Identification and Refactoring of Exception Handling Code Smells in JavaScript

Chin-Yun Hsieh[1], Canh Le My[2], Kim Thoa Ho[3], Yu Chin Cheng[1]
[1]Department of Computer Science and Information Engineering, National Taipei University of Technology, Taiwan
[2]Department of Information Technology, Hue University of Sciences, Vietnam
[3]Department of Informatics, Hue University of Education, Vietnam
hsieh@csie.ntut.edu.tw, lemycanh@husc.edu.vn, kimthoasp90@gmail.com, yccheng@csie.ntut.edu.tw

Abstract

Avoid making bad smells is very important in writing exception handling code for dealing with unexpected runtime errors. The task however is challenging and demands proficient programming skills and experience. This is particularly true in developing JavaScript applications because JavaScript is very rich in features as being dynamic, interpreted and object-oriented with first-class functions. What further complicates the situation is the use of event-driven and non-blocking I/O model in Node.js, which is a runtime environment written in server-side JavaScript.

Extended from our previous work on exception handling code smells in Java, this study aims at identifying exception handling code smells that can occur in a JavaScript application at either the client side or the server side. The impact to software quality that each smell has is demonstrated with examples. Refactorings corresponding to the identified smells are proposed; their effects to the application, including the robustness level achieved and other benefits gained, are illustrated. The work is intended to serve as a guide in helping JavaScript developers avoid or discover exception handling code smells.

Keywords: Exception handling, Code smells, refactoring, Node.js, JavaScript.

1 Introduction

Code smells are known to degrade software quality in various attributes, including readability, testability and maintainability [1-3]. Code smells in exception handling create even more critical problems in that, in addition to reduce source code quality, they also affect the robustness of a program. Though getting a right design of exception handling before coding phase is always desired, it is usually difficult however, especially in agile software development which features changing customer requirements. Since upfront design of exception handling before coding is not always possible, code can become infested with exception handling code smells (EH smells). Thus, it becomes necessary for developers to restore code quality on a regular basis. This motivates the need of smell detection and the use of refactoring for smell removal so as to improve the quality of existing source code. In effect, refactoring for EH smells (EH refactoring [3]) may not only enhance the system's robustness but also improve a variety of quality attributes.

This study is extended from our previous work on exception handling bad smells in Java [3]. The objective is to investigate bad smells which may occur in a JavaScript application at either the client side (i.e., JavaScript runs as a part of web browsers) or the server side (i.e., JavaScript used in Node.js). Since client side JavaScript program is subject to frequent changes and such changes can easily propagate to the server side, detecting and removing EH smells become a crucial development activity for maintaining code quality and ensuring robustness in JavaScript programs. We first define the identified EH smells. Then, corresponding exception handling refactorings are proposed for their removal. The effects of applying each of the refactorings are also addressed.

This research is organized as follows: we first present related work in Section 2. Section 3 provides an overview of exception handling mechanism of JavaScript and exception handling robustness levels of our concern. Section 4 presents the definitions of five EH smells identified and their impacts to a JavaScript application. In Section 5, the identified EH smells are further illustrated by code examples; the corresponding refactorings for their removal are presented together with the resulting effects achieved. A result analysis is given in Section 6. Finally, some conclusions are drawn and future work is listed in Section 7.

2 Related Work

A lot of research work has been done on improving the code quality by detecting code smells and providing refactoring methods for their removal [1-2]. However, most studies target on main logic code smells and only a very few ones focus on code smells in exception handling [3]. Among those few previous studies, Michael Feathers

*Corresponding author: Chin-Yun Hsieh; E-mail: hsieh@csie.ntut.edu.tw
DOI: 10.6138/JIT.2017.18.6.20160118

in Clean Code [2] pointed out some techniques to write a good code for exception handling: use exceptions rather than return code, use unchecked exceptions, don't return null, and so on. However, these past works mainly focus on exception handling of static object oriented languages such as Java or C#. The code smells detection and refactorings related to exception handling in a dynamic language such as JavaScript is a new aspect to the best of our knowledge.

This study is inspired by our previous work "Exception handling refactorings: Directed by goals and driven by bug fixing" [3], which defined detection rules and refactoring mechanisms for six types of exception handling code smells in Java. In this work on JavaScript, in addition to investigating the code smells of exception handling from the viewpoint of synchronous programming, we also explore code smells associated with the asynchronous programming mechanism of JavaScript. Furthermore, the specific features of JavaScript as a dynamic language is investigated in relation to exception handling. For each of the identified smells, a corresponding refactoring for its removal is proposed. Identifying the exception handling bad smells and applying refactoring to remove them contribute to improving exception handling design.

3 Background

This section presents an overview of concepts relevant to the central theme of our research. First, some aspects of exception handling mechanism are generalized, including exception representation, exception propagation, cleanup action, reliability check and continuation of control flow. We then summarize three robustness levels of exception handling based on the previous work [3].

3.1 Exception Representation

Each programming language has its own way to represent exceptions. In sum and in short, we can classify them into two groups in terms of the way they are represented: as symbols or as objects. The representation of exceptions as symbols is a classic approach used in languages such as Guide, Smalltalk, Eiffel [4-5] where exceptions are strings or numbers. Most modern programming languages, especially Object Oriented programming languages such as C# and Java, take the second approach: exceptions are objects. JavaScript is a dynamic typed language that supports the use of any legitimate type to represent exceptions. A throw statement in JavaScript accepts either a primary JavaScript value (i.e., exception as symbol) or an object [6] as shown in the following examples.

```
throw "Index out of bound"; //throw a string
throw 404; //throw a number
throw new Error("Invalid JSON string");//throw an object
```

3.2 Exception propagation

Exception propagation is the signaling of an exception from a callee to its caller. There are two ways to propagate an exception: implicit (i.e., automatic) or explicit [4]. JavaScript supports both ways of exception propagation, with implicit propagation as the default behavior. JavaScript provides a try-catch-finally structure as a way to test a block of code for errors and handle the errors that may occur. The control flow will jump to its handler if an exception occurs in the try block; the exception will in turn be automatically propagated to higher-level components if no corresponding handler is defined. Moreover, a caught exception can also be propagated explicitly by rethrowing in a catch block.

3.3 Cleanup Action

Components in a program may terminate normally or abnormally because of exceptions. In either case, keeping the component's state in a known state is essential. It is usually carried out by performing some cleanup actions such as releasing acquired resources, restoring a snapshot of program, rolling back changes to some variables or database before the component exits. Cleanup actions may be supported in three different ways, explicit propagation, specific construct and automatic clean-up [4], depending on the design of a language. In JavaScript, the finally block of a try statement is guaranteed to be executed before the try statement exits no matter if any exception was raised or not, i.e., JavaScript uses specific construct for cleanup action.

3.4 Reliability Check

Reliability checks test for possible errors introduced by the use of an exception mechanism; for instance, throwing an object of undefined type. There are two design approaches with respect to reliability: static check and dynamic check; static check is performed by compiler while dynamic check is performed by the run-time system [4]. Dynamic check is used in JavaScript which means any error in using exception mechanism would not be detectable until runtime.

3.5 Continuation of Control Flow

After an exception has been handled, the invoker of exception signaler may continue its execution in different ways, depending on the exception handling design of the language. There are two general approaches for the continuation of control flow: resumption model and termination model [4]. In resumption model, control flow will transfer to the point immediately following the place where the exception was raised after the handler has been executed. In termination model, the control flow will continue at the statement after the protected region as if the protected region were completed normally without encountering any exception [7]. The termination model is used in JavaScript.

3.6 Exception Handling Robustness Levels

Three different robustness levels were identified to reflect the results achieved by the most common exception handling strategies in [3]. They are error-reporting (G1), state recovery (G2) and behavior-recovery (G3). In this work, an additional level, called undefined (G0), is added to represent status of application that fails to achieve G1. These exception handling robustness levels are incremental and inclusive [3] and are used as goals for guiding refactoring.

3.7 JavaScript Run-to-Completion Nature

JavaScript event loop runs under a single thread (for each process), you can't do any parallel code execution; so all operations are inherently thread safe. In JavaScript, a task runs until completion once begins. The event loop will shift a task off the task queue and execute it when a task is finished executing [8-9].

As shown in the following example, the sleep function will consume CPU for 2 seconds; while it is being executed, the current process is blocked. Another noticeable result is that it will display "First" then "Second" instead of "Second" then "First." Since the second argument for setTimeout is 1000, one may expect that "Second" should appear before "First," while it is actually not. Note also that setTimeout will enqueue printSecond as a task onto task queue, the current code will continue in a manner of run-to-completion, which prints out "First" after 2 seconds for executing sleep (2000). Now, if there isn't any other task in the queue, the task printSecond will be dequeued and processed right away; otherwise, the printSecond task will have to wait for other tasks to be processed.

```
function sleep(time) {
   var stop = new Date().getTime();
   while(new Date().getTime() < stop + time) {
      ;
   }
}
function printSecond() {
   console.log("Second");
}
setTimeout(printSecond, 1000);
sleep(2000); //sleep for 2 seconds
console.log("First");
/* output:
First
Second */
```

3.8 Non-Blocking I/O

As stated previously, JavaScript (including Node.js) guarantees that a task is processed until it is completed before any other task is processed [8-10]. However, almost all I/O operations (e.g., network requests, database operations, disk input/output operations) are non-blocking (i.e., are executed in parallel). Specifically, when an I/O operation is requested on the main thread of execution, it will be delegated to the JavaScript runtime along with a callback function. The main thread now continues its own processing. When the operation has been completed, a task is enqueued along with the provided callback function. This task later on will be dequeued by the event loop and the callback is processed.

Non-blocking I/O especially brings to Node.js

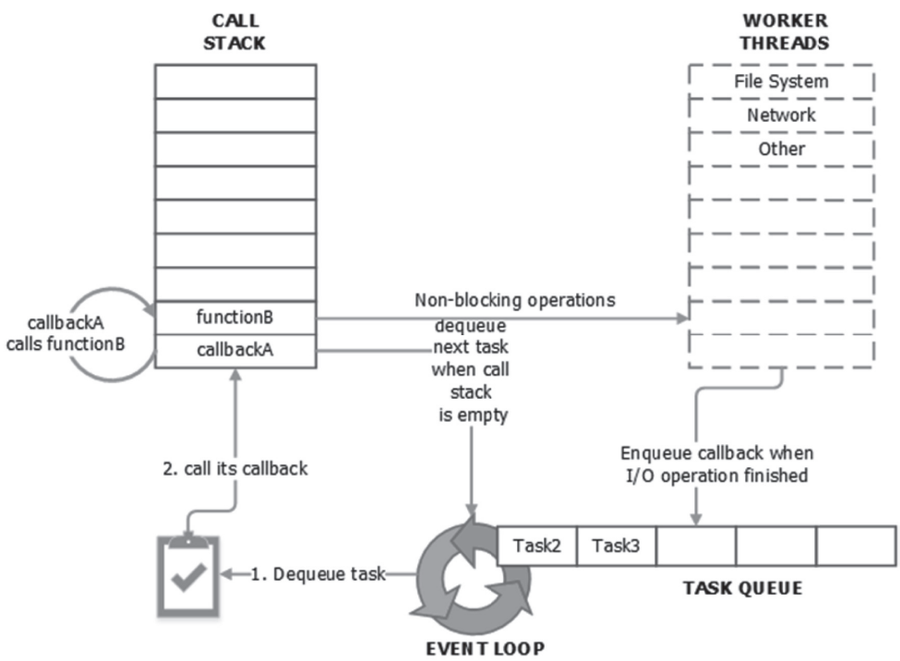

Figure 1 Scheme of the Non-Blocking I/O Model in Node.js

significant benefits in performance [10-11]. Programmers needn't to worry that doing I/O will block other requests. Since an I/O operation is several orders of magnitude slower than a non-I/O operation, a server-side Node.js program can handle a lot of parallel I/O operations.

4 Exception Handling Code Smells

In this section, five exception handling code smells identified in the study are defined, they are *ignoring exception, try/catch outside asynchronous callback, no global uncaught exception handler, throwing symbol instead of error object* and *dummy handler*. The negative effects that each of the smells has to a JavaScript application are illustrated.

4.1 Ignoring Exception

ignoring exception may appear as an Empty Catch Block (ECB) as mentioned in [3]. Leaving a catch block empty usually is not a good idea in that an exception is simply ignored silently. A thrown exception means that there is something wrong with the current application state, using ECB means all is fine however. A program executing with an error being ignored may lead to unexpected behaviors or failures without leaving a clue for debugging [3].

A variant of *ignoring exception* at server side in Node.js is skipping the check of the error-first parameter in callback. Most functions in Node.js are asynchronous and the Error-First callback pattern is applied throughout the framework, which means one has to check the first argument in the callback to determine whether the function was completed with or without an error. Skipping the checking of the first error parameter results in the same problem as leaving the catch block empty, .i.e., making it more time-consuming for tracing the failure back to the place where an exception was raised.

4.2 Try/Catch Outside Asynchronous Callback

JavaScript's event loop is designed as a run-to-completion environment in which each task is processed completely before another task is processed. When a task processing ends, the event loop will remove the next task from the queue and process it [8]. Enclosing an asynchronous callback function with a try/catch statement won't be able to catch any exception from the call back function because, when the callback is dequeued from task queue and executed, the try statement has already exited. Consequently, the exception may become an uncaught exception and the program will terminate. When this happens, the application is considered poor in quality. It is essentially a programming error that inexperienced programmers tend to make.

4.3 No Global Uncaught Exception Handler

At the client side, uncaught exceptions will terminate the current JavaScript program in an unexpected way and an error message is logged to browser's error console. Since client-side web application contains multiple JavaScript programs, uncaught exceptions also may leave the web application in an unknown state that later result in unexpected behaviors. Moreover, those uncaught exceptions will only show up at user's console without being reported to developers. Conclusively, the lack of global error handler degrades the robustness of a client-side web application.

As to the server side, a Node.js application runs on a single process by default. The process will be terminated on any uncaught exception. This is troublesome because we mostly run Node.js application as a web server. Certainly, an application will always have exceptions and some of them could possibly be uncaught. An uncaught exception will bring down the whole application and disable it from serving any further incoming requests. This situation should be considered as having poor software quality.

4.4 Throwing Symbol Instead of Error Object

A JavaScript throw statement may throw either a symbol or an object. Nevertheless, inexperienced developers usually tent to throw an error as a symbol. Although this is supported by the language, it leads to other issues in error handling. First, it is ambiguous when handling individual exception type, because it is hard to determine the type of exception from a string or a number. Comparing strings or numbers to have corresponding error handling is less readable. The caller of the error signaler may perform an instance of Error check for having a corresponding handling. Therefore throwing an error as a symbol would make the check fails in its caller. As a result, the interaction between modules is reduced. In addition, passing symbols to asynchronous callbacks has the same effect as throwing symbols.

```
throw "error message";
callback("error message");
```

Secondly, throwing an error as a symbol provides no stack trace, which specifies where the error was raised. This leaves us less information for debugging.

Thirdly, we can't attach additional context related information to an error which is a symbol. Consequently, throwing an error as a symbol is considered a bad practice for error debugging.

4.5 Dummy Handler

A *dummy handler* is an exception handler that does nothing more than logging the exception [3]. Consequently, *dummy handler* will have the same problems as *ignoring exception* in making the program assume that the error has

been handled while it is actually not. Continued execution of the program in those cases will make the program fail slowly and possibly result in unexpected behaviors (e.g., client gets a weird result displayed on the screen or just part of the result is delivered to the client).

5 Exception Handling Refactorings

In this section, a refactoring method is proposed for each of the five exception handling code smells presented in Section 4; the effects of applying each refactoring are addressed. Table 1 summarizes the proposed refactorings and the resulting effects.

Table 1 Proposed Refactorings and Their Effects

EH smell	Refactoring	Effects
Ignoring exception	Remove try statement or check for error in callbacks	• Achieves robustness level G1
Try/catch outside asynchronous callback	Move try/catch statement into asynchronous callback	• The programming error is fixed
No global uncaught exception handler	Implement global uncaught exception handler	• Keeps server alive • Gets developer to notice of the error • Achieves robustness level G1
Throwing symbol instead of error object	Replace symbol with error object	• Increases readability • Achieves robustness level G1
Dummy handler	Rethrow error or pass error object to callback	• Achieves robustness level G1

When presenting a refactoring method, we use the format of previous work [3] which is originated from Martin Fowler's Refactoring book [1]. The format is composed of four elements: name, summary, motivation, mechanics and example.

5.1 Remove try statement or check for error in callbacks

You have a try statement with an empty catch block, *remove the try/catch statement and let the exception be automatically propagated to higher levels in the call chain*.

```
try {
   /* Code may throw error */
} catch (e) {}
            ↓
/* Code may throw error */
```

In the case of ignoring error in an asynchronous callback, *check for the error and handle it or deliver it to a higher-level callback*.

```
doAsynchronousFunction(inputdata, function(err, data){
   /* Code that accesses returned data */
});
                                    ↓
doAsynchronousFunction(inputdata, function (err, data) {
   if (err) {
 // Handler the error or
 // Deliver it to higher callback then return or
 // Leave a comment to explain the reason of ignoring.
   }
   /* Code that accesses returned data */
});
```

5.1.1 Motivation

First, some ECBs are acceptable but some are not. In JavaScript, there is no checked exception like Java; therefore developers are not required to catch the exception. The reason of acceptable ~~ECB~~ varies and depends on the source code semantics. Anyway, leaving a comment in catch block to explain why the error is being ignored would increase source code readability. However, we consider other cases such as an exception caused from converting a variable, which contains a malformed JSON string, to a JSON object or; blindly ignoring all errors caused by invalid user input values. ECBs in those situations will cause slow-failing and thus should be removed. To remove ECB, Chen et al. proposed "Replace ignored checked exception with unchecked exception" [3]. However, as JavaScript has no checked exception, removing try/catch statement and letting the error automatically propagate to its caller have the same effects. Caller can deal with that exception by handling it or, at least if no one handles the exception, letting the application automatically fail fast [12] (i.e., achieves robustness level G1); in this case, an uncaught exception will be fired, thus providing us with useful information for debugging. In addition, we can implement a default global uncaught exception handler for further processing such as logging and restarting the process to make sure the server never goes offline in the case of a server-side application; or logging (posting to server), notifying users and possibly reloading client-side JavaScript application.

Secondly, developers should always check for errors in asynchronous callbacks. As stated above, usually the first parameter of any callback function is an error. Checking this parameter helps us ensure that if an error did happen, it should have already been dealt with before continuing to access the returned data.

5.1.2 Mechanics

For an empty catch block code smell:

- Remove the try/catch statement outside the source code that may throw exception.
- Run tests.
- Or leave a comment for the reason of ignoring.
- For ignoring error first parameter in asynchronous callback:
- Create an if statement to check if the error parameter is null and make it the first statement in the callback.
- If the error is not null:
  - Handle the error, or
  - Pass the error to higher callback then return immediately, or
  - Leave a comment for explaining why the error is ignored.
- Run tests.

5.1.3 Example

The example below is an empty catch block in a synchronous function. We use parseJson function of jQuery to parse a JSON string; an exception Error ("Invalid JSON: " + data) will be thrown for a malformed JSON string. Functionally, leaving the catch block empty implies giving up converting the data from string to JSON object if an exception was thrown. Consequently, any subsequent access to properties of data JSON object will end up with an undefined value or some other unexpected behaviors (e.g., Object doesn't support property or method) that may take time to debug, because the invalid JSON format error now hides itself under another type of error.

```
try {
   data = jQuery.parseJSON(data);
} catch (e) {}
```

In this case, developers cannot recover the state of application since the input data may come to our application from other services. The best thing to do is letting the exception be automatically propagated to its caller and terminating the current execution by removing the outer try/catch statement.

```
data = jQuery.parseJSON(data);
```

In the second example with an asynchronous callback as shown below, the purpose is to open a file and parse the string content to a JSON object. The file reading may fail in some way, and skip checking the error parameter can make the call to function JSON.parse end up with throwing Invalid JSON error, which could cause confusion in debugging.

```
fs.readFile(filePath, function(err, data) {
  data = JSON.parse(data);
});
```

For this second example, checking the first parameter in the callback can help avoid unexpected behaviors. A refactoring is simply to deliver it to a higher-level callback to handle the error and exit the current callback.

```
function readJSON(filePath, callback) {
 fs.readFile(filePath, function(err, data) {
  if (err) {
   return callback(err);
  }
  data = JSON.parse(data);
  …
 });
}
```

5.2 Move try/catch statement into asynchronous callback

As shown in the following example, the enclosing try/catch statement of the asynchronous callback is expected to catch all errors from the callback with a catch block following the asynchronous call. In reality, those errors will become uncaught exceptions. Therefore, *move the try/catch into the most outer level of your asynchronous callback will fix the problem*.

```
try {
  doAsyncOperation(function () {
    /* Code may throw error */
  });
} catch (ex) {
  // Handle the error
}
                ↓
doAsyncOperation(function () {
  try {
    /* Code may throw error */
  } catch (ex) {
    // Handle the error
  }
});
```

5.2.1 Motivation

Instead of using a try/catch statement at outer scope of asynchronous callback, we move it into the asynchronous callback function to form the outermost level. When an asynchronous callback is executed, the try/catch statement will then be able to catch all errors occurred inside its try block. In this way, the programming error "*try/catch outside asynchronous callback*" is fixed. To prevent this error from happening again in the future, developers need to well understand the event loop of JavaScript.

5.2.2 Mechanics

- Create a try/catch statement as the first statement in the callback.
- Move all other statements in the callback into try block of the try statement stated previously.
- Move the code in the catch block of the try/catch outside callback to the catch block of the newly created try statement (it may need to resolve any error related to the scope of functions and variables, among other things).
- Remove the original try/catch that encloses the asynchronous callback.

- Run tests.

5.2.3 Example

```
try {
   setImmediate(function () {
      …
      throw (err);
   });
} catch (ex) {
   //Handle the exception
}
```

In the above example, people may think that the error in the asynchronous function will be caught by the enclosing try statement and handle it in the catch block. But this is not what actually happens. In fact, setImmediate will enqueue the provided callback to the task queue of the event loop; then the control flow continues to execute the current source code, this will exit the try statement and execute any statements after that; until the current source code finished, the event loop will check the task queue to pick up next task; if there is no other task, the previous callback will be dequeued and processed; at this moment there is no existing try/catch outside it. Therefore, any exception raised in the asynchronous callback won't be caught by the enclosing try statement.

This programming error can be fixed by moving the try/catch statement into the callback function and making it the outermost level of the function. In this way, any error occurring inside the function will certainly be caught.

```
setImmediate(function () {
   try {
      …
      throw (err);
   } catch (ex) {
      //Handle the exception
   }
});
```

5.3 Implement Global Uncaught Exception Handler

The strategies for implementing global uncaught exception handlers are proposed for both the server-side Node.js application and the client-side JavaScript application as follows.

a. For server-side Node.js application

An uncaught exception will terminate Node.js process and possibly bring down the server. Implement a global uncaught exception handler to make a log, notify the administrator and restart the process.

Developers can implement a listener for uncaughtException event of process as follows:

```
process.on('uncaughtException', function(err) {
  /* log the err and notify administrator */
  /* Code to restart the process */
});
```

or subscribe to error event of domain as follows:

```
var server = http.createServer(function (request, response) {
   var d = domain.create();
   d.on('error', function (err) {
     /* log the err and notify administrator */
     /* Code to restart the process */
   });
   d.add(request);
   d.add(response);
   d.run(function () {
            handleRequest(request, response);
   });
});
server.listen(PORT);
```

b. For client-side JavaScript application

An uncaught exception from a JavaScript program may lead client-side web application to unexpected behaviors. Implement a global error handler to catch all uncaught exceptions, log them on the server for developers to debug and possibly reload the application.

```
window.onerror = function myErrorHandler(errorMsg, url, lineNumber)
{
  //log error to server and may reload web application
}
```

5.3.1 Motivation

a. For server-side Node.js application

When an uncaught exception occurs, it is not recommended to keep the process running. uncaughtException is an event triggered away from the original source of the exception. All you get at this point is the error object with stack trace of where it was raised. Most likely no reference is available for returning to the objects surrounding the error to clean up the application state or other resources [13]. As a result, it is best to exit the undergoing process and fork a new one. This would keep the server from going crashing.

b. For client-side JavaScript application

A message of uncaught exception is logged to browser's error console window. Although this would help developers be notified of the error in development time, it is thoroughly transparent to application users as only developers are aware of the error console window and the meaning of those messages. A JavaScript application with no global error handler fails to achieve G1 since error information becomes lost. Consequently, a global error handler to deal with uncaught exception is necessary for error reporting.

We should implement a global error handler by using window.onerror event handler to catch all uncaught exceptions thrown by a client-side web application. Since uncaught exception event (window.onerror) is invoked in a context different from the context where the error has occurred, we cannot process the error (i.e., handle every error in a particular way) but report it to developers or notify users of the error condition and reload the web application (automatically or manually by giving a recommendation to users).

5.3.2 Mechanics

The mechanics respectively used in the server-side and the client-side is as follows:

a. For server-side Node.js application

- Implement domain.on('err', …) or a listener for process.on('uncaughtException', …)
- When uncaught exception event is fired:

o Stop listening new requests on that process
o Wait for some time for other requests to finish
o Exit the process
o Create a new process to serve later requests

b. For client-side JavaScript application

- • Implement window.onerror event handler to catch all uncaught exceptions.
- • Log exception information to the server
- • Implement the function of reloading the web application (optional).

5.3.3 Example

5.3.3.1 For Server-Side Node.js Application

To keep an application always online we can refactor the application to handle any uncaught exception as shown below [14]. Note that the implementation of uncaughtException event handler of a process is similar.

```
var cluster = require('cluster');
var cpus = require('os').cpus().length;
if (cluster.isMaster) {
   for (var i = 0; i < cpus; i++) {
      cluster.fork(); // Start a process per core
   }

   cluster.on('disconnect', function (worker) {
      cluster.fork();
   });

} else {
   var server = http.createServer(function (request, response) {
      var d = domain.create();
      d.on('error', function (err) {
         try {
            /* setTimeout for calling process.exit(1) after a waiting time
*/
            server.close(); //stop taking new requests
            /*log the error and notify administrator*/
         } finally {
            cluster.worker.disconnect();
         }
      });

      d.add(request);
      d.add(response);

      d.run(function () {
         handleRequest(request, response);
      });
   });
   server.listen(PORT);
}
```

5.3.3.2 For Client-Side JavaScript Application

As mentioned in Section 5.3.1, all error information should be accessible by developers. This can be achieved by posting the information to the server.

```
window.onerror = function myErrorHandler(errorMsg, url, lineNumber,
columnNumber, error) {
   if(!!error) {
      JL().fatalException(error.message, error);
   } else {
      JL().error("Fatal Exception: " + errorMsg + "\n at " + url + ", line:
" + lineNumber);
   }
   //Other processing
   return false;
}
```

5.4 Replace Symbol with Error Object

You have a throw statement with an argument which is a symbol (string, number, and so on), replace this symbol with an instance of Error type or its subtype as shown below:

```
throw "Error message";
          ↓
throw new Error("Error message");
```

Or, in an asynchronous callback, replace first argument which is a symbol with an instance of Error or its subtype as shown below:

```
callback("Error message");
          ↓
callback(new Error("Error message"));
```

5.4.1 Motivation

Instead of throwing an error symbol, we should throw an instance of built-in Error type or an instance of custom error type that inherits from Error. Developers now can determine the type of exception and handle it in a proper way by making an instanceof checking. This would make error handling of the program more readable and robust.

The fundamental benefit of using Error objects is that they automatically keep track of where they were built and originated, which provides valuable information for debugging. Developers can further define a custom error type that inherits from Error type to add more context related information to error objects.

A JavaScript application with *throwing symbol instead of error object* fails to achieve robustness level G1, because doing this way can be seen as not providing enough information for debugging and error handling.

5.4.2 Mechanics

- Create an instance of Error type or any subtype of it.
- Pass the symbol from throw statement or callback to the constructor of the error in previous step.
- Replace the symbol in throw statement or callback with the newly created error.
- Run tests.

5.4.3 Example

```
function A() {
   throw "Error message";
}

try {
   A();
} catch (ex) {
   if (ex === "Error message") { //Compare string
      //Handle the error
   }
}
```

It can clearly be seen that, in the case of throwing symbol, the handler may need to compare strings or numbers to handle the error. However, doing so is not only error-prone but also reducing code readability. A better way to deal with it is as follows:

```
function A() {
   throw new CustomErrorType("Error message");
}

try {
   A();
} catch (ex) {
   if (ex instanceof CustomErrorType) {
      //Handle the error
   }
}
```

5.5 Rethrow Error or Pass Error Object to Callback

You have a dummy handler which does nothing more than logging an error, *rethrow the error or pass it to callback*. Specifically, rethrow error in catch block in the case of synchronous operations as shown below:

```
try {
   //code that may throw error
} catch (e) {
   // Code for logging error
}
↓
try {
   //code that may throw error
} catch (e) {
   // Code for logging error
   throw e; // Rethrow error after logging
}
```

or, in the case of asynchronous operations in Node.js, deliver the error to a higher-level callback then return or, for some errors that cannot be handled, rethrow them to make uncaught exceptions.

```
doAsyncOperation(inputParam, function (err, data) {
   if (err) {
      // Code for logging error
   }
   ...
});
                                    ↓
doAsyncOperation(inputParam, function (err, data) {
   if (err) {
      // Code for logging error
      return callback(err);
   }
   ...
});
```

5.5.1 Motivation

We can eliminate the risk of unexpected behaviors resulting from slow-failing by *rethrowing the error and letting its callers handle it or* by *letting the error become uncaught exception*. This is similar to delivering the error to a higher-level callback. The robustness level of program is upgraded from G0 to G1 in so doing.

Dummy handlers in finally block are acceptable because throwing an exception out of the finally block may overwrite any error raised in a catch block [3].

5.5.2 Mechanics

- For synchronous operations: if the error cannot be handled in the catch block, rethrow it to make the program fail fast or let an error handler at a higher level in the call chain to handle it.
- For asynchronous operations (with error-first callback):
- If the callback cannot handle the error, it can deliver the error to higher-level callbacks then exit the current function (callback); or
- For some critical errors that cannot be handled properly, throw them so that the program will get uncaught exception and make the current process restart as described in Section 5.3.

5.5.3 Example

```
try {
   data = jQuery.parseJSON(data);
} catch (e) {
   logger.error("Can not parse album data", e);
}
```

In the example code shown above, instead of logging the error and continuing the execution, the exception should be rethrown. The throw statement will propagate this exception to a higher level in the call chain and terminate current execution. This will prevent anything unexpected from happening if the execution is continued instead after the error.

```
try {
   data = jQuery.parseJSON(data);
} catch (e) {
   logger.error("Can not parse album data", e);
   throw e;
}
```

In case that a callback is provided, the function should exit immediately after invoking a callback with error object.

```
try {
   data = jQuery.parseJSON(data);
} catch (e) {
   logger.error("Can not parse album data", e);
   return callback(e);
}
```

The second example is taken from Strider CD project [15], which shows the case of a critical error that cannot be handled but need to be rethrown in order to restart the process. Specifically, if the password entered by a user is less than 6 characters long, an error message will be delivered to the user to indicate a failed validation. If the execution of saving to database failed after all validations have passed, the best thing to do is to throw an error to restart the process since this is a critical low level error and we don't know what exactly the reason of this failure is.

```
exports.account_password = function(req, res) {
  var password = req.body.password;

  if (password.length < 6) {
    res.statusCode = 400;
    res.end(JSON.stringify({status:"error",
       errors:[{"message":"password must be at least 6 characters
long"}]}));
    return;
```

```
  }
  req.user.password = password;
  req.user.save(function(err, user_obj) {
    if (err) throw err;
    email.notify_password_change(req.user);
    res.end(JSON.stringify({status:"ok", errors:[]}));
  });
}
```

6 Analysis

As stated previously, this research work is extended from our previous work on Java [3]. In this study, the focus is on the investigation of exception handling code smells and their refactorings in JavaScript. Being a dynamic programming language, JavaScript has many features which are different from Java's. For comparison purpose, the five exception handling code smells addressed in this work are classified into three groups:

i. smells derived from those in Java, there are three of them;
ii. new code smell that may also happen in Java, there is one;
iii. smell that is unique to JavaScript, there is one.

Code smells in group (i) include *ignoring exception, no global uncaught exception handler* and *Dummy handler*. The proposed refactoring methods for their removal are different from those for Java, because the mechanisms for exception handling are different in the two languages. The detailed information is given below:

- *ignoring exception*: a variant of ignoring checked exception defined in [3] is ignoring error-first parameter checking in callback; it is equivalent to an Empty Catch Block in that all they do is ignore the exception.
- *no global uncaught exception handler*: this code smell is similar to "unprotected main" in [3]. In Java, besides using a big outer try/catch statement at every entry point of each thread, we can also set an exception handler for each thread by using Thread. setUncaughtExceptionHandler or set a default uncaught exception handler for all threads by using Thread. setDefaultUncaughtExceptionHandler. From this point of view, they are equivalent to the methods used in our proposed refactoring: implementing handler for error event of domain, uncaughtException event of process in Node.js or, implementing window.onerror at the client-side web application.
- *dummy handler*: another refactoring method applied in Node.js is passing an error object to callback (in addition to rethrowing in synchronous operation). A return statement is placed right after the callback invocation to prevent slow-failing.

Group (ii) contains only one smell, the *try/catch outside asynchronous callback*. There is a correspondence between using *try/catch outside asynchronous callback* and misusing try/catch to catch exceptions from other threads in Java. In the case of using *try/catch outside asynchronous callback*, it is considered as a programming error since developers may think that the callback function will be executed in the same synchronous context as the try statement. Since a Java application contains multiple threads, misusing try/catch to catch exception from other threads may result in uncaught exceptions. As the following example illustrates, if an exception is thrown inside run method, it will not be caught by the enclosing try/catch statement.

```
try {
  Thread cpuIntensiveThread = new Thread() {
    @Override
    public void run() {
      // Doing intensive work here
      throw new RuntimeException("uncaught exception [ue]");
    }
  };
  cpuIntensiveThread.start();
} catch (Exception ex) {
  Log.e("MainActivity", "This will not catch [ue]", ex);
  throw ex;
}
```

Group (iii) contains one code smell, the *throwing symbol instead of error object*, which is unique to JavaScript. While Java supports the representation of exceptions as data objects only [4], JavaScript provides two ways to throw an exception, i.e., by a symbol or by an object. For this reason, *throwing symbol instead of error object* only happens in JavaScript.

7 Conclusion and Future Work

Our research investigated exception handling code smells that may happen in JavaScript as well as the impact they have to a JavaScript program. Refactorings corresponding to the identified smells are proposed for their removal. The difference and similarity between Java and JavaScript in terms of exception handling code smells is also addressed. The results of the work should be able to help developers of Java or JavaScript applications avoid or discover bad smells in designing or maintaining exception handling code.

For future work, we plan to continue identifying more possible smells that can happen in exception handling code of a JavaScript application. An automated tool for detecting and refactoring the identified EH smells will be implemented to reduce detection cost and effort. In addition, an empirical study will be conducted to detect exception handling code smells in some commonly-used open source JavaScript projects.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of*

Existing Code, Addison-Wesley Professional, 1999.
[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.
[3] C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh and I.-L. Wu, Exception Handling Refactorings: Directed by goals and Driven by Bug Fixing, *Journal of Systems and Software*, Vol. 82, No. 2, pp. 333-345, February, 2009.
[4] F. Garcia, C. M. Rubira, A. Romanovsky and J. Xu, A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-oriented Software, *Journal of Systems and Software*, Vol. 59, No. 2, pp. 197-222, November, 2001.
[5] R. Miller and A. Tripathi, Issues with Exception Handling in Object-oriented Systems, *ECOOP '97–Object-Oriented Programming: 11th European Conference*, Jyväskylä, Finland, 1997, pp. 85-103.
[6] Ecma International, *ECMAScript Language Specification*, Standard ECMA-262, 5.1 Edition, Ecma International, June, 2011.
[7] P. A. Buhr and W. Y. Russell Mok, Advanced Exception Handling Mechanisms, *IEEE Transactions on Software Engineering*, Vol. 26, No. 9, pp. 820-836, September, 2000.
[8] MDN web docs, *Concurrency Model and Event Loop*, 2017, https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
[9] M. E. Daggett, Expert JavaScript, *JavaScript IRL*, Apress, 2013, pp. 107-130.
[10] C. L. My, *Identification and Refactoring of Exception Handling Code Smells in Node.js*, Master Thesis, National Taipei University of Technology, Taipei, Taiwan, 2015.
[11] Node.js, *Node.js Community Wiki*, https://github.com/joyent/node/wiki
[12] J. Shore, Fail Fast, *IEEE Software*, Vol. 21, No. 5, pp. 21-25, September-October, 2004.
[13] *StrongLoop, Building Robust Node Applications: Error Handling*, https://strongloop.com/strongblog/robust-node-applications-error-handling/
[14] Node.js, *Domain Node.js v8.7.0 Documentation*, 2015, https://nodejs.org/api/domain.html
[15] GitHub, *Strider-CD/strider*, https://github.com/Strider-CD/strider
[16] *K. T. Ho, Identification and Refactoring of Exception Handling Code Smells in JavaScript Based on jQuery and AngularJS Libraries*, Master Thesis, National Taipei University of Technology, Taipei, Taiwan, 2015.

Biographies

Chin-Yun Hsieh is Professor of Department of Computer Science and Information Engineering at National Taipei University of Technology, Taiwan. He received his PhD degree in Computer Science from the University of Oklahoma. His research interests include pattern languages, software testing, and object-oriented design.

Canh Le My is a lecturer in the Department of Information Technology at Hue University of Sciences, Vietnam. He earned his MSc in Electrical Engineering & Computer Science from National Taipei University of Technology, Taiwan. His research fields are software engineering and information systems.

Kim Thoa Ho is a lecturer in the Informatic Department, Hue University of Education, Vietnam. She obtained her MSc in Electrical Engineering & Computer Science from National Taipei University of Technology, Taipei, Taiwan. Her research interests include software engineering and information systems.

Yu Chin Cheng, PhD, is a professor of Department of Computer Science and Information Engineering at Taipei Tech, Taiwan. He served as Director of Board of Governors, Software Engineering Association of Taiwan (SEAT) from 2011/7 to 2014/7. His main research interests are software engineering, pattern languages, and pattern analysis.