# Exceptions all Over the Shop

## Modular, Customizable, Language-Independent Exception Handling Layer

Walter Cazzola
Università degli Studi di Milano, Italy
cazzola@di.unimi.it

Luca Favalli
Università degli Studi di Milano, Italy
favalli@di.unimi.it

## Abstract

The introduction of better abstractions is at the forefront of research and practice. Among many approaches, domain-specific languages are subject to an increase in popularity due to the need for easier, faster and more reliable application development that involves programmers and domain experts alike. To smooth the adoption of such a language-driven development process, researchers must create new engineering techniques for the development of programming languages and their ecosystems. Traditionally, programming languages are implemented from scratch and in a monolithic way. Conversely, modular and reusable language development solutions would improve maintainability, reusability and extensibility. Many programming languages share similarities that can be leveraged to reuse the same language feature implementations across several programming languages; recent language workbenches strive to achieve this goal by solving the language composition and language extension problems. Yet, some features are inherently complex and affect the behavior of several language features. Most notably, the exception handling mechanism involves varied aspects, such as the memory layout, variables, their scope, up to the execution of each statement that may cause an exceptional event—*e.g.*, a division by zero. In this paper, we propose an approach to untangle the exception handling mechanism dubbed the *exception handling layer*: its components are modular and fully independent from one another, as well as from other language features. The exception handling layer is language-independent, customizable with regards to the memory layout and supports unconventional exception handling language features. To avoid any assumptions with regards to the host language, the exception handling layer is a stand-alone framework, decoupled from the exception handling mechanism offered by the back-end. Then, we present a full-fledged, generic Java implementation of the exception handling layer. The applicability of this approach is presented through a language evolution scenario based on a Neverlang implementation of JavaScript and LogLang, that we extend with conventional and unconventional exception handling language features using the exception handling layer, with limited impact on their original implementation.

***CCS Concepts:*** • **Software and its engineering** → **Abstraction, modeling and modularity**; **Compilers**; **Extensible languages**.

***Keywords:*** Language Modularization, Exception Handling.

**ACM Reference Format:**
Walter Cazzola and Luca Favalli. 2023. Exceptions all Over the Shop: Modular, Customizable, Language-Independent Exception Handling Layer. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623513

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SLE '23, October 23–24, 2023, Cascais, Portugal*
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623513

## 1 Introduction

Programming language development is a complex activity. It involves the development of an ecosystem of varied software artifacts, such as, parsers, optimizers, translators and development environments. The traditional approach towards language development is monolithic: language constructs and their semantics are planned during the design phase and rarely change overtime. The monolithic approach is considered easier to develop and more performant; however, the final products are hard to change, update and evolve. Developing different languages with similar constructs can provide reuse opportunities that are only possible if the implementation is modularized, so that it is easier to extract and reuse in different contexts [41]. *Language workbenches* [22, 25] are a common approach to this problem.

However, tool support does not suffice due to the inherent complexity of some language features. Take exception handling as an example. Introducing an exception handling mechanism in an existing language implementation has a drastic impact on the way a program executes, because each statement might throw an exception: exception handling is a *crosscutting feature*. Crosscutting features are language features whose code is scattered across the implementation: they are known to reduce the flexibility and maintainability of software systems [19], and can also affect parse-tree rewriting contexts [32]. Moreover, each language has a different memory layout and handles exceptions in a different way.

This makes a strictly modular and reusable implementation of exception handling challenging. In this work, we discuss a framework for the generalization of the exception handling crosscutting feature, to implement it without affecting the original implementation of the language and without making any assumptions on the structure of the original language.

Our contribution is a general, customizable, reusable, and extensible exception handling conceptual framework—dubbed as *exception handling layer*—that can be adopted by language workbenches to untangle the exception handling concern from the code of other language features.

The applicability of the proposed framework is demonstrated through a full-fledged Java implementation, then used to refactor the exception handling mechanism of a JavaScript [12] interpreter written in Neverlang [8, 14, 48] Moreover, we present the flexibility of our proposal by implementing unconventional exception handling language features such as the retry and resume statements, and implementing a recovery procedure for the LogLang [13] declarative domain-specific language. On each step of the language evolution scenario, we keep track of the required development effort in terms of lines of code and modified files. This work is validated by answering these research questions:

**RQ$_1$. How hard is it to refactor an existing language implementation so that it can be used in tandem with the exception handling layer?**
**RQ$_2$. How hard is it to add exception handling support to a language implementation using the exception handling layer?**
**RQ$_3$. How much is the achieved modularization reliant on Neverlang-specific mechanisms?**
**RQ$_4$. Does the conceptual framework support varied exception handling mechanisms and their language features?**

The remainder of this paper is structured as follows. Sect. 2 contains any background information relevant to this work, including language workbenches, their capabilities and the basics of exception handling. Sect. 3 presents the exception handling layer as the main contribution of this work. In Sect. 4 we present language evolution scenarios based on the introduction and extension of the exception handling layer. Finally, in Sect. 5 and Sect. 6 we will respectively discuss any related work and draw our conclusions on this research.

## 2 Background

In this section, we discuss the background information on language workbenches and exception handling.

### 2.1 Language Workbenches

Modular language development benefits from the creation of *sectional compilers* [8] defined in terms of independently developed language features. Each language feature is a reusable piece of a language specification, formed by a syntactic asset and a semantic asset, representing a language construct and its behavior respectively. Language workbenches [23] embrace this philosophy to improve reusability and maintainability of linguistic assets. The term language workbench was firstly introduced by Fowler [25] for the tools suited to support the language-oriented programming paradigm [55], in which complex software systems are built around a set of domain-specific languages, each used to express the problems and the solutions of a portion of the complex system. Nowadays, language workbenches are used to facilitate the development of modular programming languages and the reuse of software artifacts through better abstractions. These abstractions are designed to support five different composition mechanisms among programming languages: language extension, language restriction, language unification, self-extension, and extension composition [21]. There are several language workbenches in literature, each proposing its own flavor of language composition. Some examples (among many others) are: Melange [20], MPS [52], MontiCore [33], Neverlang [48], Rascal [31], Silver/Copper [51], and Spoofax [54].

### 2.2 Exception Handling

Software exceptions are anomalies that can occur during the execution of any instruction of a program. When an exception occurs, the application state does not conform to the continuation of its normal execution flow [27]. Instead, exceptions are handled through dedicated language control structures—called exception mechanisms—that replace the standard continuation with an exceptional continuation. Most modern programming languages provide such exception mechanisms, yet adequate exception handling has been proven difficult [18]. Sub-optimal exception handling practices are associated to low software quality and post-release defects [44]. Therefore, it is vital that each programming language provides the exception mechanisms that are the most appropriate with respect to the intended behavior. Relying on the abstractions provided by the back-end is still the most common practice in the development of DSLs but this limits the capabilities of the exception handling mechanism to those offered by the back-end. Each language has its own constructs and uses a different notation, although three common elements have been identified [27]:

— a part of a program or an operation that brings an exceptional event to the attention of the caller; this is called *throwing* or *raising* an exception and can either be implicit (e.g., a division by zero) or explicit (e.g., a `throw` call in Java);

— the *handler* is a part of a program that must be executed to handle an exceptional event; exception handling can either be explicitly defined or provided by default;

— the handler's *reach* is a syntactic construct or part of a program (such as a block) that can launch the associated handler if the activation point of the exceptional event falls within it.

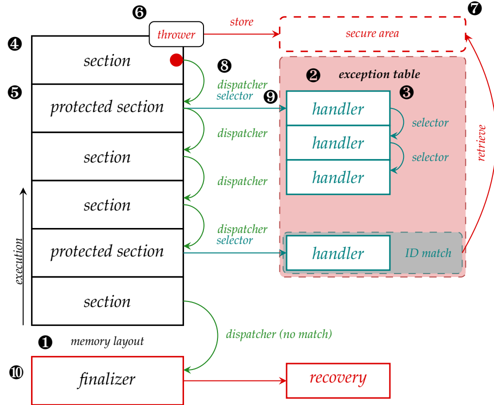


**Figure 1.** EHL general architecture and process.

## 3 The Exception Handling Layer

In this section, we discuss the conceptual framework for the implementation of a portable exception handling mechanism whose code is not scattered across the language implementation. This conceptual framework provides language developers with a template to design modular exception handling language extensions. We dubbed this conceptual framework as *exception handling layer* (EHL). We discuss EHL's fundamental data structures and procedures. We also present a fully modular decomposition that decouples exception handling from the other language features.

### 3.1 Architecture

Fig. 1 depicts the general architecture of EHL, its components and their interaction. Note how an exception-unaware language is a corner case of this architecture where all elements except the memory layout are omitted.

**Memory layout (❶).** The memory layout abstracts the stack of the machine the program is running on. We do not make any assumptions with regards to data structure used to represent the memory layout. Instead, the memory layout is split into *sections*.[1] According to this architecture, sections are nodes arranged within a directed graph that abstracts the entire memory. For simplicity, Fig. 1 shows the memory layout as a stack, a common memory layout among general purpose programming languages [1]. In fact, a stack can be viewed as a directed acyclic graph whose nodes are arranged in a chain—*i.e.*, each section is connected to the previous element of the stack. Arbitrary memory layouts allow to abstract unconventional exception handling mechanisms such as the `exit` function in Erlang: if a process calls `exit(kill)` and does not catch the exception, it will terminate and emit exit signals to all linked processes.[2]

Each section of the memory layout is either protected or normal, depending if it is within a handler's reach or not. However, the memory layout itself does not hold this piece of information. In fact, to properly modularize the exception handling concern, the memory layout is unaware of the existence of exceptions at all. Instead, any information regarding exceptions—such as, any exception handlers that can reach a section—resides in a distinct data structure. The same memory layout can therefore be used in a programming language without exception support. Notice that the memory layout of an exception-unaware language implementation contains only normal sections.

**Exception table (❷).** The *exception table* contains all references to the location of the exception handlers so that the correct handler can be executed for each exception type. In this context, the exception type is not a data type provided by the back-end but a more general and arbitrary descriptor. There is no assumptions wrt. the implementation of this table, e.g., it may be a list with an index associated to each handler or a bi-dimensional map associating each pair (section, exception ID) to a handler.

**Exception handler (❸).** As in [27], an *exception handler* refers to the code to be executed when an exception is caught, e.g., a **`catch`** block in Java. The nature of the handler depends on the language the exception mechanism is plugged on: in a compiler, it can be the method or the function to be executed; in an interpreter it can directly hook the AST node representing the code to execute. A custom implementation of the exception handler could even store the instructions to be executed directly inside the exception table.

**Normal section (❹).** Each *normal section* coincides with the memory reserved to the execution of a function or method call, or, in some languages, to a block of code—*e.g.*, a delimited sequence of instructions. Each section also serves as a namespace: it contains a symbol table with an arbitrary number of scopes, each with the named constants, variables, structures and procedures that are visible within that scope. A normal section is unaware of the existence of exceptions and is not within any exception handler's reach. The memory layout contains a reference to the currently executing section. In languages using a single stack, the current section coincides with the top of the stack, whereas in other cases it can be set by an external program, e.g., by the scheduler.

**Protected section (❺).** *Protected section*s represent the handler's reach from [27] and are parts of the program that are capable of capturing and handling exceptions, such as a **`try-except`** block in Python. A protected section is identical to a normal section but it is within the reach of an exception handler. If a section is normal or protected is determined by a procedure called *selector* (more on that later).

**Thrower (❻).** The *thrower* is a section that threw an exception, according to [27]. The thrower may be implicit, such as a section that caused a division by zero, or explicit, such as a section containing a **`throw`** statement in Java.

[1] Please refer to the corresponding paragraphs for more details on sections.
[2] https://www.erlang.org/doc/man/erlang.html#exit-1

**Secure area (❼).** The *secure area* is a special buffer reserved to the thrower to store all the relevant information (if any) for the exception handling mechanism. The secure area is needed to avoid any assumptions on the exception handling mechanism: the EHL treats exceptional events as simple signals, whereas any additional information is carried by the secure area. E.g., in Java the secure area would hold an instance of the `Throwable` class, whereas JavaScript applications can throw the result of any expression. The secure area is also used to store other information, such as the list of all sections visited during the exception handling event.

**Dispatcher (❽).** The *dispatcher* is an arbitrary graph traversal algorithm that is fired when an exception is thrown. Starting from the current section, the dispatcher navigates the memory layout. On each section, the dispatcher runs a secondary procedure called *selector* to determine if the current section is protected. The traversal ends when a protected section has a viable exception handler, as determined by the selector. Otherwise—*i.e.*, when there are no more sections to be visited in the queue—the dispatcher terminates abruptly and delegates to a procedure called *finalizer*. For instance, a stack-based implementation of the dispatcher may pop frames from the stack until finding a handler.

**Selector (❾).** The *selector* is an arbitrary procedure returning a viable handler for the thrown exception when a section is protected. Its result may be determined by inspecting the contents of the symbol table—*i.e.*, a section is protected if the exception table maps that section to at least one handler. The selector could be customized to implement implicitly protected sections—*i.e.*, sections with a default exception handler without a need for the programmer to declare one—even without inspecting the exception table.

**Finalizer (❿).** The *finalizer* is an arbitrary procedure that runs when no viable handler is found and the dispatcher cannot reach any more sections within the memory layout. The finalizer implements the ultimate recovery procedure and allows the runtime environment to smoothly shut down the application when a thrown exception cannot be handled by any handler of any protected section. Finalizers can also be used to attempt restoring the application to a suitable state without stopping the execution (see Sect. 4).

### 3.2 Process

In this section, we discuss the life-cycle of an exceptional event according to EHL. This process evolves according to four sequential phases: *normal execution*, *exception throwing*, *exception carrying* and *exception handling*.

**Normal execution.** During normal program execution, the memory expands and shrinks according to the creation and destruction of sections. Traditionally—*i.e.*, when the memory is a stack—a new section is created and pushed on the top of the stack upon entering a new scope, such as at the beginning of a block or on function calls. Sections are then popped from the stack after their code completes its execution. Different languages may use a different memory layout, adding and removing sections accordingly. For instance, if a section spawned several threads, the memory layout graph can contain several sections with an edge towards that section, one for each thread. In EHL, each time a new protected section is added to the memory layout, any handler for that protected section is added to the exception table. When a protected section ends its execution, it is removed from the memory layout and its handlers are (optionally) unregistered from the exception table. This execution flow is continued until a thrower is encountered, as shown by the red dot in Fig. 1, then an exceptional event occurs and the execution proceeds to the exception throwing phase.

**Exception throwing.** The system halts the normal execution when an exception is thrown. Such an exception is identified with a descriptor—e.g., its class in Java and the secure area is populated. Finally, a signal is sent to the EHL runtime to start the exception carrying phase.

**Exception carrying.** The thrown exception travels across the system according to the dispatcher algorithm until the correct handler is found, if it exists. During this phase, it must be possible to inspect the memory state, to feed handlers with any relevant information—*e.g.*, the variables in scope. Fig. 1 shows that the exception carrying phase is handled by the dispatcher and the selector. The dispatcher traverses the memory layout graph; on each visited section, the dispatcher delegates to the selector to determine if the current section is protected and if any of its handlers can handle the carried exception. This is usually done by inspecting the exception table, but some languages, particularly DSLs, may implement default handler procedures that are not held within the exception table. For instance, if the layout is a stack, the dispatcher may iteratively inspect the section on top of the stack, popping any normal section and any protected section with incompatible handlers. The process proceeds to the exception handling phase if a compatible handler is found (as shown by the blue box in Fig. 1). Otherwise the dispatcher is resumed to continue browsing the memory layout according to the traversal algorithm. The exception carrying mechanism fails and control is given up to the finalizer when the dispatcher ends its execution—*i.e.*, when there are no more sections to be visited. The finalizer performs any procedure needed to ensure that the system is safely shut down or recovered, possibly reporting any failures to the user. For instance, in Java the finalizer prints the stack frame before terminating the execution. A finalizer could also be used to roll back to a globally-known safe state.

**Exception handling.** The exception handling procedure starts when the selector finds a handler that is compatible to the thrown exception. The handling procedure retrieves the

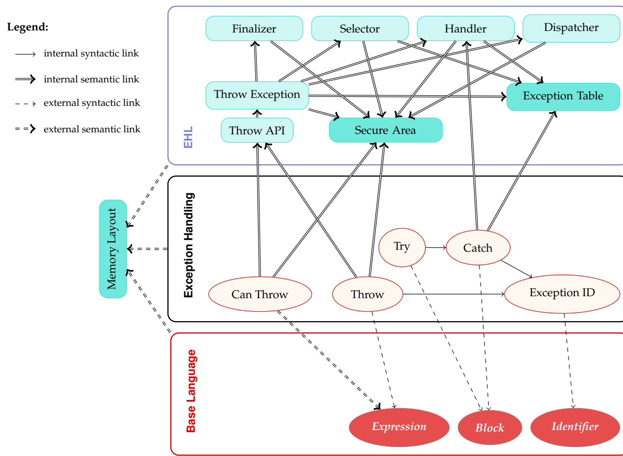


**Figure 2.** Language modularization according to the EHL.

information stored inside the secure area and—optionally—the scope of the protected section. There is no requirement on how the secure area is implemented: two possible options are either a globally accessible object or a instance that is created upon exception riding and then tunneled across the dispatcher, the selector and eventually the handler or the finalizer. Once the exception has been handled, the normal application flow is resumed. In many programming languages the execution flow is resumed after the end of the protected section, but other resumption mechanisms may be in place. Some examples are resumption from the instruction in which the exception has been thrown (resume) and from the first instruction of the thrower (retry). These mechanisms usually assume the handler changed the memory to a safe state or with additional information before resuming the normal execution. The EHL is agnostic wrt. exception handling mechanism chosen by the language and supports these mechanisms (see next sections).

### 3.3 Exception Handling Modular Decomposition

A language can be decomposed to leverage the EHL, untangling the exception handling code from the code of unrelated language features. Fig. 2 depicts such a modularization. Fig. 2 is comprised of two main components: the base language (red box) and the exception handling implementation built on its top. Fig. 2 also splits the exception handling mechanism into exception handling language features (black box) and EHL (blue box). The part about the base language implementation is not relevant to this discussion and is omitted. Each node in Fig. 2 is either a language feature (oval shape) or a component of the EHL architecture (rectangular shape, data structures are represented with a darker color and algorithms with a lighter color). The EHL components mirror the architecture discussed in Sect. 3.1. Each arrow represents a dependency between coupled components. Double and single arrows represent semantic and syntactic dependencies respectively. Dashed and normal arrows represent dependencies to external and internal components respectively.

The key element of this decomposition is dependency management. To minimize coupling between components and to maximize reuse, the decomposition uses EHL data structures as adapters [26], so that exception handling language features are not directly coupled with the underlying exception handling mechanism and the semantics can be changed freely. In fact, no feature from the base language depends on the exception handling, neither syntactically nor semantically. Exception handling features can instead depend on features of the base language, which they can extend, override and specialize depending on the exception handling mechanism to be implemented. For instance, in Fig. 2, the `Throw` language feature syntactically depends on the `Expression` language feature, because in this case the throw statement can throw an exception based on the return value of an expression. Similarly, the `CanThrow` language feature extends the semantics of any expression by declaring that its evaluation may cause an exceptional event—*e.g.*, a division by zero. Notice how such a decomposition is completely modular, so that exception handling features can be used independently. E.g., it is possible to create languages where i) the **`throw`** statement is present but expressions can never cause an exception, ii) expressions can cause exceptions but the **`throw`** statement is absent, and iii) exceptions can be thrown but never caught (no **`try-catch`** statements).

Internal dependencies among components of the EHL are similarly structured: the `Throw Exception` component acts as glue code that depends on all other elements of the exception handling mechanism, namely exception table, dispatcher, selector, handlers, finalizer and secure area. A different exception handling mechanism can be deployed by swapping the `Throw Exception` component with a similar component that shares the same interface but connects different elements. For instance, it is possible to create a new exception handling in which the dispatcher is replaced whereas all other elements remain the same. Similarly, elements can be shared across several `Throw Exception` components, possibly pertaining different programming languages. To ensure that data is properly carried throughout the entire exception handling process, all elements depend on the exception table, on the secure area, and on the memory layout. Thus, in the EHL the dependencies between components are limited to the data representation (darker color), rather than on the behavior: as long as the data representation stays the same, the exception handling mechanisms can be extended and replaced at will. Some specific language features may break this rule. For instance, the `Catch` language feature depends on the `Handler`, because it needs to register a new handler within the exception table upon entering a protected section. However, such dependencies are always limited to one language feature and do not impact the rest of the language: in

```java
public class JEL {
  public static void raise ( ExceptionID exceptionID,
      ExceptionTable exceptionTable, Memory memory,
      Section thrower, Dispatcher dispatcher,
      Selector selector, Optional<Finalizer> finalizer,
      Optional<SecureArea> secureArea) {
    var handler = dispatcher.dispatch (
      exceptionID, exceptionTable, memory,
      thrower, selector, secureArea);
    handler.ifPresentOrElse (
      h -> h.handle(secureArea),
      () -> finalizer.ifPresent (f ->
        f.finalize(secureArea, exceptionID)));
  }
}
```

**Listing 1.** Glue code that connects all the elements of JEL. Please note that redundant generic data types are omitted to save space; refer to the text of this section for the generic data types associated to each element.

Fig. 2, the `Catch` feature can be replaced with a different one without affecting the `Throw` and `CanThrow` features.

### 3.4 Exception Handling Layer in Java

In this work, we implemented the EHL as a library dubbed as *Java exception layer* (JEL). JEL behaves like an intermediate layer between the running application and the underlying JVM execution environment. The JEL library is intended to be used instead of the default exception handling mechanism offered by the JVM to support additional language features, such as retry/resume operations and handlers for arbitrary types—instead of just members of the `Throwable` hierarchy. While the underlying Java exceptions still exist within the runtime environment, they should be transparent for the user: assuming the language provides a full-fledged implementation of its exception handling mechanism using JEL, all Java exceptions are captured and translated into a JEL exceptional event. According to the modularization constraints discussed in Sect. 3.3, notice how JEL does not refer to any language-specific implementation aspect, such as the supported operations and their syntax. JEL is implemented as a library of generic interfaces with a default implementation.

**Generic Data Types.** Since the dependencies among components is EHL are based on the data representation, JEL data structures and algorithms can be customized according to five different data types (classes, in Java):

- `EX_ID` the exception identifier;
- `SEC_ID` the unique identifier for a section;
- `VAR_NAME_TYPE` the type used for variables identifiers;
- `VAR_TYPE_TYPE` the type used for variables types;
- `PAYLOAD` the type of data carried by the secure area.

**Default implementation.** Mirroring the EHL architecture, JEL provides the interfaces for three data structures: i) the memory layout, ii) the exception table and iii) the secure area, as well as their default implementations.

The *memory layout* is shared between the base language and the exception handling module. To fit the modularization requirements, the memory layout is implemented in an agnostic way wrt. the exception handling and is populated by generic `Section` objects. JEL provides two default implementations for the memory layout interface: a graph and a stack sharing the same `Section` objects. Both implementations can be adapted to the language by specifying the `SEC_ID`, `VAR_NAME_TYPE` and `VAR_TYPE_TYPE` generic data types.

The default *exception table* is implemented as a two-dimensional look-up table that takes the ID of the exceptional event and the ID of the thrower `Section` and maps them to the respective handler method. The exception table can be interacted with to register and unregister exception IDs and exception handlers upon entering and exiting sections during execution. The default exception table can be customized according to all five generic data types.

The default *secure area* is implemented as a wrapper for an object with store and retrieve operations. The type of wrapped data is set by specifying the `PAYLOAD` generic data type. The responsibility of correctly populating this data structure is delegated to the thrower, which will change depending on the language the EHL is being plugged on.

JEL also provides a default implementation for dispatcher and selector routines that can be customized according to all five generic data types. The default *dispatcher* is a traversal algorithm for a stack-based memory layout, that pops elements from the top, delegating to the selector on each element, until finding a handler or reaching the bottom. The default *selector* queries the two-dimensional exception table for the handler for a (`SEC_ID`, `EX_ID`) pair, if any.

Finally, JEL provides a static `raise` method that is in charge of starting the exception throwing event and that acts as the glue code connecting all the elements, as shown in Listing 1. Given this code, the execution of an exceptional event from the perspective of a language feature coincides with a call to the `raise` method with the correct arguments.

## 4 Case Study and Discussion

This section presents and discusses a language evolution scenario that uses EHL to add exception handling support to a base exception-unaware language. The Neverlang language workbench [48] is used to implement both the exception-unaware base language and its exception-aware variants.

### 4.1 Neverlang Overview

Neverlang [48] is a language workbench for the modular development of programming languages and their ecosystems. It is based on the language feature concept [9], each developed as separate units called *slices* that can be independently compiled, tested, and distributed. Syntactic and semantic

```
1  module nl.jel.JELTryStatement {
2    reference syntax {
3      try_part: TryPart <- "try" ProtectedSection;
4      protected: ProtectedSection <- Block;
5    }
6    role (evaluation) {
7      try_part: .{
8        eval $try_part[1];
9        Section<Long,String,JSReference> section =
10         $try_part[1].section;
11       JELSymbolTable jst = (JELSymbolTable)$$SymbolTable;
12       Stack<Long, String, JSReference> stack =
13          jst.getStack();
14       stack.push(section);
15       stack.peek().get().getCode().execute();
17       if($try_part[1].shouldRaise) {
18         JSSecureArea secure = $$SecureAreaBuilder.build (
19             $try_part[1].error;
20         );
21         JSJEL.raise($$JSExceptionTable, stack,
22             section, $$JSDispatcher, $$JSSelector,
23             $$JSFinalizer, secure
24         );
25       }
26       stack.pop();
27     }.
28     protected: .{
29       /*Code to generate the Section executable object*/
30     }.
31   }
32 }
33 endemic slice nl.jel.JELEndemic {
34   declare {
35     static JSExceptionTable: nl.jel.JSExceptionTable;
36     static SecureAreaBuilder: nl.jel.SecureAreaBuilder;
37     static JSDispatcher: nl.jel.JSDispatcher;
38     static JSSelector: nl.jel.JSSelector;
39     static JSFinalizer: nl.jel.JSFinalizer;
40   }
41 }
42 language nl.jel.JSLangJEL {
43   slices nl.jel.JELTryStatement /* ... */
44   endemic slices nl.jel.JELEndemic /*...*/
45   roles syntax <+ evaluation
46 }
```

**Listing 2.** Syntax and semantics for the JavaScript try block language feature in Neverlang.

assets are contained in a compilation unit called *module*. A module contains a **`reference syntax`** block—in which the productions are defined—and any number of roles. Each role, is preceded by the **`role`** keyword, and represents a visit of the parse tree: each role is made by one or more semantic actions [1] that are executed when some nonterminal symbol is encountered in the parse tree. Syntactic definitions and semantic roles are exogenously composed using slices. Please refer to [48] for a full Neverlang overview.

**Basic capabilities.** Listing 2 shows a modular implementation of the **`try`** statement and part of the composition mechanisms offered by Neverlang. The `Try` module (lines 1-32) declares a reference syntax for the try part of a **`try-catch`** block (lines 2-5), made of two production rules. The first is labeled "try_part" (line 3) and the second is labeled "protected" (line 4). The semantics are declared within a **`role`** block (lines 6-31) defining several semantic actions; each action is attached to a nonterminal of any of the productions of the reference syntax by referring to their label[3]—*e.g.*, the semantic action at line 7 refers to the production at line 3, whereas the semantic action at line 28 refers to the production at line 4. Nonterminals within a production are accessed using square brackets, in an array-like fashion as highlighted by the red arrows in Listing 2. Following the syntax directed translation technique [1], attributes are accessed from nonterminals by dot notation as done for retrieving the `section` attribute on line 10. Neverlang semantic actions are written in Java with some syntactic sugar. The semantic action at lines 7-27 retrieves the `Section` object from a child node (line 10), pushes it on the stack (line 14), and tries to execute it (line 15). A new exception is thrown (line 21) if any error occurs. Regardless of the result, the section is eventually popped from the stack (line 26).

**Other capabilities.** Neverlang supports composition between **`module`** units using other units called **`slice`** and **`bundle`**, hereby not shown for brevity. Neverlang **`endemic slices`** units can be used to declare instances that are globally accessible throughout all semantic actions within the language. E.g., lines 33-41 of Listing 2 declare several instances needed for the correct execution of JEL, one for each of its customizable elements, as discussed in Sect. 3.4. These instances can be accessed using the `$$` operator, as done when throwing an exception on lines 21-24. Thanks to this mechanism, the exception handling process can be customized simply by swapping the `nl.jel.JELEndemic` endemic slice with a different endemic slice that re-declares the same instances by changing their class. Instead, the semantic action of module `nl.jel.JELTryStatement` remains unaltered. Modules, bundles, slices, and endemic slices are composed into a complete and executable language specification using the **`language`** unit, as done in lines 42-46. Neverlang supports language product line engineering [10, 11] through AiDE [34, 35, 49, 50], FeatureIDE [24], and the Gradle build tool.

### 4.2 JavaScript Evolution Scenario

We present a three-staged evolution for a JavaScript interpreter written in Neverlang [12] conform to the EcmaScript 3 specification, with the exception of part of the standard library. The variant $V_1$ is the base language with its own implementation of both the memory layout and of the exception handling. Variant $V_2$ removes exception handling support and replaces the original implementation of the memory layout with a JEL-based symbol table. Variant $V_3$ adds several exception handling language features on top of variant $V_2$.

[3] Neverlang also provides an alternative mechanism, based on absolute position of nonterminals within the reference syntax, not discuss for brevity.

**Table 1.** The effort to evolve JavaScript from $V_1$ to $V_2$ and from $V_2$ to $V_3$ wrt. the memory layout (Memory) and exception handling features (Exceptions). Data are collected by using the `svn diff` and the `diffstat` Linux commands.

| Code edit | Evolution step effort | | |
|---|---|---|---|
| | **Change type** | $V_1 \rightarrow V_2$ | $V_2 \rightarrow V_3$ |
| **Files changed** | **Total** | 7 | 1 |
| | **Memory** | 6 | 0 |
| | **Exceptions** | 0 | 0 |
| | **Glue** | 1 | 1 |
| **Files added** | **Total** | 7 | 20 |
| | **Memory** | 5 | 0 |
| | **Exceptions** | 0 | 17 |
| | **Glue** | 2 | 3 |
| **Insertions (LoC)** | **Total** | 369 | 489 |
| | **To new files** | 292 | 485 |
| | **Memory** | 322 | 0 |
| | **Exceptions** | 0 | 448 |
| | **Glue** | 47 | 41 |
| **Deletions (LoC)** | **Total** | 71 | 0 |
| | **Memory** | 70 | 0 |
| | **Exceptions** | 0 | 0 |
| | **Glue** | 1 | 0 |

On both evolution steps, we measured the implementation effort, as summarized in Table 1. The first step is intended to measure the effort needed to render an existing language compliant to JEL and, by extension, to the EHL, thus answering $RQ_1$. The second evolution step aims at measuring the effort needed to implement varied exception handling language features in JEL, thus answering $RQ_2$. In both steps, we discuss how much this refactoring is affected by Neverlang, thus answering $RQ_3$. The resulting implementation of JavaScript $V_3$ is available on Zenodo.[4]

**JavaScript $V_1$.** JavaScript $V_1$ supports many of the most important features offered by the language, including (but not limited to):

- numeric, boolean, string and reference types;
- prototype-based classes and constructors;
- expressions between basic, reference and object types;
- if-else, switch, while, and for statements;
- standard output;
- functions declaration and invocation;
- throw, try, catch, and finally statements.

Most notably, JavaScript $V_1$ uses a custom memory layout based on a linked list; since this interpreter runs on the Neverlang runtime and therefore on JVM, it leverages the default exception handling mechanisms provided by Java to implement **`throw`** and **`try-catch`** statements. While this allows for a easy solution, the end result is hard to extend, due to Java not supporting unconventional exception handling language features by default.

[4]https://doi.org/10.5281/zenodo.8328246

```
1  public class JSStack
2      extends Stack<Long, String, JSReference>
3      implements JSEnvironment.Instance<JSStack> {
4    @Override
5    public Class<JSStack> genericType() {
6        return JSStack.class;
7    }
8  }
```

**Listing 3.** Adapting generic JEL datatypes to the needs of a specific language interpreter.

Overall, JavaScript $V_1$ is comprised of 144 Neverlang units—for a total of 5,409 lines of code (LoC)—and 73 Java classes—for a total of 6,475 LoC: 318 LoC are needed to implement the memory layout, with an additional 1,983 LoC to represent types and variables within memory. 43 LoC are needed to implement the **`throw`** statement, and 135 LoC are needed to implement **`try`**, **`catch`**, and **`finally`**, with an additional 109 LoC to implement the errors of various types.

This is the baseline against which the following variants will be evaluated, to measure the effort of replacing this implementation with a JEL-based one.

**JavaScript $V_2$.** The JavaScript $V_2$ interpreter replaces the default implementation of the linked list symbol table provided by JavaScript $V_1$ with the default stack memory layout provided by JEL. Moreover, it removes any support for exception handling, meaning that JavaScript $V_2$ programs cannot throw nor catch any exceptions. To minimize the impact on the original code, the symbol table was implemented as an adapter [26]—dubbed `JELSymbolTable`—that extends the `LinkedListSymbolTable` class, so that the old implementation still works just by changing the runtime class of the symbol table. Then, calls to the `JELSymbolTable` are delegated to the actual JEL stack. As shown in Listing 3, the implementation of this stack is minimal because it does not provide any functionality, but simply specifies the generic types introduced in Sect. 3.4 according to data types needed by JavaScript, in particular:

- `SEC_ID` is instantiated to `Long`;
- `VAR_NAME_TYPE` is instantiated to `String`;
- `VAR_TYPE_TYPE` is instantiated to `JSReference`.

The `JSReference` class is particularly important in this context, because we could reuse most of the existing types with the new data structures. The `JSEnvironment.Instance` interface replaces the naïve singleton `LinkedListSymbolTable` instance with a more customizable alternative that allows instances of any subclass to be registered as the singleton.

Although limited, some modifications were required to change the original implementation of JavaScript $V_1$. Table 1 reports the effort required to make these changes. The refactoring required the modification of 7 files (6 Java classes and 1 Neverlang unit) and the creation of an additional 7 files (5 Java classes and 2 Neverlang units). All three Neverlang units are a form of glue code: we implemented a new language unit and two endemic slices, but no modules. This

refactoring required 369 insertions and 71 deletions, for a total of 440 modifications; removing the existing implementation of the exception handling mechanisms required no modifications. Considering the size of the whole JavaScript $V_1$ project, changing the memory layout to support JEL required a modification of $440/(5,409 + 6,575) = 3.67\%$ of the project. Moreover, Table 1 shows that out of 369 insertions, 292 were made to newly created files, therefore changes to existing files are limited to 71 deletions and 77 insertions. In fact, using the `diffstat` command with the `-m` flag, reveals that the actual results are 306 insertions, 8 deletions and 63 modifications. We can now answer $RQ_1$.

> **How hard is it to refactor an existing language implementation so that it can be used in tandem with the exception handling layer?**

Refactoring an existing and full-fledged implementation of a language interpreter such as JavaScript $V_1$ so that it can be used in tandem with a EHL implementation for Java requires modifying about 3.67% of its code. We can conclude that an existing memory layout can be replaced with a memory layout based on EHL with limited effort, especially if part of the default implementation can be reused. Of course, different languages may require a different effort—*e.g.*, a smaller project may require more changes wrt. the project total size.

**JavaScript $V_3$.** The JavaScript $V_3$ interpreter adds several exception handling language features on top of JavaScript $V_2$. This includes the generic types specification for the JEL exception table, dispatcher, selector, finalizer and secure area according to the following types:

- `EX_ID` is instantiated to String;
- `PAYLOAD` is instantiated to `JSExceptionPayload`.

The remaining three generic types must conform to the memory layout definition and therefore they are the same used in JavaScript $V_2$. We used the default JEL implementation for all data structures and algorithms (akin to what shown in Listing 3), with the exception of the finalizer and the secure area. The finalizer stops the application, whereas the secure area holds a `JSExceptionPayload` that keeps the stack trace during the exception handling process. We also implemented the following exception handling language features: **`throw`** statement, **`retry`**, and **`resume`** statements, and **`try catch`** block, **`finally`** blocks. Most notably, **`throw`**, **`try`**, **`catch`**, and **`finally`** are fairly common exception handling language features, whereas **`retry`** and **`resume`** are rather unconventional. Both are resumption mechanisms that drive the execution of the program after running an exception handler. The **`retry`** (inspired by design by contract [40]) continues the execution from the first statement of the thrower and the **`resume`** (inspired by hardware pipelines) continues the execution from the next statement after the one causing the exceptional event. The implementation of the **`try`** block was already shown in Listing 2 and discussed in Sect. 4.1. We do not report the implementation of all other features for brevity and

```
1  module neverlang.js.jel.exceptions.JELStatementList {
2    reference syntax from neverlang.js.JSStatementList
3    role (evaluation) {
4      s_list_0: .{
5        ▶baseActionList;
6        JSCompletionValue s = $s_list_0[0].cvalue;
7        if (s.getType() == JSCVType.THROW)
8          $$JELResumeArea.push($s_list_0[2]);
9      }.
10     s_list_1: .{ ▶baseAction; }.
11   }
12 }
13 slice neverlang.js.jel.exceptions.JELResumeBlock {
14   concrete syntax from neverlang.js.JSStatementList
15   module neverlang.js.jel.exceptions.JELStatementList
16     with role evaluation delegates {
17       baseActionList ⇒
18         neverlang.js.JSStatementList ▶ evaluation[0],
19       baseAction ⇒
20         neverlang.js.JSStatementList ▶ evaluation[3]
21     }
22 }
23 module neverlang.js.jel.exceptions.JELResumeStatement {
24   reference syntax {
25     stat: Statement ← ResumeStatement;
26     resume: ResumeStatement ← "resume" SemiColonOpt;
27   }
28   role(evaluation) {
29     resume: .{
30       ASTNode resume = $$JELResumeArea.peek();
31       $ctx.eval(resume);
32       $resume.cvalue = resume.getValue("cvalue");
33     }.
34   }
35 }
```

**Listing 4.** Throw statement using Neverlang and JEL.

```
1 var a = 1;
2 try {
3   throw 1;
4   a = a + 42;
5 } catch (x) {
6   a = a + x;
7   resume;
8 }
```

**(a)** JavaScript program using the **`resume`** statement.

```
1 var a = 1;
2 try {
3   if ( a <= 10 )
4     throw 1;
5 } catch (x) {
6   a = a + x;
7   retry;
8 }
```

**(b)** JavaScript program using the **`retry`** statement.

**Listing 5.** Unconventional exception handling language features in JavaScript $V_3$.

instead we focus on the most interesting aspects. With regards to the **`catch`** and **`finally`** blocks, both are registered as handlers for the corresponding protected section within the exception table, with the difference that the handler for the finally block is always executed, regardless of an exception being thrown or not. From an implementation standpoint, this was achieved by creating a composite [26] handler that runs both the catch part and the finally part when an exception is caught whereas only the finally part is executed if

no exception occurs. To implement the **throw** statement in a way that also supports the **resume** statement, we leveraged the original implementation provided by the JavaScript $V_1$ interpreter: the node of the parse tree associated to each statement is assigned an attribute called `cvalue` that marks if that statement keeps the normal flow (`JSCVType.NORMAL`) or not, such as upon execution of a **break** or a **continue**. Similarly, when a throw statement is found, the exception is not thrown right away, rather the `cvalue` attribute is set to `JSCVType.THROW`. If that value is found when executing a statement, then all other statements within the same block are skipped. Listing 4 shows how to achieve reuse for this implementation. The `JELStatementList` imports its syntax from the JavaScript $V_1$ statement lists (line 2) and it overrides its semantics by leveraging the delegation operator [4] (lines 5 and 10). Upon encountering the delegation operator, the actual semantic actions to be executed are specified within the slice unit, as shown at lines 17 and 19. Therefore, the same code may use different delegates by using a different slice with a different **delegates** block. While the semantics of the semantic action labeled as `s_list_1` stay the same (it calls the delegate but it adds no code), the semantic action labeled as `s_list_0` is overridden. After the delegation, if the current statement of the list was a **throw** (line 7), then the parse tree node for the next statement is pushed to an endemic instance called `JELResumeArea`: this information is the resumption point of any **resume** statements within the exception handlers (line 8). This information is then retrieved (line 30) and executed (line 31) in the current context by the semantic action of the **resume** statement itself. Listing 5(a) shows a JavaScript $V_3$ program in which variable $a$ evaluates to 44 because the execution of the **try** block is resumed after executing the exception handler. The **retry** statement leveraged a similar technique: the node associated to the first statement of a protected section is pushed to the `JELRetryArea` endemic instance and can be retrieved upon executing the **retry**. In Listing 5(b) the protected section is retried until $a > 10$. For both the `JELResumeArea` and the `JELRetryArea` the latest resumption point is popped upon exiting the protected section.

Table 1 reports the effort associated to the implementation of the language features hereby discussed. Overall, the implementation required the creation of 20 new files and the modification of just one existing file—*i.e.*, the Neverlang **language** unit was updated to include the new language features. Notice how no files had to be modified to achieve these results, therefore this evolution step required no deletions. This result is important because it shows that once the memory layout for JEL is in place, the exception handling language features can be implemented without further modifications to existing code. Moreover, out of the 489 insertions (only 81.6 LoC per exception handling language feature on average), none was used to add features to the memory layout–*e.g.*, by adding additional information within the sections on the stack—instead insertions were limited to the implementation of the language components and their interaction with JEL. We can now answer $RQ_2$ and $RQ_3$.

**How hard is it to add exception handling support to a language implementation using the exception handling layer?**

Adding an exception handling language feature to a language that uses a memory layout compliant to the EHL such as JavaScript $V_2$ took 81.5 LoC per feature on average. In total, we implemented 6 different language features: **try**, **catch**, and **finally** blocks, and **throw**, **retry**, and **resume** statements using 489 LoC and without affecting any of the pre-existing implementation, except for 4 lines of glue code. The total effort may increase if the interpreter needs to implement different exception handling mechanisms, such as a dispatcher different from the JEL default. However, such a change can also be achieved without changing the original code, but simply by adding pieces of glue code such as Neverlang endemic slices.

**How much is the achieved modularization reliant on Neverlang-specific mechanisms?**

In the first evolution step ($V_1 \rightarrow V_2$), we achieved the refactoring by writing only 3 Neverlang units used as glue code. Most of the modifications involved the reliance of Java classes on the singleton instances; we refactored this into a more customizable mechanism using only Java, as exemplified in Listing 3. Thus, we believe that the same refactoring could be performed in other language workbenches with similar results. In the second evolution step ($V_2 \rightarrow V_3$) we could add the exception handling features without changing the original code partly thanks to Neverlang features, especially the delegation operator shown in Listing 4. Although delegation can be used to compose semantic actions [4], in this context it was simply used as an overriding mechanism for semantic actions, a feature that is supported by most language workbenches such as MontiCore [29], MPS [7], Lisa [42], Spoofax [30], and Melange [20]. The same can also be achieved with aspect-oriented superimposition [36]. Similarly, Listing 2 shows the composition among JEL elements using the Neverlang endemic slice construct. However, the same result can be achieved with an additional layer between the semantic action and the JEL interface, as shown in Fig. 2 with the `Throw Exception` component: instead of performing the composition directly within Neverlang, the same composition could be performed within a Java class. We conclude that the reliance of our implementation on Neverlang was very limited and was a form of opportunistic reuse rather than an actual requirement.

### 4.3 LogLang Evolution Scenario

The EHL and JEL by extension are intended to be used across different languages with different characteristics. Since this model is meant to work in tandem with the modularization options offered by language workbenches, it is particularly

```
1 task SomeTask {
2   backup "/foo/bar.txt" "/backup/bar.bak"
3   remove "/foo/bar.txt"
4 }
```

**(a)** Exemplary task written in LogLang.

```
./gradlew runLogLang
> Task :runLogLang
executing task SomeTask
File ./foo/bar.txt does not exist, do you want to create it?
```

**(b)** LogLang finalizer in action.

**Listing 6.** LogLang with exception handling support.

relevant that it is compliant to the needs of domain-specific languages (DSL), that are typically the main output of language workbenches. To test this, we stretched JEL capabilities to implement a recovery procedure for the LogLang DSL, used for file system tasks declarations [13]. LogLang tasks are declarative, do not support any form of exception handling nor run on any memory layout. The DSL also does not include any language features to catch errors, nor to define handlers. In this case, the idea was to verify if the same model is applicable to a scenario in which most elements of the EHL can be omitted. To achieve this goal, we extended LogLang, so that LogLang tasks throw a JEL exception whenever the file on which the task must be performed does not exist. Such an example is shown in Listing 6. Since there are no memory, no sections, and no handlers, both the dispatcher and the selector always fail their search, therefore according to Listing 1, control is eventually taken by the finalizer. The finalizer is a custom procedure that prompts the users by asking them if they want to create the file. For instance, when running the task reported in Listing 6(a), file "foo/bar.txt" does not exist, therefore the user is prompted accordingly, as in Listing 6(b). The users can either accept or decline: in the former case the file is created and the task execution is resumed normally; in the latter the finalizer stops the application. This language evolution required the creation of just one class[5] of 70 LoC, added 1 LoC to three different Neverlang modules—*i.e.*, the code needed to throw the exception—and modified an endemic slice to include the JEL-related endemic instances. In total, the refactoring took 77 LoC. We can now answer $RQ_4$.

**Does the conceptual framework support varied exception handling mechanisms and their language features?**

Thanks to JEL we could implement varied exception handling language features. Some features were fairly traditional, such as the `throw` statement, and the `try`, `catch`, and `finally` blocks in JavaScript, while others are rather unconventional, such as the `retry` and `resume` statements. Most notably, these features are not supported by the exception handling mechanisms offered by the JVM, but they could be easily implemented with JEL in a modular way. Moreover, we applied the same architecture for the implementation of an unconventional recovery procedure for a DSL without any memory layout and that does not support any exception handling by default. We believe that these two evolution scenarios prove the applicability of the EHL to the creation of varied exception handling mechanisms and their features.

[5] For simplicity, we created only one class that implements the `Dispatcher`, `Selector`, and `Finalizer` interfaces at the same time.

### 4.4 Threats to Validity

**External validity.** The language evolution scenarios are based on Neverlang and Java and they may not be possible to reproduce it in different contexts. To prevent this issue, we implemented JEL without making any assumptions neither with regards to the base language nor to the language workbench. In particular, JEL does not rely on the Neverlang runtime and can be used by any program running on the JVM. We also tried to limit our reliance on Neverlang to perform the language evolution, as discussed in the answer to $RQ_3$. Similarly, JEL was created from scratch and does not rely on specific characteristics of Java to work. Even the infrastructure based on the five generic types is just a programmer convenience to improve error messages at compile time, since those types are affected by type erasure and do not exist at runtime. In summary, we believe that a similar library with data structures and algorithms compliant to the EHL could be implemented in any other general purpose programming language other than Java.

**Construct validity.** The answer to $RQ_1$ and $RQ_2$ is based on a specific evolution scenario, therefore a similar implementation of the EHL on other languages may require additional effort. To avoid this issue, we strictly defined the EHL first, then developed JEL following the EHL and finally performed the evolution experiment. We never reiterated on any prior step to accommodate the evolution experiment. Whenever a change was necessary to perform the evolution, it was made to the host language implementation and never to the library nor to the EHL therefore our measures should be able to represent the actual development effort.

**Internal validity.** We defined the EHL and answered $RQ_3$ and $RQ_4$ based on our experience with exception handling mechanisms and with language workbenches. This may cause an issue due to exotic exception handling mechanisms and language workbenches we may be unaware of. To prevent this issue, we did not make any assumptions on the language, even allowing for arbitrary memory layouts; we also tested an unconventional exception handling mechanism in which most of the elements of the EHL are optional. This convinced us of the generality of our architecture, as well as of its implementation.

## 5 Related Work

Modular language development is a popular research topic. Many language workbenches have been proposed, each with its own take on language composition. Their contribution is related to ours due to their focus on providing ways to

avoid the monolithic approach to language implementation. Melange [20] integrates tools from the Eclipse Modeling Framework (EMF) ecosystem [46]and supports language extension and language merge [20]. Meta Programming System (MPS) [52] is a development environment for non-textual DSLs based on projectional editing [53] using concepts (abstract syntax nodes) and behaviors (semantics). MontiCore [33] generates abstract data types for the parse tree and uses Java visitors for the semantics. It supports reuse through the extension of abstract data types and grammar inheritance. Rascal [31] is a meta-programming language that supports the implosion of parsed text and parse tree transformations. The evaluation leverages the pattern-based dispatch technique [3]. Spoofax [54] provides several DSLs for language development; the semantics are called rules and strategies and can be defined as a sequence of functions over the AST. To the best of our knowledge, there are no contributions using language workbenches to directly address the exception handling mechanism and its portability, but, as we discussed in Sect. 4.2, we believe that the EHL is applicable to all aforementioned language workbenches and Java-based ones could even use the JEL library.

Development of crosscutting features such as exception handling have been discussed mainly with regards to aspect-oriented programming. For instance, Liebig *et al.* use the superimposition operator to handle crosscutting features in Mobl [37]. Hadas and Lorenz switch the perspective by introducing language oriented modularity [28]: instead of tackling the problem of crosscutting features in languages, they leverage the ease of use of language workbenches to create several DSLs, each tackling a different crosscutting concern in other systems. However, interactions among language-based tools are hard to understand without good integration [5]; compared to the EHL, their work is not applicable to the definition of modular exception handling language features in a unique language.

On the topic of exception handling, some contributions focus on exception handling in management systems: Chiu *et al.* [17] address the importance of reusing exception handlers to deal with workflow exceptions and propose the ADOME exception handling environment for the definition of dynamic bindings for exception handlers, run-time modifications of exception handlers and exception handler reuse. Similarly, the VIEW scientific workflow management system provides customizable and hierarchical exception handlers; the authors also propose a language for user-defined exception handling mechanisms [45]. Celovic and Soukouti [15] describe the proper use of exception handlers for the development of large scale enterprise systems. In their work, they defined six groups of responsibilities, including the thrower and the catcher; our conceptual framework is similar and reflects these responsibilities. In all these cases, the applicability to traditional programming languages is not discussed. The contribution from Ogasawara *et al.* [43] addresses on the optimization of stack unwinding and stack cutting in Java and could be used to create an optimized version of the exception handling layer. More in general, using an intermediate layer to abstract the memory layout and exception handling introduces an overhead that requires optimizations such as by limiting the costs of metaprogramming capabilities used by the language workbench [39] and using optimized AST interpreters with partial evaluation [38].

Cabral and Marques implement retry semantics on languages lacking this language feature using aspect-oriented programming [6]. Bagge *et al.* [2] present a layer that can be used on top of any platform-specific error reporting to generalize error reporting and handling through the alert concept. The proposed implementation also supports retry semantics, but it is implemented as an extension to the C language, therefore replication in other languages requires the development of a similar extension. Chase [16] also discussed exception handling in C, although his remarks are general enough to be valid for any language. Chase observes that the exception handling mechanism should be smoothly integrated with the rest of the host programming language, but the contribution focuses on low level details instead of defining a general and abstract framework such as the EHL.

In their contribution, Brinke *et al.* [47] discuss a tailorable control flow, including exceptional flow. In their view, all exception handling mechanisms should be supported within the same language and application programmers should be able to choose which kind of exception handling mechanism they want to use. Their work is closely related to ours, since they propose an intermediate layer to customize exception handling, but the code is written using continuations, thus compared to the EHL the base language must support first-order functions and the resulting code may be less readable.

## 6 Conclusions

Exception handling is a collection of language features whose implementation is usually hard to reuse because scattered across several parts of the implementation. The EHL framework permits to untangle the code of the exception handling language features from the code of other language features. The EHL architecture is very flexible, allows for arbitrary memory layouts, dispatching algorithms and handling procedures, and most of its elements are optional. We proved the EHL applicability by developing the JEL library and using it to add exception support to a full-fledged implementation of JavaScript without changing its implementation. Our experience shows that several exception handling language features—both conventional and unconventional—can be achieved with an high degree of modularity and with limited development effort.

## Acknowledgments

This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

## References


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (second ed.). Addison-Wesley, Boston, MA, USA.

[2] Anya Helene Bagge, David Valentin, Magne Haveraaen, and Karl Trygve Kalleberg. 2006. Stayin' Alert:: Moulding Failure and Exceptions to Your Needs. In *GPCE'06*. ACM, Portland, OR, USA, 265–274.

[3] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. 2015. Modular Language Implementation in Rascal—Experience Report. *Science of Computer Programming* 114 (Dec. 2015), 7–19.

[4] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. 2023. On the Granularity of Linguistic Reuse. *Journal of Systems and Software* 202 (Aug. 2023). https://doi.org/10.1016/j.jss.2023.111704

[5] Barret Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. 2015. Globalized Domain Specific Language Engineering. In *Globalizing Domain-Specific Languages (Lecture Notes in Computer Science 9400)*, Benoît Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe (Eds.). Springer, 43–69.

[6] Bruno Cabral and Paulo Marques. 2009. Implementing Retry—Featuring AOP. In *LADC'09*. IEEE, João Pessoa, Brazil, 73–80.

[7] Fabien Campagne. 2016. *The MPS Language Workbench*. Vol. 1. CreateSpace Independent Publishing.

[8] Walter Cazzola. 2012. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *SC'12 (Lecture Notes in Computer Science 7306)*. Springer, Prague, Czech Republic, 162–177.

[9] Walter Cazzola, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. 2018. $\mu$-DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures* 51 (Jan. 2018), 71–89. https://doi.org/10.1016/j.cl.2017.07.003

[10] Walter Cazzola and Luca Favalli. 2022. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering* 27, 4 (April 2022). https://doi.org/10.1007/s10664-021-10074-6

[11] Walter Cazzola and Luca Favalli. 2023. Scrambled Features for Breakfast: Concept, and Practice of Agile Language Development. *Commun. ACM* (Nov. 2023).

[12] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (Sept. 2016), 404–415. https://doi.org/10.1109/TETC.2015.2446192

[13] Walter Cazzola and Davide Poletti. 2010. DSL Evolution through Composition. In *RAM-SE'10*. ACM, Maribor, Slovenia.

[14] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2: Componentised Language Development for the JVM. In *SC'13 (Lecture Notes in Computer Science 8088)*. Springer, Budapest, Hungary, 17–32.

[15] Dino Celovic and Nader Soukouti. 2004. *About Effective Exception Handling*. White Paper. Sanabel Solutions.

[16] David Chase. 1994. Implementation of Exception Handling. *The Journal of C Language Translation* 5, 4 (June 1994), 229–240.

[17] Dickson Chiu, Qing Li, and Kamalakar Karlapalem. 2000. A Logical Framework for Exception Handling in ADOME Workflow Management System. In *CAiSE'00 (Lecture Notes in Computer Science 1789)*. Springer, Stockholm, Sweden, 110–125.

[18] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. 2017. Exception Handling Bug Hazards in Android. *Empirical Software Engineering* 22 (June 2017), 1264–1304.

[19] Adrian Colyer, Awais Rashid, and Gordon Blair. 2004. *On the Separation of Concerns in Program Families*. Technical Report 107. Lancaster University, Lancaster, United Kingdom.

[20] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In *SLE'15*. ACM, Pittsburgh, PA, USA, 25–36.

[21] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *LDTA'12*. ACM, Tallinn, Estonia.

[22] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerrtsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, and Eelco Visser. 2013. The State of the Art in Language Workbenches. In *SLE'13 (Lecture Notes on Computer Science 8225)*. Springer, Indianapolis, USA, 197–217.

[23] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Alex Kelly, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures* 44 (Dec. 2015), 24–47.

[24] Luca Favalli, Thomas Kühn, and Walter Cazzola. 2020. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In *SPLC'20*. ACM, Montréal, Canada, 285–295.

[25] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog. http://www.martinfowler.com/articles/languageWorkbench.html

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Ma, USA.

[27] John B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (Dec. 1975), 683–696.

[28] Arik Hadas and David H. Lorenz. 2016. Toward Pratical Language Oriented Modularity. In *Modularity'16*. ACM, Málaga, Spain, 94–98.

[29] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. 2016. Compositional Language Engineering Using Generated, Extensible, Static Type-Safe Visitors. In *ECMFA'16 (Lecture Notes in Computer Science 9764)*. Springer, Vienna, Austria, 67–92.

[30] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *OOPSLA'10*. ACM, Reno, Nevada, USA, 444–463.

[31] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM'09*. IEEE, Edmonton, Canada, 168–177.

[32] Jan Kort and Ralf Lämmel. 2003. Parse-Tree Annotations Meet Re-Engineering Concerns. In *SCAM'03*. IEEE, Amsterdam, The Netherlands, 161–170.

[33] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 5 (Sept. 2010), 353–372.

[34] Thomas Kühn and Walter Cazzola. 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *SPLC'16*. ACM, Beijing, China, 50–59.

[35] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *SPLC'15*. ACM, Nashville, TN, USA, 71–80.

[36] Ralf Lämmel. 2003. Adding Superimposition to a Language Semantics. In *FOAL'03*. Boston, MA, USA, 65–70.

[37] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *VaMoS'13*. ACM, Pisa, Italy.

[38] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *OOPSLA'15*. ACM, Pittsburgh, PA, USA, 821–839.

[39] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. In *PLDI'15*. Portland, OR, USA.

[40] Bertrand Mayer. 1992. Applying 'Design by Contract'. *Computer* 25, 10 (Oct. 1992), 40–51.

[41] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoît Baudry. 2016. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures* 46 (Nov. 2016), 206–235.

[42] Marjan Mernik. 2013. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software* 86, 9 (Sept. 2013), 2451–2464.

[43] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. 2001. A Study of Exception Handling and Its Dynamic Optimization in Java. *Sigplan Notices* 36, 11 (Oct. 2001), 83–95.

[44] Guilherme de Pádua and Weiyi Shang. 2018. Studying the Relationship between Exception Handling Practices and Post-Release Defects. In *MSR'18*. ACM, Gothenburg Sweden, 564–575.

[45] Dong Ruan, Shiyong Lu, Aravind Mohan, Xubo Fei, and Jia Zhang. 2012. A User-Defined Exception Handling Framework in the VIEW Scientific Workflow Management System. In *SC'12*. IEEE, Honolulu, Hawaii, USA, 274–281.

[46] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework*. Addison-Wesley.

[47] Steven te Brinke, Mark Laarakkers, Christoph Bockisch, and Lodewijk Bergmans. 2012. An Implementation Mechanism for Tailorable Exceptional Flow. In *WEH'12*. Zürich, Switzerland, 22–26.

[48] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures* 43, 3 (Oct. 2015), 1–40. https://doi.org/10.1016/j.cl.2015.02.001

[49] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *SPLC'14*. ACM, Florence, Italy, 167–176.

[50] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *SLE'13 (Lecture Notes on Computer Science 8225)*. Springer, Indianapolis, USA, 76–95.

[51] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1-2 (Jan. 2010), 39–54.

[52] Markus Völter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *ICSE'12*. IEEE, Zürich, Switzerland, 1449–1450.

[53] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *SLE'14 (Lecture Notes in Computer Science Volume 8706)*. Springer, Västerås, Sweden, 41–61.

[54] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (Sept./Oct. 2014), 35–43.

[55] Martin P. Ward. 1994. Language Oriented Programming. *Software—Concept and Tools* 15, 4 (Oct. 1994), 147–161.


Received 2023-07-07; accepted 2023-09-01