3.2.3 Error Handling

3.2.3.1 Purpose

A robust error handling mechanism is crucial for providing good user experience and ensuring the reliability of the system. This section outlines the design for implementing a centralized error handling mechanism for users.

3.2.3.2 Best Practices

1. Use Error Objects for Clearer Messages

Use error objects with clear messages and custom names to make your code easier to understand.

2. Stop the Program with Throw Statements

The throw statement stops your program and shows the nearest catch block, helping you find and fix errors more easily.

3. Track Errors Using the Call Stack

The call stack helps track down where an error started by showing the sequence of function calls that led to it.

4. Name Your Functions for Better Error Messages

Naming your functions clearly makes the error messages easier to read when debugging.

5. Handle Asynchronous Code with Promises and Async/Await

For asynchronous code, use promises or async/await to catch errors properly with .catch() or within a try/catch block.

Source: <https://stackify.com/node-js-error-handling/>

3.2.3.3 Components

1. Error Repository: A centralized storage for storing error messages and codes.

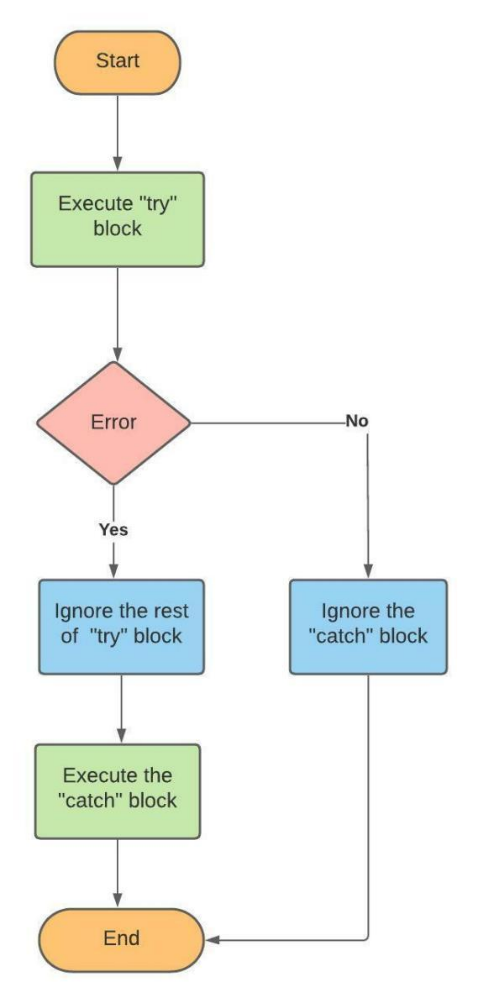
2. Error Handler: Responsible for retrieving error information from the repository, parsing it, and displaying it to the user.

3. User Interface: The interface through which users interact with the system, displaying error messages and allowing them to recover from errors.

3.2.3.4 Error Handling Flow

1. Error Occurrence: An error occurs within the system (e.g., database connection failure, invalid input).

2. Error Detection: The system detects the error and triggers the error handling mechanism.

 Source: <https://linuxhint.com/error-handling-in-javascript/>

3. Error Retrieval: The error handler retrieves error information from the repository.

4. Error Parsing: The error handler passes the retrieved error information to extract relevant details.

5. User Feedback: The user interface displays the parsed error message to the user.

6. Error Recovery: The system provides options for the user to recover from the error (e.g., retry, cancel).

3.2.3.5 Error Types

1. Operational Errors: Errors that occur within the system's internal workings (e.g., database connection failure).

2. Input Errors: Errors caused by invalid or malformed input data.

3. Functional Errors: Errors in the actual application.

3.2.3.6 Error Handling Strategies

1. Display Error Message: Display a clear and concise error message to the user.

2. Provide Recovery Options: Offer options for the user to recover from the error (e.g., retry, cancel).

3. Log Errors: Log errors for debugging and monitoring purposes.

3.2.3.7 Example Use Case & Pseudocode

function isValidInput(userInput):

# Define a regular expression pattern to match an integer

define integerReg as a pattern that matches only digits (0-9) and optionally a negative sign at the start

# Check if userInput matches the integer pattern

if userInput matches integerReg:

return true # Input is an integer with no special characters

else: return false # Input is not a valid integer # Check that user input falls within a valid numeric range

convert userInput.input to an integer

if integer user input is less than -10000 or greater than 10000:

return false

# If all validations pass, input is valid

return true

function generateUserInput(userInput) {

try {

# Check if user input is valid

if (!isValidInput(userInput)) {

throw "Invalid input"

}

# Proceed if the input is valid

print "User input is valid"

} catch (error) {

# Handle invalid input by printing a message and prompting for new input

Print Error message

Prompt for new input }

}