Milestone 4 – Final Documents

Team 10 – Nathan Adler & Campell Garvin

4/25/24

Contents

Introduction

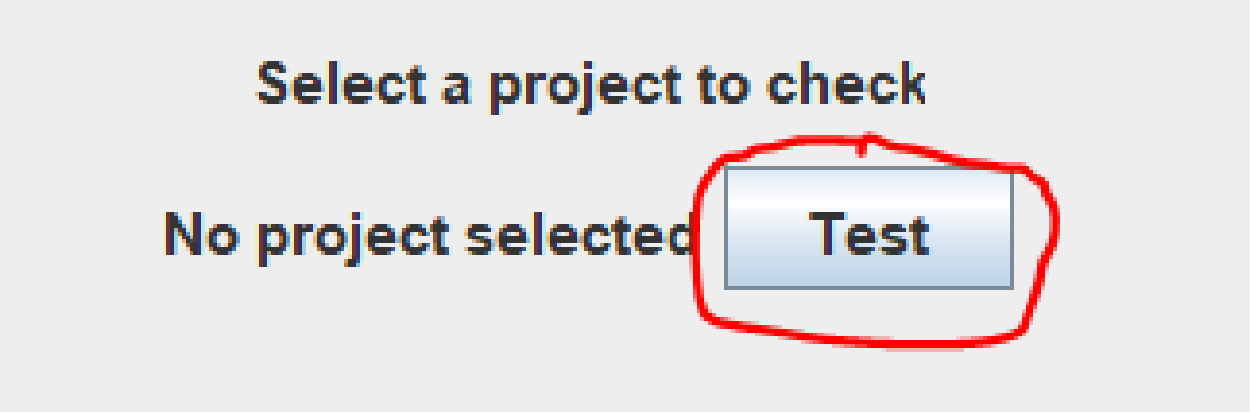
This document serves as a guide to the LintCheck software system. The LintCheck system is a software made to analyze the design paradigms of Java .class files. The system performs a variety of style, principle, and design checks to inform the user of design violations. The purpose of the system is to aid other developers in finding issues in the design of their Java code. This document contains User, Installation, and Maintenance Guides, Requirements and Architecture specifications, and a test plan description.

User Guide

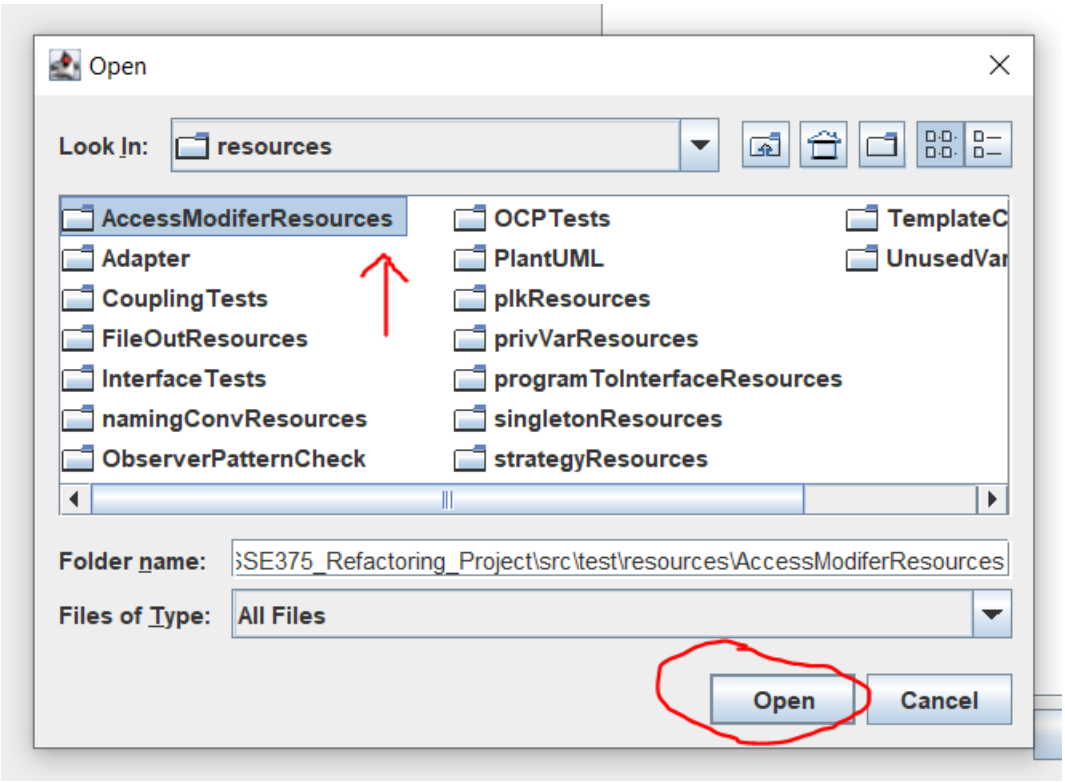
1. After following the installation guide to create the "LinterProject.jar" file, run this file to start the program.

2.Once launched the following window should appear:

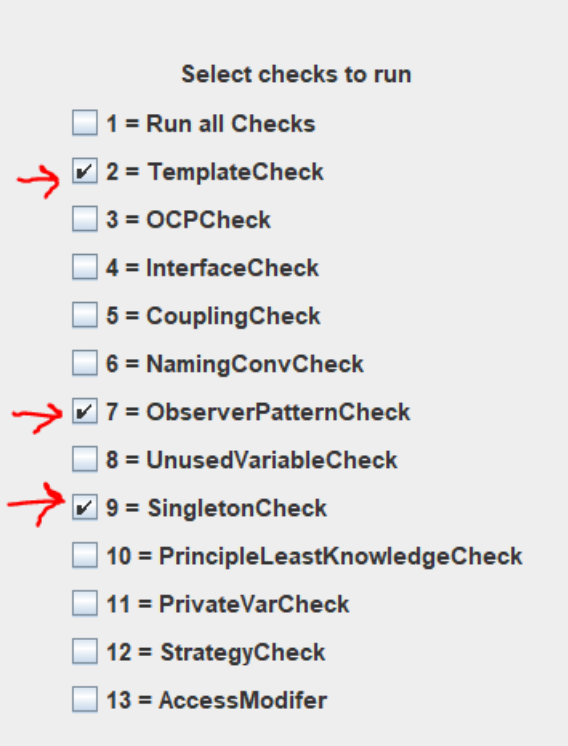
3. Next, click on the “Test” button to select the package containing Java .class files to run checks on.



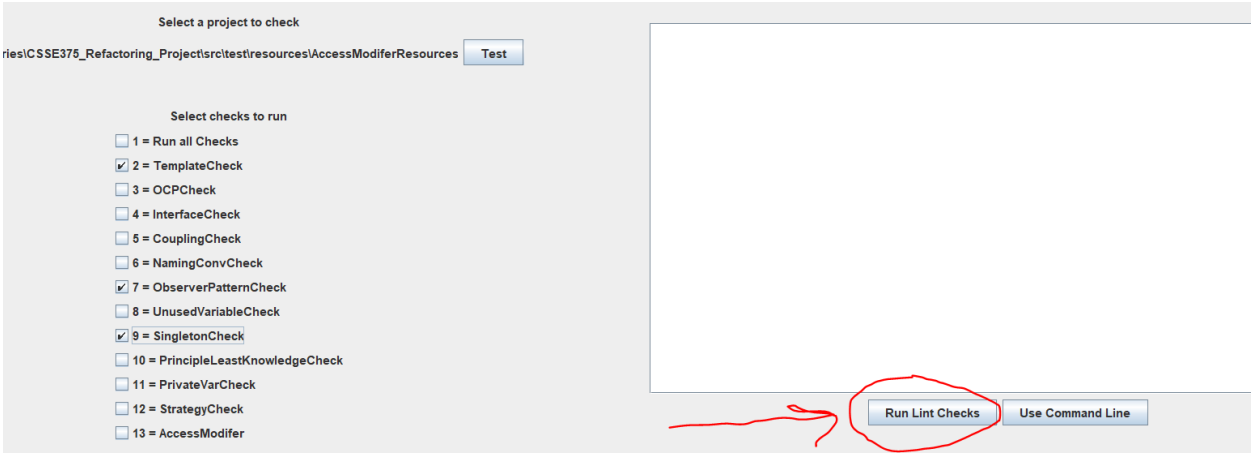
4. Navigate through your device’s file system to the desired package. Click on the package and select open.



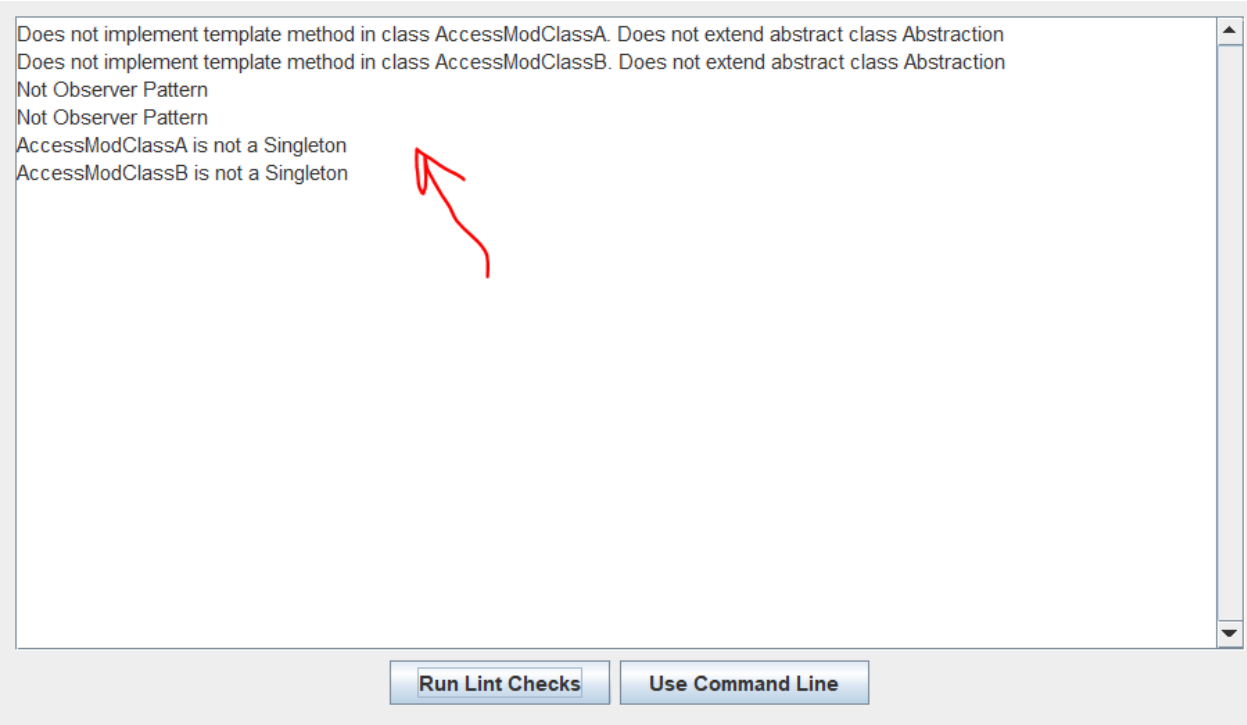
5. Next, click on the check boxes to select the checks to run, in this example checks 2, 7 and 8 are selected.



6. Now, click “Run Lint Checks” to run the lint checks on the selected files.



7. The lint checks will run and display messages in the dialog box above the buttons. If a large number of files or checks are used, you may need to scroll down to see all messages.



8. These messages as well as changed files from checks that alter files can also be found in the "files" folder in the form of a CSV file in the same directory as the .jar file.

9. An additional note on running the program through the "Use Command Line" button: you must launch the application through a command line to enable this functionality. The command line dialog will guide you through the same process as this guide.

Installation Guide

We are assuming that a version of Java exists on the system. Since our program operates on existing Java projects in development, we do not think it would make sense for a user to download this system without a way to create or modify Java files.

1. Unpack the compressed zip file.
2. Open a new terminal window.
3. Navigate to the Linter directory.
4. Run the command `java -jar ./LinterProject.jar .`
5. To enable AI-integrated checks, modify the file `files/key.txt` to contain only an OpenAI secret key.
    a. Go to https://platform.openai.com/docs/overview and either create an OpenAI account or sign in with an existing one.
    b. Navigate to API keys in the sidebar and click the button to create a new secret key.
    c. Continue with the prompts provided until you generate a secret key. This will be a long string of characters starting with "`sk-` ".

Maintenance Guide

This guide is designed to provide instructions for maintaining the functionality and performance of the software. The guide offers perspectives for helpdesk, technical support, and maintenance contributors. The bulk of this sections contains information for maintainers on the development environment and tools, dependencies, planned changes for future updates, common sources of change, where to implement new functionality, and other answers to maintenance related questions.

Helpdesk

The role of the helpdesk will primarily be to aid in the use of the system by the users, any technical issues or system defects should be reported to the technical support staff so they can log and convey the issues to maintenance. The main issues that the helpdesk will cover are installation and use issues encountered by users. Both issues can be addressed with the use of the Installation Guide and the User Guide.

Technical Support

The role of technical support will be to aid in deeper issues with the system itself, including bug reporting, linting insufficiencies, and possible compatibility problems in addition to others. The main issues that will need to be addressed by technical support will be file loading issues and check issues.

1) File Loading errors: If the system has issues loading a user-specified file, ask the type, java version, and relative location of the file to the executable. Issues may be caused by attempting to load a file the system does not have access to in which case it should be moved to a suitable location. There may also be issues with loading older versions of java .class files in which case the user will need to transfer to a version of at least JDK 8 or newer. For any further issues, there may be issues in the system that need to be resolved by maintainers, if this is the case collect the necessary information as well as the .class file (if possible) and submit a bug report.

2) Check issues: If the user is having issues with checks, ask them for the checks they are running and the contents of the file they are trying to run a check on. If a user is having trouble understanding the result of a check, communicate the purpose and the function of the check in question. If the user is having issues with getting the correct result from the check collect the .class file if possible and attempt to resolve the issue with the client, if the issue lies in the check itself, submit a bug report to the maintainers with the aforementioned .class file. If the user believes there's a valid issue with a check communicate the concern to the maintenance team for resolution. Keep in mind that the scope of the checks is limited and that not all valid java classes will generate useful linting results depending on the implementation and syntax of user classes.

3) AI check issues: If the user is having issues specifically with the checks incorporating ChatGPT, ensure that they can successfully run other checks. If they can and the issue only arises from the AI checks, ensure that the user has input a working OpenAI secret key in the file `files/key.txt`. If the user is still having issues, allow the system time to process the HTTP request to OpenAI. This process should take longer than a traditional check, but it should never take longer than a few minutes. The progress of this request can be monitored if the user initiates the program through the terminal. If the user believes that the HTTP request is returning unsuccessfully, a .class file may be unable to be read by ChatGPT. In this case, please submit a bug report to the maintainers with the invalid .class file.

Maintainer

The role of the maintainer will be to preserve and maintain the system's functionality, resolve bugs, and introduce new features to the system. The guide to introducing changes in addition to other important artifacts and considerations will constitute the rest of the maintenance guide.

Environment and Tools

Tool	Name	Version
Development Environment	Visual Studio Code IDE	v1.88.1
Language	Java	JDK 8 (build 1.8.0_202-b08)
Language Support	Redhat	v1.30.0
Change Management and Version Control	GitHub	n/a
Dependency Management	Maven	v0.44.0
Continuous Integration	Maven	V0.44.0
Collaboration Tools	Jira (Scheduling) MS Teams (Communication)	n/a

Dependencies

The LintCheck system has several outside dependencies that are essential to the function of the system. Adding or modifying dependencies can be done in the pom.xml file. The existing dependencies are outlined below.

ASM

Version 9.2

ASM is a bytecode manipulation and analysis framework. It's used to view, modify, and create classes in binary. In the system ASM is used to parse input class files into "ClassNode" objects so they can be analyzed and modified by the various lint checks. The classes that rely on ASM are ASMAdapter class where it's used to parse .class files and the classes in the model package which wrap the "nodes" in model objects to be used by checks.

OpenCSV

Version 5.6

OpenCSV is a CSV file parser library used to read and write from CSV files. In the system, it's used by the FileOutput class to save linting results to a file.

JUnit Jupiter

Version 5.8.2

JUnit is a testing framework for testing locally on the JVM. In the system JUnit is used throughout the testing package to make assertions and verify behavior.

TestNG

Version 7.9.0

TestNG is a testing framework that adds various functionalities to JUnit. It's also found throughout the testing

package where JUnit is used.

Mockito

Version 3.12.4

Mockito is a mocking library that's used to mock components to break dependencies for testing. It's used primarily in model tests to mock the ASM nodes that the models recreate.

Planned Changes

LintCheck Web App

Currently the only planned change is the expansion of the system into a fully web-based application. There is already a web front interface in the WebUserInterface class that can only be run on a local machine that serves itself by accessing local files.

Implementing the system as a web app would overcome the main issue of the software, that it can only run checks on files in the local system. A web app would allow users to run checks class files without having to install the software making the system more accessible and easier to use.

There are several steps that would have to be taken to implement the web app including serving, databases, file transfer, security, and refactoring for a web-based implementation.

Common Sources of Change

Changes to Java Language

As the system is based around analyzing and changing Java class files, any changes made to the syntax, representation, or standards of Java will impact the functionality of the system. Checks may need to be updated to maintain behavior when searching for different patterns in the bytecode or making changes to the representation.

Changes to ASM

Since the model classes wrap ASM nodes the logic in the models are heavily dependent on the ASM representation. Any changes made to ASM may affect the functionality of the models and checks. Models will need to be maintained with new versions of ASM to ensure the behavior remains consistent.

Changes to APIs

There are multiple checks that utilize the ChatGPT API to perform checks, because of this they are sensitive to changes to the API. When the API changes, there will likely need to be changes made to how the system communicates with the API to prevent issues from arising.

Troubleshooting Guide

1. ClassNotFound Errors

(Example Trace)

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/objectweb/asm/ClassVisitor
    at analysis.CCMetric.main(CCMetric.java:5)
Caused by: java.lang.ClassNotFoundException:org.objectweb.asm.ClassVisitor
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
```

A common issue that might be experienced when adding new functionality using the ASM library are ClassNotFound errors.

A class not found error can happen when accesses or testing a class that has a dependency to another user-defined class.

To resolve this issue, ensure that dependencies are inside the analyzed package, if this does not fix the error check to make sure they appear before the class they are referenced in. There are other less common reasons for this error as well, for these its recommended to look at the official ASM documentation and forums.

2. ASM Implementation Issues – Tree Nodes and Types

ASM is a complex bytecode manipulation library and attempting to adapt nodes for functionality in the system can be difficult. There are two main ideas to understanding the library for its uses in the system, the tree hierarchy, and the type annotations. When parsing a class into the system, the class node and its subservient elements are wrapped in matching model classes to add "intelligence" to enable functionality with the linting algorithms. Some nodes may not act in expected ways and because of this there is no one size fits all solution, instead its recommend to read the ASM tree documentation: https://asm.ow2.io/javadoc/org/objectweb/asm/tree/package-summary.html. Additionally for issues determining structure and types of .class files, the following graphics provided by ASM of the JVM specification may prove useful:

Compiled Class Structure:

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

Type Descriptors:

Java type	Type descriptor
`boolean`	`Z`
`char`	`C`
`byte`	`B`
`short`	`S`
`int`	`I`
`float`	`F`
`long`	`J`
`double`	`D`
`Object`	`Ljava/lang/Object;`
`int[]`	`[I`
`Object[][]`	`[[Ljava/lang/Object;`

Software Requirements Specification

The following section contains a breakdown of the software needs, features, and requirements used for the system. Additionally, it outlines the design purpose, quality attributes, constraints, and concerns for the system.

Needs

1. Users can select a .class file from the local system to run lints on
2. Offers multiple principle, cursory, and pattern style checks.
3. Can parse a .class file into a PlantUML representation.
4. Can check code with the assistance of AI
5. Users can get feedback from the checks in a GUI or file

Features

1. Multiple Cursory Style Checks
2. Multiple Principle Style Checks
3. Multiple Pattern Style Checks
4. Automated UML production
5. Multiple Interactive GUI with feedback
6. Generates a file with check feedback
7. AI assisted Checks
8. Alteration of class files

Functional Requirements

R1	The system should be able to read Java .class files from a folder.
R2	The system should have the ability to analyze Java Classes and provide feedback.
R3	The system should be able to produce a file containing feedback.
R4	The system should be able to generate AI-assisted feedback.
R5	The system should have the ability to generate a UML diagram of classes.
R6	The system should have the ability to alter and save Java .class files

Non-Functional Requirements

R7	The system should have an interface that is easy to use.
R8	The system should have an interface that has a quick response time.
R9	The system should have the ability to run multiple checks on multiple classes (5 classes, <300 lines) in under 1 minute

Design Purpose

The purpose is to create an exploratory prototype of a brownfield system based in a mature domain.

Quality Attributes

- Low Latency: All checks shall be identified and presented to the user within 1 second of the error occurring.

- High Learnability: The system shall be able to be learned in less than 1 hour.
- Visible Feedback: Each check shall be clear and visible such that the user can identify the error within 1 minute of it occurring.
- Stable and Flexible Logic: The system shall be flexible such that any new feature can be implemented in less than 8 hours.

Constraints

- Project Timeline
- Feature feasibility

Concerns

- Refactoring established code base
- Allocation of functionality to systems
- Organization of the code base
- Meeting internal functionality requirements
- Integrating webserver functionality

Software and Architecture Design Specification

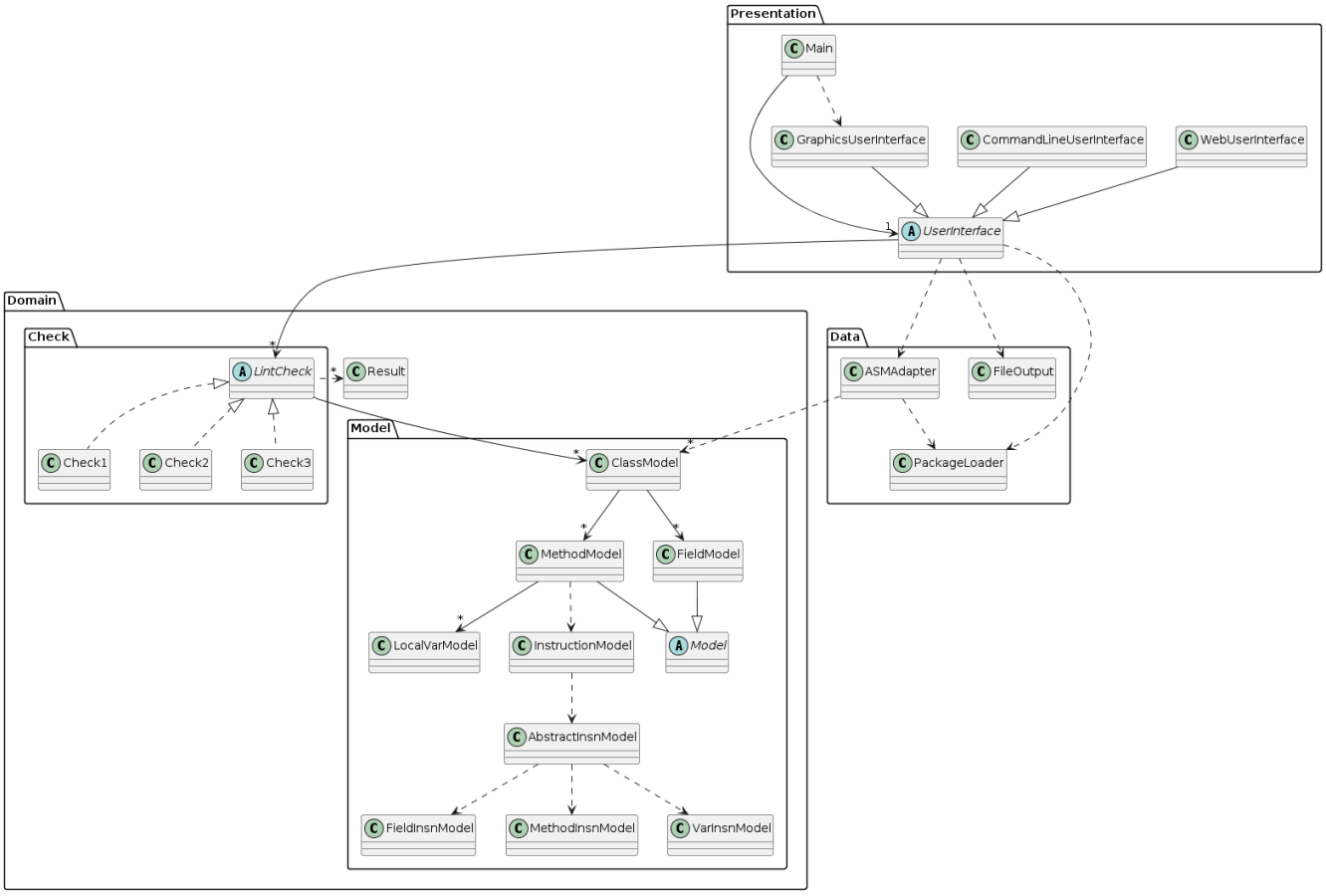
The following section contains the reference architecture for the system, a high-level look at the class diagram of the system, a detailed list of the various lint checks and their functions, and a record of the design decisions and changes made during each milestone.

Reference Architecture

- Reference Architecture: Rich Client
- Presentation Layer: Command Line, Java graphics, and Web User Interfaces; handles user inputs, outputs lint analysis.
- Domain Layer: Handles all linter checks. Has Models to wrap ASM objects used by checks. Accumulates result objects for each check.
- Data Source Layer: Handles any possible data loading/saving required by the operations in the domain layer as well as file parsing.

High Level Class Diagram

This diagram represents the high-level relations between each component of the system. Many relations have been removed to clarify the overall relationships between different classes. A fully detailed class diagram can be found in the GitHub repository at https://github.com/rhit-adlerbn/CSSE375_Refactoring_Project/blob/main/design.puml



Description of Lint Checks

Style Checks	
NamingConvCheck	Checks linted classes for naming of class, methods, and fields. Class names are checked for a pascal pattern while methods and fields are checked for a camel-case pattern.
InterfaceCheck	Checks if classes that implement an interface contain all the interface's methods.
PrivateVarCheck	Checks a collection of classes for non-privatized fields that could be made private. (See Also: AccessModifer).
UnusedVariableCheck	Checks linted classes for unused variables inside of methods. These variables are those that are declared but not used inside the method.
Principle Checks	
OCPCheck	Checks a linted class for adherence to the Open-Closed Principle, that classes should be open for extension, but closed for modification.
PrincipleLeastKnowledgeCheck	Checks a collection of classes for adherence to the Principle of Least Knowledge, that classes should interact with other classes in well-defined way. Asserts that classes don't operate on fields of other classes directly.
CouplingCheck	Checks linted classes for how closely coupled they are to other classes. Generates a numerical score based on the number amount of non-primitive references made in the class.
Pattern Checks	
SingletonCheck	Checks a linted class for the presence of the singleton design pattern. Searches for the presence of a private constructor, private instance of the class, and a public getInst()/getInstance() method that returns the instance.
TemplateCheck	Checks a linted class for the presence of the template design pattern. Checks that a concrete class implements the abstract methods of its super class.
StrategyCheck	Checks a linted class for the presence of the strategy design pattern. Checks that a class does not call methods on a concrete instance of an interface instead of the interface.
ObserverPatternCheck	Checks a linted class for the presence of the observer design pattern. Determines if a class is a subject or observer class based on the implementation of subject/object interface, and existence of subscribe, unsubscribe, and notify methods.
Other Checks	
AccessModifer	Checks a collection of classes for access modifier violations in methods and fields, altering the classes access modifiers as needed and saving the changed .class files. Makes accessed private variables and methods public and privatizes unused and non-externally referenced public variables and methods.
PlantUML	Generates a simple PlantUML UML class diagram of linted classes.
ChatGPTCouplingCheck	Checks linted classes for how closely coupled they are to other classes. Class information is given to ChatGPT so that it can return a message approximating the amount of coupling.
ChatGptObserverCheck	Checks linted classes for an observer pattern. Class information is given to ChatGPT so that it can predict whether each class is a subject, an observer, or neither.

ChatGPTSingletonCheck	Checks linted classes for a singleton pattern. Class information is given to ChatGPT so it can check whether each class contains a singleton pattern.

Milestone 4

External File Capabilities	Allows the user to specify a project path from the local system using a file selector in the GUI rather than requiring them to import it into the project **or** use a textual file path.

Changes: This milestone was focused on resolving remaining issues with the system, setting up continuous integration, finalizing features, expanding testing, and user testing. The majority of the work for this milestone was done in the AI Checks, GUI classes, and testing package. Additionally, the feature list was finalized, and new exception handling was added for user inputs.

Final Feature List:

1. Webfront
2. CSV Output
3. AI Integration
4. Access Modifier Check
5. Additional Custom AI Lints
6. External File Capabilities

User Testing:

Our first round of user testing focused on our GUI. In the first test, the user was asked to run all checks on a specified CouplingTest project. The user had no issues completing the task but recommended that we use the JFileChooser Swing component instead of having the user manually type the file path. The second user was asked to run tests 2 and 5 on the same project. The user was confused on which directory to use as the root in the pathname and the user mistyped test 3 instead of test 2. Based on these tests, we decided to improve the UI by implementing the JFileChooser as suggested. This would reduce the confusion users felt while selecting a file path. Additionally, we created a list of checkboxes for uses to select their checks. This limited the possibility of users mistyping a check number.

Exception Handling:

Previously, the user was able to run the linter without selecting a project or without selecting specific checks to run. With the GUI overhaul, we were able to implement error checking for these scenarios. Now, when the user attempts to run the linter without providing critical information, the linter will show a pop-up window telling the user to input the necessary information.

```
private static JButton createRunButton() {
    JButton runLinter = new JButton( text: "Run Lint Checks");

    runLinter.addActionListener(e -> {
        if (selectedFile == null) {
            JOptionPane.showMessageDialog(mainPage,  message: "Please select a project to check!",
                    title: "Linter", JOptionPane.ERROR_MESSAGE);
            return;
        }
        else if (checksToRun.isEmpty()) {
            JOptionPane.showMessageDialog(mainPage,  message: "Please select the checks to run!",
                    title: "Linter", JOptionPane.ERROR_MESSAGE);
            return;
        }
        resultText.removeAll();
        latch.countDown();
    });

    return runLinter;
}
```

Milestone 3

Access Modifier Check	A check that extends the LintCheck interface. Searches through classes to find access modifier violations. Reports found access modifier issues in methods and fields. Resolves issues by altering bytecode of original class files then converts the bytecode back into .class files, saving the changed file.
Custom AI Lints	A modification of the AI linter checks that more easily allow for custom prompts. Defers the lint checking process to a new overarching interface that deals specifically with these AI checks. The AI checks also conform to the output structure from the previous features and have additional error handling

Changes

The main changes for this milestone were made in the domain layer, specifically in the various model components where new methods had to be added to check and modify accessor parameters. Additionally, there were changes made in the AI Linter checks where the ChatGPTCheck was converted to an abstract class which several classes extend. Work was also done in the data source layer for file output and refactoring to add new tests.

Refactorings

1. Shotgun Surgery in AbstractInsnModel, FieldInsnModel, and MethodInsnModel
    a. Description: All three classes use similar functions of AbstractInsnNode, but any changes to this functionality would require changing all three classes.
    b. Solution: Make MethodInsnModel and FieldInsnModel extensions of AbstractInsnModel
2. Shotgun Surgery/Primitive Obsession in LintCheck child class results
    a. Description: All the lintchecks handled the output of results by building up and returning strings that described the result of the check in different ways.

   b. Solution: centralized and encapsulated message generation with new Result.java which tracks which check was run, which class it was run on and the output of each check.
3. Large Method/Duplicated Code in AccessModifer isAccessed() method
   a. Description: The AccessModifer class (originally privatizer) was non-functional, and had large, duplicated methods that assessed if a method or field were accessed.
   b. Solution: Extracted the central logic of the algorithm out and added a common interface for MethodModel and FieldModel, then used polymorphism to avoid duplication. This was part of a larger effort to implement a functional AccessModifer class and test it.

Milestone 2

Web Front	A websocket based java server that extends the new UserInterface class. The server serves a HTML page to the client over a local network allowing them to run lint checks from a browser. This feature was tested using exploratory tests and postman to verify HTTP requests.
CSV Output	A file writer implemented with the use of OpenCSV library to write linting results to a CSV file. A new test class was created for this class to verify the file output contained the correct information.
AI Integration	An AI-based linter checks that queries ChatGPT for an analysis of a codebase's coupling levels. The check implements the LinterCheck interface, which is designed to output a List of Strings, one for each class in the package. This is realized in the ChatGPTCheck by asking ChatGPT to analyze each class through an HTTP request. The response is parsed and output by the overridden checking function.

1. Code Duplication in Presentation Package
   a. Description: Multiple classes in the Presentation Package shared code
   b. Solution: Extracted Duplicated Code to a common interface (UserInterface)
2. Long Methods in Presentation Package
   a. Description: Multiple classes in the Presentation Package had excessively long methods
   b. Solution: Extracted Methods and delegated responsibility to the superclass
3. Inappropriate Intimacy in ASMAdapter
   a. Description: Classes could access any information about ASMAdapter freely
   b. Solution: Remove static references and force classes to create instances
4. Long Methods in ClassModel
   a. Description: The constructor of ClassModel was unnecessarily long and confusing
   b. Solution: Extract some of the code into methods with descriptive names
5. Dead Code in CouplingCheck
   a. Description: Some code in Coupling Check had no impact on the success of the program
   b. Solution: Safely delete the unused code
6. Long Methods and Temp Field in Template Check and Interface Check
   a. Description: The linting functionality for both classes was held entirely within one method. Additionally, Template Check created the same temp variables in every loop.
   b. Solution: Extract code into more descriptive helper methods and create instance fields.

Milestone 1

Candidate List of Features

- Webfront
- HTML Output
- Custom Lints
- Code autocorrect
- Live analysis
- Ai integration

Prioritized List of Features

1. Webfront
2. HTML Output
3. Ai integration
4. Code autocorrect
5. Live analysis
6. Custom Lints

Potential Code Smells

1. Code duplication in model classes opcode checks and class information algorithms in lint checks
1. Long methods/Large Class in Model (ClassModel and MethodModel)
2. Dead Code (FieldModel, CouplingCheck)
3. Long Method:
    1. InterfaceCheck, CouplingCheck, NamingConvCheck, ProgramToInterfaceCheck, TemplateCheck lintCheck Methods
    2. PlantUml.CreateUML
    3. ObserverPatternCheck.checkObserverPattern
    4. PrivateVarCheck.findViolations
    5. PrincipleLeastKnowledgeCheck.classLevelCheck
    6. UnsedVariableCheck.printVariables
4. Potential Inapropriate Intamacy between models and checks
5. High Coupling between ASMAdapter and PackageLoader
6. Code Duplication in graphicsUI and comandlineUI as well as large main methods for both

Test Plan

As part of the original use case, a test suite was already developed to monitor the functionality of the linter checks and the system. Therefore, the code passed the test cases before any refactoring. After refactoring, the system remained completely operational.

For new features, test cases were added where necessary to verify new behavior. Additionally, some test cases were added for untested existing functionality, and some were modified to remove dependencies that interfered with component tests.

The lint checks were tested as integration tests of the system because it was not possible to accurately mock the level of bytecode information needed in each node to get an accurate assessment of the linting algorithms. For each check, test classes were written with the desired format that was being checked and compiled into .class files. These files were then loaded into classModels and passed to the tests in the way the system would regularly function, then assertions were made on the results of these checks.

A particular sticking point was creating tests for the AI linter checks. Since we use generative AI to generate responses, the output of these checks was deterministic, so traditional, comprehensive tests could not be used effectively. We implemented two solutions. In the first solution, the tests would check to make sure that the number of checked classes and the number of responses were equal, but this did not cover the specifics of the response. In the second solution, the tests would run the responses through another instance of ChatGPT so that the response could be interpreted into a one-word answer, but this also used generative AI. While this improved test accuracy, the results were still deterministic. Therefore, we defaulted to using the first solution while providing the option to use the more advanced second solution.

Test Name	Test Description	Result
Data Source		
parseAsm_ValidPath_ExpectSuccess	Verify that parseASM() can parse a given file into a classModel.	Pass
fileOut_SaveResults	Verify that saveResults() can save a list of string arrays to a CSV file.	Pass
fileOut_SaveClass	Verify that saveClass() can save a classModel to a .class file.	Pass
loadPackage_ExpectSuccess	Verify that when given a valid file path, loadPackage() can load a package of files int a list of byte arrays.	Pass
loadPackage_ExpectError	Verify that when given an invalid file path, loadPackage() throws an IOException.	Pass
Lint Checks		
accessModifierTest_ClassAChanged	Verify that when linting a collection of classes with access modifier violations, the check identifies them, resolves them, and saves the resolved class files.	Pass
TestNoCoupling	Verify that when linting a collection of uncoupled classes, the check identifies a coupling score of 0.	Pass

TestSomeCoupling	Verify that when linting a collection of low coupled classes, the check identifies a low coupling score of <3.	Pass
TestHighCoupling	Verify that when linting a collection of highly coupled classes, the check identifies a coupling score of >10.	Pass
interfaceTest_ExpectWarning	Verify that when linting a class with unimplemented interface methods, the check identifies and specifies the methods that are unimplemented.	Pass
interfaceTest_ExpectNoWarning	Verify that when linting a class with all interface methods implemented, the check identifies that the methods are implemented.	Pass
namingTest_ expectWarning	Verify that when linting a class with a poor class name, method name, and field name conventions, the check identifies the poorly named constructs.	Pass
namingTest_ expectNoWarning	Verify that when linting a class with good naming conventions, the check identifies correctly named constructs.	Pass
TestObserver	Verify that when linting a class with the “Observer” pattern, the check correctly identifies the pattern in that class.	Pass
TestSubject	Verify that when linting a class with the “Subject” pattern, the check correctly identifies the pattern in that class.	Pass
TestNothing	Verify that when linting a class without the “Observer” or “Subject” pattern, the check identifies that there is no pattern in that class.	Pass
OCPTest_ExpectWarning	Verify that when linting a class with final methods, the check identifies that the final methods may constitute an OCP violation.	Pass
OCPTest_ExpectWarning2	Verify that when linting a class declared as final, the check identifies that the class may constitute an OCP violation.	Pass
OCPTest_ExpectNoWarning	Verify that when linting a class that does not appear to violate OCP, the check does not identify any potential OCP violations.	Pass
TestSimple	Verify that a simple class (no dependencies) is accurately parsed into PlantUML code.	Pass
TestExternal	Verify that a class with a single dependency is accurately parsed into PlantUML code.	Pass
TestComplex	Verify that a more complex class with multiple methods and dependencies is accurately parsed into PlantUML code.	Pass
plkTest_Fail	Verify that when linting classes that access each other fields, the check identifies the fields that are inappropriately accessed.	Pass

plkTest_Pass	Verify that when linting classes that have no PLK violations, the check correctly identifies the lack of violations.	Pass
privVarTest_Fail	Verify that when linting a class with a unused public variable, the check identifies that the variable is not used but public.	Pass
privVarTest_Pass	Verify that when linting a class with no private variable violations, the check identifies the lack of violations.	Pass
singletonTest_notSingletons	Verify that when linting classes without the “Singleton” design pattern, the check correctly identifies that the classes are not singletons.	Pass
singletonTest_Singleton	Verify that when linting a class with the “Singleton” design pattern, the check correctly identifies that the class is a singleton.	Pass
strategyTest_Fail	Verify that when linting a class that references a concrete instance of an interface, the check identifies the reference as a potential “Strategy” pattern violation.	Pass
strategyTest_Pass	Verify that when linting a class that references interfaces, the check identifies the class as having no “Strategy” pattern violations.	Pass
TemplateTest_ExpectWarning	Verify that when linting a class that does not extend an abstract class or implement methods from an abstract class, the check identifies the class as not having the “Template” pattern.	Pass
TemplateTest_ExpectNoWarning	Verify that when linting a class that extends an abstract class and implements its abstract methods, the check identifies the class as having the “Template” pattern.	Pass
TestUnused	Verify that when linting a class with unused variables, the check identifies and references the unused variables.	Pass
TestUsed	Verify that when linting a class with no unused variables, the check identifies the lack of unused variables.	Pass
Models		
testClassModel	Verify the adapter functionality of ClassModel’s methods, ensuring that they properly adapt ClassNode methods.	Pass
testFieldModel	Verify the adapter functionality of FieldModel methods, ensuring that they properly adapt FieldNode methods.	Pass
testMethodModel	Verify the adapter functionality of MethodModel’s methods, ensuring that they properly adapt MethodNode methods.	Pass
AI Checks		

TestCouplingOutput	Verify that the ChatGPT Coupling check can detect multiple levels of coupling. Since we are testing against ChatGPT's output, this test is deterministic. We can mitigate this somewhat by using ChatGPT to interpret the output.	Deterministic
TestObserverOutput	Verify that the ChatGPT Observer check can detect observer patterns in code. Since we are testing against ChatGPT's output, this test is deterministic. We can mitigate this somewhat by using ChatGPT to interpret the output.	Deterministic
TestSingletonOutput	Verify that the ChatGPT Coupling check can detect singleton instances in code. Since we are testing against ChatGPT's output, this test is deterministic. We can mitigate this somewhat by using ChatGPT to interpret the output.	Deterministic