Construction Lab 3

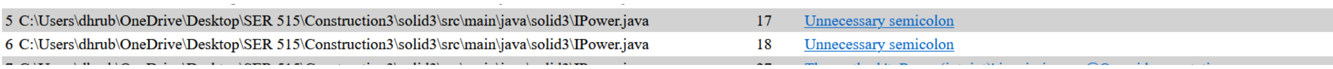
Suparno Roy Chowdhury

Part 2:

f)

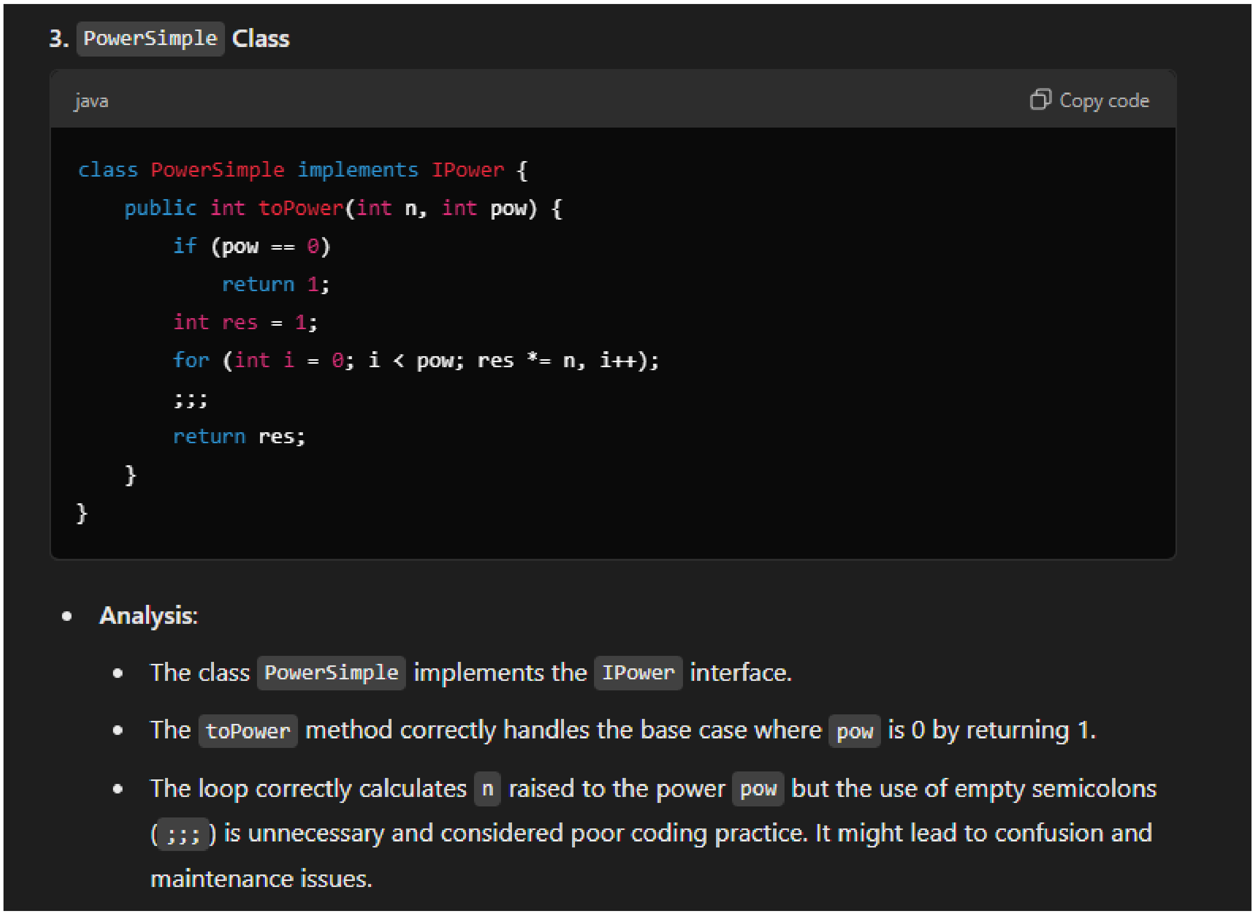
a) Copilot, PMD, and Microsoft both pointed out weird formatting issues with the original IPower code. These are the easiest observations to make, even to the human eye. All three systems made the same suggestion on fixing the semi-colon overuse in the for loop at PowerSimple.

PMD on semi-colon formatting:



5	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	17	Unnecessary semicolon
6	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	18	Unnecessary semicolon

ChatGPT on semi-colon formatting:

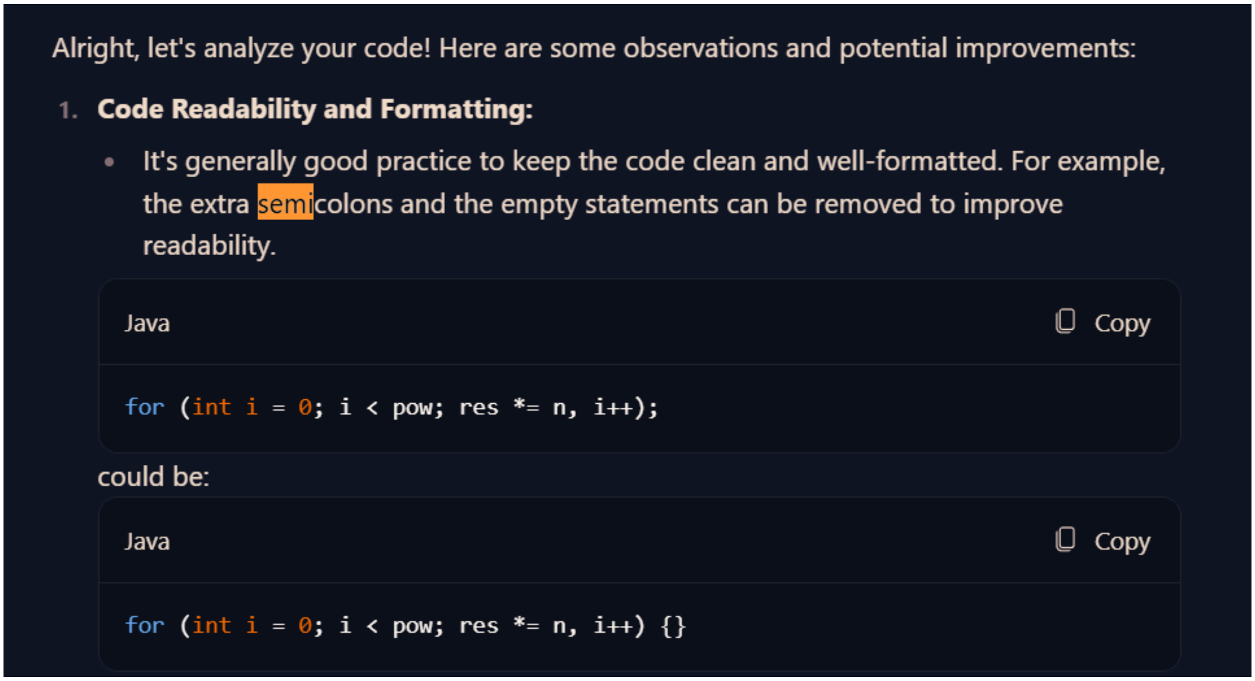


3. `PowerSimple` Class

```java
class PowerSimple implements IPower {
    public int toPower(int n, int pow) {
        if (pow == 0)
            return 1;
        int res = 1;
        for (int i = 0; i < pow; res *= n, i++);
        ;;;
        return res;
    }
}
```

- **Analysis:**
  - The class `PowerSimple` implements the `IPower` interface.
  - The `toPower` method correctly handles the base case where `pow` is 0 by returning 1.
  - The loop correctly calculates `n` raised to the power `pow` but the use of empty semicolons (`;;;`) is unnecessary and considered poor coding practice. It might lead to confusion and maintenance issues.

Copilot on semi-colon formatting:



Alright, let's analyze your code! Here are some observations and potential improvements:

1. **Code Readability and Formatting:**
    - It's generally good practice to keep the code clean and well-formatted. For example, the extra semicolons and the empty statements can be removed to improve readability.

```java
for (int i = 0; i < pow; res *= n, i++);
```

could be:

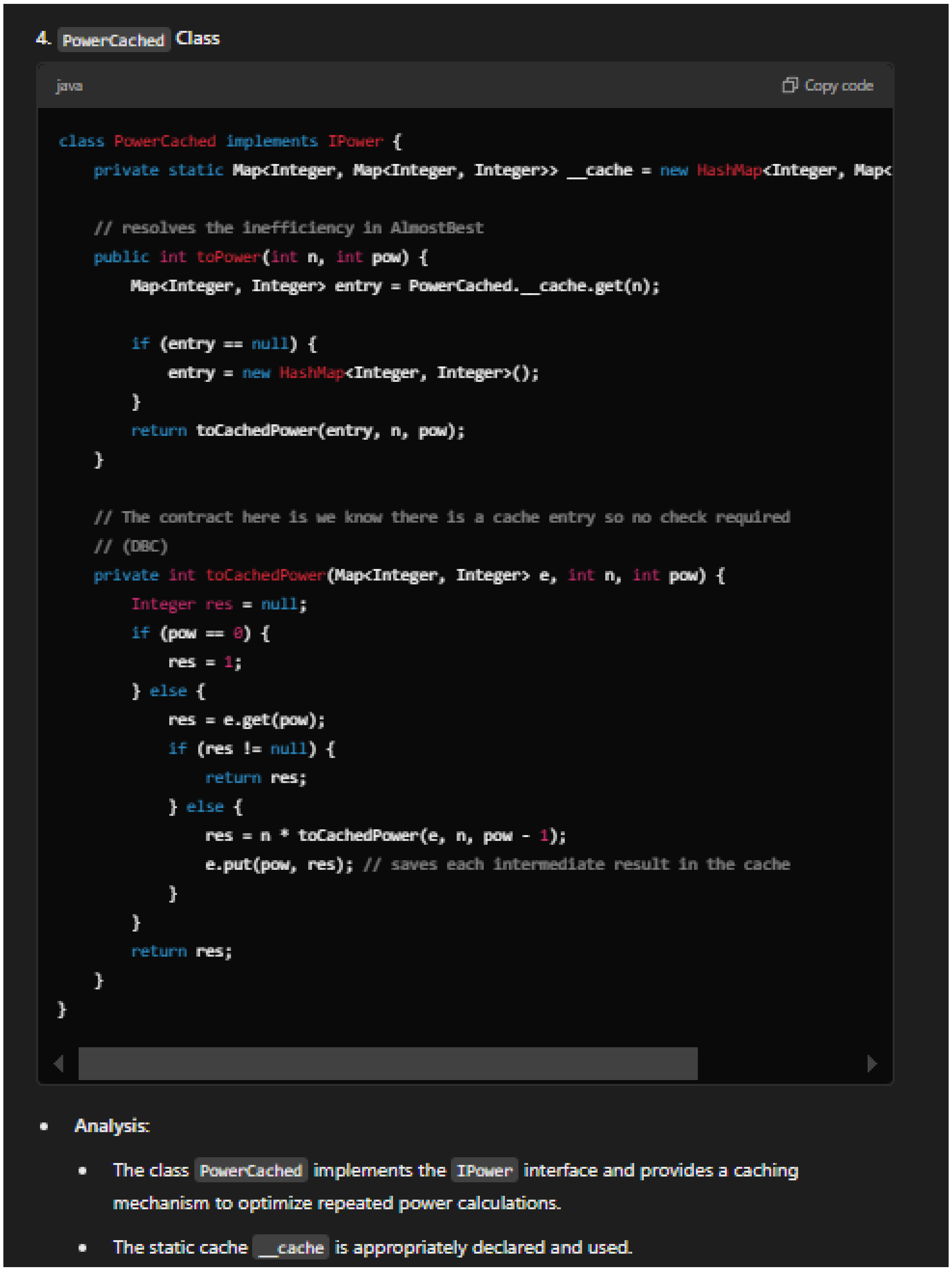
```java
for (int i = 0; i < pow; res *= n, i++) {}
```

b)

PMD differed from ChatGPT and Copilot with the handling of __cache.get(). Originally, it was PowerCached.__cache.get() but PMD highlighted that it was an unnecessary qualifier because it was already declared in the scope. Removing PowerCached resulted in PMD no longer throwing up an error on it.

Neither ChatGPT nor Microsoft Copilot noticed this error. ChatGPT in fact states cache as a whole is appropriately used and copilot says nothing in that regard.:

From conversation 1 in part b:



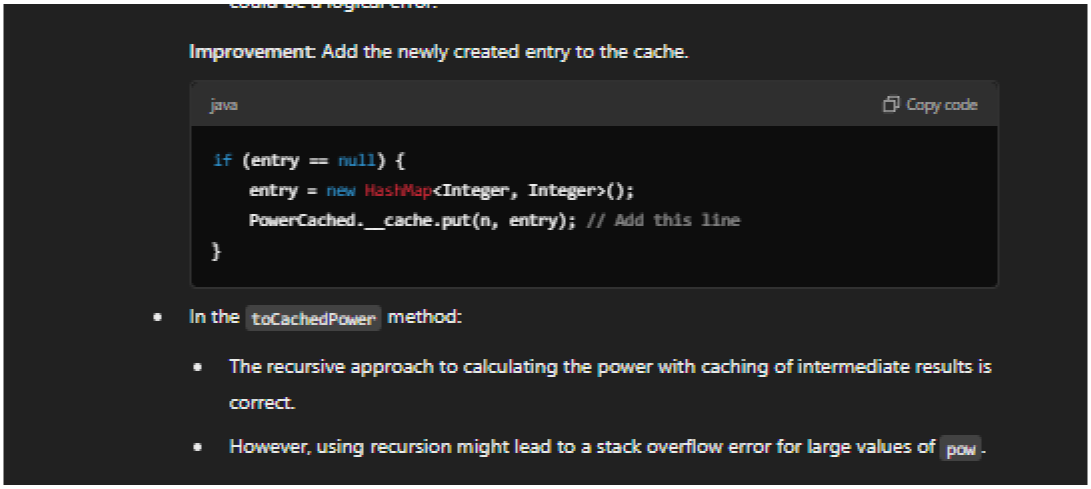
4. `PowerCached` Class

```java
class PowerCached implements IPower {
    private static Map<Integer, Map<Integer, Integer>> __cache = new HashMap<Integer, Map<

    // resolves the inefficiency in AlmostBest
    public int toPower(int n, int pow) {
        Map<Integer, Integer> entry = PowerCached.__cache.get(n);

        if (entry == null) {
            entry = new HashMap<Integer, Integer>();
        }
        return toCachedPower(entry, n, pow);
    }

    // The contract here is we know there is a cache entry so no check required
    // (DBC)
    private int toCachedPower(Map<Integer, Integer> e, int n, int pow) {
        Integer res = null;
        if (pow == 0) {
            res = 1;
        } else {
            res = e.get(pow);
            if (res != null) {
                return res;
            } else {
                res = n * toCachedPower(e, n, pow - 1);
                e.put(pow, res); // saves each intermediate result in the cache
            }
        }
        return res;
    }
}
```

- Analysis:
  - The class `PowerCached` implements the `IPower` interface and provides a caching mechanism to optimize repeated power calculations.
  - The static cache `__cache` is appropriately declared and used.

However, both Copilot and ChatGPT noticed that if a null is detected, code must be written to account for a new entry into the cache. This was not pointed out by PMD.

From conversation 1 in part b and c

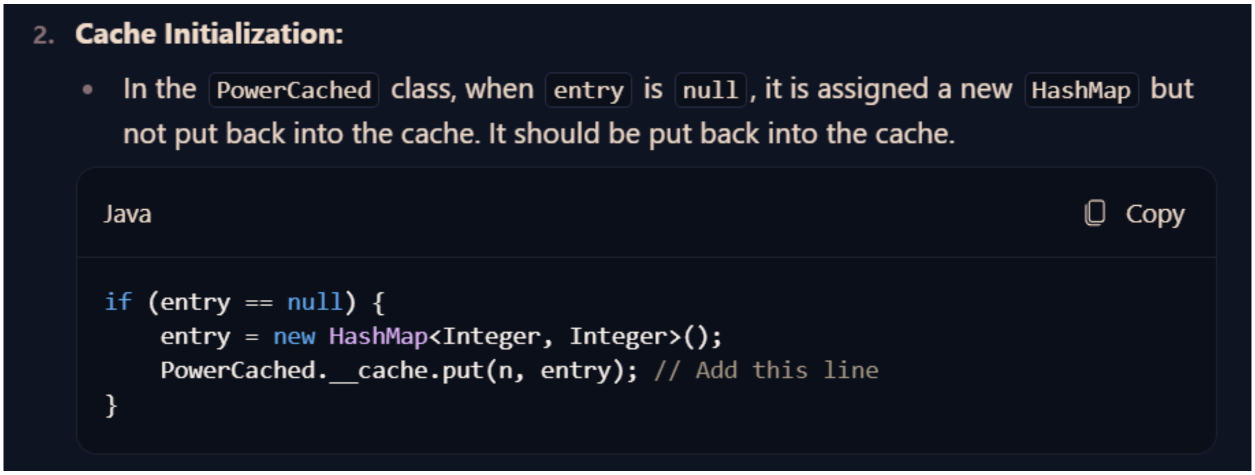


could be a logical error.

Improvement: Add the newly created entry to the cache.

```java
if (entry == null) {
    entry = new HashMap<Integer, Integer>();
    PowerCached.__cache.put(n, entry); // Add this line
}
```

- In the `toCachedPower` method:
  - The recursive approach to calculating the power with caching of intermediate results is correct.
  - However, using recursion might lead to a stack overflow error for large values of `pow`.



2. **Cache Initialization:**
   - In the `PowerCached` class, when `entry` is `null`, it is assigned a new `HashMap` but not put back into the cache. It should be put back into the cache.

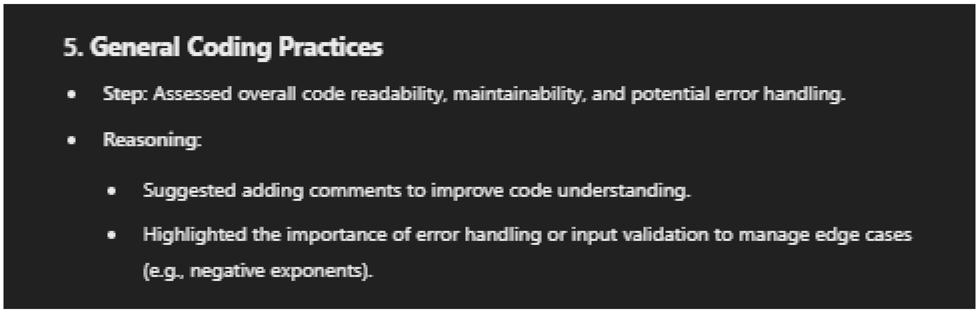
```java
if (entry == null) {
    entry = new HashMap<Integer, Integer>();
    PowerCached.__cache.put(n, entry); // Add this line
}
```

c)

Like mentioned in a, there are some overlaps in the way it performs static analysis. All three systems look to see if general coding practices are adhered to.

Chat GPT:

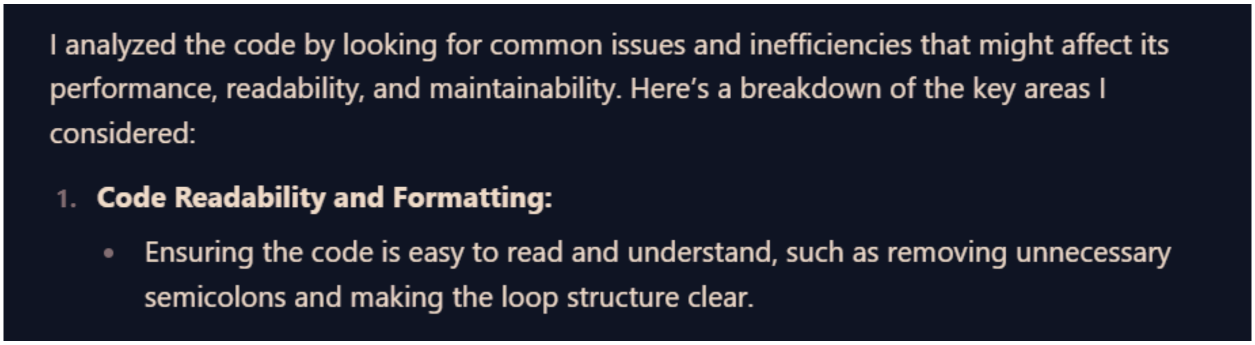
From conversation 1 in part b



5. General Coding Practices

- Step: Assessed overall code readability, maintainability, and potential error handling.
- Reasoning:
  - Suggested adding comments to improve code understanding.
  - Highlighted the importance of error handling or input validation to manage edge cases (e.g., negative exponents).

Copilot:

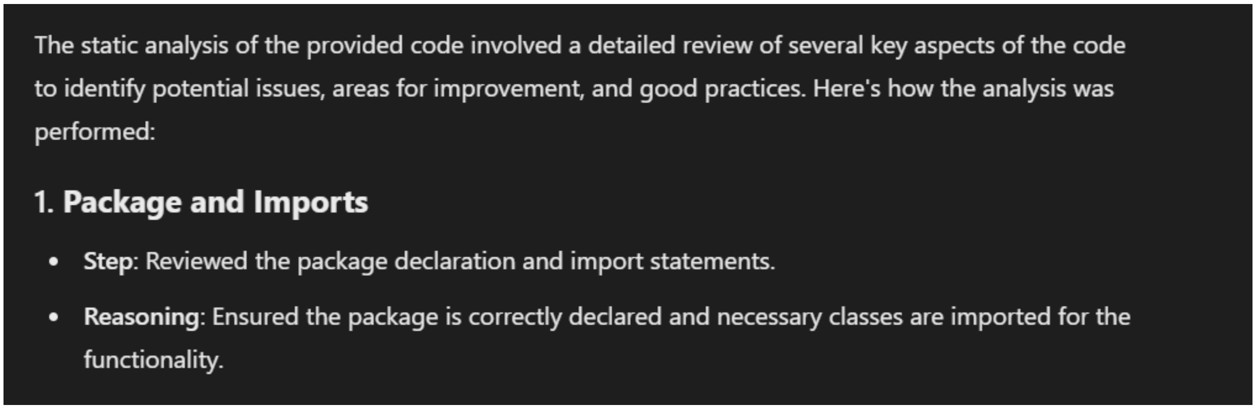


I analyzed the code by looking for common issues and inefficiencies that might affect its performance, readability, and maintainability. Here's a breakdown of the key areas I considered:

1. **Code Readability and Formatting:**
   - Ensuring the code is easy to read and understand, such as removing unnecessary semicolons and making the loop structure clear.

These are where some of the similarities end. ChatGPT, Copilot, and PMD have different ways of doing deep dives with their static analysis.

ChatGPT will be thorough with its analysis, even going as far as to analyse packages. From conversation 1b:



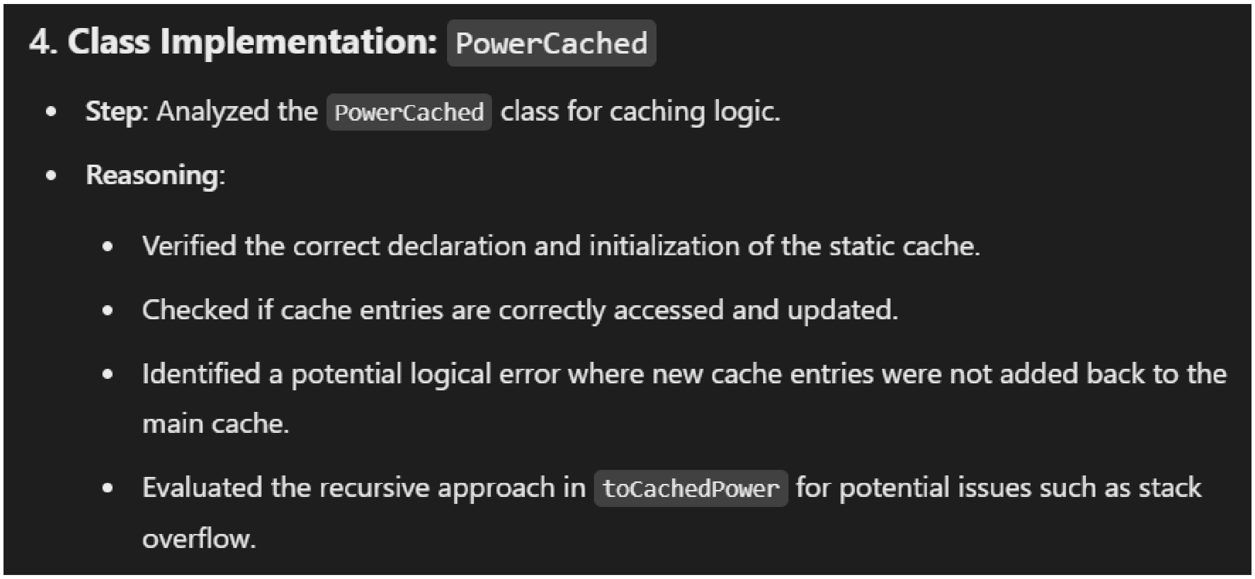
The static analysis of the provided code involved a detailed review of several key aspects of the code to identify potential issues, areas for improvement, and good practices. Here's how the analysis was performed:

1. Package and Imports

- **Step**: Reviewed the package declaration and import statements.
- **Reasoning**: Ensured the package is correctly declared and necessary classes are imported for the functionality.

It will also point out inefficiencies, but rather than classifying them as a part of a problem set, it will analyze the segment of code with an inefficiency and point it out.

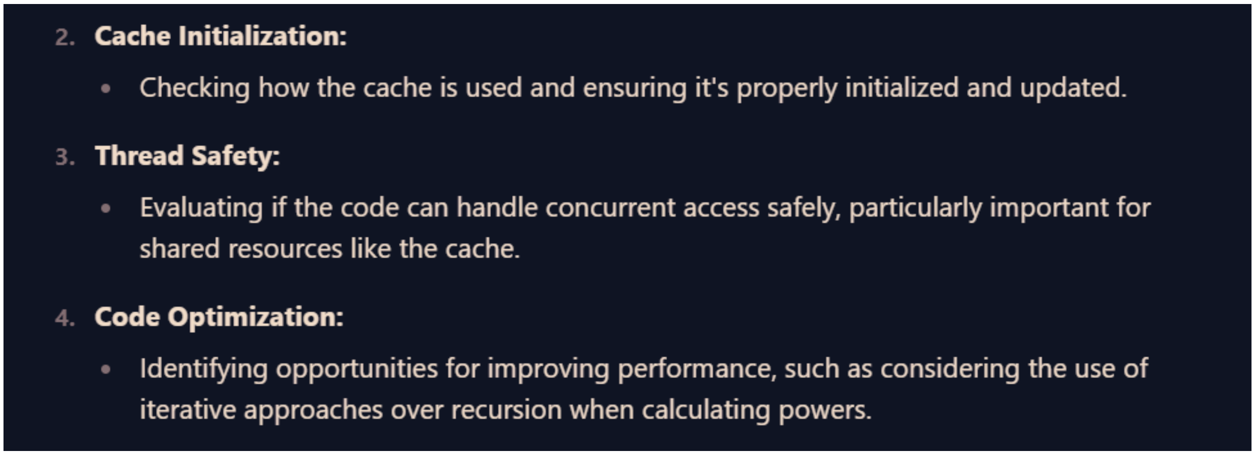
From conversation 1b:



4. Class Implementation: `PowerCached`

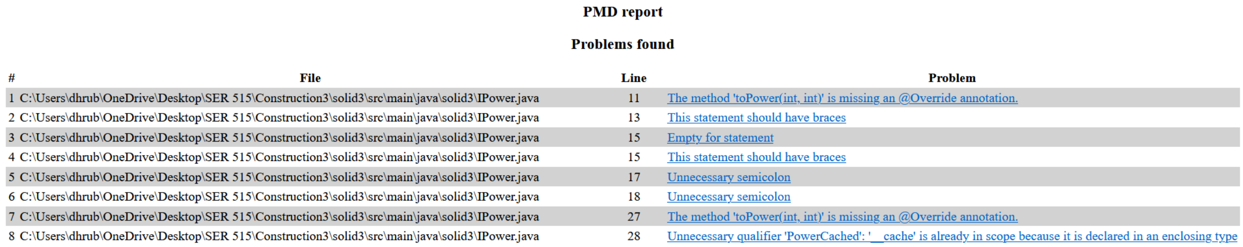
- **Step**: Analyzed the `PowerCached` class for caching logic.
- **Reasoning**:
  - Verified the correct declaration and initialization of the static cache.
  - Checked if cache entries are correctly accessed and updated.
  - Identified a potential logical error where new cache entries were not added back to the main cache.
  - Evaluated the recursive approach in `toCachedPower` for potential issues such as stack overflow.

Copilot is more adept at pointing out general inefficiencies.
From conversation 1c:



2. **Cache Initialization:**
   - Checking how the cache is used and ensuring it's properly initialized and updated.
3. **Thread Safety:**
   - Evaluating if the code can handle concurrent access safely, particularly important for shared resources like the cache.
4. **Code Optimization:**
   - Identifying opportunities for improving performance, such as considering the use of iterative approaches over recursion when calculating powers.

PMD may be limited by its quickstart ruleset, which is why it doesn't provide stronger analysis on the cache aspect of IPower.java and only provides a surface level analysis. If we were to integrate more rulesets, it may have a better shot at detecting the inefficiencies ChatGPT and Copilot noticed.



PMD report

Problems found

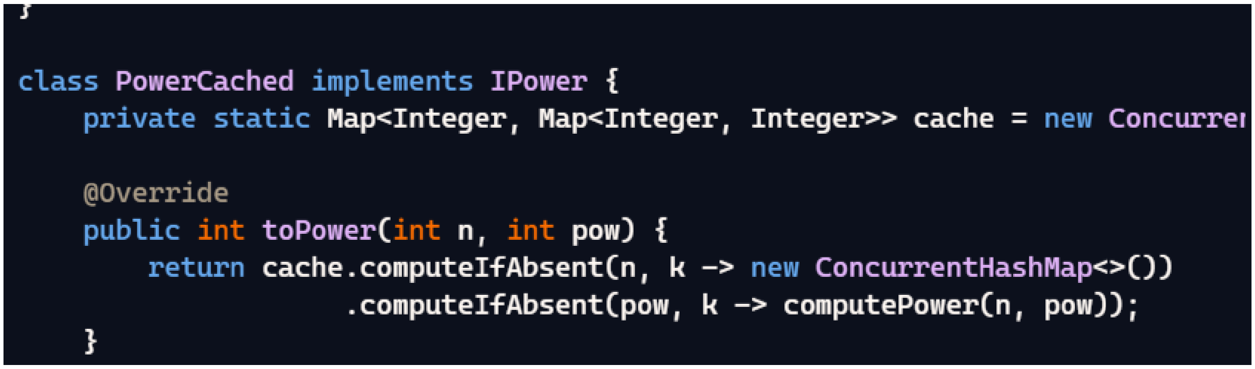
#	File	Line	Problem
1	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	11	The method 'toPower(int, int)' is missing an @Override annotation.
2	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	13	This statement should have braces
3	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	15	Empty for statement
4	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	15	This statement should have braces
5	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	17	Unnecessary semicolon
6	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	18	Unnecessary semicolon
7	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	27	The method 'toPower(int, int)' is missing an @Override annotation.
8	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	28	Unnecessary qualifier 'PowerCached': '__cache' is already in scope because it is declared in an enclosing type

d)

I would prefer to use PMD. The issues with ChatGPT and Copilot is its tendency to introduce context to pre-existing code. It will draw from past knowledge of previous problems and attempt to fix it.

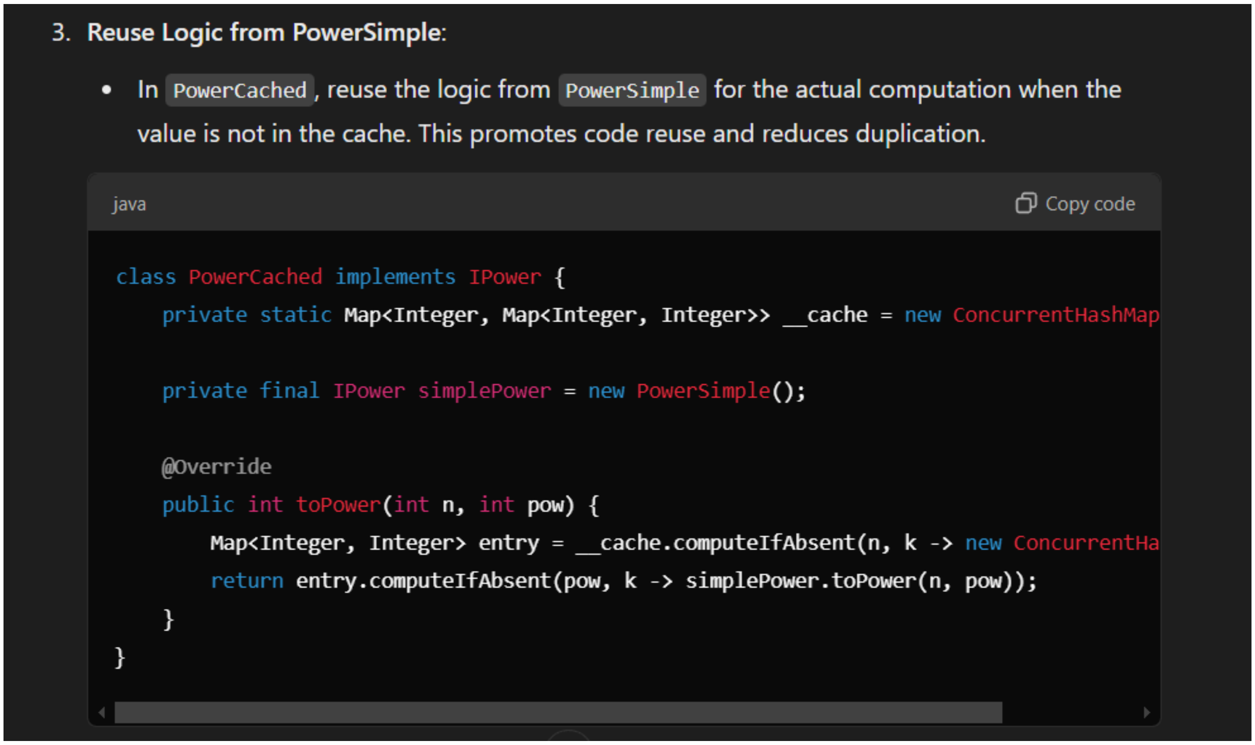
This is observed from after implementing the PMD fixes in step e. Copilot and ChatGPT suggested the usage of concurrent hash maps which, while useful, can end up introducing more complexities if it doesn't follow expected behaviour.

Copilot step e) conversation after implementing PMD fixes:



```
}

class PowerCached implements IPower {
    private static Map<Integer, Map<Integer, Integer>> cache = new Concurre

    @Override
    public int toPower(int n, int pow) {
        return cache.computeIfAbsent(n, k -> new ConcurrentHashMap<>())
                    .computeIfAbsent(pow, k -> computePower(n, pow));
    }
```

ChatGPT step e) conversation after implementing PMD fixes:



3. **Reuse Logic from PowerSimple:**
   - In `PowerCached`, reuse the logic from `PowerSimple` for the actual computation when the value is not in the cache. This promotes code reuse and reduces duplication.

```java
class PowerCached implements IPower {
    private static Map<Integer, Map<Integer, Integer>> __cache = new ConcurrentHashMap

    private final IPower simplePower = new PowerSimple();

    @Override
    public int toPower(int n, int pow) {
        Map<Integer, Integer> entry = __cache.computeIfAbsent(n, k -> new ConcurrentHa
        return entry.computeIfAbsent(pow, k -> simplePower.toPower(n, pow));
    }
}
```

Furthermore, production code could come in large files that ChatGPT might not parse all off without there being a cut off. For a more automated system, PMD would be better to include, especially within a CI/CD pipeline.

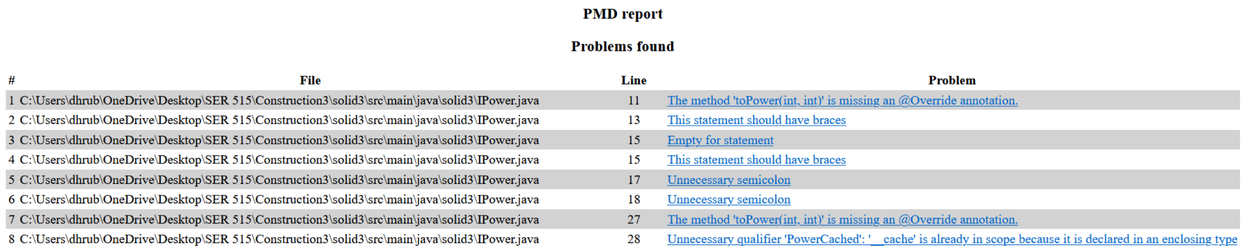
Conversations and responses for parts a to e

a)

```
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:11:	MissingOverride:
The method 'toPower(int, int)' is missing an @Override annotation.
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:13:
ControlStatementBraces: This statement should have braces
```

```
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:15:
EmptyControlStatement:  Empty for statement
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:15:
ControlStatementBraces: This statement should have braces
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:17:
UnnecessarySemicolon:   Unnecessary semicolon
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:18:
UnnecessarySemicolon:   Unnecessary semicolon
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:27:       MissingOverride:
The method 'toPower(int, int)' is missing an @Override annotation.
C:\Users\dhrub\OneDrive\Desktop\SER
515\Construction3\solid3\src\main\java\solid3\IPower.java:28:
UnnecessaryFullyQualifiedName:  Unnecessary qualifier 'PowerCached': '__cache' is
already in scope because it is declared in an enclosing type
```

In a nicer interface thanks to using pmdMain from Gradle:

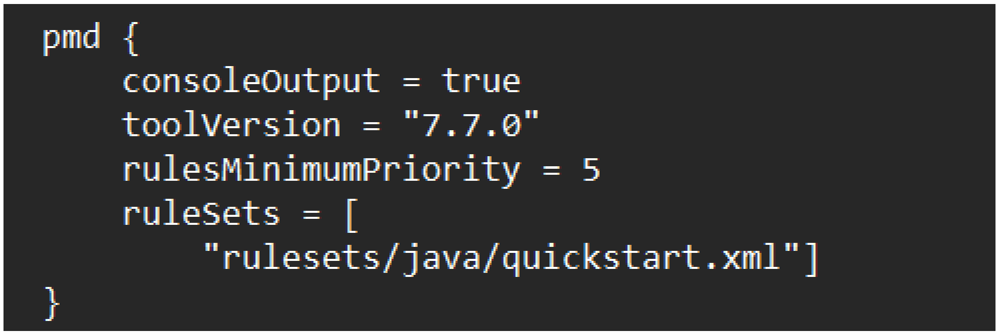


PMD report

Problems found

#	File	Line	Problem
1	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	11	The method 'toPower(int, int)' is missing an @Override annotation.
2	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	13	This statement should have braces
3	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	15	Empty for statement
4	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	15	This statement should have braces
5	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	17	Unnecessary semicolon
6	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	18	Unnecessary semicolon
7	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	27	The method 'toPower(int, int)' is missing an @Override annotation.
8	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\IPower.java	28	Unnecessary qualifier 'PowerCached': '__cache' is already in scope because it is declared in an enclosing type

The gradle configuration used:



```
pmd {
    consoleOutput = true
    toolVersion = "7.7.0"
    rulesMinimumPriority = 5
    ruleSets = [
        "rulesets/java/quickstart.xml"]
}
```

b)

Two prompts asked:

- Run static analysis on the following code: <included code for IPower.java>
- How did you perform your analysis for this code?

https://chatgpt.com/share/67469d21-3ff0-8008-bf0b-2cb08a9aa61c

What I've observed from ChatGPT 4o (Human observation here by human Suparno): It offers a piece by piece static review of code. It will handle each segment of code, looking out for bad practices that go beyond just bad formatting.

c)

My Input:

Run static analysis on the following code: package solid3; import java.util.Map; import java.util.HashMap; public interface IPower {   int toPower(int n, int pow); } class PowerSimple implements IPower {   public int toPower(int n, int pow) {   if (pow == 0)   return 1;   int res = 1;   for (int i = 0; i < pow; res *= n, i++)   ;   ;   ;   return res;   } } class PowerCached implements IPower {   private static Map<Integer, Map<Integer, Integer>> __cache = new HashMap<Integer, Map<Integer, Integer>>();   // resolves the inefficiency in AlmostBest   public int toPower(int n, int pow) {   Map<Integer, Integer> entry = PowerCached.__cache.get(n);   if (entry == null) {   entry = new HashMap<Integer, Integer>();   }   return toCachedPower(entry, n, pow);   }   // The contract here is we know there is a cache entry so no check required   // (DBC)   private int toCachedPower(Map<Integer, Integer> e, int n, int pow) {   Integer res = null;   if (pow == 0) {   res = 1;   } else {   res = e.get(pow);   if (res != null) {   return res;   } else {   res = n * toCachedPower(e, n, pow - 1);   e.put(pow, res); // saves each intermediate result in the cache   }   }   return res;   } }

Microsoft Copilot:

Alright, let's analyze your code! Here are some observations and potential improvements:

1. **Code Readability and Formatting:**
    - It's generally good practice to keep the code clean and well-formatted. For example, the extra semicolons and the empty statements can be removed to improve readability.

java

```
for (int i = 0; i < pow; res *= n, i++);
```

could be:

java

```
for (int i = 0; i < pow; res *= n, i++) {}
```

2.
3. **Cache Initialization:**
    - In the `PowerCached` class, when `entry` is `null`, it is assigned a new `HashMap` but not put back into the cache. It should be put back into the cache.

java

```
if (entry == null) {
   entry = new HashMap<Integer, Integer>();
   PowerCached.__cache.put(n, entry); // Add this line
}
```

4.
5. **Thread Safety:**
    - The cache implementation in `PowerCached` is not thread-safe. If multiple threads access and modify the cache simultaneously, it could lead to incorrect behavior. You might consider using `ConcurrentHashMap` or synchronizing access to the cache.
6. **Code Optimization:**
    - In the `PowerCached` class, the `toCachedPower` method uses recursion, which might not be the most efficient way to calculate powers due to the overhead of recursive calls. An iterative approach could be more efficient.
7. **Interfaces and Implementation:**
    - The code properly uses interfaces and different implementations, which is good for adhering to SOLID principles. However, the interface could be extended with additional methods if needed for future scalability.
8. **Magic Numbers:**

    - Avoid magic numbers (like `0` and `1` in the code) by defining them as constants.

java

```
private static final int BASE_CASE = 1;
```

9. Then use `BASE_CASE` instead of `1` in the `toPower` and `toCachedPower` methods.
10. **Documentation and Comments:**
    - Adding comments and Javadoc to explain the purpose and usage of classes and methods would make the code easier to understand and maintain.

Here's an improved version of your code with some of these suggestions implemented:

java

```
package solid3;

import java.util.Map;
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public interface IPower {
    int toPower(int n, int pow);
}

class PowerSimple implements IPower {
    public int toPower(int n, int pow) {
        if (pow == 0) return 1;
        int res = 1;
        for (int i = 0; i < pow; res *= n, i++) {}
        return res;
    }
}

class PowerCached implements IPower {
    private static final int BASE_CASE = 1;
    private static Map<Integer, Map<Integer, Integer>> __cache = new
ConcurrentHashMap<>();

    public int toPower(int n, int pow) {
        Map<Integer, Integer> entry = PowerCached.__cache.get(n);
        if (entry == null) {
            entry = new HashMap<>();
            PowerCached.__cache.put(n, entry);
        }
        return toCachedPower(entry, n, pow);
    }

    private int toCachedPower(Map<Integer, Integer> e, int n, int pow) {
        Integer res = e.get(pow);
        if (res != null) {
            return res;
        }
        if (pow == 0) {
            res = BASE_CASE;
        } else {
            res = n * toCachedPower(e, n, pow - 1);
            e.put(pow, res);
        }
        return res;
    }
}
```

I hope this helps! Let me know if you need any further assistance. 😊

My Input:

How did you perform your analysis for this code?

Microsoft Copilot:
I analyzed the code by looking for common issues and inefficiencies that might affect its performance, readability, and maintainability. Here's a breakdown of the key areas I considered:

1. **Code Readability and Formatting:**
    - Ensuring the code is easy to read and understand, such as removing unnecessary semicolons and making the loop structure clear.
2. **Cache Initialization:**
    - Checking how the cache is used and ensuring it's properly initialized and updated.

3. **Thread Safety:**
    - Evaluating if the code can handle concurrent access safely, particularly important for shared resources like the cache.
4. **Code Optimization:**
    - Identifying opportunities for improving performance, such as considering the use of iterative approaches over recursion when calculating powers.
5. **Interfaces and Implementation:**
    - Ensuring the code adheres to good object-oriented principles, like using interfaces to define contracts and allowing for flexible implementation.
6. **Magic Numbers:**
    - Avoiding the use of hard-coded values in the code, replacing them with constants for better clarity and maintenance.
7. **Documentation and Comments:**
    - Suggesting the addition of comments and documentation to make the code easier to understand for others who might work on it in the future.

These points help ensure that the code is not only functional but also maintainable and efficient in the long run. If you have any more code you'd like me to review or if there's something specific you'd like to dive deeper into, feel free to let me know! 😊

<u>What I have observed from Microsoft Copilot (Human observation here by human Suparno):</u>

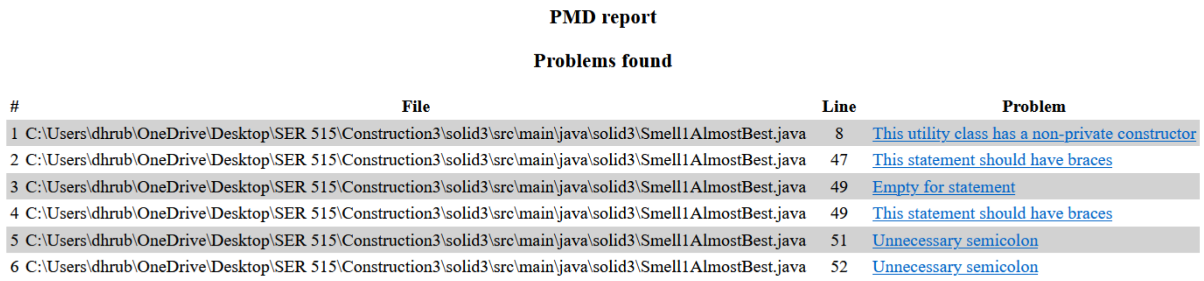
Copilot is less analytical of every aspect of the code, but signals out the most obvious flaws. It will not highlight the problems with each aspect, but it will make brief mentions of what is problematic with it.

d)

Commit for PMD fixes:
bb3099702bb4a69e58598954826564c6153baf93

e)

All the PMD problems highlighted for IPower.java are gone. All that remained were Smell1AlmostBest.java's problems.



PMD report

Problems found

#	File	Line	Problem
1	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\Smell1AlmostBest.java	8	This utility class has a non-private constructor
2	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\Smell1AlmostBest.java	47	This statement should have braces
3	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\Smell1AlmostBest.java	49	Empty for statement
4	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\Smell1AlmostBest.java	49	This statement should have braces
5	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\Smell1AlmostBest.java	51	Unnecessary semicolon
6	C:\Users\dhrub\OneDrive\Desktop\SER 515\Construction3\solid3\src\main\java\solid3\Smell1AlmostBest.java	52	Unnecessary semicolon

For chatGPT, I went the extra step of setting up a new chat and asking it to perform static analysis. This way, it avoids the biases from the previous chat and tries to perform static analysis without checking to see if I've implemented the fixes it suggested, therefore solely judging the PMD fixes.

https://chatgpt.com/share/67476ac5-3728-8008-8950-a81c401ea280

For Microsoft Copilot, I've done the same as well, starting a new chat and having it analyze the problems without previous biases.

My Prompt

Run static analysis on the following code: package solid3; import java.util.Map; import java.util.HashMap; public interface IPower { int toPower(int n, int pow); } class PowerSimple implements IPower { @Override public int toPower(int n, int pow) { if (pow == 0) { return 1; } int res = 1; for (int i = 0; i < pow; i++) { res *= n; } return res; } } class PowerCached implements IPower { private static Map<Integer, Map<Integer, Integer>> __cache = new HashMap<Integer, Map<Integer, Integer>>(); // resolves the inefficiency in AlmostBest @Override public int toPower(int n, int pow) { Map<Integer, Integer> entry = __cache.get(n); if (entry == null) { entry = new HashMap<Integer, Integer>(); } return toCachedPower(entry, n, pow); } // The contract here is we know there is a cache entry so no check required // (DBC) private int toCachedPower(Map<Integer, Integer> e, int n, int pow) { Integer res = null;

```
        if (pow == 0) {                         res = 1;                   } else {
    res = e.get(pow);                     if (res != null) {                               return res;
                } else {                                      res = n * toCachedPower(e, n, pow - 1);
                        e.put(pow, res); // saves each intermediate result in the cache
                }                 }                   return res;     } }
```

Copilot Response:

Let's perform a static analysis on your code. Here are a few points to consider:

Potential Issues:

1. Duplicate Key Handling in Cache (PowerCached)
    - When `PowerCached` fetches an entry from the cache, it does not update the cache if the entry was `null`. It only updates it in `toCachedPower`.
    - Solution: Update the cache entry right after fetching it if it is `null`.
2. Concurrency Issues (PowerCached)
    - If `PowerCached` is accessed by multiple threads simultaneously, the cache can become corrupted, leading to incorrect results or exceptions.
    - Solution: Use synchronization or a concurrent data structure like `ConcurrentHashMap`.

Optimization Opportunities:

1. Initial Cache Population (PowerCached)
    - When `PowerCached` calculates a new power, it only updates the entry for the specific power. Future requests for intermediate powers still require recomputation.
    - Solution: Populate all intermediate powers during the first calculation.
2. Loop Unrolling (PowerSimple)
    - To improve performance, the loop in `toPower` could be unrolled for small fixed powers to reduce loop overhead.

Code Quality Improvements:

1. Variable Naming
    - The variable `__cache` in `PowerCached` does not follow standard Java naming conventions. Consider renaming it to `_cache` or `cache`.
2. Redundant Null Check (PowerCached)

    - The null check on `res` can be eliminated by using Java's `Optional` class to handle the presence or absence of values more cleanly.

Enhanced Version

Here’s a slightly improved version of your code, addressing some of the issues mentioned:

java

```
package solid3;

import java.util.Map;
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public interface IPower {
    int toPower(int n, int pow);
}

class PowerSimple implements IPower {

    @Override
    public int toPower(int n, int pow) {
        if (pow == 0) {
            return 1;
        }
        int res = 1;
        for (int i = 0; i < pow; i++) {
            res *= n;
        }
        return res;
    }
}

class PowerCached implements IPower {
    private static Map<Integer, Map<Integer, Integer>> cache = new
ConcurrentHashMap<>();

    @Override
    public int toPower(int n, int pow) {
        return cache.computeIfAbsent(n, k -> new ConcurrentHashMap<>())
```

```
                .computeIfAbsent(pow, k -> computePower(n, pow));
    }

    private int computePower(int n, int pow) {
        if (pow == 0) {
            return 1;
        }
        int res = 1;
        for (int i = 1; i <= pow; i++) {
            res *= n;
        }
        return res;
    }
}
```

This improved version should handle concurrency more effectively and provide cleaner, more maintainable code. If you need further adjustments or more analysis, feel free to ask!

My Prompt

How did you perform your analysis for this code?

Copilot Response:

How did you perform your analysis for this code?

I performed the static analysis of your code by examining several aspects:

1. Logic and Functionality:
    - Ensured the methods perform their intended calculations correctly.
    - Checked for common pitfalls like infinite loops or incorrect base cases.
2. Code Readability and Maintainability:
    - Looked at variable naming conventions.
    - Considered the organization and structure of the code.
3. Performance Considerations:
    - Analyzed potential inefficiencies, such as repeated calculations or redundant code.
    - Identified opportunities for optimization, like caching intermediate results.
4. Concurrency and Thread Safety:

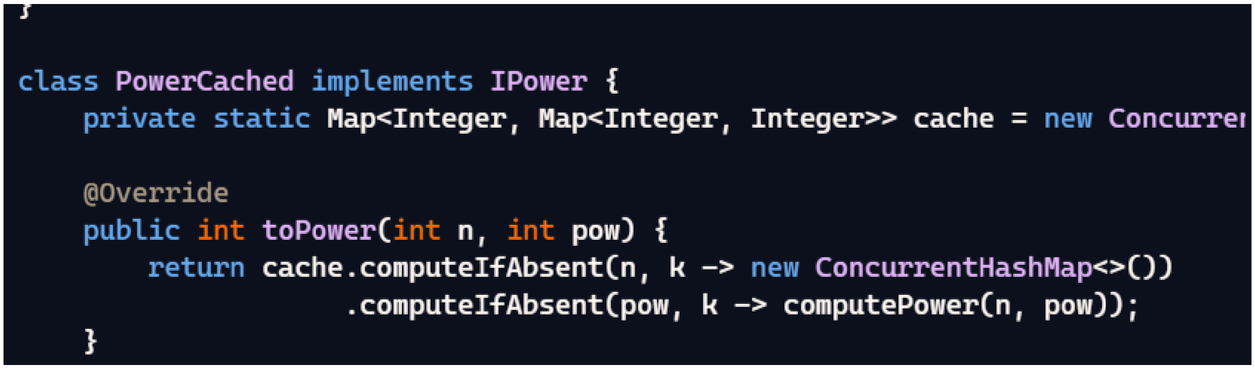
    - Checked if the code could handle simultaneous access by multiple threads.
    - Suggested improvements to avoid issues like data races or deadlocks.
5. Best Practices and Conventions:
    - Ensured the code followed standard Java practices.
    - Recommended improvements for better code quality.

I did not run the code; rather, I inspected it to identify areas for improvement. If you want, I can help you with more detailed analysis or specific improvements!

<u>An interesting observation (by me Human Suparno)</u>

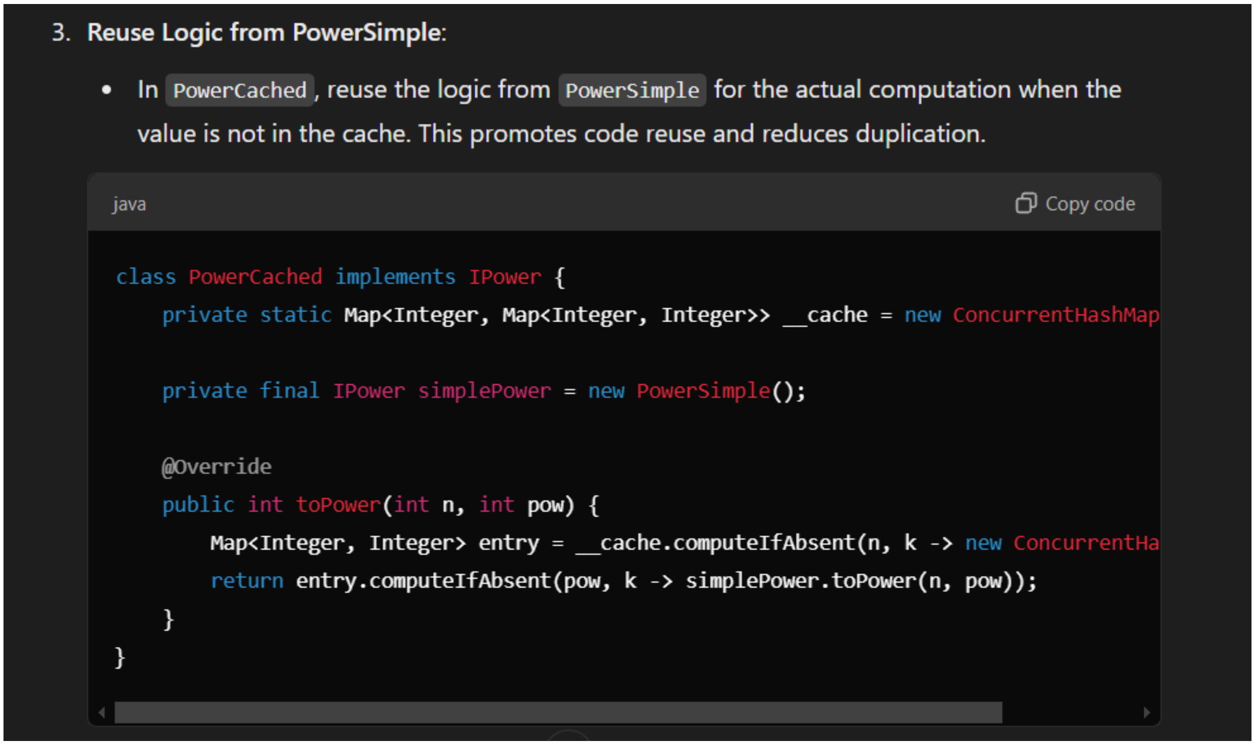
After fixing the issues, Copilot and ChatGPT noted the introduction of using ConcurrentHashMaps to better handle code reuse.

Copilot:



```
}

class PowerCached implements IPower {
    private static Map<Integer, Map<Integer, Integer>> cache = new Concurre

    @Override
    public int toPower(int n, int pow) {
        return cache.computeIfAbsent(n, k -> new ConcurrentHashMap<>())
                    .computeIfAbsent(pow, k -> computePower(n, pow));
    }
```

ChatGPT:



3. **Reuse Logic from PowerSimple**:
   - In `PowerCached`, reuse the logic from `PowerSimple` for the actual computation when the value is not in the cache. This promotes code reuse and reduces duplication.

```java
class PowerCached implements IPower {
    private static Map<Integer, Map<Integer, Integer>> __cache = new ConcurrentHashMap

    private final IPower simplePower = new PowerSimple();

    @Override
    public int toPower(int n, int pow) {
        Map<Integer, Integer> entry = __cache.computeIfAbsent(n, k -> new ConcurrentHa
        return entry.computeIfAbsent(pow, k -> simplePower.toPower(n, pow));
    }
}
```