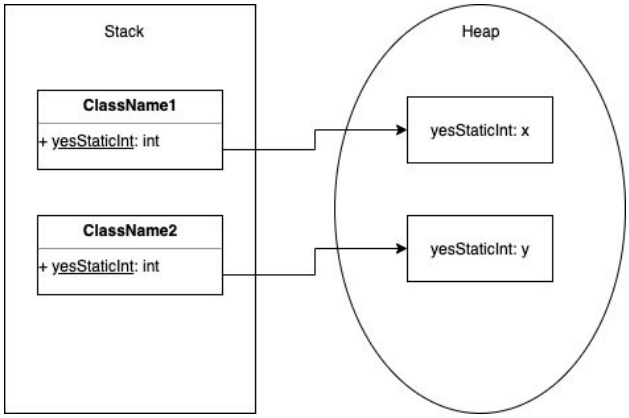
Tabassum Fabiha
tf2478

1.1 - Fields

- No, other classes can also have static fields called yesStiticInt, because Java will refer to them as ClassName1.yesStaticInt and ClassName2.yesStaticInt
-



1.2 - Design Rules

1) The technical debt is the effort it will take to rewrite a simpler algorithm to cover more complex aspects.
2) We leave by technical debt when we write for the mainline and leave exceptions for later. The main line may have limitations that don't allow it to cover those exceptions, and the code may have to be rewritten at a later time to fully cover all the exceptions in the use case.
3) We should record this debt in the javadoc of the methods the limitations reside in. The should be stated by clearly writing that these are the limitations of the system and what the algorithms do not cover.
4) Design smells are designs that make the codebase difficult to maintain.
5) The codebase for the Duck example would have been difficult to maintain if we hadn't used the plugin method, since the addition of so many inherited ducks would have made it incredibly difficult for the programmer to know what types of quacking and flying behaviors already existed and what had yet to be implemented.

1.3 - Interfaces: Comparable, in Ass. 2

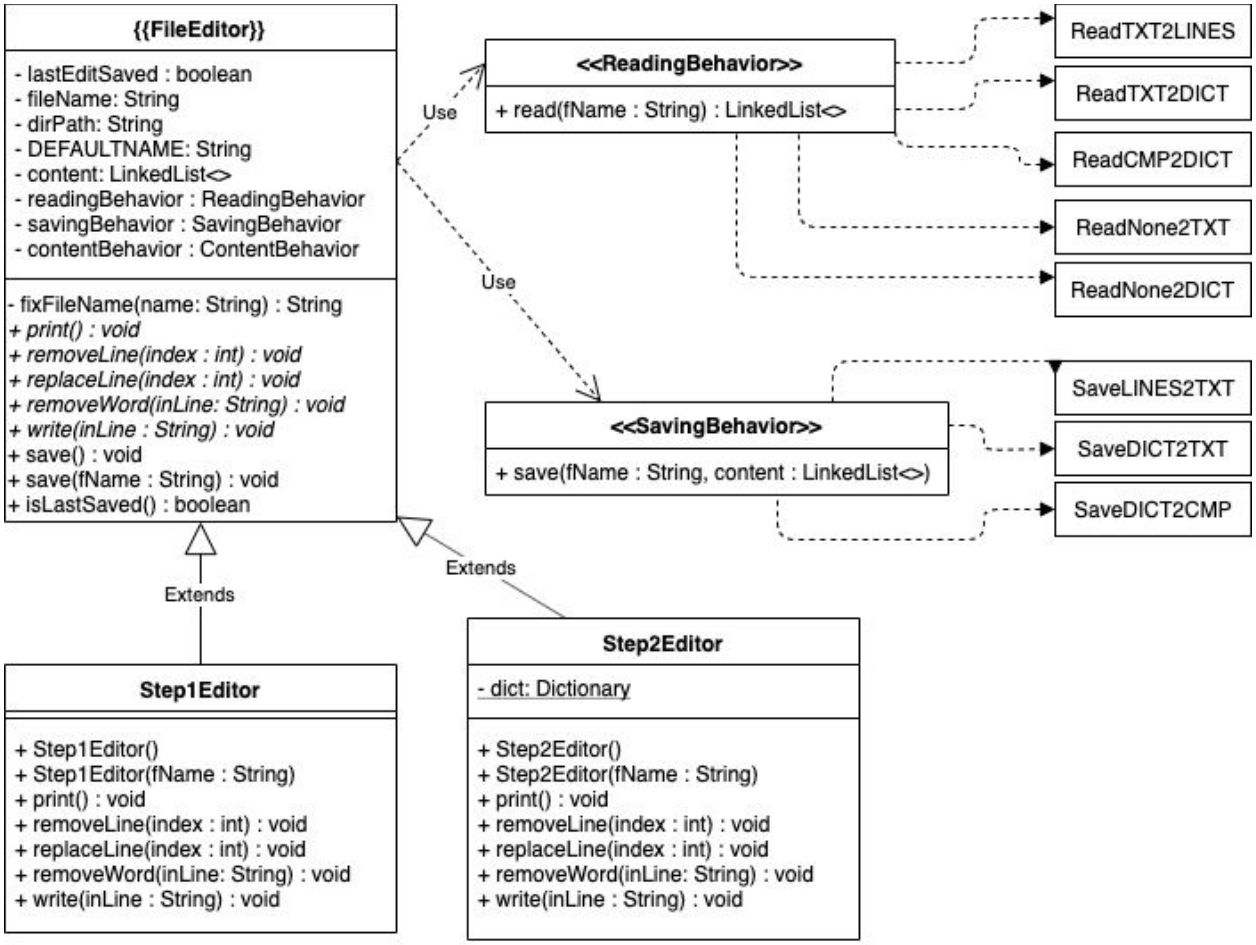
For NET A and NET B

- If A.s < B.s
    - Return -1
- If A.s > B.s
    - Return 1
- If A.s == B.s
    - If A.x < B.x
        - Return -1

- If A.x > B.y
    - Return 1
- If A.x == B.x
    - If A.y < B.y
        - Return -1
    - If A.y > B.y
        - Return 1
    - If A.y == B.y
        - Return 0

This is not really useful to figure out the position of one instance of NET to another instance of NET, since an answer of -1 or 1 doesn't really tell you which of the 3 values are the same. The only way to know the positions relative to each other would be to get a 0 when you use compareTo(). However this is useful for sorting, since you will always be able to order these in a certain way based on the size of the NET, then the x position, and then the y position.

1.4 - Patterns: Strategy, in Ass. 2



1.5 - Patterns: Lazy Evaluation

```
public interface SquareRoot {
```

```
        public double squareRoot(int input);
    }

    public class EagerEvaluation implements SquareRoot {
        public double squareRoot(int input) {
            return Math.sqrt(input);
        }
    }

    public class AnticipatoryEvaluation implements SquareRoot {
        private double storedRoots;
        public AnticipatoryEvaluation(int N) {
            storedRoots = double[N];
            for (int i = 0; i < storedRoots.length; i++) {
                storedRoots[i] = Math.sqrt(i);
            }
        }
        public double squareRoot(int input) {
            return storedRoots[input];
        }
    }

    public class LazyEvaluation implements SquareRoot {
        private double storedRoots;
        public AnticipatoryEvaluation(int N) {
            storedRoots = double[N];
            for (int i = 0; i < storedRoots.length; i++) {
                storedRoots[i] = -1;
            }
        }
        public double squareRoot(int input) {
            if (storedRoots[input] < 0)
                storedRoots[input] = Math.sqrt(input);
            return storedRoots[input];
        }
    }
```

1.6 - Patterns: Pipes&Filters, leading to Programming Part

```
grep “Columbia” inputfile.txt | sort | sed 's/\t/ /g' | uniq | tail -n 3
```

1.7 - Patterns: Builder, leading to Programming Part

```
public class Name {
```

```
private Name( Builder inBuilder ) {
        firstName = inBuilder.firstName;
        middleName1 = inBuilder.middleName1;
        middleName2 = inBuilder.middleName2;
        lastName = inBuilder.lastName;
}

@Override
public String toString() {
        String middles = “”;
        if ( middleName1.length() > 0)
                middles += middleName1 + “ “;
        if ( middleName2.length() > 0)
                middles += middleName2 + “ “;

        return firstName + “ “ + middles + lastName;
}

public static class Builder {
        public Builder() {
                firstName = “John”;
                middleName1 = “”;
                middleName2 = “”;
                lastName = “Doe”;
        }

        public Builder given( String inName ) {
                firstName = inName;
        }
        public Builder middle( String inName ) {
                middleName1 = inName;
        }
        public Builder middle2( String inName ) {
                middleName2 = inName;
        }
        public Builder family( String inName ) {
                lastName = inName;
        }
        public Name build() {
                return new Name( this );
        }

        private String firstName;
```

```
        private String middleName1;
        private String middleName2;
        private String lastName;

    }

        public final String firstName;
        public final String middleName1;
        public final String middleName2;
        public final String lastName;
}
```

- The Builder class will handle default values by initially setting the first name to “John”, the last name to “Doe” and 2 middle names as empty Strings. If the user wishes to change these default names then they may do so with the given, middle, middle2, and family methods. When creating the toString, the Name class checks to see to see if the 2 middle names exist, and each that does is added to a variable called middles with an additional space in the end. The returned value is (firstName + “ “ + middles + lastName). If we end up having no middle names then middles will consist of an empty String and what will be returned will be synonymous to firstName + “ “ + lastName. If the first middle name or second middle name exists but not both, what will be returned will be synonymous to firstName + “ “ + middleName1 + “ “ + lastName or firstName + “ “ + middleName2 + “ “ + lastName. If both middle names exist then the return value will be synonymous to firstName + “ “ + middleName1 + “ “ + middleName2 + “ “ + lastName.
- new Name.Builder().given(“Sarah”).family(“Stout”).build() will return an instance of Name with a toString of “Sarah Stout”.
- new Name.Builder().build() will return an instance of Name with a toString of “John Doe”.