Legacy Code Repair Using LMM Generated Acceptance Criteria

Jessyca Ferreira <jfs7>
Lucas Lins <lll2>
Paulo Sérgio <psgs>
Rafael Moura <rnm4>

Motivation

Motivation

- **Wicked Problems**: These are challenges that are extremely difficult/impossible to solve
  - There is no single correct answer, an unlimited number of approaches can be taken, and requirements are not well defined

- **Modernizing Legacy Systems**: Upgrading outdated software has become a widely recognized wicked problem, resisting decades of well-intentioned initiatives and policies [1]

Motivation

- **Security Risks**: Legacy systems are highly vulnerable to security threats due to their outdated implementations
- **Maintenance Challenges**: These systems are difficult to maintain and upgrade over time.
- **Largely prevalent** : Many U.S. federal government legacy systems are 30 to 60+ years old [1]
- **Operational Impact:** Outdated systems create major efficiency and staffing challenges, in addition to the security ones

Motivation

- Traditional approaches to converting legacy code into modernized analogues either involve highly specialized rule-based refactoring, static pattern matching, or complete system rewrites

- There is a cautiousness brought on by the risks that can be introduced by even the smallest of changes, which pose a great challenge to modernizing critical systems

Motivation

- To validate the modernization, specially considering the criticity of most legacy systems, there needs to be a robust criteria

- A robust test set might be the answer, and LLMs may also help both with the task of the test suite generation as well as the modernization itself

Study setting

Settings

- Given the complexity of many real in-use legacy systems, we had to choose representative systems in their steads

- The projects chosen are in their majority small projects of reduced complexity levels
    - They were, however, written by humans with legacy versions of Java and may grant us insights that could be generalized to a larger-scale version of the problem

- The systems chosen were Hipparcos [2] and F-sharp [3]

Settings

- **Hipparcos**: A series of the European Spacial Agence tools written originally in 1997 and revisited, later, 9 years ago*. The **plot** module of the tools were chosen to be modernized: it is responsible for generating graphs based on input data and it implements POO concepts.

- **F-sharp**: Music Library Manager with native *MySql* database connection written 16 years ago*. It also allows us to test how the LLMs deal with outdated libraries

Settings

RQ1 **How effective are LLMs at generating acceptance criteria for code repair?**

RQ2 **How well do the APR capabilities of LLMs work on legacy code?**

RQ3 **How do the tests affect the repair process?**

Settings

- Reasoning models are described as better suited for complex tasks such as mathematics, puzzle solving and coding, therefore, our chosen models all belonged to this class

- Chosen models:
    - ***Gpt 3-o***
    - ***Deepseek R1***
    - ***Gemini Flash Thinking***

- Default parameters used for all

ChatGPT

deepseek-R1

Gemini

Method

Workflow

Generation of acceptance criteria

Utilization of LLMs for repair

Iterative validation and refinement of the generated patches

Generation of acceptance criteria

PROMPT

Analyze the code below, including its purpose, inputs, and expected outputs. Then, generate a detailed natural language description of all the necessary scenarios that verify whether the code functions correctly and meets its intended requirements. Consider that the code will be transformed for the Java 17 language.

SCENARIO

GUI Initialization and Component Verification:

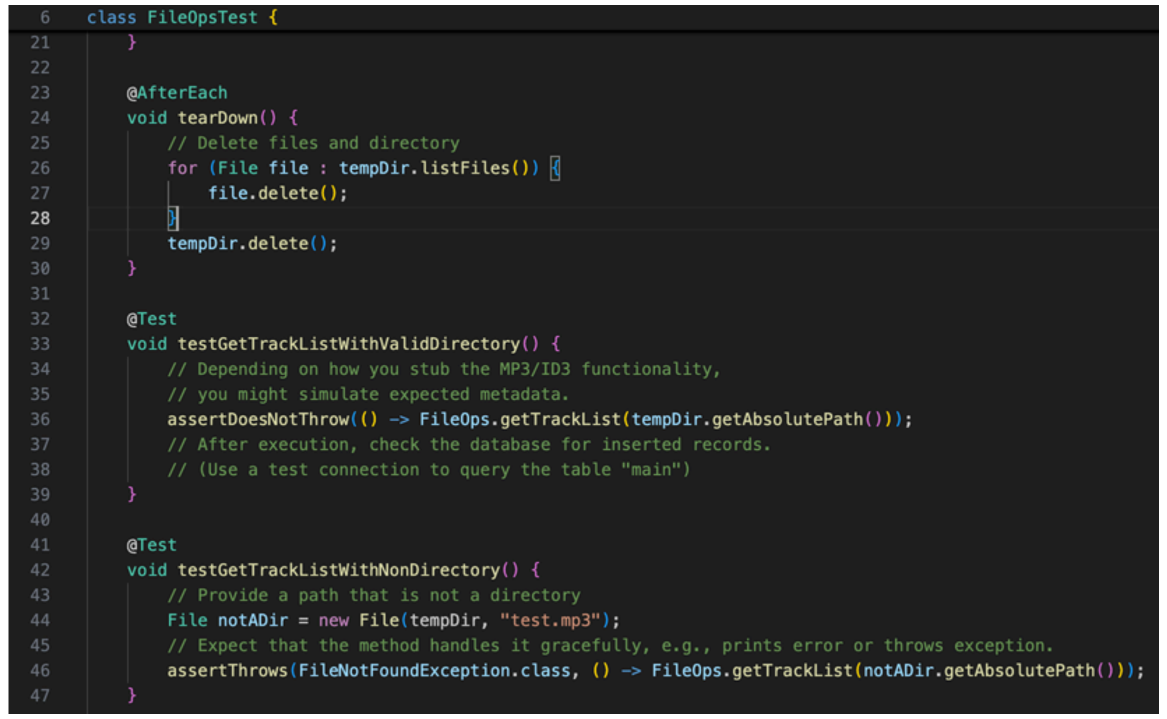
Scenario: Launch the application.
Test: Verify that the window appears with the expected components:
A title label with the header text.
A text field preset with “Browse Directory to list.”
A “Scan” button.
Three scrollable lists (for artists, albums, and tracks).
A status label initially showing “Idle.”

Generation of acceptance criteria

PROMPT

Analyze the code below, including its purpose, inputs, and expected outputs. Then, create a comprehensive test suite consisting of unit and/or integration tests that fully cover all aspects of the code's functionality, ensuring it produces a valid solution. Consider that the code will be transformed for the Java 17 language.

SCENARIO



```java
class FileOpsTest {
    }

    @AfterEach
    void tearDown() {
        // Delete files and directory
        for (File file : tempDir.listFiles()) {
            file.delete();
        }
        tempDir.delete();
    }

    @Test
    void testGetTrackListWithValidDirectory() {
        // Depending on how you stub the MP3/ID3 functionality,
        // you might simulate expected metadata.
        assertDoesNotThrow(() -> FileOps.getTrackList(tempDir.getAbsolutePath()));
        // After execution, check the database for inserted records.
        // (Use a test connection to query the table "main")
    }

    @Test
    void testGetTrackListWithNonDirectory() {
        // Provide a path that is not a directory
        File notADir = new File(tempDir, "test.mp3");
        // Expect that the method handles it gracefully, e.g., prints error or throws exception.
        assertThrows(FileNotFoundException.class, () -> FileOps.getTrackList(notADir.getAbsolutePath()));
    }
```

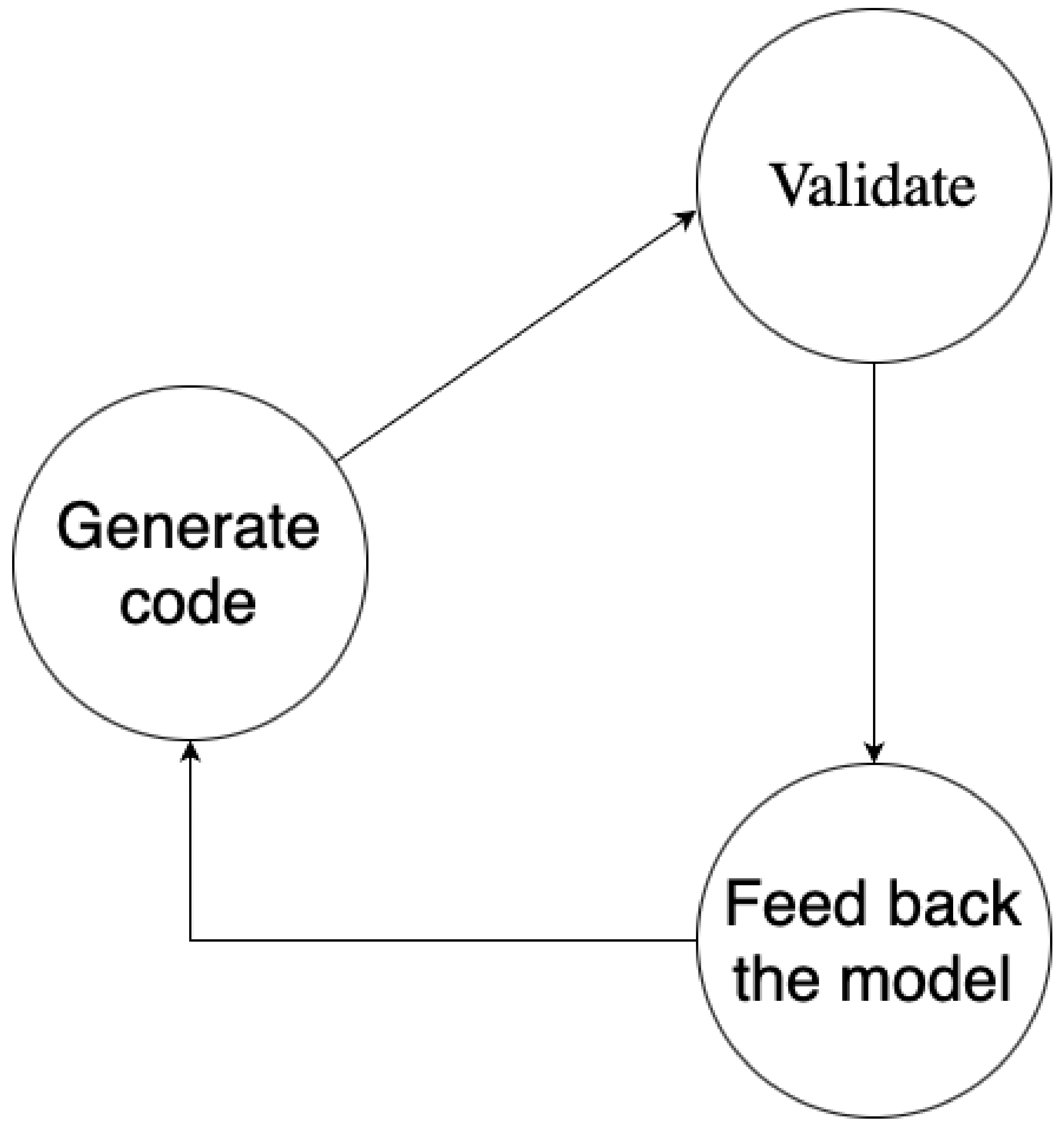
Utilization of LLMs for repair

Test suit only

Natural Language Only

Baseline

Iterative Repair and Validation Process



Evaluation Metrics and Reproducibility

Hipparcos

coverage

patches

tests

F-sharp

coverage

patches

tests

Results and findings

Validation criteria for Hipparcos

- 1: Data points are correctly plotted without any distortion
- 2: Graph is accurate to the mathematical relationship between variables
- 3: Scale and proportion are maintained

- 4: Handle plot values outside the graph's established limits
- 5: Independent use of classes
- 6: Lack of interference when multiple classes are used

Hipparcos Assessment

- No model has verified the independent and integrated uses of the Hipparcos plotting classes have no unwarranted effect on each other

- All of the predefined tests for Hipparcos were covered except for the above

New additions for Hipparcos

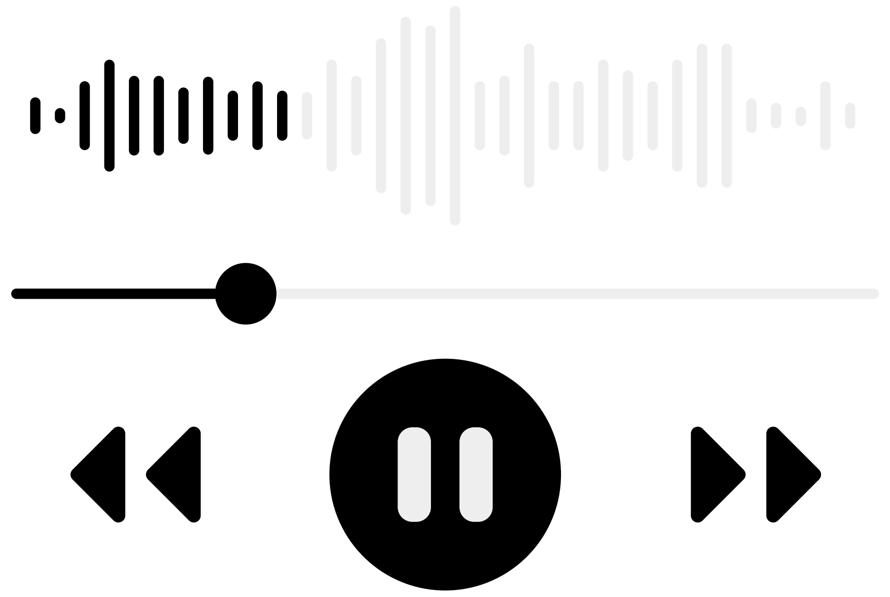
- 1: Copying integrity
- 2: Text (title and labels) positioning
- 3: GUI reset
- 4: Real time dimension changes
- 5: No Data warning
- 6: Coloring test

- 7: Rendering speed for large amount of data
- 8: Thread safety

Assessment: Hipparcos

Coverage by model	Predefined test amount	Good additions	Poor additions	Additions	Total
NL-GPT	4	4	0	1, 2, 3, 4	8
NL-DEEPSEEK	4	3	1	5, 4, 7, 8	8
NL-GEMINI	4	3	1	1, 3, 5, 6	8
TS-GPT	4	2	1	1, 3, 5	7
TS-DEEPSEEK	4	1	0	1	5
TS-GEMINI	4	4	1	1, 2, 6, 5, 3	9
Unique total	6	7	1		

Validation criteria for F-Sharp



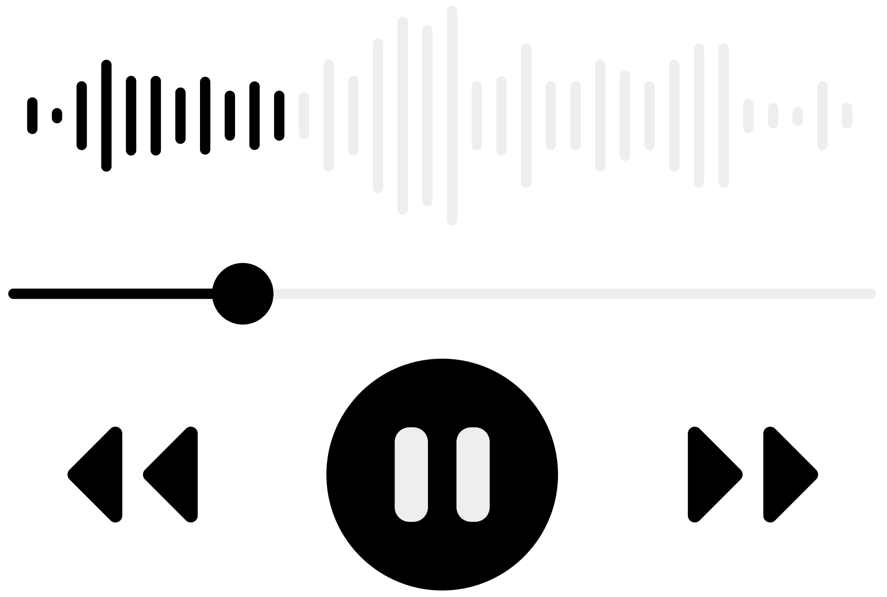
- 1: Handle missing information
- 2: Provide warning and alerts for denied access

- 3: Validate locked/concurrent directory uses
- 4: Handle denied access directory reads
- 5: Unusual tag formats
- 6: Inexistent directories
- 7: Correct database initialization
- 8: Storage and retrieval of DB info
- 9: Duplicate filename handling
- 10: Large files handling
- 11: Special characters handling

Fsharp Assesment

- More diversified coverage of the predefined set
- Common problems: special characters, locked/concurrent access, duplicate file names
- Deepseek unable to generate the test suit (possible hang up due to database mocking)

New additions for F-sharp



- 1: Ensure the GUI updates on rescan
- 2: Verify label integrity
- 3: Verify button functionalities

- 4: Check how it handles large directories
- 5: Recursive search
- 6: Mixed directories
- 7: Close all database connection after operation
- 8: Check for database network failure behavior
- 9: Subsequential search on different directories
- 10: Empty directory handling

Assessment: F-Sharp

Coverage by model	Predefined test amount	Good additions	Poor additions	Additions	Total
NL-GPT	7	6	0	1, 4, 7, 5, 6, 2	13
NL-DEEPSEEK	7	4	0	5, 6, 8, 4	11
NL-GEMINI	7	5	0	9, 1, 6, 10, 5	12
TS-GPT	7	2	0	2, 3	9
TS-DEEPSEEK	7	0	0	-	7
TS-GEMINI	7	2	0	2,10	9
Total	11	10	0		

Modernization findings

Hipparcos - OOP errors:

- Attempt to override funcions present in multiple classes without defining its abstract implementation
  - ex: ChatGpt and *ResetGraph* for alls classes; Gemini and both ResetGraph and PlotGraph; Deepseek for only one class.

Modernization findings

Fsharp - Outdated dependencies:

- Updated syntax with outdaded library
  - No listing of dependencies can have impact on undocumented systems

- Ignoring different metadata types

- Observations are true for all approaches and models

- For F-Sharp, Deepseek was the one that most differed, breaking up the code in a series of functions even when not asked to do so

- No great alteration between all models other than GUI alignment - could still be significant for larger applications

Findings

- Although the natural language prompt did not specify how to define the systems’ requirements, most models took it to mean unit manual assertion tests.

- Instructions sometimes lack tutorial for dependendencies installation, but providing exceptions as feedback to the model generate the necessary guidance as to what dependencies have to be installed

Conclusion

Questions & Answers

RQ1 How effective are LLMs at generating acceptance criteria for code repair?

RQ2 How well do the APR capabilities of LLMs work on legacy code?

RQ3 How do the tests affect the repair process?

Similarities and general discussion

- All codes can be fixed once the exception is fed to the llm
- The outputs are extremely similar

- All the modernizations present small errors, these might pile up in a real system where requirements are crispily defined

Future opportunities

- Fast evolution of LLMs and GenAI in general might be able to get rid of the minor errors in a near future

- As most of the errors were, in nature, due to syntax, using a traditional APR tool alongside an LLM might help expedite the process (as suggested by the article Automatic Repair of Programs from Large Language Models [4])

References

References

1: **Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation**
https://arxiv.org/pdf/2411.14971

2: **HipparcosJava**
https://github.com/esa/hipparcosJava/

3: **F-Sharp**
https://github.com/saahil/F-Sharp

4: **Automatic Repair of Programs from Large Language Models**
https://ieeexplore.ieee.org/document/10172854/