ELASTIC: Evaluation of Large Language Models and Abstract Syntax Tree Parsing for Identification of Code Smell Detection

Northeastern University

Yash Mahesh Burshe

Advised by: Joydeep Mitra, Assistant Teaching Professor, Khoury College of Computer Science

Introduction

Static Analysis
The process of analyzing source code directly at compile time to perform techniques that can extract certain metrics or facts that can be used to modify, understand and evaluate the source code.

Abstract Syntax Tree
A data structure that is used to represent the structure of any piece of source code. Parsing the source code, we can identify the relationship of syntax and arrange them in a tree structure to operate on it easier.

Large Language Models (LLMs)
A transformer-based language model that can perform reasoning tasks based on training on a large amount of input data. Capable of using the trained data to provide more context in relation to the nature of the source code.

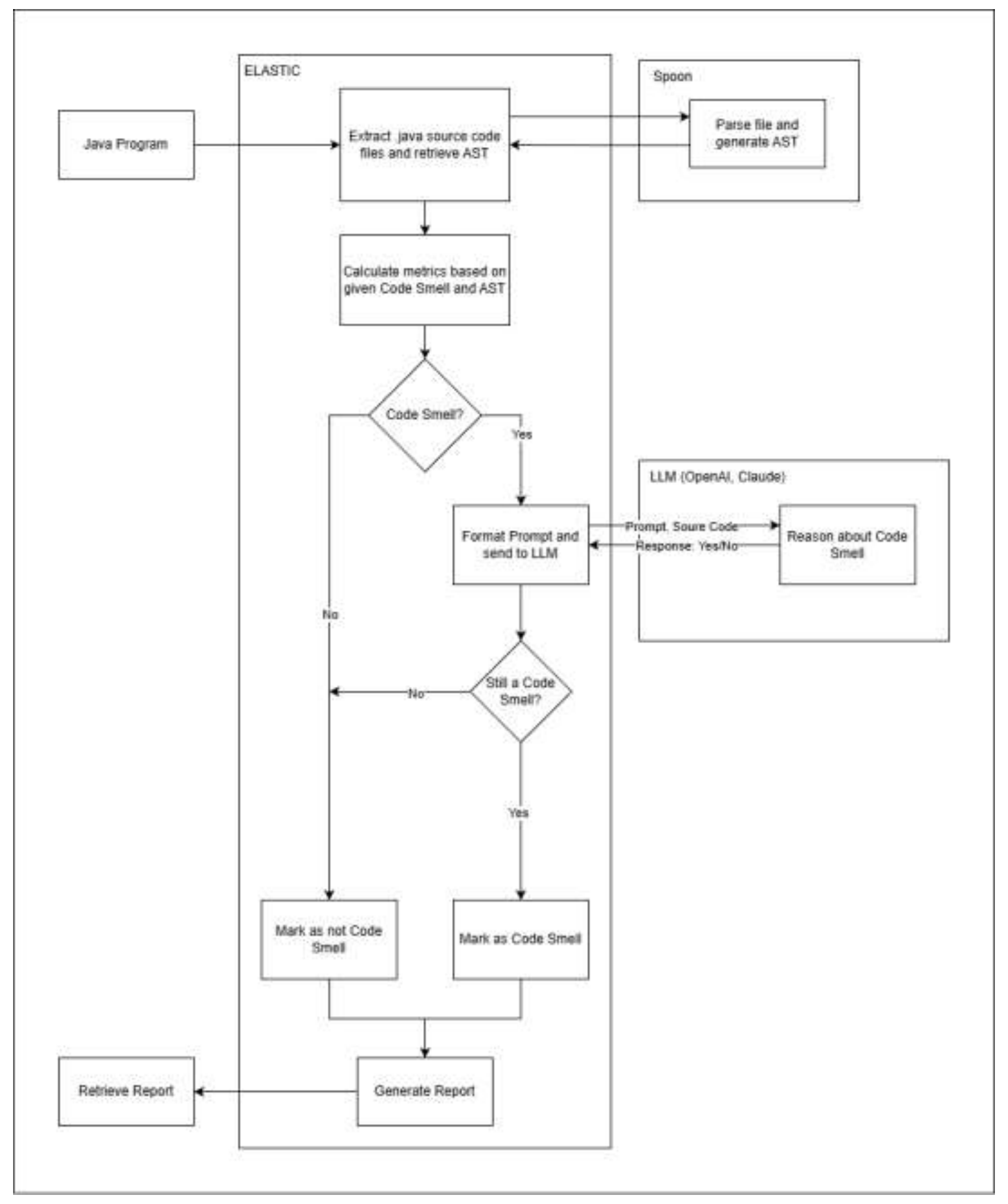
Code Smell
Coined by Kent Beck in Martin Fowler's book 'Refactoring', it is a flaw in the design of a part of source code which is prone to make the code difficult to read, maintain now or in the future. Described by Martin Fowler as a surface indication that usually corresponds to a deeper problem. Martin Fowler identified 23 such code smells.

ELASTIC is a tool that is based on Spoon which is an open-source library that provides the AST for given Java source code and performs calculations and derives metrics to determine whether a piece of code constitutes a "Code Smell". However, using static analysis alone is prone to "playing it safe" by minimizing the number of false negatives as compared to false positives. Here we introduce LLMs by allowing them to reason on the false positives to weed out malicious actors that slip through Static Analysis and thus increase the precision of the tool while maintaining high recall.

Research Questions

1. Do industry leading tools detect all 23 of Martin Fowler's code smells?
2. Why are certain code smells not detected by industry leading tools?
3. Is Abstract Syntax Tree parsing enough to detect these code smells?
    1. What metrics are required to detect such code smells
4. How do LLMs perform in code smell detection?
5. Can LLMs help Abstract Syntax Tree Parsing in increasing the tools accuracy?

Methodology

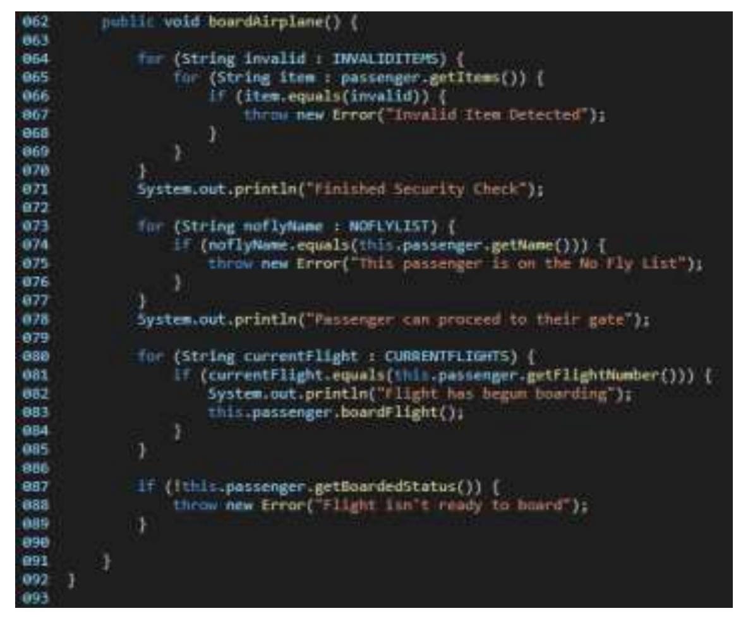


System Architecture Diagram for ELASTIC

The prototype addresses two code smells so far that have been identified as requiring extra context:

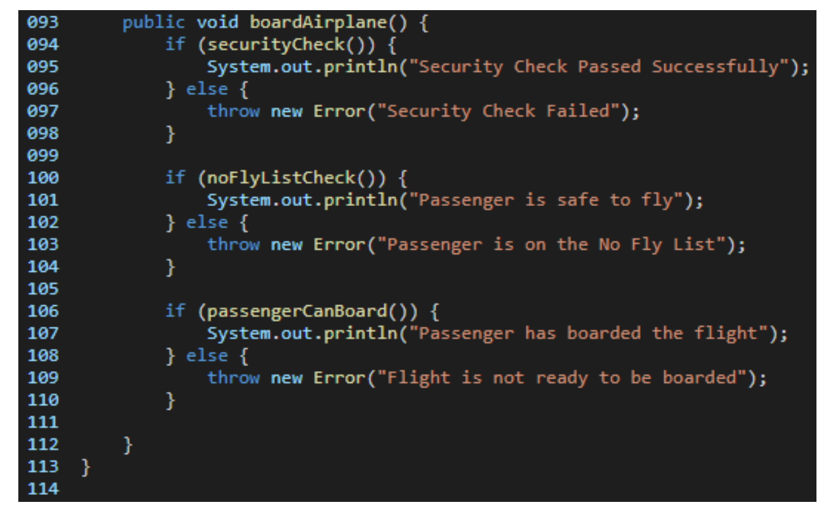
- Long Method
- Middle Man

- Long Method is defined as a method that is too long and is breaking the rule of **Single Responsibility** (i.e. doing more than one logically assigned task).
- The metrics chosen are
    - **Cyclomatic Complexity** greater than 10
    - The **length of the method** is greater than or equal to 30



```java
    public void boardAirplane() {

        for (String invalid : INVALIDITEMS) {
            for (String item : passenger.getItems()) {
                if (item.equals(invalid)) {
                    throw new Error("Invalid Item Detected");
                }
            }
        }
        System.out.println("Finished Security Check");

        for (String noflyName : NOFLYLIST) {
            if (noflyName.equals(this.passenger.getName())) {
                throw new Error("This passenger is on the No Fly List");
            }
        }
        System.out.println("Passenger can proceed to their gate");

        for (String currentFlight : CURRENTFLIGHTS) {
            if (currentFlight.equals(this.passenger.getFlightNumber())) {
                System.out.println("Flight has begun boarding");
                this.passenger.boardFlight();
            }
        }

        if (!this.passenger.getBoardedStatus()) {
            throw new Error("Flight isn't ready to board");
        }

    }
}
```

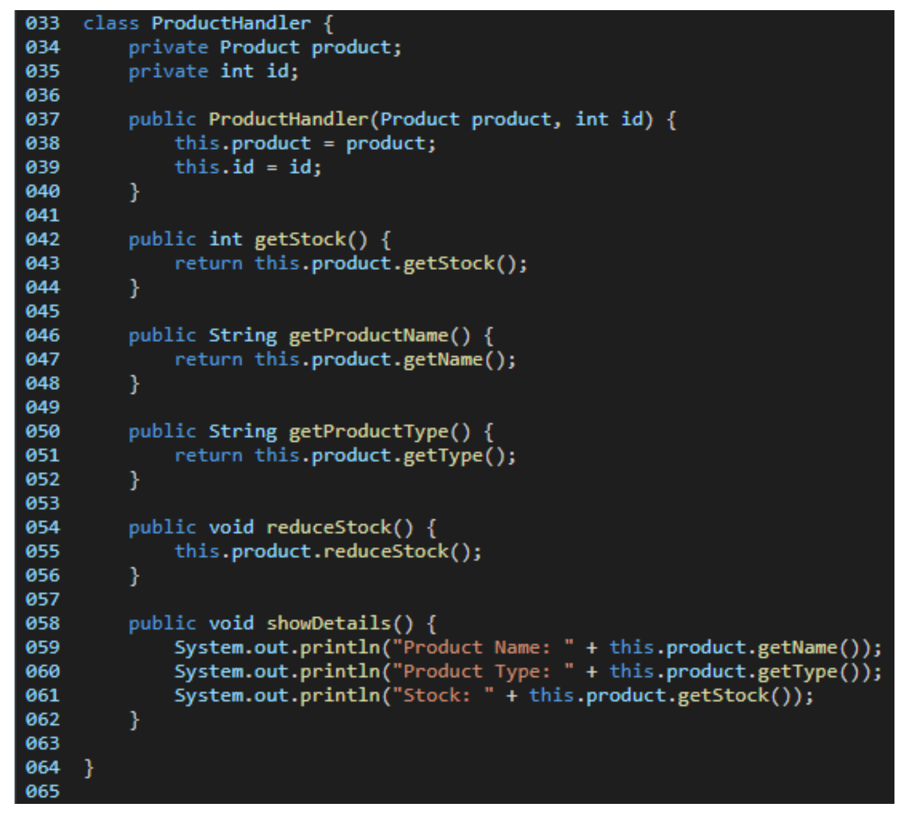
An example of the Long Method Code Smell



```java
    public void boardAirplane() {
        if (securityCheck()) {
            System.out.println("Security Check Passed Successfully");
        } else {
            throw new Error("Security Check Failed");
        }

        if (noFlyListCheck()) {
            System.out.println("Passenger is safe to fly");
        } else {
            throw new Error("Passenger is on the No Fly List");
        }

        if (passengerCanBoard()) {
            System.out.println("Passenger has boarded the flight");
        } else {
            throw new Error("Flight is not ready to be boarded");
        }

    }
}
```

An example of refactoring the Long Method Code Smell

- **Middle Man** is defined as a class that exists for the sole purpose of acting as an **intermediary** between two other classes. This can be thought of as a **wrapper** for a class that does **not** perform any other functions.
- The metrics chosen are
    - **Single Line Ratio** greater than or equal to 0.5
    - **Fan Out** should be greater than 0.



```java
class ProductHandler {
    private Product product;
    private int id;

    public ProductHandler(Product product, int id) {
        this.product = product;
        this.id = id;
    }

    public int getStock() {
        return this.product.getStock();
    }

    public String getProductName() {
        return this.product.getName();
    }

    public String getProductType() {
        return this.product.getType();
    }

    public void reduceStock() {
        this.product.reduceStock();
    }

    public void showDetails() {
        System.out.println("Product Name: " + this.product.getName());
        System.out.println("Product Type: " + this.product.getType());
        System.out.println("Stock: " + this.product.getStock());
    }

}
```

An example of the Middle Man Code Smell

Prompting Techniques have been taken from a previous tool **SmeLLM**[1] which acts as a precursor to the development of this tool. The following is a part of the prompt sent to the LLM along with the source code for the Middle Man code smell:

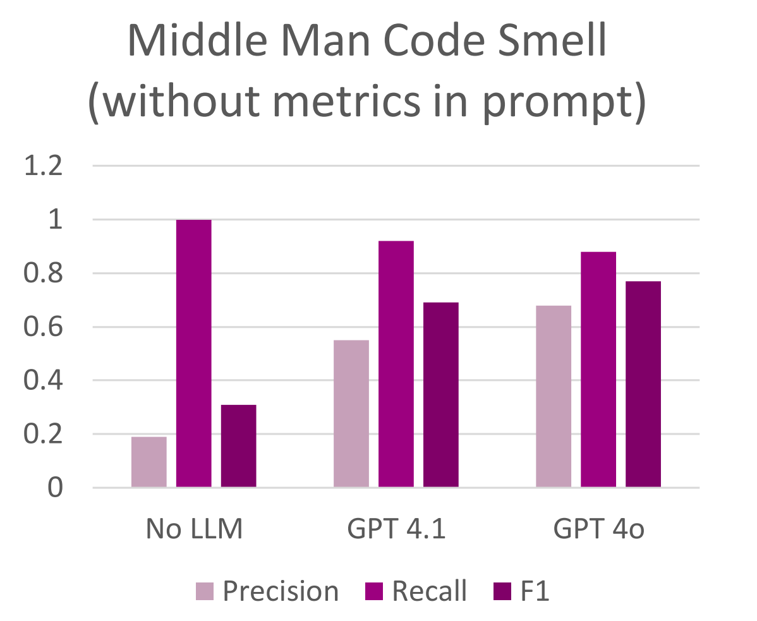
A code smell is an anti-pattern in code based on Martin Fowler's 23 code smells.
You are an expert at detecting the Middle Man code smell. A Middle Man code smell is defined as follows:
"If a class performs only one action, delegating the work to another class, why does it exist at all?".
You will be given source code of a class that is suspected to be a Middle Man, however it also may not be one.
You are to check if it is strictly a Middle Man class or not.

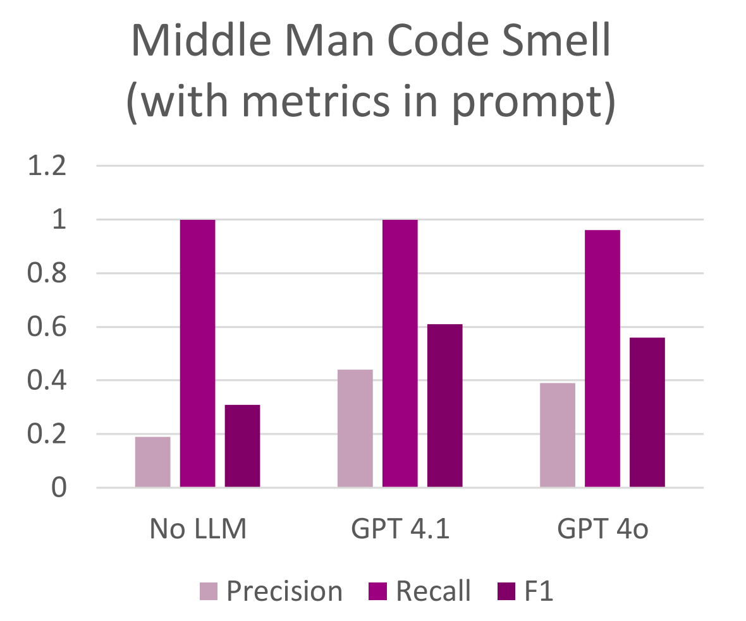
One effort made in prompting techniques was to determine whether the LLM benefits from additional context in the case of metrics. Another version of the prompt contained the metrics along with an explanation of the metric and how it affects the existence of the code smell. This turned out be less effective as it signaled to the LLM to forego its own reasoning about the intent and rely more on the metrics.

1. https://github.com/refactoring-assistant/SmeLLM

Results

We can see that metrics are sent in the prompt, precision drops which means the LLM is now allowing more false positives in its evaluation because it more closely follows how the tool detects Code Smells





Conclusion

So far, the prototype has shown that LLMs can indeed help improve the accuracy of Code Smell detection when used to detect intent and address the contextual void that often comes with Static Analysis tools.

SmeLLM earlier showed that simply using LLMs to detect Code Smells proved to be a challenge, but building on that, when we use Static Analysis to weed out false negatives first and then only use LLMs in cases of ambiguity, they seem to help increase detection rates than hurt them. Prompting also plays a huge part when it comes to the success of working with LLMs. Static Analysis can thus help reduce the cost and runtime associated with SmeLLM.

Future Work

- Extend the capabilities of the tool to address more of the 23 Code Smells defined by Martin Fowler
- Integrate the tool into Pawtograder, an open source automated platform for assignments and grading student submissions.
- Conducting a Usability Study of the ELASTIC integration with Pawtograder