b
UNIVERSITÄT
BERN

Modular Exceptions

A system for handling exceptions in a modular way

Bachelor Thesis

Indermühle Patrick
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

August 28, 2018

Prof. Dr. Oscar Nierstrasz

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Exception handling is an integral part of programming. However, it is often not written in a way that makes it easily reusable. We have found exception handling code to often be copy pasted across multiple catch blocks instead of being made into a method. We also found that there are certain patterns across different methods when it comes to exception handling. That is why reusable exception handling would be a helpful feature for software development. By creating Modular Exceptions we offer a solution that enables programmers to easily apply and reuse exception handling to multiple methods. We achieved this by analyzing the knowledge gathered in previous research about exception handling and performing our own research of exception handling in Smalltalk. We then studied different implementation approaches such as dynamically rewriting the source code and method wrappers until we found the optimal approach. Our final product is written in Java and uses AspectJ in order to dynamically insert try-catch blocks into methods and to add exception handling into already existing catch blocks. These handler blocks are compatible with many methods and classes, and the user only has to write a few lines of code to get a specific method covered.

Contents

1 Introduction

Software stability and error tolerance are important aspects of software development. No program runs 100% error free, and we cannot guarantee that the data fed to the program is valid, so the ability to recover from exceptions is vital.

That is why exception handling is so extensively studied [8] [5] [6] [7] [2], especially exception handling in Java. Several studies [1] [9] in exception handling reveal that exceptions are often handled in similar ways and that certain patterns exist in exception handling. The fact that these patterns exist lead us to the idea that an exception handling approach that focuses on re-usability would be highly desirable.

In our own study of exception handling in Smalltalk using the Moose-6.1 image we found that exception handling in Smalltalk had many similar patterns of exception handling. Yet still exception handling itself was rarely written in a reusable way. In fact in our own study we saw many exception handling blocks that seemed almost copy pasted across multiple handler blocks. Good examples of this in Smalltalk are the methods `AbstractNautilusUI>>addNewGroup` and `AbstractNautilusUI>>renameGroup`. Their catch blocks are identical, as shown by their method definitions below.

```
addNewGroup
   | group |
   [ group := self groupsManager createAnEmptyStaticGroup ]
      on: GroupAlreadyExists
      do: [ :ex | self alertGroupExisting: ex groupName ].
   GroupAnnouncer uniqueInstance announce: (AGroupHasBeenAdded group:
     group into: self)
```

```
renameGroup
   | group |
   group := self selectedGroup.
   group ifNil: [ ^ self ].
   [self groupsManager renameAGroup: group ]
      on: GroupAlreadyExists
      do:[ :ex | self alertGroupExisting: ex groupName ].
   self updatePackageGroupAndClassList
```

The programmer/s of these methods decided to simply copy paste the handler block instead of using a handler method. This makes it inflexible as any change in exception handling would have to be implemented and then copy pasted to the other method. A far more effective approach to exception handling would be to write handler methods that can be used across many methods or classes so that exception handling can be reused or even inherited.

That is why in this thesis we present modular exceptions, a model with which we can easily apply exception handling to different methods and reuse handler blocks instead of copy pasting code across multiple methods. The advantages of such a model would be more flexibility in exception handling since changes to the handler block would no longer have to be copy pasted to other methods. Thanks to this increased flexibility we hope that software created with our system would be more error tolerant and much more adaptable to unexpected behavior.

In order to create a solution to this problem our plan was as follows. First we study how exceptions are commonly handled, to determine whether there are patterns to exception handling and if yes, which patterns are the most common. This is a necessary step to understand what our solution must be able to do. There is no point creating a system to handle exceptions if no programmer will use it. That is why we need to understand how exceptions are commonly handled.

Once we understand this we define our requirements to our solution. Once we know what the requirements are we start testing out approaches to implement exception handling in order to see which approaches are even viable. Using the requirements found earlier we weed out approaches that will not work. Eventually we find an approach that fulfills all our requirements. Once this approach has been found we create an example project to truly prove that it can actually solve our problem. Our example project must contain examples that show how our solution handles the most common types of exceptions.

2
Related Work

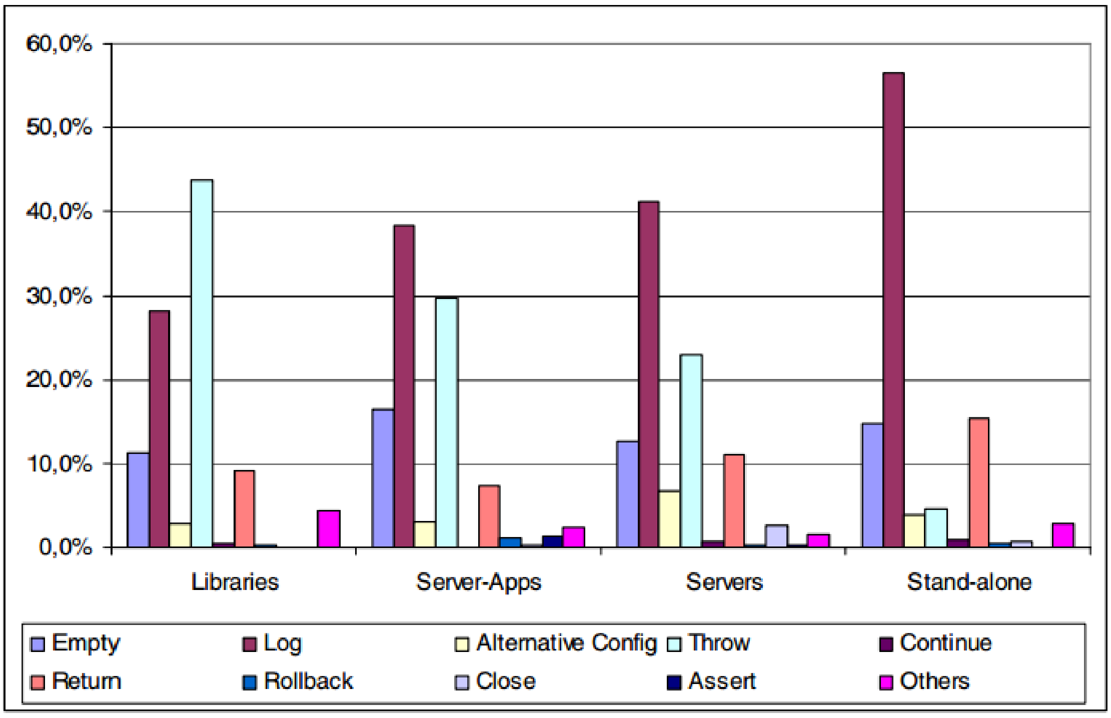
In this chapter we list the previous research that we analyze in detail. The research in question consists of the papers “Exception Handling: A Field Study in Java and .NET” [1] and “On the Evolution of Exception Usage in Java Projects” [2]. We chose the first paper in order to figure out what patterns exist in Java when it comes to exception handling. The second paper was chosen because we needed to figure out how exception handling changes over a project’s lifetime.

2.1 Exception Handling: A Field Study in Java and .NET

In the paper “Exception Handling: A Field Study in Java and .NET” [1] the researchers analyzed 16 Java applications of which 4 were libraries, 4 server-apps, 4 servers and 4 were stand-alone software. The goal of the research was to figure out how programmers commonly use exceptions. They categorized exception handling patterns in Java into ten different non-exclusive categories.

Type of Exception Handling	**Description**
Empty	Handler block is empty and only catches the exception
Log	Logs the error
Alternative state	Uses a predetermined alternative object state
Throw	Throws the Exception caught or throws a new one
Continue	Continue with the next iteration if inside a loop
Return	Return to the caller or exit the application
Rollback	Undo all changes made to an object inside the method
Close	Closes open connections to streams
Assert	Makes an assertion and often throws a new Exception if false
Others	Any Exception handling that does not belong to a category

The ratio's between the different categories are displayed in Figure 3 of the paper. [1, 5.2, p. 162]



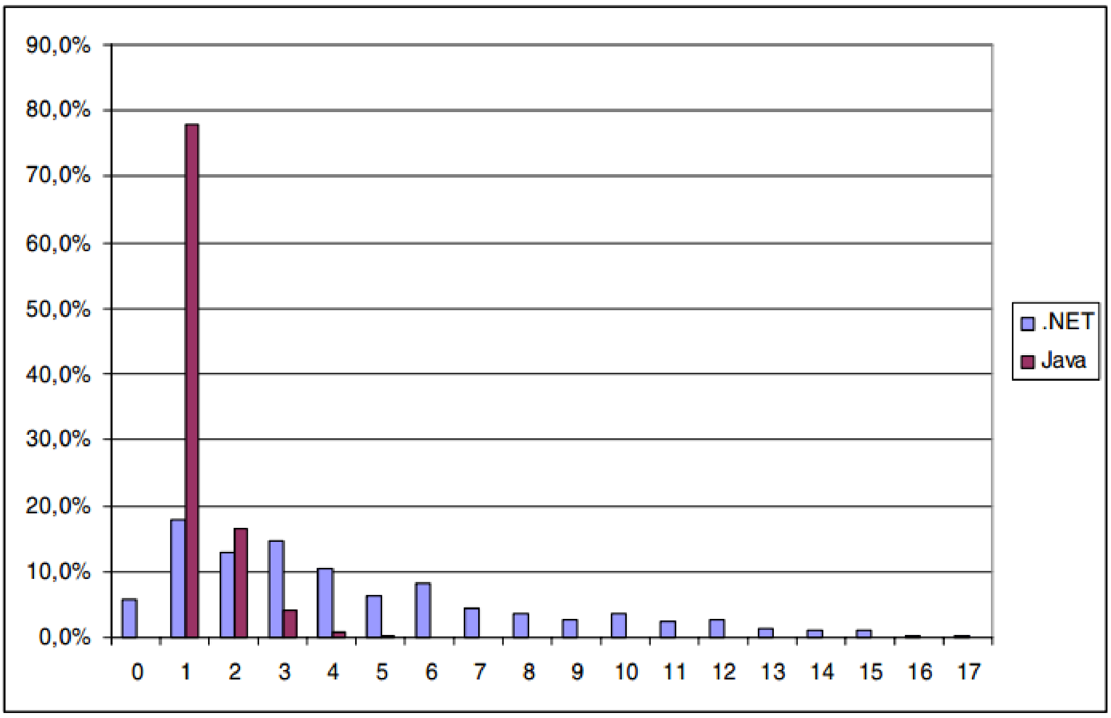
This clearly shows that exception handling has repeated patterns and that modular exceptions or any other model that helps us reuse exception handling is needed. On top of that it also shows us that certain exception handling patterns are used more often than others. Based on this figure we can say that Java developers most commonly handle exceptions in these kinds of ways.

- Logging the exception

- Re-throwing the exception
- Returning to the caller
- Catching the exception but simply resuming the execution of the method
- Using an alternative object configuration

Other kinds of exception handling are used less than 5% of the time. If our solution can handle exceptions in these ways it will be useful for handling 95% of all exception handling cases.

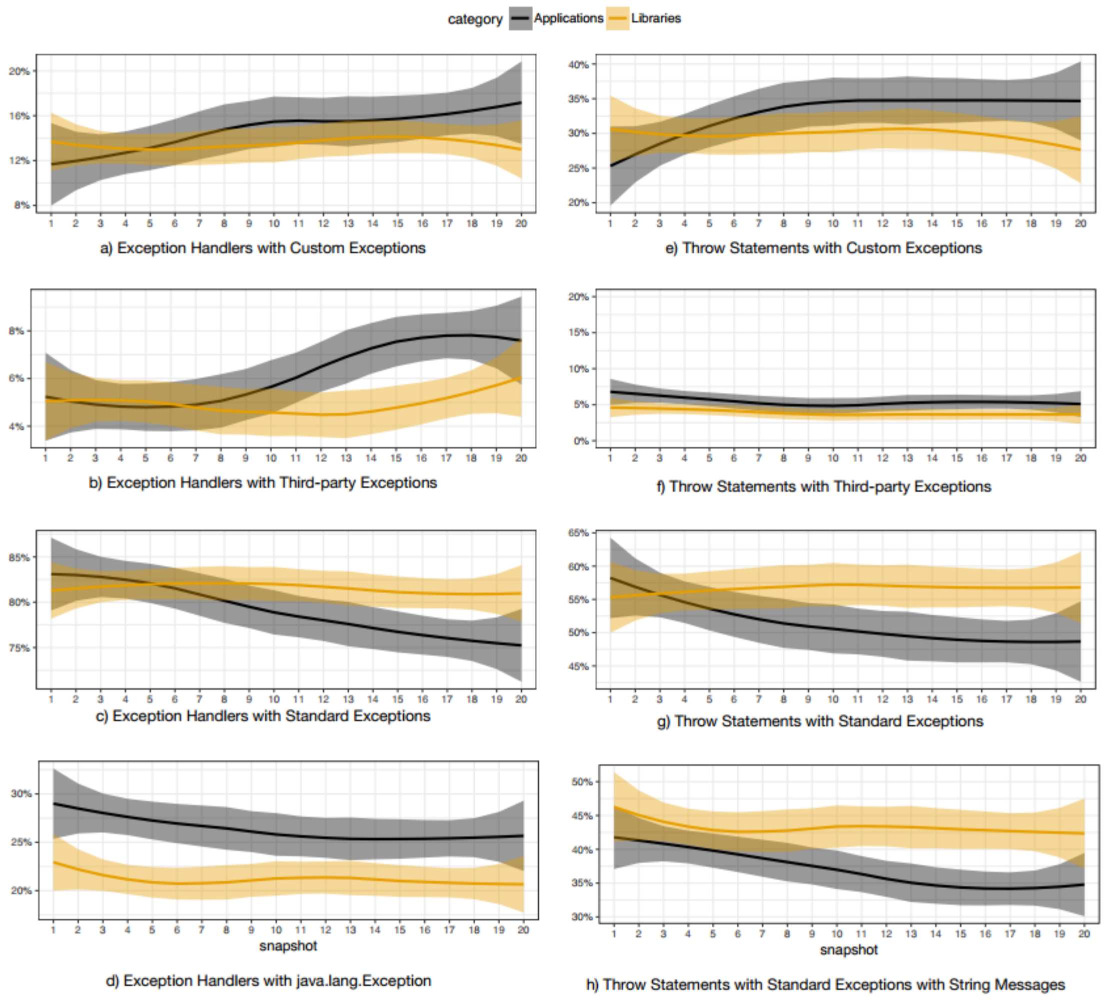
Figure 11 of the paper [1] reveals to us that almost 80% of Java exceptions are caught immediately by the caller instead of being thrown further up the stack. About 95% of the exceptions are re-thrown less than 5 times.
[1, 5.4.3, p. 170]



This information shows us that exceptions are handled in Java commonly where the exception originates from. This means that at least in Java we can expect a try-catch block to already be present where an exception could occur. This means that our Java solution will mostly focus on handling exceptions rather than catching them.

2.2 On the Evolution of Exception Usage in Java Projects

We consult another relevant paper called “On the Evolution of Exception Usage in Java Projects” [2]. The authors analyzed the usage of exceptions over a certain time frame for 15 Java applications and 15 Java libraries. They did this to figure out whether programmers used standard Java exceptions or custom exceptions. The time frame was set to be at least five years long with 20 snapshots being chosen in those five years in order to evaluate the changes over time. What follows is a graphic of their findings provided in their paper [2, p. 5].



Comparing the first snapshot with the last it becomes clear that exception handlers still mostly handle standard exceptions. The usage of standard exceptions however has fallen in Java libraries from about 83 % to about 75 % over the timespan of the 20 snapshots. Meanwhile the usage of custom exceptions in handler blocks in libraries rose from about

12 % to about 18 %, which is still a lot lower than the 75 % of handler blocks that handle standard exceptions in Java libraries.
Overall this paper shows us that standard exceptions are the most commonly used type of exceptions and that we need to design our solution to address that fact.

3 Analyzing exception handling

In this chapter we describe the work we have done, our findings and results.

3.1 Exception handling patterns in Smalltalk

In order to find common patterns in exception handling we analyze the usage of exceptions in Smalltalk Pharo. To find these common patterns in exception handling we look at how the method `BlockClosure>>on:do:` is used in Smalltalk. The method `BlockClosure>>on:do:` in Smalltalk effectively does the same as a try-catch block would in Java. It executes a block of code and in case an exception is raised executes a handler block. In Smalltalk all users of `BlockClosure>>on:do:` can be found by using the SystemNavigation class. This class has an instance method called `SystemNavigation>>allSendersOf:` that takes as its argument the selector of a method and returns all method definitions that use that method. As code base we used the Moose-6.1 image. We use the following line of code to get a list of all methods in Smalltalk that handle exceptions in some way.

```
SystemNavigation new allSendersOf: #on:do:
```

This method gives us a list of 851 methods that all use the method `BlockClosure>>on:do:` in some way. We cannot use all these methods for our analysis because not all of them actually handle exceptions. For example unit tests do not handle exceptions but rather test a target method to see if it throws an exception when it should. An example of this is the method `AnnouncerTest>>testSymbolIdentifier`.

```
testSymbolIdentifier
   | passed |
   passed := false.

   [announcer
      when: #FOO send: #bar to: nil;
      announce: #FOO ]
      on: MessageNotUnderstood
      do: [ :ex | passed := (ex message selector = #bar). ].

   self assert: passed
```

Or the method `ExceptionTests>>testResumableOuter`.

```
testResumableOuter

   | result |
   result := [Notification signal. 4]
      on: Notification
      do: [:ex | ex outer. ex return: 5].
   self assert: result = 5
```

These methods are useless to us as they do not show how exceptions are handled by Smalltalk programmers.
Next there are methods that handle exceptions by allowing the user to give the method a handler block through one of the method parameters. These kinds of methods are not useful to us either since they do not handle exceptions by themselves. An example of this is the method `AbstractCompiler>>silentlyDo:exceptionDo:`.

```
silentlyDo: aBlock1 exceptionDo: aBlock2
   aBlock1
      on: Error, SyntaxErrorNotification, OCSemanticWarning,
   OCSemanticError
      do: aBlock2
```

Another example of this is the method `GTCurrentSpotterExceptionHandler>>use:during:`.

```
use: anObject during: aBlock

   ^ aBlock on: self do: [ :notification | notification resume:
   anObject ]
```

Then there are also methods that use exceptions as makeshift events, which makes them worthless for our study. An example of such is the method `GoferMerge>>execute`.

```
execute
   [ self model merge ]
      on: MCMergeResolutionRequest
      do: [ :request |  request autoMerge ].
   self gofer cleanup
```

Or the method `GTEventTool>>download`.
There is also also a series of classes that seem to be almost copy pasted clones of each other. This series of classes is the "ConfigurationOf" classes. These include the following classes.

- ConfigurationOf class
- ConfigurationOfAnnouncerCentricDebugger class
- ConfigurationOfBitmapCharacterSet class
- ConfigurationOfFuel class
- ConfigurationOfFuelPlatform class
- ConfigurationOfGlamour class
- ConfigurationOfGofer class
- ConfigurationOfGrease class
- ConfigurationOfMagritte3 class
- ConfigurationOfMerlin class
- ConfigurationOfMocketry class
- ConfigurationOfMoose class
- ConfigurationOfPharoDocComment class
- ConfigurationOfProtocolAnalyzer class
- ConfigurationOfRoelTyper class
- ConfigurationOfVersionnerTestBitmapCharacterSet class
- ConfigurationOfVersionnerTestXMLParserTemplate class

- ConfigurationOfVersionnerTestXMLWriter class
- ConfigurationOfXMLParser class
- ConfigurationOfXMLWriter class
- ConfigurationOfZinchHTTPComponents class

All of these classes have a method called `ensureMetacello:` which uses the method `BlockClosure>>on:do:`.
The method `ConfigurationOf>>ensureMetacello:` is copy pasted across multiple classes. If we counted all these methods we would heavily distort our findings. If they were programmed by conventional design they would have used inheritance to inherit the method from some base class instead. In that case we would have only counted it once, so we decide to not count copy pasted code as separate methods. The method `ConfigurationOf>>ensureMetacello:` is therefore only counted once. In the end we ignore the following kinds of methods in our study.

- Unit tests
- Tester class methods
- Methods that take a handler block as a parameter
- Methods that use exceptions as makeshift events
- Methods that were copy pasted

From the leftover methods we analyze 163 methods. We categorize exception handling based on actions the exception handling block performs. Some of the exception handling types were only used in methods with a return value (like the "return default value" exception handler) while others were only used in methods have no return value (like the "Return immediately" exception handler).

Handling type	**Description**
Alert user	Alerts the user of the exception by using the console or through some pop-up
Return immediately	Returns to the caller.
Restore object	Changes the data of an object in order to execute the method properly
Alternative code	Use a block of code or a different method in order to perform the original task of the method where the exception happened
Resume execution	Resumes the method by skipping a part of the execution. This is often used in for-each loops
Run handler method	Executes a handler method with the exception in question as a parameter
Return default value	Returns a default value which is either a constant, a value calculated through a different method or a parameter
Return null	Returns a null value signaling the caller method that something has gone wrong
Other	All cases of exception handling that do not fit into any of the previous categories

What follows is an example method of each category of exception handling.

- “Alert user” handling is used in the method `EpLogNodeGraphModel>>refreshLogNodesTreeModel`.

```
refreshLogNodesTreeModel

   | nodes |
   nodes := #().
   [  nodes := EpFileLogNode fromAllLogsIn: self directory.
      nodes := nodes sorted: [ :a :b | a modificationTime >= b
   modificationTime ] ]
         on: FileSystemError
         do: [ :error | self inform: error asString ].

   self flag: #fix. "hacky"
   self isMonitorDirectory ifTrue: [
      | currentGlobalName |
      currentGlobalName := self monitor sessionStore store
   globalName.
      nodes := nodes reject: [:each | each globalName =
   currentGlobalName ].
      nodes := { EpMonitorLogNode for: self monitor }, nodes.
   ].

   hiedraCheckBoxModel state ifTrue: [
      nodes
         do: [ :node | node populateReferencedNodesWith: nodes ]
         displayingProgress: ('Analyzing ', directory asString)].


   logNodesTreeModel roots: nodes.
```

- “Return immediately” handling is used in the method `FontChooserMorph>>pointSizeString:`.

```
pointSizeString: aText

   | string number |
   string := aText asString trimBoth.
   string isEmpty ifTrue: [^self].
   string
      detect: [:c | c isDigit not and: [c ~= $.]]
      ifFound: [^self].
   [number := string asNumber asFloat] on: Error do: [:e | ^self
   ].
   (number < 1 or: [number > 1024])
      ifTrue: [^self].

   pointSizeMorph ifNotNil: [pointSizeMorph hasUnacceptedEdits:
   false].
   model pointSize: number
```

- “Restore object” handling is used in the method `GlobalIdentifierPersitence>>load:`.

```
load: existingDictionary
   "It loads stored information into existingDictionary."
   self isEnabled ifFalse: [ ^ self ].
   self preferences exists ifFalse: [
      "This is a new computer, so we define new computer UUID.
      User still has to agree about sending data if it is not
   has been done yet."
      ^ self save: existingDictionary ].
   [ (self mergeExisting: existingDictionary stored: self load)
         ifTrue: [ self save: existingDictionary ].
   ] on: Error do: [
      "Preferences likely contains a different settings version,
    so we store the actual one.
      We should keep the preferences as stable as possible."
      self mayOverwrite: existingDictionary ]
```

- “Alternative code” handling is used in the method `ConfigurationCommandLineHandler>>metacelloVersion:`.

```
metacelloVersion: aVersionName
   | project |
   project := self project.
   ^ [ project version: aVersionName ]
      on: MetacelloVersionDoesNotExistError do: [ :error |
         aVersionName = 'last'
            ifTrue: [
               "manual fallback since there is no symbolic name
   for lastVersion"
               project lastVersion ]
            ifFalse: [
               "symbols and strings are not equal in Meteacello
   ..."
               project version: aVersionName asSymbol ]].
```

- “Resume execution” handling is used in the method `GLMSingleUpdateAction>>computeAnnouncerObjects`.

```
computeAnnouncerObjects
   self flag: 'We catch the error because if there is a problem
   in the computation of the announcer object, we still want to
   be able to continue the execution'.
   ^ OrderedCollection with:
      ([self transformation glamourValue: self presentation
   entity]
         on: Error
         do: [:e | self presentation entity. e resume])
```

- “Run handler method” handling is used in the method `CommandLineTestRunner>>runCase:`.

```
runCase: aTestCase
   self increaseTestCount.
   self printTestCase: aTestCase.

   [[ aTestCase runCaseManaged.
      self printPassOf: aTestCase ]
      on: Halt , Error, TestFailure
      do: [ :err | self handleFailure: err of: aTestCase ]]
      on: TestSkip do: [ :skip| self handleSkip: skip of:
   aTestCase ]
```

The method `CommandLineTestRunner>>handleFailure:` itself handles the exception by alerting the user

```
handleFailure: anError of: aTestCase
   (anError isNil or: [aTestCase isExpectedFailure])  ifTrue: [
   ^ self ].
   (anError isKindOf: TestFailure)
      ifTrue: [
         suiteFailures := suiteFailures + 1.
         self printFailure: anError of: aTestCase ]
      ifFalse: [
         suiteErrors := suiteErrors + 1.
         self printError: anError of: aTestCase ].

   self shouldSerializeError
      ifTrue: [ self serializeError: anError of: aTestCase ]
```

- “Return default value” handling is used in the method `GtExampleProcessor>>canProcess`.

```
canProcess
   "I verify if this processor can execute on the given example.
    I do this by running the processor
   and checking if any example-specific specific is raised. Such
    exceptions indicate a problem with
   the given example that does not make it possible to apply the
    processor. "

   [ self value ]
      on: GtExampleError
      do: [ ^ false ].
   ^ true
```

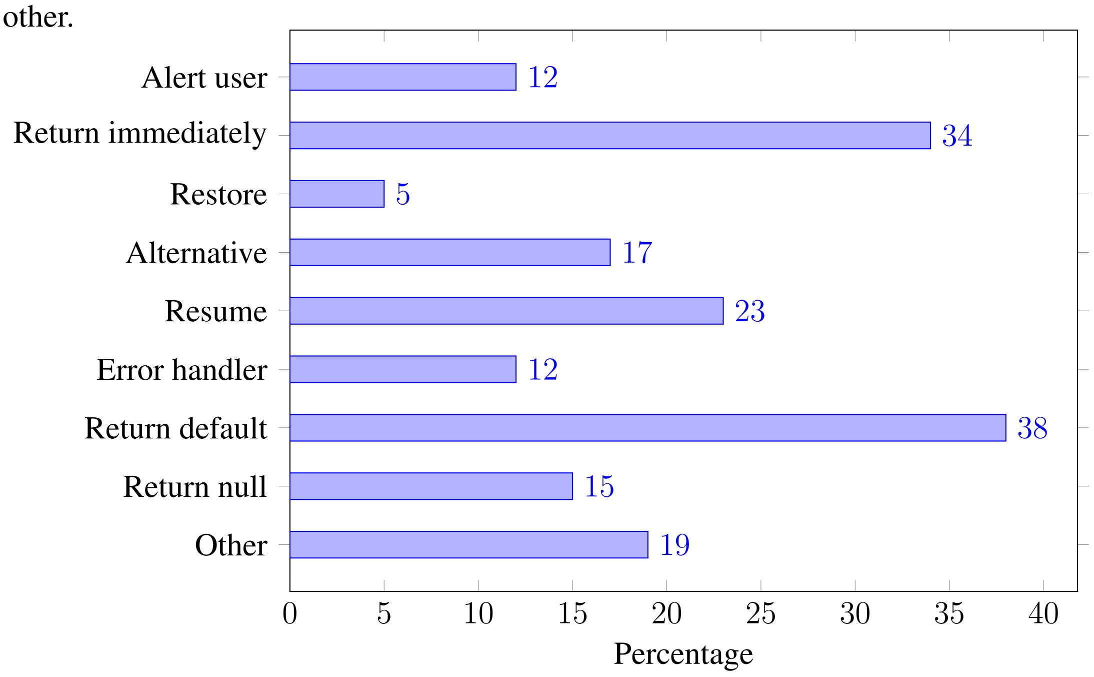
- “Return null handling” is used in the method `Alien>>lookupOrNil:`.

```
lookupOrNil: symbol  "<String>" inLibrary: libraryName "<String>
     ^<Alien>"
   "Answer the address of symbol in libraryName, or nil if it is
    not in the library.
   Exceptions will be raised for invalid libraries, symbol names,
    etc."
   ^[self lookup: symbol inLibrary: libraryName ]
      on: NotFoundError
      do: [:ex| nil]
```

We analyzed 163 methods and categorized 175 ways to handle exceptions. The reason why these numbers differ is because the 12 methods that alert the user also implement a second form of Exception handling. An example of this is the method `AbstractStoredSetting>>updateSettingNode` which alerts the user and also cancels the method execution in case of an exception.

```
updateSettingNode: aSettingNode
    | value |
    [ value := self realValue ] on: Error do: [ :exception | "ignore
    and leave"
       self inform: 'Cannot read stored value of ', self
    settingNodeIdentifier, '. Exception: ', exception printString.
       ^ self ].
    [ aSettingNode realValue: value ] on: Error do: [ :exception | "
    inform and ignore"
       self inform: 'Cannot update ', self settingNodeIdentifier, '.
    Exception: ', exception printString ]
```

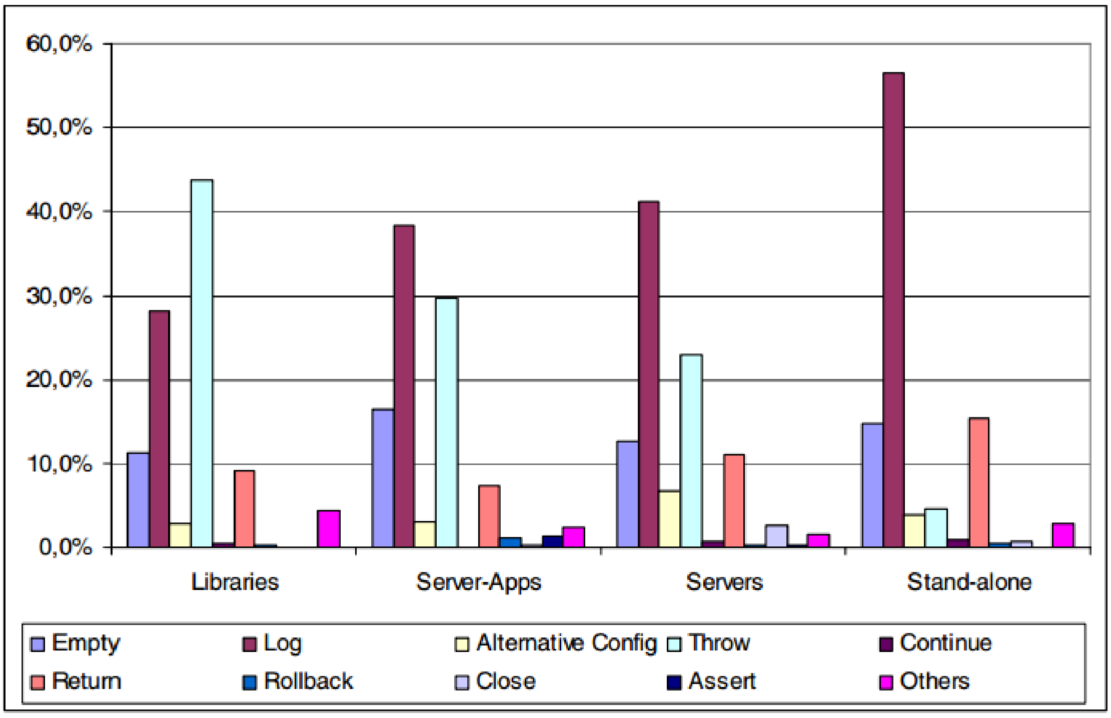
The following table shows the ratios of each exception handling type compared to each other.



As seen in the table the most commonly used types of exception handling are

1. Returning immediately to the caller
2. Returning a default value
3. Resuming the method

We compare our results to the results of the paper "Exception Handling: A Field Study in Java and .NET" [1] which are displayed in Figure 3 in the paper [1, 5.2, p. 162].



We have found the following similarities and differences.

- Logging the exception in Java and alerting the user in Smalltalk are both common, though in Java it is a lot more common. This shows that feedback to the developers concerning exceptions that have occurred is important when handling exceptions.
- In Java exceptions seem to be often re-thrown while in Smalltalk exceptions are rarely re-thrown. re-throwing the exception in Smalltalk is so uncommon that we categorized it into the “other” category together with other exception handling types that we were unable to put into other categories.
- Many exception handler blocks in Java do nothing since they are empty. We found this to be similar to the “resume execution” exception handling type in Smalltalk as those kinds of handler blocks often did nothing more than catch the exception and continue the method.
- Returning a default value or a null value in case of an exception is a common way to handle exceptions in Smalltalk. In Java however it is rare for a method to return to the caller in case of an exception.

- Using an alternative configuration in Java is very rarely used. Using an alternative code in Smalltalk is however very common. The problem with both is that they are very individualized. In Java the exception handling blocks of this type are very individualized [1, 5.2, p. 161]. In Smalltalk methods that handle exceptions this way use methods or code that is not available to multiple classes. An example of this are the method `ConfigurationCommandLineHandler>> metacelloVersion:`.

```
metacelloVersion: aVersionName
   | project |
   project := self project.
   ^ [ project version: aVersionName ]
      on: MetacelloVersionDoesNotExistError do: [ :error |
         aVersionName = 'last'
            ifTrue: [
               "manual fallback since there is no symbolic name
   for lastVersion"
               project lastVersion ]
            ifFalse: [
               "symbols and strings are not equal in Meteacello
   ..."
               project version: aVersionName asSymbol ]].
```

  Another example is the method `ConfigurationCommandLineHandler>> handleMergeConflictDuring:`.

```
handleMergeConflictDuring: aLoadBlock
   [aLoadBlock
   on: MCMergeOrLoadWarning do: [ :mergeConflict | mergeConflict
    merge ]]
   on: MCMergeResolutionRequest do: [ :request |  request
   autoMerge ].
```

  Another example is the method `HismoAbstactHistory>> allEarliestEvolutionOfPropertyNamed:`.

```
allEarliestEvolutionOfPropertyNamed: aPropertyName

   | ene currentValue previousValue |
   ene := self first propertyNamed: aPropertyName.
   self
      versionsIndexFrom2Do:
         [:i |
         currentValue := (self versions at: i) propertyNamed:
   aPropertyName.
         previousValue := (self versions at: i - 1)
   propertyNamed: aPropertyName.
         [ene := ene + ((currentValue - previousValue) abs * (2
   raisedTo: 1 - i))]
```

```
            on: ArithmeticError
            do: [:exc | ene := ene + 0]].
   ˆene asFloat
```

These three different methods all handle exceptions in wildly different ways, despite them being in the same category. This makes it very hard to handle these kinds of exception in a modular way since a handler block that handles one of the exceptions would be completely incompatible with the other two methods. We therefore conclude that handling exceptions with alternative code/configuration exception handling type is not possible to do in a modular way.

We also found that many exception handler blocks were copy pasted across methods. We count a handler block as copy pasted if it fulfills the following conditions.

1. It uses at least one method call.
2. The method/s it calls are not explicitly for exception handling.
3. At least a portion of the code is identical to code in another handler block.

We found the following methods have copy pasted exception handling code.

- `AbstractNautilusUi>>addNewGroup:`
- `AbstractNautilusUi>>renameGroup:`
- `CatalogBrowser>>installStableVersion:onSuccesss:`
- `CatalogBrowser>>loadConfiguration:onSucess:`
- `CatalogBrowser>>refresh`
- All variations of the ConfigurationOf classes also copy pasted the handler blocks
- `FileReference>>spotterPreviewContentsIn:`
- `FileReference>>gtInspectorContentsIn:`
- `GTObjectPrinter class>>asTruncatedTextFrom:`
- `GTObjectPrinter class>>asNonTruncatedTextFrom:`
- `GTSpotterProcessor>>filterInContext:`
- `GTSpotterProcessor>>continutFilterInContext:`
- `GtExampleFactory>>initializeSubjects:forExample:`

- `GtExampleFactory>>initializeSubjectsForMethod:forExample:`
- `GtExampleFactory>>initializePragmas:forExample:`
- `HismoAbstractHistory>>allEarliestEvolutionOfPropertyNamed:`
- `HismoAbstractHistory>>latestEvolutionOfPropertyNamed:`
- `HismoAbstractHistory>>allLatestEvolutionOfPropertyNamed:`
- `HismoAbstractHistory>>earliestEvolutionOfPropertyNamed:`

With this information we can finally define our requirements to our solution

3.2 Requirements to our solution

We have defined the requirements to our solution as follows.

1. Our solution must be modular. This means ...
   (a) ... it must be easy to add modular exceptions to a method.
   (b) ... it must be compatible with many methods instead of being tailored around one specific one.
2. We must be able to handle exceptions in multiple ways, like...
   (a) ... logging the exception.
   (b) ... re-throwing the exception.
   (c) ... returning to the caller with a default value.
   (d) ... catching the exception and simply resume the execution of the method.
3. Our solution must not be error prone itself, which means that...
   (a) ... modular exceptions must not allow users to insert code that cannot be compiled.
   (b) ... exceptions or other irregular program flows caused by our modular exceptions must be possible to debug.
   (c) ... modular exceptions must be stable. They should not crash the editor, IDE or other programming tools or cause corruption of save files.
4. (Optional) Changes to the code made by our solution must be documented or displayed somewhere.

We go through the reasoning for each requirement.

R. Modularity	The definition of modular requires the user to be able to add exception handling easily. It also requires these exception handling blocks to be compatible with many methods. Without this requirement being fulfilled it would not be "modular" exception handling.
R. Common cases	Modular exceptions must be able to perform the most common cases of exception handling, otherwise it would not be modular "exception handling". In Java the following exception handling types are very common. • Logging the exception is used in at least 28% of Java exception handling. • Re-throwing the exception is used at least 20% of the times in Java servers, server-apps and libraries. • Returning to the caller is used in around 10% of Java exception handling blocks and therefore common enough to be something we must support. • In Smalltalk resuming the method is a commonly used way to handle exceptions. We assume this type of exception handling to be useful in Java for handling exceptions within a loop. • In Java catching and re-throwing an exception is used at least 20% of the times in Java servers, server-apps and libraries.
R. Error tolerance	This requirement we found while creating prototypes in Smalltalk. The dynamic code recompilation approach we chose in our first prototype was able to insert uncompilable code into methods which would completely break that method. We also found that the MetaLink approach could fulfill all the other requirements but that exceptions caused by the MetaLink class crashed the editor and caused the save files to become corrupt. We realized that there was no point in creating a new approach to handle exceptions if that approach itself causes more instability than it fixes. That is why requirement 3 exists.

R. Display	This requirement is optional but a very welcome feature to have in our modular exceptions. When we created the AspectJ prototype we realized that it automatically showed us which methods were affected by our modular exceptions and also showed us which methods a particular modular exception affected. We decided to give a bonus point for approaches that fulfilled this requirement

We will use this list in order to rate the quality of the implementation and the quality of our prototypes.

4 Testing implementations in Smalltalk

Before we started to implement modular exceptions we first used Smalltalk in order to test different ideas and approaches. We then used the knowledge gathered during that process in order to create an implementation in Java. Smalltalk is a programming language that allows far deeper access to the system than conventional programming languages. Smalltalk allows methods to be changed and recompiled dynamically at run time. Not only that but method definitions are objects that are attached to the method dictionary of a class, which means we can replace methods at run time the same way instance variables can be updated. It also has the MetaLink class that allows us to link method calls to other method calls, which we used in one of our prototypes in order to link exception handling to method calls. We used the Pharo IDE version 6.1 [1] for all of our experiments and prototypes. We tested the following approaches in Smalltalk through creating prototypes.

1. The usage of dynamic method recompilation to dynamically add new lines of code to existing methods that handle exceptions.
2. Wrapper objects that wrap a method inside a new object that handles exceptions thrown in the method call
3. The MetaLink class allows method calls to be linked to a target method. This way exception handling could be dynamically added to methods

[1] `https://pharo.org`

What follows is a documentation of what we did to create each prototype and what we learned from them.

4.1 Smalltalk prototype 1: Dynamic Method Recompilation

Smalltalk has the unique property among programming languages in that it allows methods to be recompiled at run time. Our idea was to use this property to dynamically rewrite method implementations in order to write Exception handling into the method definition. We could do this at run time or in response to an exception. In Smalltalk the definition of a method can be gotten in text form with the following code.

```
code := (ReadStream >> next) definition.
```

Where ReadStream is the name of the a class to which the method `ReadStream>>next` belongs to. “next” is the selector of the method that we want to rewrite. The code above can be used with any other class and any other method. This code definition can be split by line breaks through the following code.

```
code := code findTokens: Character cr.
```

This way we have an array where each element represents one line of code. We can insert new lines of code into this array, like the following line.

```
newLine := 'parameter1 isNil ifTrue: [^self].'.
```

This line cancels the method execution if the parameter named parameter1 is nil. We insert this line after the method header with the following code.

```
code add: newLine afterIndex: 1.
```

We can then compile this code through the following line.

```
ReadStream compile: code.
```

This entire process can be simplified by creating a helper method that takes a block of code and inserts it after the method header.

```
preAppendCode: newLine toFunction: aSelector ofClass: aClass
   |oldCode newCode|
   oldCode := (aClass>>aFunction) definition.
   oldCode := oldCode findTokens: Character cr.
   oldCode add: newLine  afterIndex: 1.
   newCode := ''.
   code do:[:each | newCode := newCode,each,Character cr asString].
   aClass compile: newCode.
```

This method will insert the newLine string into the method definition after the header of aSelector of class aClass. We can use this in order to insert a line of code into the method to first check the parameters and return to a default value in case the parameters are invalid. This prevents the execution of the method in case the parameters are invalid. We can also use this approach to surround the method with a try-catch block.
While this approach could be used to add exception handling in a quick way there were several flaws to it that made it useless for our project.

1. It makes assumptions about the placement of line breaks. Technically line breaks are not necessary in Smalltalk. They have no effect on the code, therefore breaking up the code into pieces based on line breaks and assuming that each line is one statement is a fallacy.

2. It is not detectable through code, therefore making it impossible to prevent the exception handling being added multiple times to the same function due to a user's mistake.

3. It is hard to remove through code without making assumptions about the structure of the code. This makes it hard to dynamically replace with other Modular Exceptions.

We use our requirements to rate this approach

R. Modularity	Fulfilled. Modular exceptions can be added with a single line of code which fulfills part a. Part b is fulfilled if both methods require the exact same code to handle the exception.
R. Common cases	Fulfilled as we have the ability to insert literally any code we want. If the exception handling can be written in code by hand it can also be written dynamically.
R. Error tolerance	Not fulfilled. It is possible to insert code into a method that does not compile therefore breaking the method. Once broken this way it is hard to remove the inserted line unless the specific line number is known.
R. Display	Fulfilled. All code that was written dynamically appears in the editor immediately. It is not necessary to refresh the window or anything. It appears the moment the method gets recompiled.

This approach fulfills all but one requirement, but requirement 3 is essential. As stated before there is no point in creating a system for handling exceptions that itself is more prone to cause exceptions. We therefore abandon this solution and investigated different approaches instead.

4.2 Smalltalk prototype 2: Wrapper objects

In Smalltalk all methods are objects and like all objects they can be replaced by other objects. Instance variables in Smalltalk are dynamically typed, which means that an instance variable can store any object regardless of class and can also have its stored object replaced by another object. Methods called on the instance variable will throw a doesNotUnderstand exception if the stored object does not implement a method with the same selector. This means that so long as an object implements the right methods it can be inserted into any method dictionary of any class and will be called when the method is triggered on an object of this class. Our idea is to create our own wrapper object class [3, 3.7, p. 402] and insert an instance into the method dictionary of a class. This wrapper object would replace a target method in the dictionary and keep the old method stored inside as an instance variable. Having the old method definition stored in a variable would allow our wrapper to still call the old method definition and therefore keep the old functionality intact but also allows us to wrap our exception handling around this old method. In order for this to work our wrapper object class must have the following instance variables.

Variable name	**purpose of the variable**
targetSelector	The selector of the target method
targetClass	The class the target method belongs
oldMethodDef	The method definition of the target method. This can be gotten through the following lines. `oldMethodDef := targetClass lookupSelector: targetSelector.`

The following methods must be implemented in our wrapper object.

Method header	**Necessary functionality**
run:with:in:	This is the message sent to the method object when the method is called. This is where we implement exception handling. These are its parameters. 1. Its first parameter is the selector of the method called “aSelector” 2. Its second parameter are the method parameters called “arguments” 3. its last parameter is the target object called “aReceiver” It must at some point call the old method definition, which can be done through the following line of code `aReceiver withArgs: arguments executeMethod: oldMethodDef`
doesNotUnderstand:	Simply call the following line `^oldMethodDef perform: aMessage selector withArguments: aMessage arguments.` Where the aMessage is the parameter of the method.
install	Has to add itself to the method dictionary of the target class and target selector through the following lines `targetClass addSelector: targetSelector withMethod: self.`
uninstall	Has to replace the object with the old method implementation. This can be done through the following lines `targetClass addSelector: targetSelector withMethod: oldMethodDef.`

The object that replaces the method effectively only needs two parameters to do its job. It needs to know the target class and the target selector. Once this has been delivered it can read the method definition from the method dictionary of the class and save it in an instance variable. With the class and selector information it can replace the old method definition with itself. On top of that it can still call the old method.

This approach gives us the ability to wrap exception handling around a target method. Within the `ReflectiveMethod>>run:With:In` method we can use the `BlockClosure>>on:do:` command of Smalltalk (which does the same as a try-catch block would in Java) around the old method call as follows

```
run: aSelector with: arguments in: aReceiver
   [^aReceiver withArgs: arguments executeMethod: oldMethodDef]
   on: Error do:[^aReceiver].
```

Wrapper object have many advantages. We rate the approach with our requirements.

R. Modularity	Fulfilled. wrapper object can be created and applied easily. If the exception handling of two or more methods are the similar we can create a new wrapper object class in order to handle them.
R. Common cases	Fulfilled. We can run any code in our wrapper object and therefore can handle the exception in any way we wish.
R. Error tolerance	Fulfilled. wrapper object do not allow code to be inserted that itself does not compile. They do not throw any exception that crashes or corrupts the editor. Exceptions thrown by our wrapper object would travel upwards towards the caller of the target method and could be debugged from there.
R. Display	Not fulfilled. Methods do not show any information about what wrapper object they are affected by.

To prove that requirement 2 has been fulfilled we will show the `ReflectiveMethod>>run:with:in:` method for each exception handling type. The other methods of the wrapper object can be the same.

- Logging the exception

```
run: aSelector with: arguments in: aReceiver
[
   ^aReceiver withArgs: arguments executeMethod:
wrappedMethod.
]
on: Error do:
[:err |
self inform: err class name.
self inform: (err signalerContext stack).
].
```

- Re-throwing the exception

```
run: aSelector with: arguments in: aReceiver
[
   ^aReceiver withArgs: arguments executeMethod:
wrappedMethod.
]
on: Error do:
[:err |
err pass
].
```

- Returning to the caller while returning a default value

  ```
  run: aSelector with: arguments in: aReceiver
  [
     ˆaReceiver withArgs: arguments executeMethod:
   wrappedMethod.
  ]
  on: Error do:
  [:err | ˆreturnValue
  ].
  ```

  Note that returnValue must contain the value that the method returns to the caller. In case of a method that should not return value it should return the receiver object instead. If the method should return a value it should return a default value.

- Catch the exception but simply resume the execution of the method

  ```
  run: aSelector with: arguments in: aReceiver
  [
     ˆaReceiver withArgs: arguments executeMethod:
   wrappedMethod.
  ]
  on: Error do:
  [:err | err resume:= returnValue.
  ].
  ```

4.3 Smalltalk prototype 3: MetaLinks

The third Smalltalk prototype makes use of the structure in which all code is interpreted by the Smalltalk compiler. Smalltalk compiles all method calls into an Abstract Syntax Tree (AST) [2], which holds the parameters of each method call in its branches and the method calls are saved as roots to these branches. Through the MetaLink class we can create a MetaLink object that can be inserted into this tree to allow us to execute a block of code whenever a target method is called. We can have this block executed before, after or instead of the method we link the MetaLink to. Our idea is to have our MetaLink code block executed instead of the method itself, so that we can wrap it into a try-catch block. The following lines of code are used to create a MetaLink

[2]http://marcusdenker.de/talks/18LectureMetaLinks/MetaLinks.pdf

```
aMetaLink := MetaLink new
   metaObject: [:aReceiver :aSelector :args :link|
      ''Execute code here''
      ];
   selector: #value:value:value:value: ;
   arguments: #(object selector arguments link);
   control: #instead.
```

Each MetaLink object has four relevant properties that are initialized here. The metaObject is a code block in which we can do whatever we wanted the system to do in case of an exception. The selector variable is the message that is used to start the code block. In Smalltalk block code is triggered using the `BlockClosure>>value:` message with the parameter following the `BlockClosure>>value:` keywords. In case there are multiple parameters we need to use the method `BlockClosure>>value:value:` or `BlockClosure>>value:value:value:`or `BlockClosure>>value:value:value:value:`. The parameters used in this example are defined in the arguments variable. Our parameters for the metaObject block are the target object, the selector, the parameters of the method call and the MetaLink object itself. In the control variable we can set the MetaLink to trigger before, after or instead of the target method. Creating a MetaLink object does not have an effect on its own. It needs to be linked to the method call manually. This is done through the following line which we performed after the MetaLink instance has been created.

```
   (ReadStream>>next) ast link: aMetaLink.
```

Where “ReadStream” is the Class of the method we link to, “next” is the selector of the method we link, and aMetaLink is the MetaLink we want to use. Like with the wrapper object approach we need to have the ability to execute our target method inside the MetaLink code block so that we can perform exception handling outside the target method. To do this we need a way to call the target method from within our MetaLink. This seems easy at first since we have the object, the selector and the arguments of the method call available as parameters in our code block. This enables us to get the method definition and call the target method from within our code block with the following metaObject example.

```
metaObject: [:aReceiver :aSelector :args :link|
   theMethod := (aReceiver class lookupSelector: aSelector).
   aReceiver withArgs: args executeMethod: theMethod.
      ];
```

This approach however has one fatal flaw. Since calling the target method always triggers our MetaLink instead and since our MetaLink now triggers the target method again we have an infinite loop. Before we can call our target method from within the MetaLink we need to make sure our MetaLink does not get triggered again. The most simple approach to this problem is to uninstall the MetaLink before calling the target method and then reinstall the MetaLink afterwards. So our metaObject must be changed to this form.

```
metaObject: [:aReceiver :aSelector :args :link|
   |result|
   link uninstall.
   theMethod := (aReceiver class lookupSelector: aSelector).
   result := aReceiver withArgs: args executeMethod: theMethod.
   (aReceiver class>>aSelector) ast link: link.
   result.
   ];
```

In our metaObject we uninstall the MetaLink object before we call the old method. MetaLink implements the uninstall message by default so we do not have to implement that ourselves. It simply unlinks the MetaLink from the target method call. After that we call the target method while saving its return value in a local variable. We then reinstall the MetaLink after the target method has been called. We saved the return value of the target method call in the local variable called “result” because the value of a block code is the return value of the last statement inside the block code. Since we need to reinstall the MetaLink again after the target method we need to temporarily save its return value so that we can have it as the last statement in the block. Through this process we can wrap any method into a MetaLink object in order to wrap it into a try-catch block. We use our requirements to rate this approach.

R. Modularity	Fulfilled. MetaLinks are very similar to WrapperObjects and therefore fulfill both parts of the requirement.
R. Common cases	Fulfilled. We are free to run any code inside the MetaLink.
R. Error tolerance	Not Fulfilled. Exceptions caused by the MetaLink crashed the editor and corrupted the save files. We found the approach to not be stable.
R. Display	Not fulfilled. Methods do not show any information about what MetaLinks they are affected by.

To prove requirement 2 we show the metaObject for each exception handling type

- Logging the exception

```
metaObject: [:object :selector :args :link|
         |aMethod result|
         link uninstall.
```

```
        aMethod := (object class lookupSelector: selector).
        [
           result := object withArgs: args executeMethod:
   aMethod.

        ]
        on: Error do: [:err |
           self inform: err class name.
           self inform: (err signalerContext stack).
        ].
        (object class>>selector) ast link: link.
        result.
     ];
```

- Re-throwing the exception

```
metaObject: [:object :selector :args :link|
        |aMethod result|
        link uninstall.
        aMethod := (object class lookupSelector: selector).
        [
           result := object withArgs: args executeMethod:
   aMethod.

        ]
        on: Error do: [:err |
           err pass.
        ].
        (object class>>selector) ast link: link.
        result.
     ];
```

- Returning to the caller with a return value.
  Note that the return value that we put into the "result" variable in case of an exception is hard coded. To dynamically set the return value one would have to create a subclass of MetaLink in order to store the return value and find a way to use it as a parameter for the metaObject code block. This would require rewriting the process that MetaLink uses to run the metaObject block code on top of having to implement the method `BlockClosure>>value:value:value:value:value` in order to run blocks that have five parameters. Due to a lacking documentation for the MetaLink class we were unable to implement these methods. That is why we decided to simply have the return value hard coded into the metaObject in our prototype.

```
metaObject: [:object :selector :args :link|
        |aMethod result|
         link uninstall.
         aMethod := (object class lookupSelector: selector).
         [
            result := object withArgs: args executeMethod:
   aMethod.

         ]
         on: Error do: [:err |
            result := 0.
         ].
         (object class>>selector) ast link: link.
         result.
      ];
```

- Catch the exception but simply resume the execution of the method

```
metaObject: [:object :selector :args :link|
         |aMethod result|
         link uninstall.
         aMethod := (object class lookupSelector: selector).
         [
            result := object withArgs: args executeMethod:
   aMethod.

         ]
         on: Error do: [:err |
            err resume: nil.
         ].
         (object class>>selector) ast link: link.
         result.
      ];
```

This shows that all the common ways to handle exceptions are possible with MetaLinks. The MetaLink approach is very lightweight and much more performant than the wrapper object approach. However it has a few drawbacks. It is impossible to stack MetaLinks, so using multiple Modular Exceptions on the same method was impossible. On top of that we find it to be very error prone. An exception inside the MetaLink is hard to debug as it often crashes the editor itself. The MetaLink class itself is barely documented at all and there is a lot of code being run behind the scenes. This makes it hard to figure what the MetaLink is doing behind the scene. We consider the MetaLink approach sub-optimal.

4.4 Conclusion of the Smalltalk Prototypes

Smalltalk's deep access to the system allows us to do things that would otherwise be impossible to implement in mainstream programming languages like Java or C++. The wrapper object approach fulfills all our obligatory requirement which means it would be sufficient to simply port it to Java. However Java does not give us deep enough access to the system in order to dynamically replace methods. Therefore we have to find different approaches in order to implement modular exceptions in Java. We did prove that it is possible to create a solution in Smalltalk that fulfills our requirements. Now we only had to find an approach in Java.

5 Testing implementations in Java

After the Smalltalk prototypes we investigated approaches to implement modular exceptions in Java. Java allows a lot less control over the execution of code compared to Smalltalk. For instance, methods in Java are not objects, so we can not change those at run time. MetaLinks also do not exist in Java. Still we needed a way to nest a target method into a wrapper method so that we could wrap it into an exception handler block. We tested two different approaches: Bytecode transformation and Aspects.

5.1 Java Prototype 1: Bytecode transformation

One approach we investigated is the usage of Bytecode transformation, which allows us to rewrite the code of methods dynamically. This can be done through BCEL[1], an external jar library. Once downloaded and unzipped the BCEL package comes with multiple jar files and multiple example projects. We use the helloify.java example to test the effectiveness of this approach. The helloify class has a main method that requires the path to and the name of a class and then creates a new class in which all methods print "hello" plus the method name. We reprogram that class to instead overwrite the old class with these changes by having the JavaClass object use the `helloify>>dump` method with the original class's file path. We realize that Bytecode transformation has many disadvantages.

[1]`https://commons.apache.org/proper/commons-bcel/`

1. It is impossible to see the changes in the source code or anywhere else. The changes made by our program are therefore effectively undocumented

2. It is impossible to prevent the transformation from being applied twice to the same code. A user could accidentally apply the transformation twice which will impact code performance.

3. Inserting new lines with Bytecode transformation requires assumptions to be made about the code structure. Inserting a handler block dynamically with Bytecode transformation requires knowledge of where exactly the catch block is within a method.

4. Bytecode transformation cannot be used to insert code directly. Instead the user must use the BCEL Instruction class and create an InstructionList object. BCEL uses its own methods to create these instructions. For example the following code is the method `helloify>>helloifyMethod` of the helloify class in the BCEL example project.

```
/**
    * Patch a method.
    */
   private static Method helloifyMethod(Method m) {
       Code code = m.getCode();
       int flags = m.getAccessFlags();
       String name = m.getName();

       // Sanity check
       if (m.isNative() || m.isAbstract() || (code == null)) {
           return m;
       }

       // Create instruction list to be inserted at method
  start.
       String mesg = "Hello from " + Utility.
  methodSignatureToString(m.getSignature(),
               name,
               Utility.accessToString(flags));
       InstructionList patch = new InstructionList();
       patch.append(new GETSTATIC(out));
       patch.append(new PUSH(cp, mesg));
       patch.append(new INVOKEVIRTUAL(println));

       // make a new method
       MethodGen mg = new MethodGen(m, class_name, cp);
       InstructionList il = mg.getInstructionList();
       InstructionHandle[] ihs = il.getInstructionHandles();
       if (name.equals("<init>")) { // First let the super or
   other constructor be called
           for (int j = 1; j < ihs.length; j++) {
               if (ihs[j].getInstruction() instanceof
   INVOKESPECIAL) {
                   il.append(ihs[j], patch); // Should check:
   method name == "<init>"
                   break;
               }
           }
       } else {
           il.insert(ihs[0], patch);
       }

       // Stack size must be at least 2, since the println
   method takes 2 argument.
       if (code.getMaxStack() < 2) {
           mg.setMaxStack(2);
       }

       m = mg.getMethod();

       il.dispose(); // Reuse instruction handles

       return m;
   }
```

It rewrites a method to print “Hello from ” + the header of the method whenever that method is called. For example the method `public static void main(String[] arg0)` would print “Hello from public static void main(String[] arg0)”. Note how the the method creates an InstructionList object and then appends three new instructions to it. These instructions do not use regular Java code and instead use BCEL’s instruction system. This makes it much more difficult to use as one would have to learn this instruction system to implement modular exceptions. Users of our solution would have to learn a completely new way of programming in order to handle exceptions this way.

Due to these drawbacks we decided to pursue a different approach without investigating if it fulfilled our requirements.

5.2 Java Prototype 2: AspectJ Development Tools

Aspects [4] in Java allow us to call an aspect method whenever a target method is called. This aspect method can either be run before, after or around the target method call. Aspects also allow us to gain access to the parameters and other data related to the method call. It is also possible to change the parameters or the target object of the method call before the execution. Java does not natively support aspects, therefore an external jar file must to be downloaded and added to the Java project. This can be done manually but it is preferable to download and install the AspectJ Development Tools (AJDT) through Eclipse. For this prototype we choose Eclipse 4.4.2 (LUNA version) with Java version 8 and AJDT for Eclipse 4.4 .
The AJDT can be installed with the “install New Software..” button in the “help” tab in Eclipse. The URL to download the add-on from depends on the Eclipse version. For version 4.4 it is the following one `http://download.eclipse.org/tools/ajdt/44/dev/update`. For manual download the URL is as follows `https://www.eclipse.org/ajdt/downloads/`.
Once installed AspectJ projects can be created that can use the Aspect files. Aspects can only be implemented in Aspect files and not in common Java Class files. Therefore all the code concerning advices and pointcuts must be written into an Aspect file. For more specific information concerning AspectJ refer to section 8.2.

5.2.1 Handling exceptions with AspectJ

To showcase how exception handling can be done with pointcuts and advices we created an example project that has methods that can throw exception and errors. We use AspectJ to handle these exceptions in a modular way to prove that this approach is viable. Our example project is a student database that allows users to create students and add them to the database. It also supports searching for a student with a certain name and last name. Here is the code of the database itself.

```
package StudentDataBaseExample;

import java.util.ArrayList;

public class StudentDatabase {

   ArrayList<Student> students = new ArrayList<Student>();

   public void addNewStudent(Student newStudent) throws Exception {
      if (doesStudentExist(newStudent))
         throw new Exception("Student " + newStudent.getName()
               + " already exists");
      students.add(newStudent);
   }
```

```
    public void renameStudent(Student student, String newStudentName,
          String newStudentLastName) throws Exception {

       //Check if the student with the new names would conflict with
    the database
       if (doesStudentExist(new Student(newStudentName,
    newStudentLastName)))
          throw new Exception("Student " + newStudentName + " already
    exists");

       student.setName(newStudentName);
       student.setLastname(newStudentLastName);
    }

    public Student getStudentByNameAndLastname(String name, String
    lastname) {
       for (Student item : students) {
          if (item.getName() == name && item.getLastname() == lastname
    )
             return item;
       }
       return null;
    }

    boolean doesStudentExist(String studentFirstName, String
    studentLastName) {
       return this.getStudentByNameAndLastname(studentFirstName,
             studentLastName) != null;
    }
}
```

This class has four methods here.

- The method `addNewStudent(Student)` adds a Student object to the database and throws an exception if a Student object with the same name and last name already exists.
- The method `renameStudent(Student,String,String)` renames a Student object and throws an exception if a student with the new name and new last name already exists within the database.
- The method `getStudentByNameAndLastname(String,String)` returns the student with the name and last name specified and returns null if no such student exists.
- The method `doesStudentExist(String,String)` returns true if a student with the same name and last name is already in the database.

The student class is defined as follows.

```
package StudentDataBaseExample;

public class Student {

   String name;
   String lastname;

   public Student(String name, String lastname){
      this.name = name;
      this.lastname = lastname;
   }

   String getName(){
      return name;
   }

   void setName(String newName){
      name = newName;
   }

   String getLastname(){
      return lastname;
   }

   void setLastname(String newName){
      lastname = newName;
   }

   public boolean equals(Object other){
      Student otherStudent = (Student) other;
      return otherStudent.getName() == this.name && otherStudent.
   getLastname()==this.lastname;
   }

   public String toString(){
      return "Student: " +name +" "+lastname;
   }

}
```

This class has the following methods

- The constructor of this class which takes tow String values, one for the name and one for last name.
- The methods `getName()`, `getLastname()`, `setName(String)` and `setLastname(String)` are typical getters and setters for the name and lastname variable.

- The overridden `equals(Object)` method is currently not used for the database but we are using it in our examples to show how to handle exceptions with AspectJ.
- The overridden `toString()` method is a helper method that we used to debug our examples.

5.2.2 Implementing “returning immediately” exception handling

For our first example we show how to implement an exception handling block that cancels the execution of a method in case of an exception and continues where the caller left off. To do this we first showcase a test unit for our exception handling.

```
package StudentDataBaseExample.DatabaseCancelExample;

import StudentDataBaseExample.Student;
import StudentDataBaseExample.StudentDatabase;

public class StudentDatabaseTestCancel {

   public static void main(String[] args) throws Exception{

      StudentDatabase DB = new StudentDatabase();
      DB.addNewStudent(new Student("Tom", "Knott") );
      DB.addNewStudent(new Student("Tom", "Knott") );
      //This should cause a "Student already exists" Exception

      Student alexSpencer = new Student("Alex", "Spencer");
      DB.addNewStudent(alexSpencer);
      DB.renameStudent(alexSpencer,"Tom","Knott" );
      //This should cause a "Student already exists" Exception

   }
}
```

This main method attempts to add a student with the name “Tom Knott” twice, which will cause the method `StudentDatabase>>addNewStudent(Student)` to throw an exception as that student is already inside the database. Then the second half of the main method creates the student “Alex Spencer” and adds it to the database. Then it attempts to rename that student to “Tom Knott” which will cause another exception to be thrown in the method `StudentDatabase>>rename(Student,String,String)`.

In order to implement the “returning immediately” Exception handling type we use a pointcut that affects both the method `StudentDatabase>>addNewStudent(Student)` and `StudentDatabase>>rename(Student,String,String)`. For this example we will restrict our pointcut to only affect these methods if they were called from within the main method of the StudentDatabaseTestCancel class listed above. The pointcut in question would look something like this.

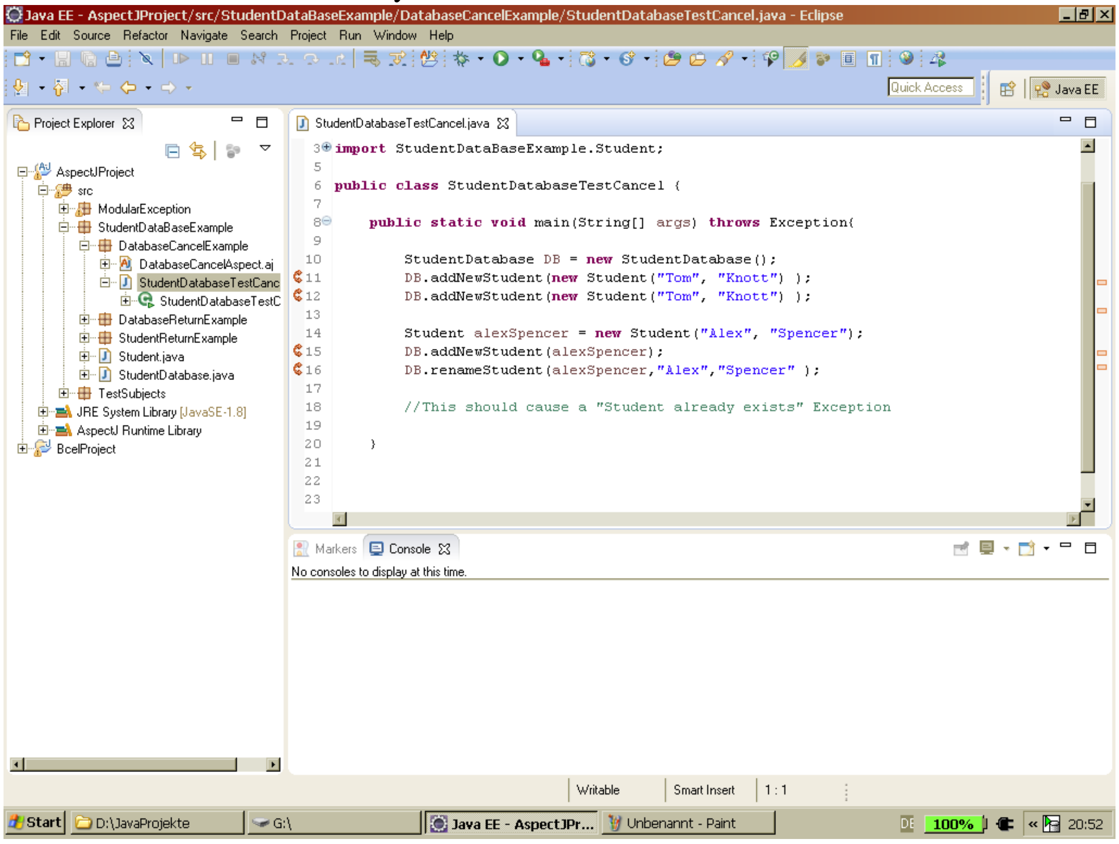
```
pointcut methodsToCancelExceptions():
      (
      call (void *.StudentDatabase.renameStudent(Student,String,
   String))
      ||
      call (void *.StudentDatabase.addNewStudent(Student))
      )
      &&
      withincode(void *.StudentDatabaseTestCancel.main(String[] ))
   ;
```

This code must be written into an aspect file in order to work. The advice we want to apply with this pointcut is one that catches the exception thrown by the method and does nothing with it. That will cancel the method execution and continue the program in the main method at the very next line.

```
void around(): methodsToCancelExceptions(){
      try{
         proceed();
      }catch(Exception e){
      }

   }
```

If everything was done correctly Eclipse will display arrow symbols next to the method calls `StudentDatabase>>addNewStudent(Student)` and `StudentDatabase>>rename(Student,String,String)`. This signals that the method calls are affected by an advice.



```java
import StudentDataBaseExample.Student;

public class StudentDatabaseTestCancel {

    public static void main(String[] args) throws Exception{

        StudentDatabase DB = new StudentDatabase();
        DB.addNewStudent(new Student("Tom", "Knott") );
        DB.addNewStudent(new Student("Tom", "Knott") );

        Student alexSpencer = new Student("Alex", "Spencer");
        DB.addNewStudent(alexSpencer);
        DB.renameStudent(alexSpencer,"Alex","Spencer" );

        //This should cause a "Student already exists" Exception

    }
```

```
No consoles to display at this time.
```

When we run the main method we notice that no exceptions reach the main method anymore. Our pointcut has successfully intercepted the exception and stopped the execution of the method. This approach handles exceptions in a modular way. The pointcut can be expanded to include any method if needed and will handle it the exact same way. It is also possible to change the handler block to print the exception that was thrown by changing the advice to the following.

```
void around(): methodsToCancelExceptions(){
      try{
         proceed();
      }catch(Exception e){
      e.printStackTrace();
      }
```

5.2.3 Implementing “return default value” exception handling

For our next example we will handle an exception with a “return default value” exception handler. This handler should cause the method to return a default value in case of an exception. To create this exception handler we first create a use case scenario as follows.

```
package StudentDataBaseExample.DatabaseReturnExample;

import StudentDataBaseExample.Student;
import StudentDataBaseExample.StudentDatabase;

public class StudentDatabaseTestReturnDefault {

   public static void main(String[] args) throws Exception{

      StudentDatabase DB = new StudentDatabase();
      DB.addNewStudent(null);
      Student alexSpencer = new Student("Alex","Spencer");
      DB.addNewStudent(alexSpencer);
      Student foundStudent = DB.getStudentByNameAndLastname("Alex", "
   Spencer");
      assert foundStudent == alexSpencer;

   }
}
```

This main method creates a database object and inserts a null student and a valid student into the database. The method `StudentDatabase>>addNewStudent(Student)` will first check if the Student in question already exists with the method `StudentDatabase>>doesStudentExist(String,String)`. This will cause a NullPointerException to be thrown when the methods `Student>>getName()` and `Student>>getLastname()` are called on the null student. To prevent this we will wrap these methods with our pointcut and advice to have them return an empty string instead. The following pointcut will affect both method calls.

```
pointcut methodResumeString():
      call (String StudentDataBaseExample.Student.getName())
      ||
      call (String StudentDataBaseExample.Student.getLastname(..))
   ;
```

The default value to return in case of an exception should be an empty string. The advice we use is as follows.

```
String around(): methodResumeString(){
      try {
         return proceed();
      } catch (Exception e) {
         return "";
      }
   }
```

This will cause the method `StudentDatabase>>addNewStudent(Student)` to check for a Student object with an empty name and last name when the parameter is a null student instead of throwing an exception. The method `StudentDatabase>>getStudentByNameAndLastname(String,String)` will compare the input strings to an empty string in case one of the elements inside the Student object ArrayList is a null pointer. This example may not be optimal as the null student was still inserted into the ArrayList of the database instead of being rejected, but it does show that the same kind of exception handling can be reused across multiple method calls and how one can fix multiple methods this way. Just as with the previous approach one can also have the exception printed before returning to the caller by changing the advice to the following.

```
String around(): methodResumeString(){
      try {
         return proceed();
      } catch (Exception e) {
         e.printStackTrace();
         return "";
      }
   }
```

5.2.4 Implementing "resume execution" exception handling

Resuming the execution after an exception is also easy to implement with our approach. We reuse the same main method that we wrote for the "return default value" exception handling implementation.

```
package StudentDataBaseExample.DatabaseResumeExample;

import StudentDataBaseExample.Student;
import StudentDataBaseExample.StudentDatabase;

public class StudentDatabaseTestResume {

   public static void main(String[] args) throws Exception{

      StudentDatabase DB = new StudentDatabase();
      DB.addNewStudent(null);
      Student alexSpencer = new Student("Alex","Spencer");
      DB.addNewStudent(alexSpencer);
      Student foundStudent = DB.getStudentByNameAndLastname("Alex", "
   Spencer");
      assert foundStudent == alexSpencer;

   }

}
```

The exceptions that this code would throw remain the same. The method `StudentDatabase>>addNewStudent(Student)` will cause an exception as it needs to get the name and lastname of a null student. To handle this we create a pointcut that affects all method calls within this main method that return void and use it in an advice that wraps the method call into a dynamic try-catch block.

```
package StudentDataBaseExample.DatabaseResumeExample;

public aspect DatabaseResumeAspect {

   pointcut methodResumeString():
      withincode(void StudentDatabaseTestResume.main(String[]))
      &&
      call (void *(..))
   ;

   void around(): methodResumeString(){
      try {
         proceed();
      } catch (Exception e) {

      }
   }
}
```

Once this has been done the exception thrown by the method `StudentDatabase>>addNewStudent(Student)` will be caught and the main method will simply resume.

5.2.5 Rating the AspectJ approach

R. Modularity	Fulfilled. Aspects and advices can be added with a single line of code. Advices themselves can be written in a way to be compatible with many methods.
R. Common cases	Fulfilled. Advices can handle exceptions in any way necessary
R. Error tolerance	Fulfilled. Advices cannot insert code that cannot be compiled. Advices themselves do not throw any exceptions, only the code written inside the advice by the user can throw an exception. Advices did not cause any crashes during our experimentation with prototypes
R. Display	Fulfilled. AspectJ shows a symbol next to any method that is affected by a pointcut. Pointcuts themselves also show how many methods they are currently affecting.

To prove that requirement 2 has been fulfilled we created show examples of pointcuts and advices handling exceptions in the student database.

- Logging the exception

```
/*
 * Logging the exception
 */

pointcut methodsToPrintExceptions():
   withincode (* StudentDataBaseExample.StudentDatabase.
addNewStudent(..))
   || withincode (* StudentDataBaseExample.StudentDatabase.
renameStudent(..))
;


pointcut printStudentDataBaseErrors(Exception exception):
   methodsToPrintExceptions()
   && handler(Exception)
   && args(exception)
;


before(Exception e): printStudentDataBaseErrors(e){
   e.printStackTrace();

}
```

- Returning a default value

```
pointcut methodsToReturnDefaultBooleanValue():
      call(boolean *.StudentDatabase.doesStudentExist(..))
      || call(boolean *.StudentDatabase.doesStudentHaveName(..))
   ;

   before(): methodsToReturnDefaultBooleanValue(){

   }

   boolean around():methodsToReturnDefaultBooleanValue(){
      try{
         return proceed();
      }catch(Exception e){
         e.printStackTrace();
         return false;
      }
   }
```

- The code of the methods affected is as follows

```
boolean doesStudentExist(Student student){
      return getStudentByName(student.getName())!= null;
   }

boolean doesStudentHaveName(Student student, String name){
      return student.getName() == name;
   }
```

- Resume the operation

```
pointcut methodResume():
      withincode (* StudentDataBaseExample.StudentDatabase.
   getStudentByName(..))
      && call (boolean *(..))
   ;

   boolean around(): methodResume(){
      try{
         return proceed();
      }catch (Exception e){
         return false;
      }

   }
```

5.3 Conclusion of the Java Prototypes

Using AspectJ we realize that it is the ideal approach to our problem. It fulfills all our obligatory requirements and even fulfills the optional requirement 4. Bytecode transformation itself has many disadvantages that make it useless to us. Therefore we decide to use AspectJ for our final product. Now that we have found the optimal approach the only thing that needs to be done is creating a project that everyone can download and use with ease.

6 The Validation

In 3.2 we defined our requirements. To showcase that our requirements have been met we use code from the student database example shown in 5.2.1. The prove that our solution fulfills all our requirements goes as follows.

6.1 Validating Requirement 1 (Modularity)

Requirement 1 is mostly concerned with usability and compatibility. Our solution must be easy to apply (requirement 1 a) and be compatible across methods and classes (requirement 1 b) .

6.1.1 Validating Requirement 1a

Modular exceptions can be added to a method with very little code. What follows is an example of how to add modular exceptions to the methods `Student>>getName()` and `Student>>getLastname()`. The following code is the content of an AspectJ file.

```
   package StudentDataBaseExample.DatabaseReturnExample;

import StudentDataBaseExample.*;

public aspect DatabaseReturnAspect {

   pointcut methodResumeString():
      (
      call (String StudentDataBaseExample.Student.getName())
      ||
      call (String StudentDataBaseExample.Student.getLastname())
      )
   ;

   String around(): methodResumeString(){
      try {
         return proceed();
      } catch (Exception e) {
         return "";
      }

   }

}
```

This is very little code and handles all occurrences of the two methods. Therefore our solution allows us to add modular exceptions with ease.

6.1.2 Validating Requirement 1b

Our modular exceptions are compatible with many methods. This is evident in the previous example where we handled exceptions in two methods with the same file. The same code shown before can be expanded to handle exceptions in more methods so that they too return a default String value in case of any exception. Our modular exceptions are therefore compatible with many methods.

6.2 Validating Requirement 2 (Common cases)

Requirement 2 is concerned with the functionality of our solution. It must be possible to handle the most common kinds of exception with our solution. To prove that each way of handling an exception is possible with our solution we simply showcase an example of an exception being handled with our solution. The examples in question are from our student database example made in 5.2.1. These examples are included in the project files in the StudentDataBaseExample package

6.2.1 Validating Requirement 2a

```
   package StudentDataBaseExample.DatabaseRethrowExample;
```

```
import StudentDataBaseExample.*;

public aspect DatabaseRethrowAspect {

   pointcut methodsToRethrowExceptions():
      (
      call (void *.StudentDatabase.renameStudent(Student,String,
String))
      ||
      call (void *.StudentDatabase.addNewStudent(Student))
      )
      &&
      withincode(void *.StudentDatabaseTestRethrow.main(String[] )
)
   ;

   void around() throws Exception: methodsToRethrowExceptions(){
      try{
         proceed();
      }catch(Exception e){
         e.printStackTrace();
         throw e;
      }
   }
}
```

6.2.2 Validating Requirement 2 b

The example shown previously can also re-throw a caught exception. This proves that our solution can handle catching exception, logging them and then re-throwing the previously caught exception.

6.2.3 Validating Requirement 2 c

The following example shows that our modular exceptions solution can handle exceptions by canceling the method and returning a default value.

```
package StudentDataBaseExample.DatabaseReturnExample;

import StudentDataBaseExample.*;

public aspect DatabaseReturnAspect {

   pointcut methodReturnDefaultString():
      (
      call (String StudentDataBaseExample.Student.getName())
      ||
      call (String StudentDataBaseExample.Student.getLastname())
      )
   ;

   String around(): methodReturnDefaultString(){
      try {
         return proceed();
      } catch (Exception e) {
         return "";
      }
   }
}
```

6.2.4 Validating Requirement 2 d

The following example shows that our modular exceptions solution can handle exceptions by resuming the method that threw the exception. Note that in order to do so we need to simply handle all the exceptions thrown by methods called from our target method.

```
   package StudentDataBaseExample.DatabaseResumeExample;

   public aspect DatabaseResumeAspect {

      pointcut methodResumeString():
         withincode(void StudentDatabaseTestResume.main(String[]))
         &&
         call (void *(..))
      ;

      void around(): methodResumeString(){
         try {
            proceed();
         } catch (Exception e) {

         }
      }
   }
```

6.3 Validating Requirement 3 (Error tolerance)

Requirement 3 is about error tolerance and safety. Our solution should not introduce more bugs and errors than it fixes. Exceptions caused by our solution must be debuggable like any other exception.

6.3.1 Validating Requirement 3 a

Advices and pointcuts are covered by the compiler. It is impossible to run code in an advice that would not compile. This makes it less error prone than solutions that manipulate source code or byte code directly as those solutions often allows one to run code with compiler errors inside.

6.3.2 Validating Requirement 3 b

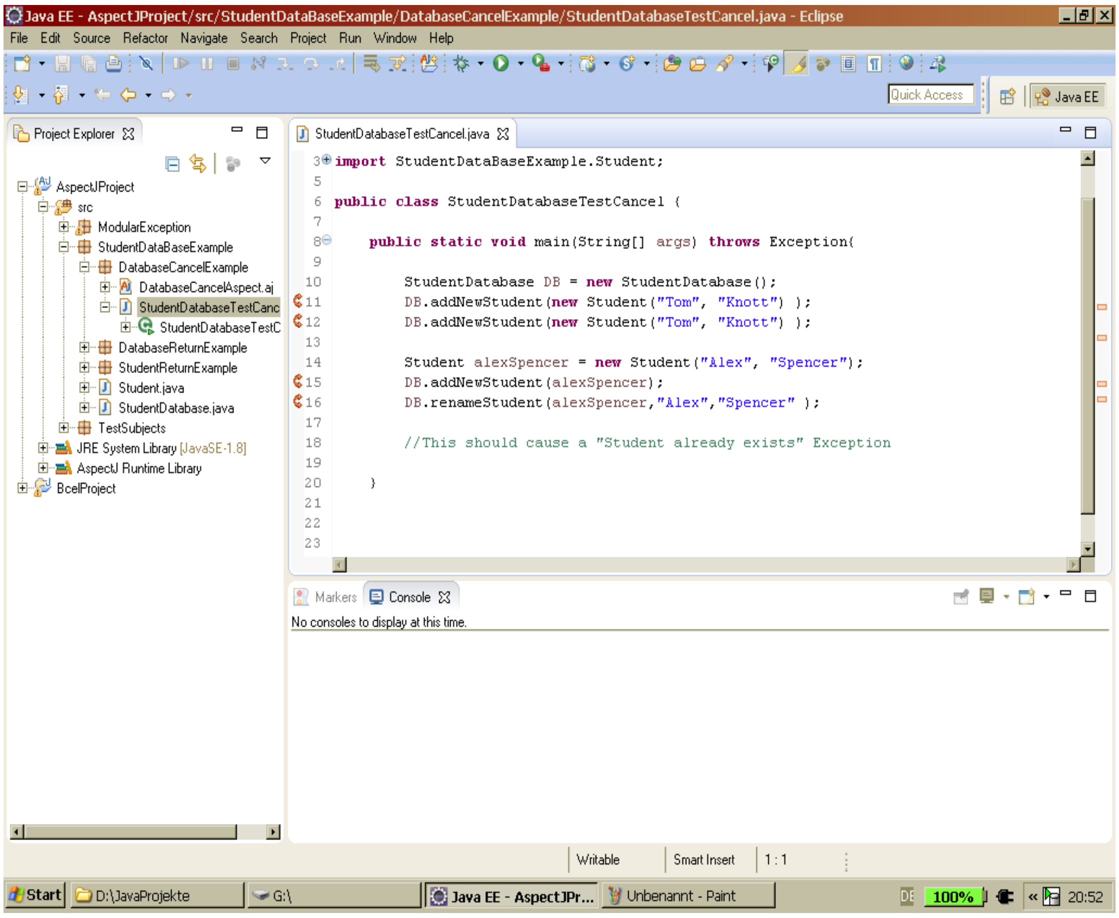
Exceptions thrown from inside the advice travel up the stack like any other exception would if thrown from within a regular method call. This is proven by the re-throw exception handling type being possible to implement at all. This makes debugging advices as easy as debugging regular code.

6.3.3 Validating Requirement 3 c

We have not experienced any crashes during our experimentation with AspectJ nor with our own solution.

6.4 Validating Requirement 4 (Display)

Requirement 4 is about user interface. The changes made by our solution to the base code should be documented in a way. The changes made to the code by our solution are documented in multiple ways. In Eclipse method calls that have been affected by an aspect have a symbol to their left.



Java EE - AspectJProject/src/StudentDataBaseExample/DatabaseCancelExample/StudentDatabaseTestCancel.java - Eclipse

```java
import StudentDataBaseExample.Student;

public class StudentDatabaseTestCancel {

    public static void main(String[] args) throws Exception{

        StudentDatabase DB = new StudentDatabase();
        DB.addNewStudent(new Student("Tom", "Knott") );
        DB.addNewStudent(new Student("Tom", "Knott") );

        Student alexSpencer = new Student("Alex", "Spencer");
        DB.addNewStudent(alexSpencer);
        DB.renameStudent(alexSpencer,"Alex","Spencer" );

        //This should cause a "Student already exists" Exception

    }



```

No consoles to display at this time.

This makes it easy to track down the aspect file that is responsible for all the changes made to the code. All the changes to the code can be seen in the aspect file. This makes it possible to figure out what code is being executed in a method affected by an aspect.

7 Conclusion and Future Work

We have studied exception handling and created a requirements list to our solution from the knowledge gathered. With this requirements list we were able to weed out undesirable approaches. Once we had found our ideal solution we created an example project that showcases how our solution fixes the problem of exception handling being hard to reuse.

Our solution can easily implement the most common types of exception handling in Java with ease. It is compatible across multiple methods and multiple classes. These modular exceptions can be implemented all in one file or split across many files if the user wants/needs to. Changes made to the code through our modular exceptions approach are signalized in Eclipse and can therefore be tracked down. This means that readability of code is not negatively affected (should mention that somewhere first!)

Modular exceptions provide an easy way to reuse exception handling across multiple methods. This massively increases flexibility in exception handling which is important since demands to software can change quickly. Code structure that changes a lot by definition needs exception handling that can change with it. Modular exceptions therefore allows exception handling to keep up with the changes in demands to software.

Future possibilities to this project include a native integration in the most common IDEs for Java, namely Eclipse and Netbeans. Currently there is no way to get an overview about the changes made to the software through modular exceptions. This can make projects confusing if there are many modular exceptions at work, so an improved interface would be very welcome.

It remains to be seen if programmers will use modular exceptions at all. All the systems in the world will not help if programmers continue to copy paste exception handling blocks across methods, as we saw it done in Smalltalk. We do know that Java programmers tend to be more concerned with exception handling [1] since exceptions in Java have to be caught at some point. This means that modular exceptions could at least become a big part of Java programming.

8 Anleitung zu wissenschaftlichen Arbeiten

This appendix features the following sections.

- a tutorial that consists of several examples on how to use modular exceptions.
- an API documentation that gives more specific information on how to use modular exceptions.
- implementation notes that describe how it was implemented.

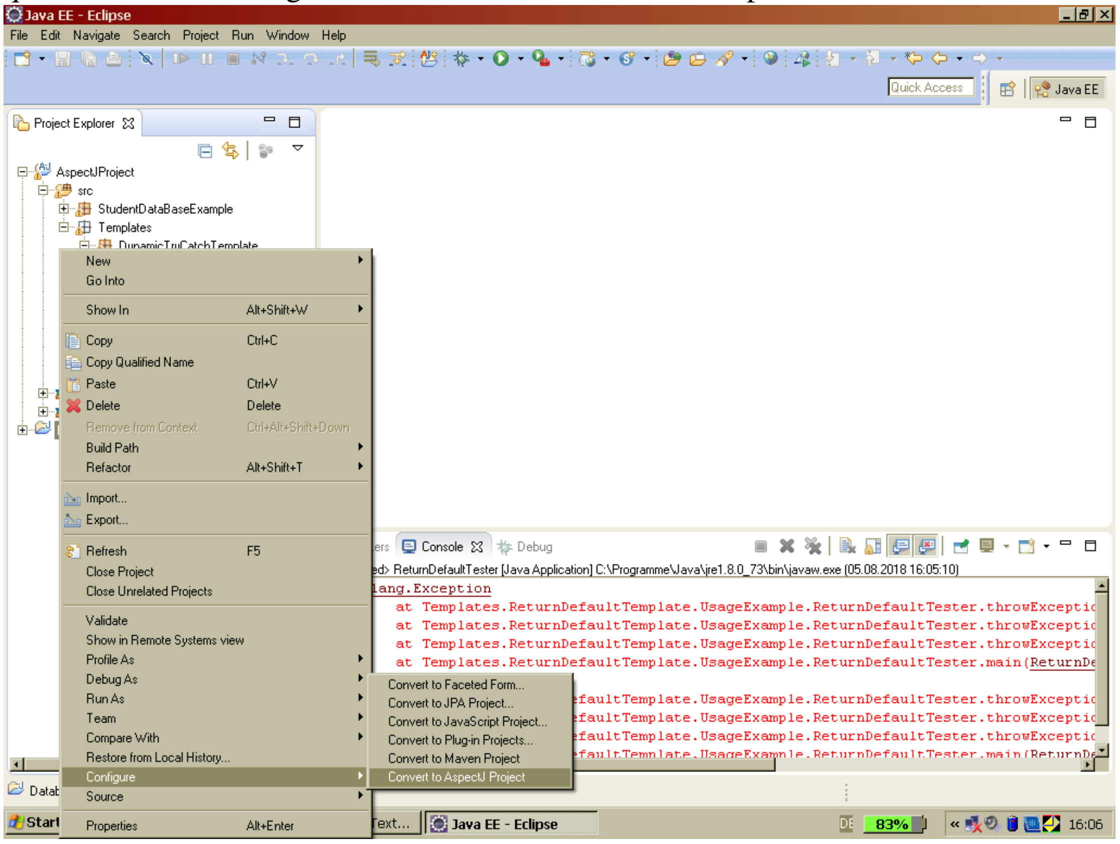
8.1 Tutorial

This tutorial consists of a setup tutorial and a few example that show how common exceptions can be handled.

8.1.1 Setup

The project can be downloaded from GitHub at the following link. `https://github.com/PIndermuehle/JavaModularExceptions`
The package can only be used with Eclipse 4.4 or higher and Java 1.8 or higher.
Our project requires AspectJ which can be downloaded at the following link
`https://www.eclipse.org/ajdt/downloads/`

It is highly recommended to install the plugin through Eclipse by selecting the "help" tab and then selecting "Install New Software". Then type in the following link into the "Install" window.
`http://download.eclipse.org/tools/ajdt/44/dev/update`
Once imported it is possible to copy paste the templates over to other projects. However this only works if the target project is an AspectJ project. In case the target project is not an AspectJ project yet it must be converted to one. To do that right click on the project in question, select "configure" and then select "convert to AspectJ".



```
ed> ReturnDefaultTester [Java Application] C:\Programme\Java\jre1.8.0_73\bin\javaw.exe (05.08.2018 16:05:10)
lang.Exception
    at Templates.ReturnDefaultTemplate.UsageExample.ReturnDefaultTester.throwExceptic
    at Templates.ReturnDefaultTemplate.UsageExample.ReturnDefaultTester.throwExceptic
    at Templates.ReturnDefaultTemplate.UsageExample.ReturnDefaultTester.throwExceptic
    at Templates.ReturnDefaultTemplate.UsageExample.ReturnDefaultTester.main(ReturnDe

efaultTemplate.UsageExample.ReturnDefaultTester.throwExceptic
efaultTemplate.UsageExample.ReturnDefaultTester.throwExceptic
efaultTemplate.UsageExample.ReturnDefaultTester.throwExceptic
efaultTemplate.UsageExample.ReturnDefaultTester.main(ReturnDe
```

Once this has been done aspect files can be created in the target project. It is also possible to copy paste one of the templates of our project over to the desired target project.

8.1.2 Most used pointcuts

In our modular exceptions project we use two types of pointcut designator combinations. One of them affects certain method calls only when they are called from within a specified other method. This allows us to only affect specific lines of code. An example of this is the TryCatchAspect aspect in the package Templates.DynamicTryCatchTemplate-.UsageExample. With this pointcut is is possible to use an "around" advice in order to wrap that method call into a dynamic try-catch block.

```
pointcut affectedMethods():
      withincode (void TryCatchTester.main(String[]))
      &&
      call(void *(..))
   ;
```

In case a try-catch block already exists we use a poincut that affects catch blocks within a target method. An example of this can be seen within the LogExceptionAspect in the package Templates.LoggingTemplate.UsageExample. This pointcut allows us to insert whatever exception handling we want through the advice that uses this pointcut.

```
pointcut affectedMethods(Exception e):
      withincode(void LogExceptionTester.main(String[]) )
      &&
      handler(Exception)
      &&
      args(e)
   ;
```

8.1.3 Advices

Advices themselves can be called at different locations relative to the pointcut. They can be called before, after or around the pointcut. Advices are always defined in aspect files. The syntax for advices is as follows.

```
advice_type(parameter_1 parameter_name_1, etc): pointcut_name(
   parameter_name_1, etc){
      //advice code here
   }
```

An example of such an advice is the advice used in the StudentDatabaseAspect within the DatabaseReturnExample package.

```
String around(): methodResumeString(){
      try {
         proceed();
      } catch (Exception e) {
      }
   }
```

What follows is a quick description of each advice type.

8.1.3.1 Before advice

The “before” advice is called at the point before the one specified by the pointcut. If there is a pointcut that has a single “call” designator the advice would be called right before the method specified in the “call” designator is executed. Note that only “before” advices can be called by pointcuts using the “handler” designator. Other types of advices do not work with such pointcuts.

8.1.3.2 After advice

“After” advices work almost like “before” advices with the main difference being that they are executed after the method call in the pointcut. As specified previously “after” advices cannot be executed after a catch block. In our solution we do not use them at all.

8.1.3.3 Around advice

An around advice wraps around the method call specified within the pointcut. This allows us to not only execute code before the method call but also afterwards. On top of that the return value of the method call in question can also be switched before returning to the caller. This is used extensively within our modular exceptions system in order to return a default value in case of an exception.

8.1.4 Implementing a dynamic try-catch block

One of the most common techniques used within our project is the ability to create dynamic try-catch blocks. This is done by creating an advice of the following form.

```
String around(): dynamicTryCatchPointcut(){
      try {
          proceed();
      } catch (Exception e) {
      }
   }
```

8.1.5 Implementing “return default” exception handling

Suppose there were two methods that could throw an exception in case of faulty parameters or in case of a null pointer reference. Suppose we wanted to return a default value in case of these exceptions. For these cases we have the “return default” exception handler type. This exception handler consists of a dynamic try-catch block being wrapped around the target method with the catch block returning a default value.

In this example we have two method we wish to have a default return value. The methods `Student>>getName()` and `Student>>getLastname()` method of the Student class. To affect both these methods we only require one pointcut and one advice. The advice in question must however affect both methods. The pointcut in question is as follows.

```
pointcut methodResumeString():
      (
      call (String StudentDataBaseExample.Student.getName())
      ||
      call (String StudentDataBaseExample.Student.getLastname())
      )
   ;
```

The advice must wrap the method in a try-catch block, therefore it must be an “around” advice. The advice must also return a default String value. The advice in question is as follows.

```
String around(): methodResumeString(){
      try {
         return proceed();
      } catch (Exception e) {
         return "";
      }

   }
```

The same process can be used for methods that do not return a String value but a different value instead. All it takes is to replace the return type of the advice and the return type of the method-signatures within the pointcut to the desired one.

8.1.6 Implementing “resume” exception handling

Implementing the “resume” exception handling type (which resumes the method in case of an exception) is done very similarly to the “return default” exception handling type. Instead of affecting only certain methods at a time we need to have all method calls within the target method affected. That way any line within the target method is covered by exception handling which allows our target method to resume the operation even in case of an exception. The pointcut in question is as follows.

```
pointcut methodResumeString():
      (
      withincode (void StudentDatabaseTestResume.main(String[]))
      &&
      call(void *(..))
      )
   ;
```

This pointcut would affect all method calls within the main method of the StudentDatabaseTestResume class that return void. In order to continue the main method in case these methods throw an exception we use an around advice to wrap a try-catch block around them.

```
String around(): methodResumeString(){
      try {
         proceed();
      } catch (Exception e) {
      }
   }
```

8.2 API documentation

This section documents the API of the modular exceptions solution. There are several templates that can be used to modified to handle common exception handling types. Most of the API is based on simple AspectJ [4] methods and function.

8.2.1 Pointcut designators

Pointcuts work by combining so called pointcut designators in order to form a condition [4]. If the condition is fulfilled the pointcut triggers. Certain pointcut designators can also get parameters for the advice instead.

Many pointcut designators (such as the "withincode" designator) require a method-signature as a parameter. The TryCatchAspect aspect in the package Templates.DynamicTryCatchTemplate.UsageExample has the following pointcut as an example of this.

```
pointcut affectedMethods():
      withincode (void TryCatchTester.main(String[]))
      &&
      call(void *(..))
   ;
```

The "withincode" designator has the description of the main method of the TryCatchTester class as a parameter. This method returns void and requires an array of String as a parameter. Note that since there is no package information in this definition the pointcut will only affect methods defined within the same package as the aspect file of the pointcut itself. If one wishes to create ones own method-signature the following generic template can be used.

```
(YourReturnType YourPackage.YourClass.YourMethod(YourParameter1,
   YourParameter2, etc.. )
```

All method-signatures in AspectJ must follow the form of the template listed above. Certain elements of this template can be replaced with a * in order to affect all methods regardless of one of the traits. For example the following pointcut would affect all main methods within the project.

```
pointcut affectedMethods():
      withincode (void *.main(String[]))
      &&
      call(void *(..))
   ;
```

8.2.1.1 call designator

This designator is fulfilled whenever the method defined with the method-signature is called. An example of the “call” designator is within the DatabaseReturnAspect aspect in the package StudentDataBaseExample.DatabaseReturnExample.

```
pointcut methodResumeString():
      (
      call (String StudentDataBaseExample.Student.getName())
      ||
      call (String StudentDataBaseExample.Student.getLastname())
      )
   ;
```

8.2.1.2 withincode designator

The “withincode” designator triggers whenever a method is called from within the method defined with the method-signature. An example of the “withincode” designator is within the TryCatchAspect aspect in the package Templates.DynamicTryCatchTemplate-.UsageExample.

```
pointcut affectedMethods():
      withincode (void TryCatchTester.main(String[]))
      &&
      call(void *(..))
   ;
```

8.2.1.3 handler designator

The “handler” designator triggers whenever the program enters a catch block. An example of this designator can be seen within the LogExceptionAspect in the package Templates.LoggingTemplate.UsageExample

```
pointcut affectedMethods(Exception e):
      withincode(void LogExceptionTester.main(String[]) )
      &&
      handler(Exception)
      &&
      args(e)
   ;
```

8.2.1.4 args designator

The “args” designator is fulfileld if the parameter of the current method call is of the same type as defined in the “args” designator. Its main purpose is getting the parameters of method calls of other designators. An example of this designator can be seen within the LogExceptionAspect in the package Templates.LoggingTemplate.UsageExample

```
pointcut affectedMethods(Exception e):
      withincode(void LogExceptionTester.main(String[]) )
      &&
      handler(Exception)
      &&
      args(e)
   ;
```

The “args” designator in the example above gets the Exception instance of the catch block of the “handler” designator. Note that the pointcut itself must have the parameter the “args” designator gets defined as the parameter to the pointcut itself.

8.2.2 StudentDataBaseExample

This package serves as an example as to how to implement modular exceptions in a project. It features several examples that show how each type of exception handling can be implemented.

8.2.2.1 Student class

Field summary

Field type	**Field name**	**Description**
String	name	The name of the student
String	lastname	The last name of the student

Method summary

Return type	**method name**
Description	
String	getName()
Gets the name of the student.	
void	setName(String newName)
Sets the name of the student.	
String	getLastname()
Gets the last name of the student.	
void	setLastname(String newName)
Sets the last name of the student.	
boolean	equals(Object other)
Returns true if the other object is a Student which has the same name and last name.	
String	toString()
Returns a string representation of this Student	
Student	defaultStudent()
Creates and returns a default Student object	

The constructor of the Student class takes two String values as parameters, one for the name and the other for the last name of the Student.

8.2.2.2 StudentDatabase class

Field summary

Field type	**Field name**	**Description**
ArrayList<Student>	students	The list of Student objects within the database

Method summary

Return type	**method name**
Description	
void	addNewStudent(Student newStudent)
Adds a student to the database after checking if that student exists in the database. Throws an Exception if that student already exists within the database.	
void	renameStudent(Student student, String newStudentName, String newStudentLastName)(String newName)
Renames a student in the database. Before renaming the student the database checks if the student with the new name and last name already exists within the database. Throws an exception in case a student like this already exists.	
Student	getStudentByNameAndLastname(String name, String lastname)
Returns the student with the name and last name given. Returns null if no student with these names can be found.	
boolean	doesStudentExist(String studentFirstName, String studentLastName)
Returns true if a student with the given name and last name exists within the database.	

8.2.2.3 DatabaseCancelExample package

This package contains an example that shows how exceptions in a method can be handled by canceling the execution of that method and resuming at the caller.

8.2.2.4 DatabaseDynamicCatchExample package

This package contains an example that shows how a dynamic try-catch block can be implemented

8.2.2.5 DatabaseResumeExample package

This package contains an example that shows how a method can be resumed at the next line in case of an exception.

8.2.2.6 DatabaseRethrowExample package

This package contains an example that shows how a dynamic try-catch block can be used to catch, log and re-throw an exception.

8.2.2.7 DatabaseReturnExample package

This package contains an example that shows how an exception can be handled by having the method return a default value.

8.2.2.8 StudentReturnExample package

This package is similar to the“DatabaseReturnExample” package in that it has an example that shows how one can return a default value in case of an exception. The difference is that it uses methods of the Student class as an example.

8.2.3 Templates

Templates are presets that are ready to be used, copied and/or modified. The user only has to insert the methods to be affected in the pointcut of the aspect file of the template. Each template comes with an example use case which is saved in the “UsageExample” package inside the package where the template is.

8.2.3.1 Templates.DynamicTryCatchTemplate

This template will wrap a dynamic try-catch block around method calls within the target method of the users choice. Insert the name of the class and method to be affected where “YourClass” and “YourMethod” is written.

8.2.3.2 Templates.LoggingTemplate

This template allows user to call an advice when entering a catch block within a target method. The advise itself will log the exception. Insert the name of the class and method to be affected where “YourClass” and “YourMethod” is written.

8.2.3.3 Templates.RethrowTemplate

This template wraps a dynamic try catch block around a target method that catches the exception, prints its stack trace and then re-throws the exception. To use it insert the name of the class and method to be affected where “YourClass” and “YourMethod” is written.

8.2.3.4 Templates.ReturnDefaultTemplate

This template allows user have a method return a default value when a target method throws an exception. Currently it supports methods that return a float or a String value. This can be easily changed by replacing the return types of the pointcut and advice with the desired return type. Insert the name of the class and method to be affected where “YourClass” and “YourMethod” is written.

8.3 Implementation notes

The project uses unmodified AspectJ. No intermediate classes have been created for this project.

Bibliography

[1] Bruno Cabral, Paulo Marques, Exception Handling: A Field Study in Java and .NET. In Proceedings of European Conference on Object-Oriented Programming (ECOOP'07), LNCS 4609 p. 151—175, Springer Verlag, 2007.

[2] Haidar Osman, Andrei Chis, Jakob Schaerer, Mohammad Ghafari, Oscar Nierstrasz On the Evolution of Exception Usage in Java Projects. In Proceedings of the 24rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), p. 422—426, February 2017.

[3] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the Rescue. In Proceedings European Conference on Object Oriented Programming (ECOOP'98), LNCS 1445 p. 396—417, 1998.

[4] Adrian Colyer, Andy Clement, George Harley, Matthew Webster. Eclipse AspectJ: aspect-oriented programming with AspectJ and Eclipse AspectJ development tools Addison-Wesley Professional, 2004.

[5] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. How Developers Use Exception Handling in Java?. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16 p. 516—519, ACM, New York, NY, USA, 2016

[6] Eiji Adachi Barbosa, Alessandro Garcia, and Mira Mezini. Heuristic strategies for recommendation of exception handling code. In Software Engineering (SBES), 2012 26th Brazilian Symposium on, p. 171—180, 2012

[7] Clément Bera, Stéphane Ducasse, Alexandre Bergel, Damien Cassou, and Jannik Laval. Handling Exceptions. In Deep Into Pharo, p. 38, Square Bracket Associates, September 2013

[8] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of Exception Handling Patterns in Java Projects: An Empirical Study. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16 p. 500—503, ACM, New York, NY, USA, 2016

[9] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16 p. 212—222, ACM, New York, NY, USA, 2016.