DESIGNLINT: DOCUMENTATION

CSSE375 Team F Project

Date: 5-13-2022

Team: Saayeh Siahmakoun, Alec Polster, Alex Westerman, Nicolas Bohner

Table of Contents

User Guide

System Requirements (User)

In order to use DesignLint, the system must meet the following requirements:

- Java Runtime 8 or Later (Any Flavor)
- At Least 1 GB RAM
- Access to a terminal

System Requirements (Developer)

In addition to the user requirements, the following is needed for developing DesignLint:

- Apache Maven
- Java Development Kit (JDK) 8 or Later (Any Flavor)

Installation and Usage

DesignLint is distributed as an executable JAR that is invoked with arguments. At minimum, it requires a path argument to a Java compiled class (`.class`) or to a directory containing the set of `.class` files. The latter type of path will automatically recurse into subdirectories of the specified path to find all `.class` files, and will throw an error if it finds a Symbolic Link. The following are example of basic invocations of DesignLint:

(NOTE - the `$` indicates that this should be inputted in a terminal)

```
$ java -jar <Path to DesignLint JAR> ./ExampleClass.class
```

```
$ java -jar <Path to DesignLint JAR> /home/user/ExampleDir
```

Advanced Usage

DesignLint offers multiple options to expand output or to only use specific analyzers. These options are detailed below, or are outputted by the program if given the `-h` argument or given bad arguments. We also provide a copy of this output below:

Usage Syntax:

```
$ java -jar <Path to DesignLint JAR> [-v[v[v]]] [-h] [[-a{XX|YY|...}] [-a...] ...] <.class file | directory>
```

Switches:

Switch	Description
`-v`	Includes summary output of analyzer-specific findings.
`-vv`	Display all errors found by the analyzers. Includes output of `-v`
`-vvv`	Display all output generated by analyzers (Errors, Warnings, Info, and Pattern Detection). Includes output of `-v` and `-vv`
`-h`	Show the help output
`-aXX`	Only run the analyzer specified by the code XX. This switch is used once for each analyzer desired, but excluding this switch will run all available analyzers.

Analyzer Codes (Used with `-a`):

Code	Associated Analyzer	Description
`GN`	Generic Name Analyzer	Checks for bad generic class names
`VN`	Variable Name Analyzer	Checks for bad variable names
`ET`	Exception Thrown Analyzer	Checks for bad exception handling practices
`EH`	`equals()` and `hashcode()` Analyzer	Throws warnings if a class doesn't override the `equals()` or `hashcode()` methods
`HC`	High Coupling Analyzer	Checks for classes with high coupling
`LK`	Principle of Least Knowledge Analyzer	Checks for proper encapsulation
`DR`	Don't Repeat Yourself (DRY) Analyzer	Checks for possible areas of repetition

Code	Associated Analyzer	Description
`CI`	Code to Interface Analyzer	Checks if object types use interfaces if they exist
`SI`	Singleton Pattern Detector	Detects if a class implements the Singleton Pattern
`OA`	Object Adapter Pattern Detector	Detects what classes represent an Object Adapter pattern
`ST`	Strategy Pattern Detector	Detects what classes/interfaces implement a Strategy pattern
`TM`	Template Method Pattern Detector	Detects what classes implement a Template Method Pattern

Developer Quickstart

This will serve as a brief primer for building, unit testing, and mutation testing the code. For more details about specific parts of DesignLint, see the developer documentation.

Obtaining the Source

The source for DesignLint is available from the Git repository at https://github.com/rhit-westeraj/DesignLint. To make it available on your local machine for modification, simply clone the repository remote:
`$ git clone https://github.com/rhit-westeraj/DesignLint`

Building The Program

DesignLint uses Apache Maven for dependency management and build pipelines. This means that compiling the program is as simple as running `<maven executable> compile` in the directory of the cloned repository. For a packaged build (i.e. a JAR Executable), this can be done with the `package` lifecycle, which will also run the `compile` step in the lifecycle, as well as testing. Compiled output can be found within the `target` subdirectory within the cloned repository once run.

It should be noted that most modern Java IDEs also have built-in support for processing Maven projects, therefore it may be worth looking at the IDE documentation to understand how it interacts with Maven.

Running Unit Testing and Mutation Testing

Because DesignLint uses Maven for the build lifecycle, it allows for automated running of test prior to packaging or deployment operations. Testing uses JUnit 5 as the unit testing library, with Maven Surefire being used as the integration in Maven for automated unit testing within the build lifecycle. Running test is as simple as running `<maven executable> test` within the cloned repository directory.

We also have added the PITesting plugin to the Maven dependency list to allow for analysis of mutation coverage of tests, improving test robustness. To invoke analysis of the PITesting plugin, it is also as simple as running `<maven executable> pitest:mutationCoverage` in the cloned repository directory. The results of the PITesting Analysis can be found within the `target/pit-reports` subdirectory within the cloned repository once run.

Developer (Maintenance) Guide

Presentation Layer Overview

The presentation layer is defined by the LinterMain and PresentationLayer classes. Descriptions for each are included below.

`LinterMain`

The LinterMain class acts as a wrapper for the PresentationLayer class. LinterMain creates an instance of the PresentationLayer class in its main method, and then uses the flags and methods defined in it to setup and run analyzers and then output linter info. This is done without LinterMain having to know anything about the implementation of the methods that accomplish these tasks. All user input is handled by the LinterMain class. Which analyzer is constructed and run is determined by the user input, which corresponds to one of the flags in the PresentationLayer class that represents each analyzer. LinterMain also allows for the user to input flags for the verbosity and help functions of the system.

Information on the specific flags that the user can input are included in the "Advanced Usage" section of the Home page.

`PresentationLayer`

This class acts as a bridge between the Presentation and Domain layers. This class handles the initialization of all analyzers that the user wants to run using the flags received via input, which is done in the setupAnalyzers method (which uses the helper initAnalyzers method). It also provides a method that runs the analyzers that were constructed and collects relevant data based on the unique implementations of these methods by the analyzer classes (runAnalyzers). Lastly, this class takes all collected data and constructs the output messages that the user will see based on the verbosity flag (vomitOutput).

Modifying the System (with respect to the presentation layer)

Adding support for new types of analyzers, or removing current types of analyzers will require a couple of changes to the classes in the presentation layer. This can be done very easily by only adding or removing a few lines of code. Firstly, you will have add a new flag representing the new analyzer type. Next, you need to modify LinterMain's setFlags method to support the adding of a new flag, as well as add a line for the new flag in the displayHelp method. Lastly, you need to add a couple lines of code in PresentationLayer's initAnalyzers method to construct the new analyzer if the input flag corresponds to it. By doing these things, the presentation layer will fully support any new analyzer implemented in the domain layer. If you are removing an existing analyzer you can simply remove its corresponding flag and the related code in the methods discussed previously.

Analyzer Layer

Currently, there are 12 different concrete analyzer classes (see _Concrete Analyzer Classes_below). Using the the abstract DomainAnalyzer, each concrete analyzer implements the template method `getFeedback(classList: String[])`, which calls three uniquely defined methods, respectively: `getRelevantData(classList: String[]), analyzeData(), & composeReturnTurn(): ReturnType`.

Methods

Method Name	Description
`getRelevantData`	collects all relevant data using `parser: ASMParser`
`analyzeData`	utilize the `getRelevantData()` to create respective linter messages
`composeReturnType`	constructs an `AnalyzerReturn`, counting the linter messages constructed

Concrete Analyzer Classes

Class Name	Description
`VarNameAnalyzer`	analyzes variable names to check for Java naming standards
`TemplateMethodAnalyzer`	analyzes sets of classes to check for Template Pattern
`StrategyAnalyzer`	analyzes sets of classes to check for Strategy Pattern

Class Name	Description
`SingletonAnalyzer`	analyzes sets of classes to check for Singleton Pattern
`PrincipleOfLeastKnowledgeAnalyzer`	analyzes module to check for any violations of the Principle of Least Knowledge
`ObjectAdapterIdentifierAnalyzer`	analyzes module to check for Adapter Pattern
`HighCouplingAnalyzer`	analyzes module to check for high coupling between classes
`GenericTypeNameAnalyzer`	analyzes module for any Java generic type names being used & their format
`ExceptionThrownAnalyzer`	analyzes module for unchecked Exceptions not being thrown properly
`EqualsAndHashcodeAnalyzer`	analyzes module to check for overriding compatibility between equals & hashcode methods
`CodeToInterfaceAnalyzer`	analyzes module to check for any violations of Coding to an interface (i.e. coding to abstraction)

Extensibility of Analyzers

If this repo is cloned, a developer can create their own Analyzer by extending the DomainAnalyzer class and implementing the respective methods. Below is an example of what this may look like for a *new* Analyzer:

```
public class MySpecialAnalyzer extends DomainAnalyzer {

         ASMParser parser = null;

         public MySpecialAnalyzer(ASMParser parser) {
              this.parser = parser;
         }

         public ReturnType getFeedback(String[] classList) {
                getRelevantData(classList);
                analyzeData();
               return composeReturnType();
         }

         public void getRelevantData(String[] classList){...}

         public void analyzeData(){...}

         public ReturnType composeReturnType(){...}
}
```

Data Source Layer

Understanding the `ASMParser` Class

Fundamentally speaking, the `ASMParser` is designed as an "interface layer" between the DesignLint Analyzers and OW2 ASM. This allows for providing a set of common utilities to retrieve information about the disassembled classes for use with analyzers. While this class does not fully utilize all the data generated by the parsing done by ASM, it can be extended by adding new methods to this class.

Construction

Briefly, it is important to understand how `ASMParser` interfaces with ASM. To minimize resource usage, `ASMParser` stores the `ClassNode` objects returned by the ASM `ClassReader` in the private hashmap `classMap`, keyed by the fully qualified *internal JVM* name of the class. For example, to retrieve the `ClassNode` returned from analyzing `java.lang.String`, the proper key to use with `classMap` would be `java/lang/String`.

There are two constructors provided by `ASMParser` that determine where to find the class data:

- The `String[]` constructor should be used if *all* classes that `ASMParser` should hold data for is in the classpath of the DesignLint project. It is only encouraged to use this with respect to unit testing.
- The `InputStream[]` constructor is used by providing some `InputStream` containing proper Java bytecode (such as that from a file) to be read by the ASM `ClassReader`. This is the recommended way to initialize `ASMParser` and an example can be found within the `PresentationLayer` class in the `setupAnalyzers()` method.

Available Methods

Analyzers are provided with the following implemented methods for obtaining information about the input classes:

Name	Parameters	Return Type	Description
`getParsedClassNames()`	N/A	`String[]`	Returns the list of classes that `ASMParser` has parsed
`getSuperName()`	`String className`	`String`	Returns the name of the superclass for the given class with name `classname`
`getInterfaces()`	`String className`	`String[]`	Returns the list of interfaces that the class with name `className` implements
`getMethods()`	`String className`	`String[]`	Returns a list of all method names defined by the class with name `className`. This only returns the name of methods and no other information.
`getMethodExceptionSignature()`	`String className`, `String methodName`	`String[]`	Returns all the exception types that are thrown by `methodName` in the class with name `className`.
`getMethodExceptionCaught()`	`String className`, `String methodName`	`String[]`	Returns all the exception types that are caught by `methodName` in the class with name `className`.

Name	Parameters	Return Type	Description
`getStaticMethods()`	`String className`	`String[]`	Returns a list of all static method names defined by the class with name `className`. This only returns the name of methods and no other information.
`isClassConstructorPrivate()`	`String className`	`boolean`	Returns true if the class with name `className` has only one constructor and that constructor has the `private` access modifier
`getClassStaticPrivateFieldNames()`	`String className`	`String[]`	Returns a list of the names of fields declared by the class with name `className` that have both the `static` and `private` access modifiers. This only returns the name of fields and no other information.
`getFieldNames()`	`String className`	`String[]`	Returns a list of all field names defined by the class with name `className`. This only returns the name of fields and no other information.
`getGlobalNames()`	`String className`	`String[]`	Returns a list of all field names defined by the class with name `className` with the `static` access modifier. This only returns the name of fields and no other information.

Name	Parameters	Return Type	Description
`findCorrectMethodInfo()`	`String className`, `boolean names_and_vars`	`Map<String, List<String>>`	Returns a map containing the names of local variables in methods implemented in the class with name `className`. Setting `var_and_names` to true will mean that the values of each entry in the map will represent the list of local variable names in the method specified by the key. Setting `var_and_names` to false will instead return the types of those local variables instead, duplication of entries is intentionally included. It should be noted that the list values of the two maps (one from each option of `var_and_name`) will have a one-to-one correlation assuming the lists are from the same specified key.
`getClassFieldTypes()`	`String className`	`List<String>`	Returns the de-duplicated list of types used by fields in the class with name `className`
`getInterfacesList()`	`String className`	`List<String>`	Returns a list of all interfaces implemented by the class with name `className`. Unlike `getInterfaces()`, this method will also attempt to parse classes in the classpath and not yet parsed by the `ASMParser`, thus being more extensive and useful for more in-depth analysis of JRE packages

Name	Parameters	Return Type	Description
compareMethodFromInterface()	String className, String methodName, String interfaceName	boolean	Returns true if the return type of methodName is identical between the definition in the interface interfaceName and the implementation in class className. Returns false otherwise. This assumes that className implements interfaceName
getAbstractMethods()	String className	List<List<String>>	Returns a list of methods with return type descriptors in the class className that have the abstract access modifier. The second dimension of this return will have the name of the method at index 0 and the return type descriptor at index 1.
getConcreteMethods()	String className	List<List<String>>	Returns a list of methods with return type descriptors in the class className that do not the abstract access modifier. The second dimension of this return will have the name of the method at index 0 and the return type descriptor at index 1.
getAbstractMethodsInConcrete()	String className, List<String> methodName, List<List<String>> methodList	List<String>	Returns a list of methods from class className that contain a method call to another method that has an abstract access modifier in a superclass but calls a concrete implementation. It should be noted that List<List<String>> methodList should be the return from getAbstractMethods().

Name	Parameters	Return Type	Description
`getSignature()`	`String className`	`String`	Returns the full class signature of the class `className`
`getSignatureNonEnum()`	`String className`	`String`	If the class `className` is not an enumeration, then it returns the class signature of `className`. Returns `null` otherwise
`getMethodCalls()`	`String className`,`String methodName`	`List<MethodCall>`	Returns a list of `MethodCall` objects that contain information about calls to other methods in the method `methodName` in class `className`. `MethodCall` contains data about the "invoker" of a method.
`getFieldTypeNames()`	`String className`	`String[]`	Returns a de-duplicated list of types used by fields in the class `className`
`getAllMethodReturnTypes()`	`String className`	`String[]`	Returns a de-duplicated list of return types used by methods in the class `className`
`getAllMethodParameterTypes()`	`String className`	`String[]`	Returns a de-duplicated list of types used by method parameters in the class `className`
`getAllMethodBodyTypes()`	`String className`	`String[]`	Returns a de-duplicated list of return types used by method calls in method bodies of the class `className`
`getAllMethodLocalTypes()`	`String className`	`String[]`	Returns a de-duplicated list of return types used by local variables in method bodies for the class `className`
`getExtendsImplementsTypes()`	`String className`	`String[]`	Returns a list of classes and interfaces that the class `className` extends or implements

Name	Parameters	Return Type	Description
`isInterface()`	`String className`	`boolean`	Returns `true` if the class `className` is an interface
`isEnum()`	`String className`	`boolean`	Returns `true` if the class `className` is an enumerated type
`isFinal()`	`String className`	`boolean`	Returns `true` if the class `className` has the `final` access modifier.
`allMethodsStatic()`	`String className`	`boolean`	Returns `true` if all methods in class `className` have the `static` access modifier.

Utility Methods in `ASMParser`

The `ASMParser` class contains a few `private` methods to be used within the class to reduce code duplication. Below is the list of those methods:

Name	Parameters	ReturnTypes	Description
`getMethodNode()`	`String className`, `String methodName`	`MethodNode`	Returns the corresponding `MethodNode` object for the method `methodName` in class `className`
`getLocalVarContext()`	`MethodNode method`, `int index`	Returns a map of instruction indexes to `LocalVariableNodes`. This is primarily used in `getMethodCalls()`, which can be used as an example usage	

SRS/SADS

Requirements

By Milestone 4, DesignLint will have the following functionality

- Take a path to a Java class or directory as a command line argument to perform analysis on. If the path is a directory, it will analyze all Java classes within that directory. (*IMPLEMENTED PRIOR TO M2*)
- Have 12 analyzers that output errors, warnings, information (which is either generic or related to detecting a design pattern). (*IMPLEMENTED PRIOR TO M2*)
- Have an extensible Analyzer Structure that allows adding new Analyzers in modified builds (*IMPLEMENTED PRIOR TO M2*)
- Have a flag/parameter system that is flexibly accessible by many interfaces (*IMPLEMENTED IN M2*)
    - Have a flag to only enable certain analyzers to run on the input files. If no such flag is provided, all analyzers will run (*IMPLEMENTED IN M3*)
    - Have a flag to toggle the amount of output (verbosity) that is sent to output (*IMPLEMENTED IN M4*)
    - Have a flag to redirect program output to a file (*IN PROGRESS*)
    - Have a flag to invoke a help display on how to use the program (*IMPLEMENTED IN M4*)

    - If the program detects incorrect arguments, it will display the help output and terminate (*IMPLEMENTED IN M4*)

Architecture UML

The following serves as a generalized UML diagram of the architecture of the linter:
arch-diag

Project Hierarchy/Dependencies

This project uses Maven for build, testing, and deployment. We also have CI actions that automate the build and testing process on commits/pull requests to the `main` branch. The project is dependent on the following libraries/Maven Plugins for compilation and testing:

- OW2 ASM (Compilation): Provides Java Bytecode Analysis functionality used by the linter
- JUnit 5 (Testing): Unit Testing Framework
- Maven Surefire Plugin (Testing): Maven Plugin used to run JUnit tests
- PITest Plugin w/ JUnit 5 Extension(Testing): Maven Plugin used to generate Mutation Testing and Test Coverage Metrics

The following details the general source tree hierarchy:

- `src/main` contains all the files that will be available in the final binary
    - `LinterMain.java` is a basic CLI Wrapper that interfaces with the main Linter architecture
    - `presentation` Package contains code that serves as the interface between analyzers and a wrapper. This is done in `PresentationLayer.java`
    - `domain` package contains code related to defining analyzer logic. All analyzers will extend the `DomainAnalyzer` class which contains abstract template methods to define specific analyzer functionality.
        - `domain.analyzer` package contains the concrete implementations of analyzers
    - `datasource` package contains code that provides an adapter from the ASM library to the analyzers. The `ASMParser` class serves as this adapter and is used by all analyzers
- `src/test` contains all the relevant code for testing
- `target` is the output directory from builds

Testing Suite

Tests are created for every concrete analyzer class. These tests all follow a similar pattern and utilize the features of JUnit 5. Tests utilize the `@BeforeEach` annotation to initialize necessary items before each test method. This initialization runs automatically before every method and includes code that initializes the analyzer, parser, and any other relevant objects to the analyzer test. Parameterized tests are utilized in several test classes to reduce code duplication and run many tests with different parameters. This ensures that the developer does not have to write similar test methods many times. By utilizing these features, developers can easily add more tests where necessary.

Testing Strategies choices

- For unit testing, we have tests for the analyzers that cover all possible outputs that the system is able to produce. This is captured in 160 tests that we have implemented in the system. Our unit testing can also be scoped through checking the return after a specific parsed input. Additionally, we can test the analyzer logic itself.

- Integration testing is not directly implemented in our system, but can be done thought analyzer unit testing.

- Acceptance testing can be achieved by invoking the command line and manually checking the results.

- Performance testing is not directly tested as the linter is not designed to check the load, stress, or efficiency of the code.

License

Because this program uses OW2 ASM for bytecode-level analysis, we include the following 3-Clause BSD license declaration for ASM: