**JS Code Smell Detector**

## **Overview**

The **Code Smell Detector** is a JavaScript-based static code analysis tool designed to identify and flag potential code smells in a given codebase. The project employs various linting techniques to detect issues related to code quality, readability, security vulnerabilities, and potential performance problems.

The tool aims to help developers write cleaner, more maintainable code by automatically pointing out common mistakes and inefficiencies in JavaScript and TypeScript code.

## **Introduction**

The **Code Smell Detector** is a tool designed to help developers identify and resolve potential problems in their JavaScript and TypeScript codebases. Code smells are patterns in the code that may indicate deeper issues or areas for improvement. While not necessarily bugs, these smells can lead to maintainability challenges, security vulnerabilities, or performance issues if left unaddressed.

This project automates the detection of code smells, providing a fast and efficient way to identify problematic areas in your code. It aims to assist development teams in maintaining high-quality code standards, reducing technical debt, and improving overall project health.

## **Rationale**

In modern software development, maintaining a clean, readable, and efficient codebase is crucial for scalability and long-term success. Code smells often arise during rapid development cycles, where trade-offs are made between speed and quality. Over time, these smells can accumulate, leading to:

* **Increased Maintenance Costs**: Difficult-to-read or overly complex code can slow down development and debugging.
* **Security Risks**: Poor practices like hard-coded sensitive information can expose the project to vulnerabilities.
* **Performance Issues**: Inefficient patterns can lead to unnecessary computational overhead.
* **Team Collaboration Challenges**: Code that is hard to understand or inconsistently written can create barriers for team members.

By providing an automated way to detect these smells, this tool enables teams to:

* Proactively address potential issues before they escalate.
* Improve code readability and maintainability.
* Ensure better security and performance practices.
* Foster a culture of quality and consistency within the development team.

The **Code Smell Detector** is not only a diagnostic tool but also an educational resource, offering suggestions for resolving identified smells and encouraging developers to adopt best practices.

## **Key Features**

* **Comprehensive Detection**: Identifies 14 distinct code smells, including security vulnerabilities, readability issues, and maintainability concerns.
* **Actionable Fixes**: Provides detailed descriptions and recommendations to address identified smells effectively.
* **Customizable Configuration**: Allows users to add or modify code smells to tailor the tool to specific project needs.
* **Interactive CLI**: Guides users through code analysis with an intuitive command-line interface.
* **Detailed Reporting**: Generates both summary and detailed reports, providing insights into code quality.
* **Extensible Architecture**: Easily extend the tool by adding new detection rules or integrating it with existing workflows.
* **CI/CD Integration**: Seamlessly integrates with CI/CD pipelines for continuous quality assurance.

## **Detailed Description of Code Smells**

### **1. Lengthy Lines**

Modern coding standards emphasize readability, and excessively long lines are a common barrier. Lines with more than 80-100 characters are hard to read, making code reviews cumbersome and reducing overall maintainability. Style guides from prominent organizations like Airbnb and Node.js suggest keeping lines concise, excluding unavoidable cases like URLs or regular expressions (IEEE, 2016). By breaking long lines into smaller segments, developers can create clearer and more navigable code.

**Reference**: IEEE (2016). [Coding Standards](https://ieeexplore.ieee.org/abstract/document/7884630).

### **2. Long Parameter List**

Functions with long parameter lists are often a sign of overly complex logic. Ideal functions should have one or two parameters, or even none, to simplify their use and ensure readability. Breaking down functions or using configuration objects allows better modularity and backward compatibility (IEEE, 2016). This approach not only reduces confusion but also prepares the code for future changes without breaking existing functionality.

**Reference**: IEEE (2016). [Parameter Guidelines](https://ieeexplore.ieee.org/abstract/document/7884630).

### **3. Nested Callbacks (Callback Hell)**

JavaScript's asynchronous nature often leads to deeply nested callback functions when tasks depend on one another. Known as "callback hell," this pattern makes debugging and comprehension difficult. Modern techniques like Promises and async/await provide more structured and readable alternatives (IEEE, 2016). These tools reduce complexity and help maintain a cleaner codebase.

**Reference**: IEEE (2016). [Callback Management](https://ieeexplore.ieee.org/abstract/document/7884630).

### **4. Variable Re-Assign**

JavaScript’s flexibility allows variables to be reassigned different types at runtime, but this dynamic nature can lead to confusion and bugs. Clear and unique variable names should be used to improve code readability and avoid unintended errors (IEEE, 2016). This simple practice ensures developers can easily understand the purpose and scope of each variable.

**Reference**: IEEE (2016). [Variable Reassignment](https://ieeexplore.ieee.org/abstract/document/7884630).

### **5. Duplicate Code**

Duplicating logic across a codebase increases the risk of inconsistencies and higher maintenance costs. Repeating similar logic is especially problematic when updates are required across multiple locations. Consolidating these patterns into reusable functions simplifies changes and minimizes errors (Bitsrc, 2017). Developers can thus avoid redundancy and improve overall efficiency.

**Reference**: Bitsrc (2017). [Code Reuse](https://blog.bitsrc.io/your-javascript-smells-3574121bcfbe).

### **6. Unused Dependency**

Unused dependencies often linger in package.json files after being removed from the code. These bloated files increase load times and risk outdated or insecure packages remaining in the project. Properly categorizing runtime and development dependencies and cleaning up unused ones can streamline the build process and improve security (IEEE, 2021).

**Reference**: IEEE (2021). [Dependency Management](https://ieeexplore.ieee.org/abstract/document/9519532).

### **7. Empty Catch Block**

Handling errors gracefully is essential, and leaving catch blocks empty is a missed opportunity. Empty blocks obscure potential issues and may introduce security risks, as undetected errors can leave the application vulnerable (Arxiv, 2023). Effective logging or alternative error-handling mechanisms ensure stability and security.

**Reference**: Arxiv (2023). [Error Handling](https://arxiv.org/pdf/2411.19358).

### **8. Long Method/Function**

Functions that exceed a certain number of lines become difficult to read and maintain. This often indicates poor decomposition, leading to higher risks of bugs and vulnerabilities. Refactoring long functions into smaller, specific tasks improves modularity and simplifies testing (Arxiv, 2023). This practice also aligns with security guidelines like CWE-1080 to prevent errors.

**Reference**: Arxiv (2023). [Function Complexity](https://arxiv.org/pdf/2411.19358).

### **9. Missing Default in Case Statement**

Switch statements without a default case fail to handle all possible inputs, leaving room for unexpected behavior. This oversight can lead to exploitable vulnerabilities, such as revealing sensitive information through error traces (Arxiv, 2023). Adding a default case ensures robust and secure handling of all inputs.

**Reference**: Arxiv (2023). [Conditional Logic](https://arxiv.org/abs/2411.19358).

### **10. Large Objects**

Objects with too many properties resemble "god classes" in traditional object-oriented programming, which are difficult to test and maintain. Breaking these objects into smaller, domain-specific ones reduces complexity and minimizes potential vulnerabilities (Arxiv, 2023). This modular approach promotes better software design and maintainability.

**Reference**: Arxiv (2023). [Object Design](https://arxiv.org/abs/2411.19358).

### **11. Dead/Unused Code**

Dead code clutters the codebase and increases the attack surface. While seemingly harmless, it may be exploited through runtime execution, introducing security risks. Regularly removing or refactoring such code ensures clarity and reduces vulnerability (Arxiv, 2023).

**Reference**: Arxiv (2023). [Code Cleanup](https://arxiv.org/abs/2411.19358).

### **12. Hard-Coded Sensitive Information**

Embedding sensitive information like API keys or passwords directly in the source code is a major security flaw. Such practices make it easier for attackers to access sensitive data. Best practices recommend using environment variables or external configuration files to store these values securely (Arxiv, 2023).

**Reference**: Arxiv (2023). [Secure Data Storage](https://arxiv.org/pdf/2411.19358).

### **13. Active Debugging Code**

Leaving debugging statements like console.log in production code exposes system details to attackers. These active debug features must be removed to prevent unauthorized access to sensitive information (Arxiv, 2023). Developers should conditionally disable debugging in production environments.

**Reference**: Arxiv (2023). [Debugging Practices](https://arxiv.org/pdf/2411.19358).

### **14. Insecure File Handling**

Improper file validation and sanitization during uploads can allow attackers to execute malicious code. Implementing strict validation mechanisms reduces these risks and ensures only safe files are processed by the application (Arxiv, 2023).

**Reference**: Arxiv (2023). [File Security](https://arxiv.org/pdf/2411.19358).

## **Tools and Techniques Used**

### **Core Tools**

1. **JavaScript**: Primary language used for implementing the code smell detection logic.
2. **Node.js**: Provides the runtime environment for executing the script.
3. **Regular Expressions (Regex)**: Utilized extensively to match patterns in the source code for detecting issues like unused variables, long lines, and duplicate code. Examples include:
   * /(console\.log|console\.debug|console\.error|alert\()/g for detecting debugging code.
   * /function\s+\w\*\s\*\(([^)]\*)\)/ for finding functions with long parameter lists.
   * /catch\s\*\((\w+|\w+\s\*:\s\*\w+)\)\s\*{([\s\S]\*?)}/ for identifying empty catch blocks.
4. **Inquirer.js**: Used to build an interactive command-line interface (CLI) for guiding users through different options and functionalities.
5. **Babel Parser**: Parses JavaScript code into an Abstract Syntax Tree (AST) for analyzing and detecting complex issues like nested callbacks and duplicate code.

### **Optional Tools**

1. **ESLint**: Provides additional linting capabilities and can be integrated for further static code analysis.
2. **Third-party Modules**:
   * fs: Used for file system operations, such as reading and analyzing files.
   * path: Facilitates file and directory path manipulations.

### **Development Environment**

1. **Version Control**: GitHub for collaborative development and version tracking.
2. **NPM**: Manages dependencies for the project.
3. **VSCode/IDE**: Recommended for efficient coding and debugging.

## **Installation**

### **Prerequisites**

* **Node.js** (version 12.x or higher)
* **NPM** (Node Package Manager)

### **Steps to Install**

1. Clone the repository:

| git clone https://github.com/Mustafa-tariq23/JS-codeSmells-detector-tool.git |
| --- |

1. Navigate to the project directory:

| cd code-smell-detector |
| --- |

1. Install the required dependencies:

| npm install |
| --- |

## **Usage**

### **Running the Code Smell Detector**

1. After installation, you can run the detector via the command line:

| node detector.js path/to/your/code/files |
| --- |

Here, path/to/your/code/files is the directory where your code files are located. The script will analyze all JavaScript/TypeScript files in that directory and report any detected code smells.

### **Configuration**

The tool can be customized based on the project needs. For example, you can modify the threshold for what constitutes a "large object," or disable certain checks. The configuration is defined in the config.js file, where you can specify:

* **Thresholds** for large objects.
* Enable/Disable specific checks.
* Add custom checks using regular expressions.

### **Sample Output**

The detector will output results in the following format:

| Code Smell Detected in file 'path/to/file.js': - Line 15: Missing default case in switch statement. Consider adding a default case to handle unexpected input. - Line 38: Object has too many properties (12 properties). Consider breaking it into smaller objects. - Line 80: Hard-coded API key detected. Consider removing it and using environment variables. |
| --- |

## **Detailed Description of Code Smells Detected**

### **1. Missing Default in Case Statement**

* **Problem**: A switch statement that lacks a default case can lead to unhandled scenarios. This omission can cause the program to behave unpredictably when unexpected input is encountered.
* **Fix**: Add a default case to ensure all potential inputs are managed and prevent unexpected behavior.

**Example:**

| switch (value) {  case 'A':  // Do something  break;  case 'B':  // Do something  break;  // Missing default case } |
| --- |

### **2. Large Objects**

* **Problem**: Objects with too many properties (e.g., over 10 properties) increase complexity and reduce maintainability. Large objects are harder to test and refactor, and they often indicate poor design.
* **Fix**: Break large objects into smaller, focused objects or classes to reduce complexity and enhance readability.

**Example:**

| const userProfile = {  name: 'John',  age: 30,  address: '123 Street',  phone: '555-5555',  email: 'john@example.com',  preferences: {...}, // and many more properties }; |
| --- |

### **3. Dead/Unused Code**

* **Problem**: Dead or unused code includes unreachable code, unused variables, or functions. Such code clutters the codebase and increases the potential for errors or vulnerabilities.
* **Fix**: Regularly remove or refactor unused variables, functions, and imports.

**Example:**

| const unusedFunction = () => {  console.log('This is unused'); }; |
| --- |

### **4. Hard-Coded Sensitive Information**

* **Problem**: Hard-coding sensitive data, such as API keys, passwords, or private tokens, poses a significant security risk. Attackers can easily extract this data.
* **Fix**: Use environment variables or secure storage solutions, such as AWS Secrets Manager or HashiCorp Vault, to handle sensitive information.

**Example**:

| const apiKey = '123456789SECRET'; // Avoid this practice |
| --- |

### 

### **5. Active Debugging Code**

* **Problem**: Leaving debugging code such as console.log or alert() in production can expose internal details, potentially revealing sensitive information or weakening performance.
* **Fix**: Remove or disable all debugging statements before deploying to production.

**Example**:  
 console.log('Debug info'); // Remove this in production

### **6. Insecure File Handling**

* **Problem**: File upload methods, such as multer or express-fileupload, that lack proper validation or sanitization expose systems to malicious file uploads and potential security vulnerabilities.
* **Fix**: Implement strict file validation and sanitization mechanisms to secure file handling.

| Example:  app.post('/upload', upload.single('file'), (req, res) => {  // Validate file type and size  if (!isValidFile(req.file)) {  return res.status(400).send('Invalid file');  } }); |
| --- |

### **7. Lengthy Lines**

* **Problem**: Lines exceeding 80-100 characters reduce readability and make code reviews more challenging. Exceptions include URLs and complex regular expressions.
* **Fix**: Break long lines into multiple shorter ones or refactor code for better readability.

**Example**:

| const example = 'This is a very long line of code that exceeds the recommended limit of 80 characters'; |
| --- |

### **8. Long Parameter List**

* **Problem**: Functions with more than 2-3 parameters are harder to understand and maintain, often indicating overly complex logic.
* **Fix**: Use a single configuration object or simplify the function by breaking it into smaller units.

**Example**:

| function configure(a, b, c, d, e) { // Too many parameters  // Simplify } |
| --- |

### **9. Nested Callbacks (Callback Hell)**

* **Problem**: Excessive nesting of callbacks makes code difficult to read and debug. Known as "callback hell," this pattern often arises in asynchronous programming.
* **Fix**: Use Promises or async/await to manage asynchronous code more cleanly.

**Example**:

| asyncFunction1(() => {  asyncFunction2(() => {  asyncFunction3(() => {  // Deep nesting  });  }); }); |
| --- |

### **10. Variable Re-Assign**

* **Problem**: Reassigning variables with different types or values leads to confusion and potential bugs.
* **Fix**: Avoid reassigning variables; instead, use new variables for different purposes.

**Example**:

| let example = 42; example = 'Now a string'; // Avoid this |
| --- |

### **11. Duplicate Code**

* **Problem**: Copying and pasting similar logic across multiple places increases maintenance effort and the likelihood of bugs.
* **Fix**: Refactor repetitive code into reusable functions or components.

**Example**:

| const calculateSum = (a, b) => a + b; // Reuse this instead of repeating logic |
| --- |

### **12. Unused Dependency**

* **Problem**: Dependencies listed in package.json but not used in the code clutter the project and can introduce vulnerabilities.
* **Fix**: Regularly audit and remove unused dependencies.

**Example**:

| "dependencies": {  "unused-library": "^1.0.0" // Remove this if not used } |
| --- |

### **13. Empty Catch Block**

* **Problem**: Catch blocks without proper handling obscure errors and make debugging difficult.
* **Fix**: Include meaningful error handling or logging.

**Example**:

| try {  riskyOperation(); } catch (error) {  // Log or handle the error appropriately } |
| --- |

### **14. Long Method/Function**

* **Problem**: Long functions with many lines of code are harder to understand and debug.
* **Fix**: Break long functions into smaller, more focused functions.

**Example**:

| function longFunction() {  // Too many lines of code } |
| --- |

## 

## **How It Works**

### **Detection Process**

1. **Input**: The tool reads the JavaScript or TypeScript files from the specified directory.
2. **Parsing**: The files are parsed line by line using regular expressions. Each line is analyzed for specific patterns representing code smells.
3. **Analysis**: The tool uses pre-defined regular expressions to detect issues like dead code, missing default cases in switch statements, large objects, and more.
4. **Reporting**: For each detected issue, the tool outputs the line number, the type of issue, and a description of how to fix it.

### **Code Smell Patterns**

The core of the tool relies on a set of regular expressions designed to match specific code patterns that indicate potential issues. These patterns are periodically updated to keep up with evolving best practices and new issues.

## **Extending the Project**

### **Adding New Code Smells**

To add a new code smell detection pattern:

1. Create a new regular expression to match the pattern of the code smell.
2. Add the regular expression to the detector.js file in the check functions.
3. Define a description for the issue, and decide how to fix it in the fix section.

### **Example**

Here’s an example of how you can add a check for unused imports:

| {  name: "Unused Import",  check: (lines, filePath, projectRoot) => {  const issues = [];  const fullContent = lines.join("\n");  const importRegex = /import\s+([a-zA-Z\_][a-zA-Z0-9\_]\*)\s+from\s+/g;  let match;  while ((match = importRegex.exec(fullContent)) !== null) {  const lineNumber = fullContent.slice(0, match.index).split("\n").length;  if (!new RegExp(`\\b${match[1]}\\b`).test(fullContent)) {  issues.push({  line: lineNumber,  description: `Imported module ${match[1]} is not used. Consider removing it.`,  });  }  }  return issues;  },  fix: "Remove unused imports to reduce code complexity and improve performance.", } |
| --- |

### 

### **Extending Regular Expressions**

If you need to support additional JavaScript syntax or more complex patterns, you can modify the regular expressions used in each check. However, keep in mind that adding more complex patterns may affect the performance of the tool.

## **Configuration**

The configuration is defined in the config.js file. Here, you can customize various settings:

* **Thresholds for large objects**
* **Enable/Disable specific checks**
* **Add custom patterns or ignore specific files**

## **References**

1. **MDN Web Docs**: A comprehensive resource for JavaScript syntax and best practices. The documentation includes examples, explanations, and references for various JavaScript features.
   * [Visit MDN Web Docs](https://developer.mozilla.org/en-US/)
2. **ESLint**: A powerful linting tool to enforce coding standards and detect potential issues in JavaScript code. Integration with this project helps extend static code analysis for detecting smells and errors.
   * [Visit ESLint Documentation](https://eslint.org/)
3. **OWASP Cheat Sheet Series**: A collection of security best practices covering various areas of web development. This resource is particularly valuable for understanding secure file handling and managing sensitive information.
   * Visit OWASP Cheat Sheet Series
4. **Node.js Documentation**: Detailed documentation about the Node.js runtime, including file system operations, asynchronous programming, and modules.
   * Visit Node.js Documentation

## **License**

This project is licensed under the **MIT License**. See the LICENSE file for more details.

## **Contributing**

If you’d like to contribute to the project, please follow these guidelines:

1. Fork the repository.
2. Create a new branch for your changes.
3. Make your changes and write tests if necessary.
4. Run the tests to ensure everything works as expected.
5. Submit a pull request.

We welcome contributions to improve the tool!

## **Contact Information**

For questions, suggestions, or feedback, feel free to open an issue on GitHub or contact the project maintainer at [**SP22-BSE-119@cuilahore.edu.pk**](mailto:SP22-BSE-119@cuilahore.edu.pk) **or at** [**Mustafacui119@gmail.com**](mailto:mustafacui119@gmail.com).