throws it.

#### Note

Exception wrapping is like repositioning the problem in a context that is easier interpretable by higher level application layers, without losing information. It is also used to mask low level implementation issues from higher application layers.

Example: Imagine you are writing an application that you would like to sell to more than one customer. The business logic is the same, but some of your customers want to persist data in a relational database, some in a LDAP database. It would be nice not having to modify your business logic only because you change the data access technicalities!

Your DAO (data access layer) can catch `SQLException`-s, **wrap** them in a `PersistenceException` and re-throw them. Thus your business layer becomes unaware of the underlying persistence mechanics.

Support for exception wrapping was added to `java.lang.Throwable` since JDK 1.4.

## Responsibilities

### The wrongdoer

We mentioned earlier that the system would respond with an exception when *the system* goes wrong, or when *the user* tries to use the system in an inappropriate manner. The potential sources of the problem can therefore be **the system** or **the user**.

In both cases, we expect the system to *react gracefully*. If *the system* is in cause, we need to send a comprehensible message to the user and request help from the system support team. Or trigger an automated recovery procedure.

If *the user* is in cause, we inform the user about what he did wrong and explain why the system cannot fulfil his request.

We will see later that depending on *who* is at the origin of the exception will lead us to throwing different types of exceptions.

### The thrower

The thrower creates the exception object and is *expected* to create one that answers to all three questions: *what*, *where* and *how* the things went wrong.

The *thrower's responsibility* is to *detect* a problem and *inform* about it by instantiating and throwing an exception. The thrower does *not* know how to deal with the problem or how severe the problem is. That is why exception names such as `FatalException` do not make sense.

### The catcher

The *catcher's responsibility* is to do something meaningful with the exception in order to help the system recover gracefully.

The first question a software component should ask is "Should I catch the exception?" You can get the answer from another question: "Can I do something about the problem?" If yes - catch it; if non - don't, someone else will.

"If I don't catch it, who will?" The answer is: "It's not your problem, but someone will. Trust your application architect."

The software architect, by designing the right exception handling policy, together with the suitable exception hierarchy, will make sure exceptions do not slip through. This is where a clean layered architecture, and well distributed responsibilities come into play. Again...

Once the exception is caught, what can the component do?

There are two options:

- ✚ Deal with the problem: Trigger an alternative logic flow that will try to deal with the problem, or
- ✚ Wrap and re-throw, if you think that would help higher level catchers better understand the problem.

### The architect

The condition for building effective exception handling is to build effective application architecture in the first place. Earlier in this article we mentioned the *separation of concerns*. In an effective OO architecture, technical and business domain concerns are clearly assigned between layers, components and services. Exception handling is one of those concern.

### The business analyst

Exceptions can be thrown if certain conditions related to the **business domain** in question are not met.

Example:

- ✚ User tries to withdraw £500 from his account while the current account balance is £356 only: `throw AmountNotAvailableException`.

The business analyst needs to *analyse* those conditions and group them into *categories* of exceptions. A hierarchy of **ApplicationException**-s will be the result of that analysis.

Note that certain conditions could be verified before attempting to perform the operation and fail. See the discussion on data validation later in this article.

## The theory applied

This second part of the article will try to explain how to apply the principles presented above in order to take full advantage of the Java exception handling framework in your enterprise applications.

First we will argue the benefits of categorising all exceptions into following four categories:

- **ApplicationException**-s, for business related problems,
- **SystemException**-s, for predictable technical problems,
- **RuntimeException**-s, for bugs,
- **Assertions**, for defensive coding without being defensive.

Then we will give indications in which application layers those exceptions should be thrown, and in which they should be handled.

Exception handling being to a great extent dependent on the application architecture, we will present a sample J2EE application that allows us to better illustrate the ideas.

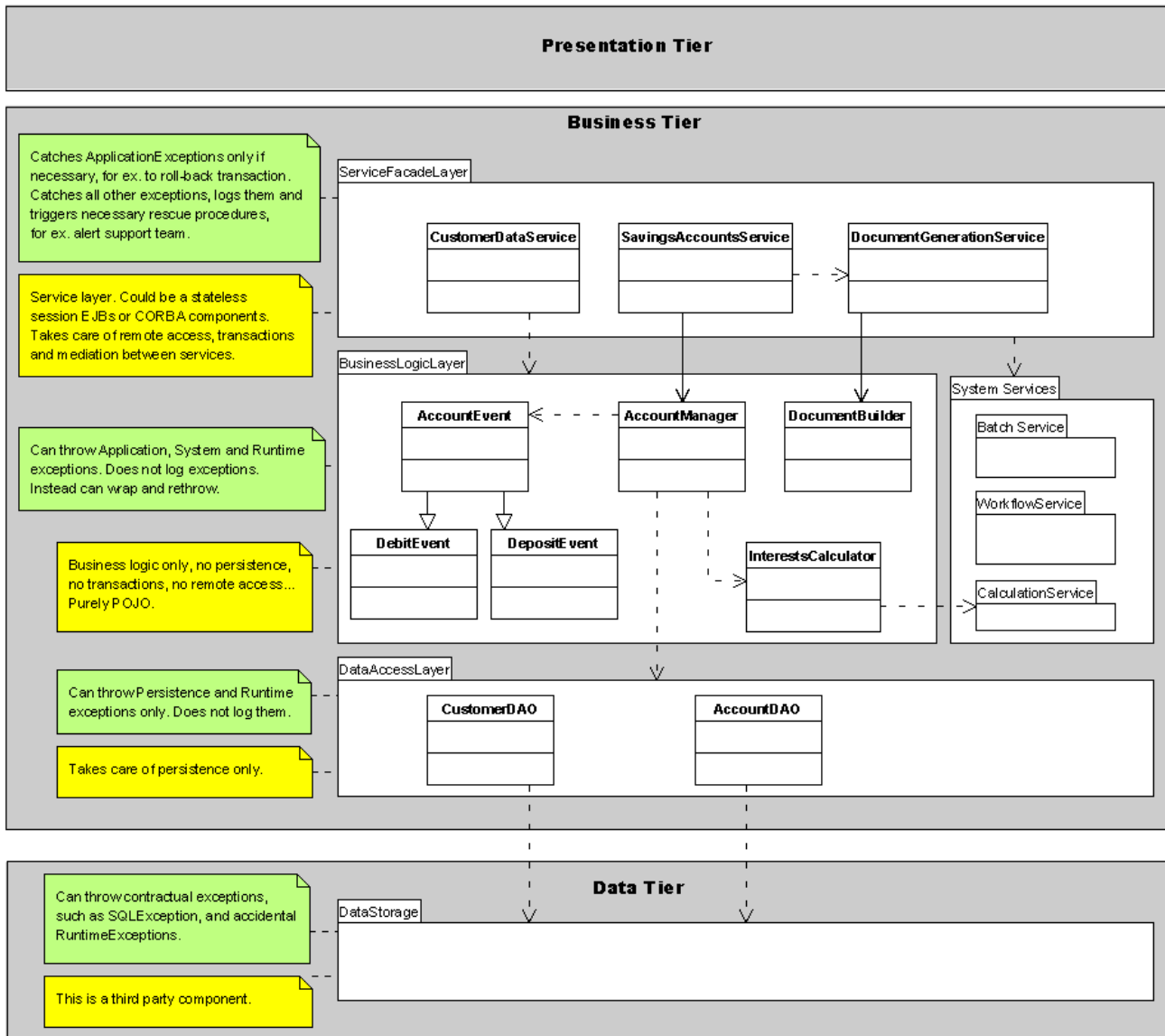
## The sample application

It is a simplified banking application that manages personal accounts for a high street bank. The application is able to perform standard banking operations, such as deposits, withdrawals, money transfers, interest calculations for savings accounts... It has a documents generation service able to format bank statements, confirmation letters, etc. Interests are calculated and deposited on the account in a nightly batch process that runs monthly.

### The Architecture

The purpose of this article not being application architecture, we are not going into discussions benefits of *layered application architectures*. We will just take for granted that it is good... Please refer to [ref5] for a discussion dedicated to enterprise application architecture.

Below is a *very* simplified package diagram showing the architecture layers. For the purpose of this example only few components that participate in the process that handles account interests are presented. Presentation tier has not been detailed.



Important application layers to notice in the above diagram are:

- Service Façade Layer (Session Beans/CORBA components)
- Business Logic Layer (pure Java: POJO)
- Persistence Layer (Java + Entity Beans/Hibernate/JDBC...)

## Categorising exceptions

**Actors:** Business analyst, application/component architect.

The above actors analyse the *business domain* that is to be automated, and together, find the best technical solution. As a result of that analysis they will draw the Use Case model and the application architecture. They will also identify potential *business related* and *technical* problems that would need to be handled. Some of those problems will probably be described in alternative flows of the use cases.

As a result, they will come up with:

- A hierarchy of ApplicationException-s
- A hierarchy of SystemException-s

## ApplicationException

Actors: Business analyst, application/component architect.

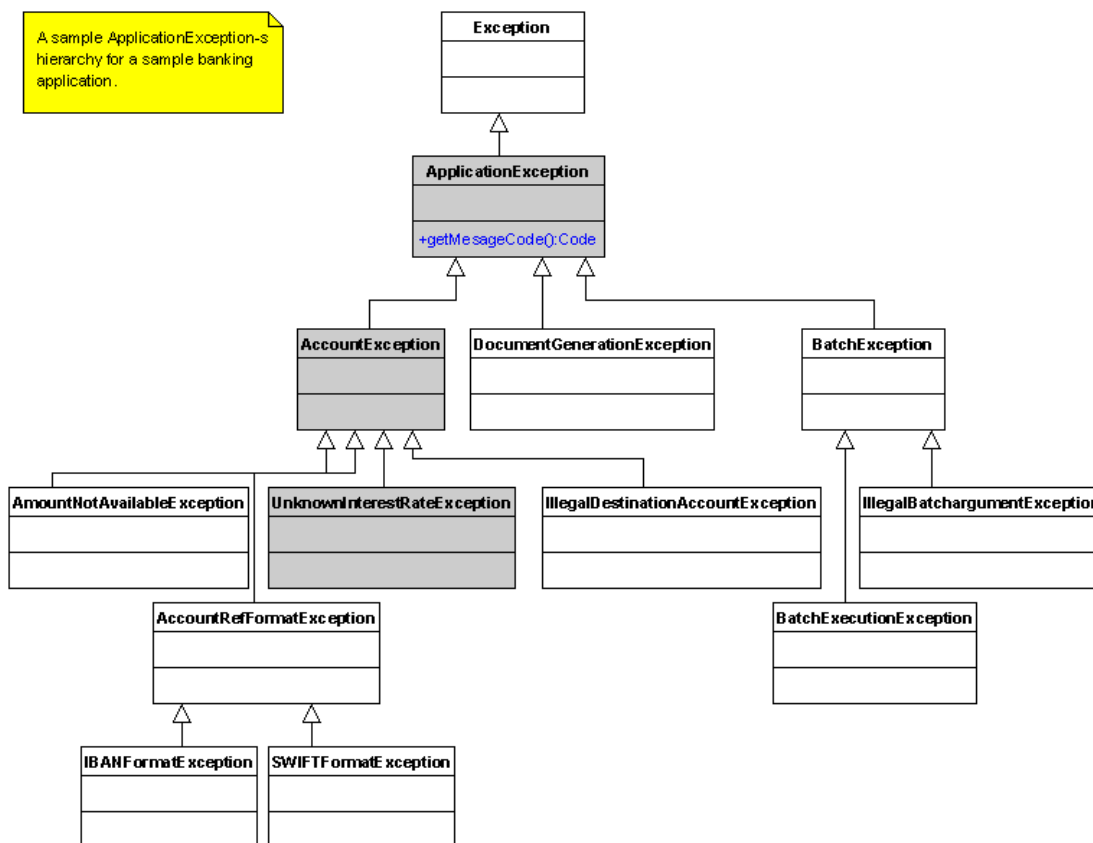
The analysis of the *business domain* will also indicate things that could go wrong. Those problems will naturally be business related and will probably nicely fit into categories of similar types of problems.

Example:

- ✚ User enters inconsistent data (marriageDate entered is anterior to the birthDate).
- ✚ User browses the online store and puts items in his shopping basket. Once finished (after a wife) he wants to check out and pay but the check out fails: an item from the basket went out of stock meanwhile. The application needs to act gracefully and maybe offer an alternative to the missing item... You might say: we could have made sure that the item does not go out of stock while a customer is still shopping, but that solution has other implications too - transaction duration, session, availability of items etc...
- ✚ On the 1st of Feb, banking application tries to calculate January interests on a savings account. The operation fails because the January interest rate was still not published for some reason.

The options:

The problem could be solved by using a hierarchy of ApplicationException-s that categorise all business related problems. In our sample banking application we could end-up with a hierarchy like this:



**Benefits:**

- ApplicationExceptions are **business related** and **predictable** at analysis time.
- They are part of the **service contract** and therefore part of the method signature.
- They are **checked exceptions**: the compiler helps us *not to forget* to handle potential problems. If we don't handle it, the application does not compile. This means that we discover bugs earlier: at compile time as opposed to runtime.
- Our business logic becomes **more robust**, because it delegates exception handling to the client code **by contract**: ApplicationException-s are part of the method signature, so the service contract clearly says to the client code what kind of problems it must handle.
- They help us extract exception handling logic to higher architectural levels (outside the main business logic), where we know better how to deal with the problem.
- We can handle problems by category.

- Our **exception handling policy** becomes simpler and clearer: we can decide in which application layer to handle exceptions of certain type. After throwing the exceptions, all we have to do is catch it in the appropriate layer. The exception handling logic is not everywhere in our code.
- We can use exception wrapping to hide implementation details of application layers.
- Different clients might decide to act differently upon problems: try again, trigger an alternative process, or simply inform the user with a nicely formatted message.
- This does **not** mean that the client code can not check for some conditions *before* it calls business methods. This is just the question of trade off between *defensive* against *optimistic* coding. See the discussion on data validation below.
- **Internationalisation**: Since application exceptions are business related, we will probably like to send meaningful messages to users about the problem. We could add support for internationalisation to all ApplicationException-s by identifying them with a unique code, which is later mapped to a message in different languages.

In the method below, the possibility that the interest rate be unknown is part of the contract:

#### Code

```
public void handleInterests(AccountNumber account, DateRange period)
throws UnknownInterestRateException;
```

Without using exceptions, we will probably end up coding a considerable amount of logic outside the business methods (`calculateInterests()` for example) in order to make sure that all conditions are met. Considering that exceptional situations are *exceptional*, this logic will in most cases be executed unnecessarily (performance issues). In addition to that, in order to implement the validation logic, we will probably need similar, if not the same data, as to actually *perform* the action. Thus we might end up retrieving the data twice, or having to pass all that data through method arguments.

#### Note

A business method should never declare `throws ApplicationException`. Throws ApplicationException does not give any information to the client code about the business problem that caused the exception to occur.

#### Logging

ApplicationException-s would in most cases be logged in INFO or DEBUG, or not logged at all. Exceptions apply.

#### Data Validation

Since we talk about *business related* problems, this is a good place to mention data validation.

In enterprise applications, where presentation and business logic are in separate tiers, it is often desirable to do some validation on data entered by users, before crossing tier boundary to call business logic. Data validation is governed by validation rules. Those rules can be dependent on the current state of business data, or not, i.e. **dynamic** or **static**.

**Dynamic rules:** In order to make sure that a withdrawal of EUR500 is possible, we must check the account balance; in order to book a plain ticket we must make sure a seat is available.

**Static rules:** The format of an IBAN account number does not depend on the state of the account (account balance). Our online banking system does not allow transfers over EUR50000 – the rule is fairly static (it could be cached). An other example of static validation is *data type* validation, such as validation of date or number formats.

Static rules are self sufficient, or depend on fairly static data that could be kept in a cache. Those rules could be verified in the presentation tier, before crossing the network and trying to invoke the business logic.

To validate dynamic rules we need to access additional business data, and therefore cross tier boundaries anyway. For that reason, it is often more suitable to implement them in the business tier only.

**NOTE that the meaning of “static” depends on the application: interest rates could be considered as fairly static in a high street banking application, but very dynamic in a portfolio management application.**

It is a good idea to implement data validation logic as a separate (and probably configurable) software component. Thus we can decide to run it in the presentation or business tier, whatever is more suitable for the given application. We could also choose to do it partly in the presentation, partly in the business tier. If the component is based on configurable rules, this becomes easier. The cost against benefits of such a component is to be taken into account though.

**NOTE: It is important to note that existence, or non-existence, of a data validation module should not in any case affect the way the business logic is implemented. Business methods cannot rely on any external logic to pre-validate data. That is why business methods declare ApplicationException-s, thus clearly stating that they are able to detect them but are not trying to handle them.**

Independently on where the data validation component will run, it should be implemented and maintained those who understand business, i.e. developers of the business tier.

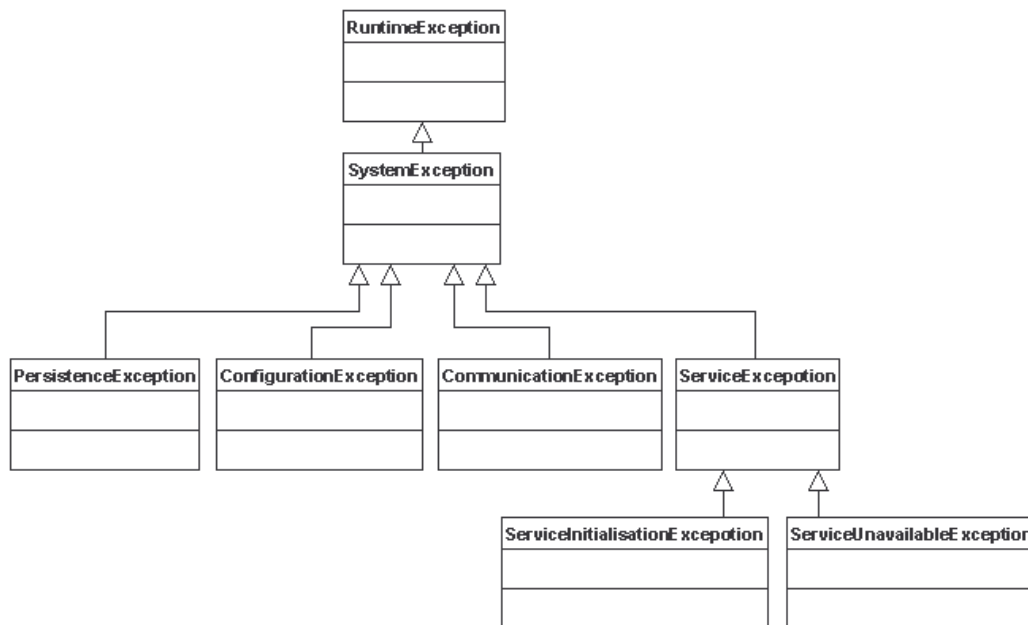
More information about data validation patterns can be found on the Internet. See also [ref4].

## SystemException

Actors: Application/component architect.

Business analyst is not in the list of actors here, because SystemException-s are related to *technical* issues. In an ideal world, those exceptions will never occur, but considering the reliability of the technical infrastructure and services we are using, we can probably expect certain type of problems to occur.

A hierarchy of SystemException-s could look something like:



Example:

- Failed to connect to the database.
- Failed to initialise workflow service.
- Failed to persist money deposit event.
- Failed to confirm payment with VISA service.

An important difference between system and application exceptions is that the latter are part of the service contract and are declared in the method signature. They are therefore designed by application architects in conjunction with the business analyst, which gives clear indications to the developer *where from* and *why* to throw them.

### Note

If it is not clear to the developer which exception to throw, he will end-up throwing SystemException directly. This would circumvent the base purpose of those exceptions. For that reason, SystemException-s hierarchy needs to be kept simple.

In addition to that, the variety of actions that we can take when a system exception is thrown is not that numerous: the most likely one is to simply alert the technical support. As a consequence, we do not need a very rich hierarchy of SystemException-s.

SystemException-s can also help to hide implementation details of lower application layers by use of exception wrapping. See examples below.

**To summarise:**

- SystemException-s are **not** part of the service contract, because they are not supposed to occur. However, we still want to act gracefully in the unlikely event...
- They are RuntimeException-s: we don't want to clutter our method declarations with those.
- They allow us to be a bit more specific about technical problems that can occur: *what*, *where* and *why*...
- They help in hiding implementation details of layers and components.

- They allow us to take the right action (alert the right person for example), based on the type of problem. E.g. for persistence problems, call Mr. A, for VISA service availability issues, alert Ms. B.
- The user does not need to know the details about the problem (we will probably be embarrassed to admit it anyway)... He will just receive a message saying that his request could not be processed at the moment and ask him to try again later. The log file however should contain more details about the problem.
- As opposed to ApplicationException-s, exception messages for SystemException-s will probably not need to be internationalised. They are intended to a technical support person who is likely to speak the language used during development.

#### Logging

SystemException-s would be logged in FATAL, ERROR or WARNING, depending on gravity.

### Why is SystemException a RuntimeException?

We do not want to clutter our method declarations with those exceptions. Public methods being part of the service contract, this contract should concentrate on **what** the service does, not **how** the service is implemented. SystemException-s are related to technical and therefore implementation issues.

### Since they are runtime exceptions, I could forget to handle them?

Those exceptions should be handled only in certain application layers, designed to handle technical issues. Since they are related to technical problems, the layers that implement business logic will not know what to do with them anyway...

The best place to handle them would be the ServiceFacadeLayer in the sample architecture presented earlier. Clear distribution of responsibilities between application layers together with code reviews should help in avoiding flaws.

### Why SQLException for example is a checked exception then?

We could think that SQLException is related to technical issues, but it is not!

The **business domain** of companies that provide JDBC drivers is data access and storage. As far as they are concerned, SQLException is part of their **business** commitment: to provide a standard way of accessing a relational data source, respecting the contract specified by the JDBC API.

Put in the context of our banking application however, SQLException becomes a technical issue: our business commitment is to handle money, not data storage. This is where we can use exception wrapping. Therefore:

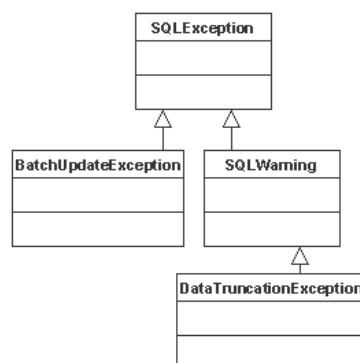
*SQLException is to JDBC driver what ApplicationException is to our application.*

**What is considered as being a business related issue by one layer or one might not be considered as business related by higher application layers. The service contract operates between adjacent application layers.**

### Why not inherit from Exception directly?

By having a hierarchy of ApplicationException-s we clearly distinguish business related problems from others, as far as our application domain is concerned. This allows us to do some specific handling for all such problems, like for example internationalisation of error messages.

By the way, all exceptions related to JDBC inherit from SQLException:



Again, *SQLException is to JDBC driver what the ApplicationException is to our application.*

Anyhow, you don't need to call them ApplicationException-s. You could name them `MyProjectBusinessRelatedException` if you wish, the whole discussion remains unchanged.

### Is there an ambiguity between system and application exceptions?

There can be a difficulty. An example could be `VISAServiceUnavailabeException`: As far as VISA Web service is concerned (on line payment service provider), this is an application exception. As far as AMAZON.com is concerned (online book store), this could be a system exception.

But there is no ambiguity, just needs a bit of reflection in the context of a given application. Business analyst can help in that.

## Other RuntimeException-s

In a 100% bug free application, they cannot happen :-). It is however a good idea to consider the possibility of runtime exceptions such as `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException` to occur.

Those exceptions are not `ApplicationException-s`, nor `SystemException-s`, and we have no control over *throwing* them - cannot predict bugs... All we can do is to make sure that we *catch* them and make our application act gracefully. The same service layer that handles `SystemException-s` should handle those as well.

### Logging

Other `RuntimeException-s` would be logged in FATAL or ERROR, depending on where you decide to log problems related to bugs.

## Other checked exceptions

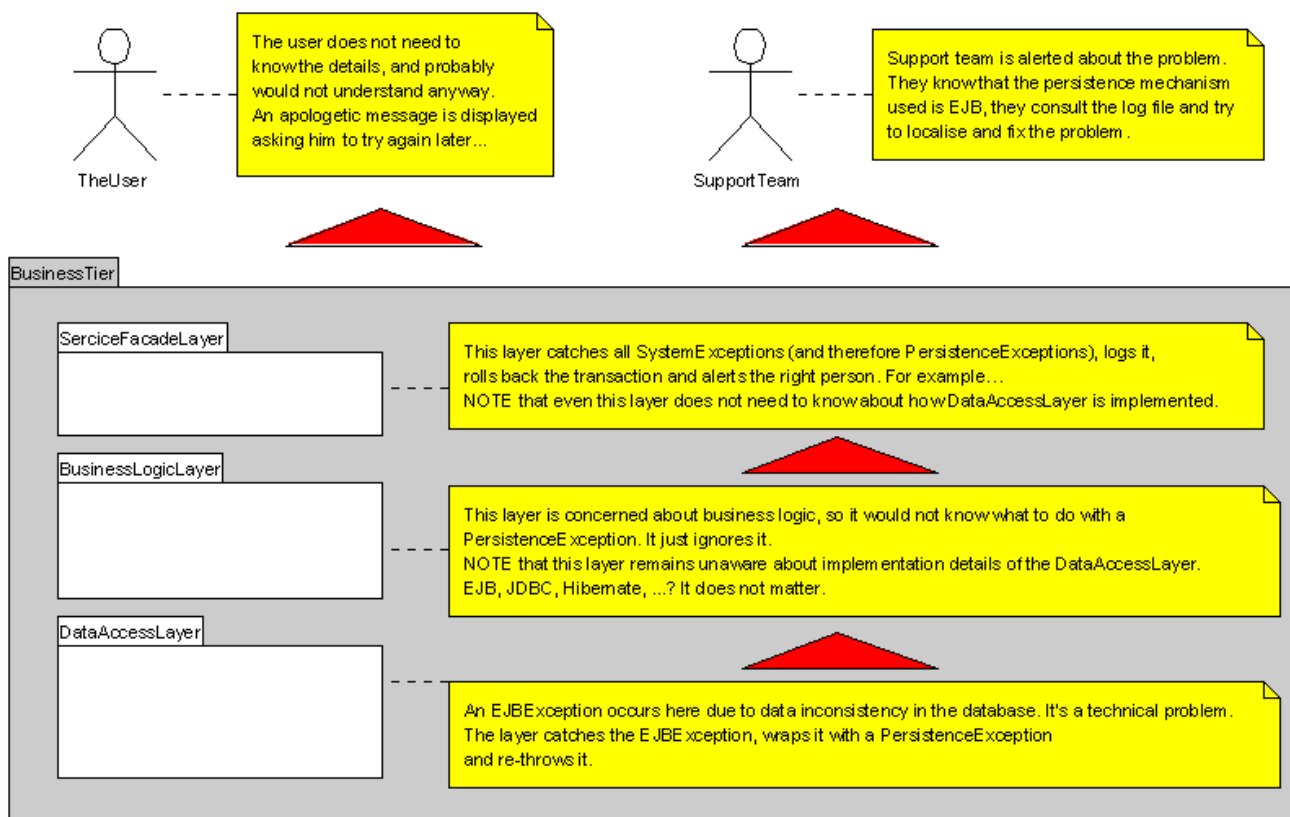
Example: `SQLException`, `ParseException`, `IOException`, `NamingException`... Just to name few.

Those exceptions can occur, but cannot slip through because the compiler enforces their handling.

### Note

If you think of your architecture in terms of layers and in terms of *service contracts* between adjacent layers, those checked exceptions could occur *within* a layer, but *should not cross layer boundaries* – unless specified otherwise in the contract. They need to be caught within the same layer that threw them, wrapped into an exception that *is* part of the contract or a `SystemException`, and re-thrown.

Example: `DataAccessLayer` throws an `EJBException` due to data inconsistency problem with the data in the database (read from bottom up...):



**Logging**

Other checked exceptions don't necessarily need to be logged. Especially if they are wrapped and re-thrown. The catcher of the wrapper exception will log it, which makes the log file smaller and easier to understand.

**Assertions**

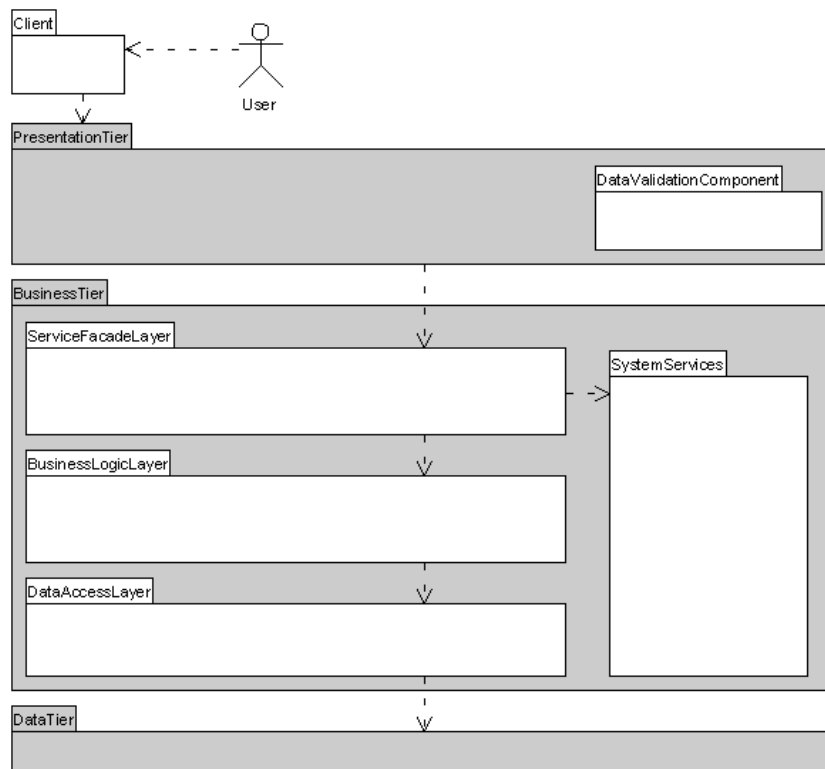
As we discussed earlier, assertions are to be used during debugging and testing phases in order to help discover bugs earlier.

**Logging**

Same treatment as runtime exceptions as far as logging is concerned.

**Putting It All Together**

Now that we know about what we can do with exceptions in Java, and that we know how to organise them, we will summarise it all in the context of the sample application architecture...

**DataTier**

Has a public service contract (EJB, JDBC, Hibernate, ...). Note that this is a *third party component*.

Depending on the implementation choice, `DataAccessLayer` uses the appropriate interface to access data.

Can throw: Any exception specified in its contract, i.e. in the public method signatures/interfaces. E.g. `SQLException`...

Catches: We don't know, we didn't write it...

**DataAccessLayer**

Has a service contract with the `BusinessLayer`. Accesses `DataTier`, as stated above.

Can throw:

- ApplicationException-s that are part of its interface,
- PersistenceException-s, for all other kind of problems, and
- Accidental RuntimeException-s, due to bugs.
- Assertion errors.

Catches:

- Checked exceptions that occur in the code, wraps them into PersistenceException and re-throws them.

**BusinessTier**

Has a service contract with the ServiceFacadeLayer. Concentrates on business logic only.

Can throw:

- ApplicationException-s as stated in its contract, i.e. in public methods signatures/interfaces.
- Accidental RuntimeException-s, due to bugs.
- Assertion errors.

Catches:

- Checked exceptions that occur in its own code. Wraps them into ApplicationException-s if business related, SystemException-s if technical, assertions/runtime if considered as bugs. And re-throws them.
- Could catch RuntimeException-s that occur in its own code and wraps them if it can add any useful information that would help higher layers and debugging. Otherwise, just lets go..

**ServiceFacadeLayer**

Has a public service contract with the outside world, such as PresentationTier, batch service or other client services.

This is probably the most important interface of the application.

Can throw:

- ApplicationException-s that are stated in its contract, i.e. in public methods signatures/interfaces.
- SystemException-s, due to *any* non-contractual problem.
- Nothing else.

Catches:

- Could catch ApplicationException-s thrown by lower layers if needs to do something (e.g. roll-back the transaction), or just lets go. This depends on the business process in question and must be decided on the case by case basis.
- *All other exceptions*. Logs them, alerts support, triggers rescue procedures... Depends on the application. Then wraps those exceptions to SystemException-s (if they are not already) and re-throws.

**Warning**

When a RuntimeException is thrown by a session EJB, transaction is rolled back automatically. With checked exceptions, this is **not** the case.

Therefore in case of ApplicationException-s the ServiceFacadeLayer must decide whether it is necessary to roll back the transaction before (re)throwing it. In case of SystemException-s this will be done automatically.

**PresentationTier** (or other clients)

Uses the application service contract to access business services and builds presentation. Could do some data validation too, as motioned earlier in this document.

Can throw: Nothing. Communicates with the user via a UI.

Catches:

- ApplicationException-s, formats an error message in the appropriate human language and presents it to the user, or triggers an alternative screen flow. As specified in the UI spec.
- All other exceptions, including SystemException-s. Formats a generic message to the user saying that the system could not fulfil her request, to try later, or similar.

## Technical considerations

### Performance

When used correctly, exception handling should not affect the performance.

Exceptions are exceptional, and when they occur the performance becomes a secondary issue – we need to deal with the problem as priority.

### Naming conventions

The name of the exception should try to describe *what* went wrong and should not imply any notion of gravity.

Having an exception called `FatalException` does not make sense: only the handler of the exception knows how fatal the exception is, not the thrower.

*Where* is answered by the stack trace, do not try to say it in the exception name. The name should be built of nouns, not verbs.

No good: `FatalException`, `CalculatingInterestException`, `IBANFormattingException`

Good: `IOException`, `UnavailableInterestRateException`, `IllegalIBANFormatException`

### Coding tips

Just few examples to make code simpler and clearer.

#### One try, multiple catch

This code example:

```
Code
try{
    calculateInterests (...);
    saveInterests (...);
    return;
}
catch (SQLException) {
    // do something
}
catch (InterestRateUnavailableException e) {
    // do something
}
```

... is easier to understand than:

```
Code
try{
    calculateInterests (...);
}
catch (InterestRateUnavailableException e) {
    // do something
}
try{
    saveInterests (...);
}
catch (SQLException) {
    // do something
}
return;
```

Note that `return` is inside the `try` block.

## Catch order

Careful not to catch what you don't want to catch...

```
Code
try{
    // some code
}
catch(Exception){
    // do something
} catch(RuntimeException){
    // do something
}
```

In the example above the catch Runtime is never reached. The first catch block catches everything.

## Never return from finally

```
Code
try{
    // some code that throws an exception
}
finally{
    closeConnection();
    return;
}
```

Even if an exception occurs, the finally block will be executed and will *return normally*, as if nothing happened. The exception is lost.

Better:

```
Code
try{
    // some code that throws an exception
    return;
}
finally{
    closeConnection();
}
```

## Never log in exceptions

Exceptions *carry* information, they do not *do* things.

Example:

```
Code
public class MyException extends Exception{

    /** CONSTRUCTOR */
    public MyException(Exception rootCause){
        super(rootCause);
        logger.error(rootException);
    }
}
```

The idea might seem practical: exception is logged as soon as you construct it.

First, logging might not be available to the component that constructs the above exception, so your exception constructor could throw an exception... The original exception is lost.

Second, your log file will become a mess.

Exception logging is part of exception *handling*, not exception *throwing*.

### Exception message

Be brief in exception messages. Do not format it, let the logger or other exception handling components do it.

Take as example `FileNotFoundException` where the `getMessage()` method returns the file name only:

```
>> missing_file.txt  
toString() would return:  
>> java.io.FileNotFoundException: missing_file.txt
```

Notice that `toString()` returns `className + message`.

## Conclusion

When used effectively, exception handling is an excellent aid in making applications more robust. Exceptions are another application of the principle of separation of concerns: problem detection vs. problem handling. We hope that we were convincing enough about benefits of a clear separation between business related and system exceptions too.

### About the authors

*Dino Celovic* and *Nader Soukouti* are senior software architects for enterprise applications at Sanabel Solutions in Geneva. *Dino* specialises in services such as workflow, calculation engines and batch management. He has previously worked on a variety of projects in the pension management, finance and insurance industries.

*Nader* has a long experience in building distributed enterprise applications in CORBA and J2EE. He has previously worked in domains of banking, pension fund management and telecoms and has co-authored the book "EJB 2.0 – Mise en Oeuvre" in French.

### References

- [ref1] "Three Rules for Effective Exception Handling" by Jim Cushing.  
<http://today.java.net/pub/a/today/2003/12/04/exceptions.html>
- [ref2] "Handling Errors with Exceptions" on <http://java.sun.com/docs/books/tutorial/essential/exceptions/>
- [ref3] "Applying UML and Patterns", by Craig Larman
- [ref4] "EJB best practices: The fine points of data validation" on <http://www-128.ibm.com/developerworks/java/library/j-ejb1217.html>
- [ref5] "About Effective Enterprise Application Architecture" on <http://www.sanabel-solutions.com>