Software Testing Report: Part 1 - Functional Partitioning

Project: TuxGuitar (Open Source Tablature Editor)
Course: SWE 261P
Member: Xiyao Li & Ping Lu
Date: January 23, 2026

1. Introduction

1.1 Project Background

TuxGuitar is a widely adopted open-source application designed for creating, editing, and playing multi-track musical tablature. It functions as a versatile platform for music notation, compatible with numerous guitar score formats and offering both desktop and mobile interfaces. Its acceptance within the global music community is largely due to its cross-platform nature and robust handling of intricate musical compositions.

1.2 Technical Analysis and Scope

An examination of the project repository confirms that TuxGuitar qualifies as a substantial Java-based system, comfortably meeting the specified complexity thresholds:

- **Core Technology:** The application is developed chiefly in Java. The build incorporates the Standard Widget Toolkit (SWT) for the graphical user interface and allows for integration with optional native sound libraries such as FluidSynth or Jack for audio playback.
- **Codebase Volume:** The project contains roughly 266,800 lines of Java source code.
- **Architectural Complexity:** With a total of 2,658 distinct Java files, the system's scale significantly surpasses the baseline requirements of 15,000 lines of code and 100 classes.

2. Build and Environment Documentation

2.1 Prerequisites

To establish a functional build environment, the following dependencies must be configured:

- **JDK:** Version 9 or higher.
- **Build Tool:** Apache Maven 3.3 or higher.
- **GUI Library:** SWT 4.
- **Optional Components:** FluidSynth / Jack (for advanced audio features).

2.2 Compilation and Execution (macOS Example)

The following steps outline the procedure for building the project on a macOS environment:

1. **Dependency Installation:** Install `openjdk`, `maven`, and `wget` via Homebrew: `brew install openjdk maven wget`
2. **SWT Configuration:** Download the platform-specific SWT zip and install the `.jar` into the local Maven repository:

```
TUX_ARCH=`uname -m | sed 's/arm64/aarch64/'`
wget https://download.eclipse.org/eclipse/downloads/drops4/R-4.37-202509050730/swt-4.37-cocoa-macosx-${TUX_ARCH}.zip
unzip swt-4.37-cocoa-macosx-${TUX_ARCH}.zip
mvn install:install-file -Dfile=swt.jar -DgroupId=org.eclipse.swt -DartifactId=org.eclipse.swt.cocoa.macosx -Dpackaging=jar -Dversion=4.37
```

3. **Core Build Execution:** Navigate to the build directory and execute the Maven lifecycle:

```
cd desktop/build-scripts/tuxguitar-macosx-swt-cocoa
mvn -e clean verify -P native-modules
```

4. **Running the SUT:** Launch the generated `.app` located in the `target/` directory: `open desktop/build-scripts/tuxguitar-macosx-swt-cocoa/target/TuxGuitar.app`

---

3. Study of Existing Testing Practices

3.1 Testing Frameworks and Implementation

TuxGuitar utilizes **JUnit 5 (junit-jupiter-engine)** as its primary testing framework. The testing architecture follows a modular approach, where test suites are decentralized into individual component directories to ensure isolated verification of logic.

3.2 Key Testing Modules and File Locations

Based on our repository analysis, the existing test cases are primarily concentrated in the following core logic modules:

- **TuxGuitar-lib (`common/TuxGuitar-lib/src/test/java`)**: Houses critical music model tests, such as `TestTGDuration` and `TestTrackManager`.
- **Testing Resources**: The module utilizes dedicated resource files for file format verification, including `test_20.xml` and `test_midi_20.tg`.

3.3 Build Verification and Troubleshooting

During the initial execution of the test suite via Maven, we encountered a build hurdle that required technical intervention.

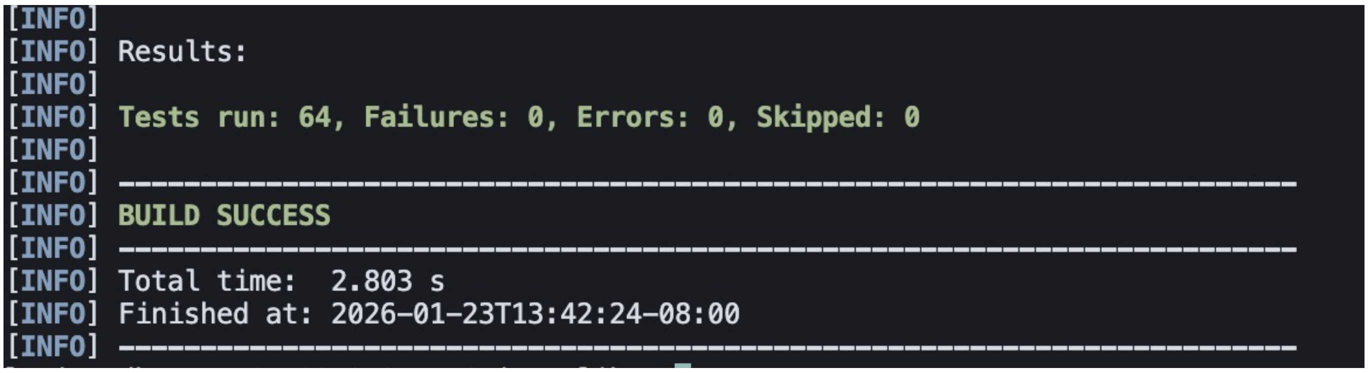
Technical Challenge: Path Encoding Conflict

When running `mvn test` within the `TuxGuitar-lib` module, the build initially failed with multiple `java.io.FileNotFoundException` errors.

- **Symptom**: The Maven Surefire plugin reported that it could not locate standardized music resource files (e.g., `Untitled_20.xml`).
- **Root Cause Analysis**: The local directory path contained special characters and spaces (`&`, `:`, and `%20`), which prevented the Java `FileInputStream` from correctly resolving absolute paths to the `target/test-classes/` directory.
- **Resolution**: We performed a "clean relocation" by moving the project to a standardized alphanumeric directory path. We then executed `mvn clean test` to purge corrupted build artifacts and re-index the resource metadata.

Successful Test Execution

After resolving the environment-specific pathing issues, we successfully executed the full test suite for the core library.



```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 64, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.803 s
[INFO] Finished at: 2026-01-23T13:42:24-08:00
[INFO] ------------------------------------------------------------------------
```

As shown in the execution log above, the system verified **64 test cases** with zero failures or errors, confirming the integrity of the core music engine, file I/O operations, and duration management logic.

---

4. Partitioning

4.1 Motivation and Concepts (Systematic Functional Testing & Partition Testing)

Because the input space of a real software system is vast, exhaustive testing is not feasible. **Systematic functional testing** addresses this by selecting representative inputs that reflect how the system should behave according to its specification. **Partition testing** divides the input space into subsets (partitions) that are expected to behave similarly; we then test one or more representative values from each partition, with extra attention to **boundary values** where failures are more likely. This provides high defect detection efficiency with a manageable number of test cases.

4.2.1: TGDuration (TestTGDuration.java)

Feature chosen: `TGDuration.splitPreciseDuration(total, max, factory)`
This method is suitable for partitioning because the outcome depends on the relationship between the total duration and the maximum allowed duration, and whether the total can be expressed using valid note divisions (including dotted values).

Partitioning scheme and partitions:

- **P1 Valid, simple exact split:** Total duration can be evenly decomposed into identical pieces within the `max` limit.
- **P2 Valid, fine subdivision required (boundary):** Total duration can be decomposed only using finer rhythmic divisions (e.g., dotted values).
- **P3 Invalid / impossible to split:** Total duration cannot be expressed using allowed note values.
- **P4 Valid with non-power-of-two max (boundary):** `max` is a non-power-of-two constraint; all pieces must be ≤ `max`.
- **P5 Large input robustness:** Very large total duration to ensure no overflow or runtime failure.

How partitions differ:
P1 is a straightforward exact split; P2 is valid but requires finer subdivisions; P3 is invalid and should return `null`; P4 stresses boundary behavior on `max`; P5 stresses robustness under large values.

Relationship to existing tests:
The original `TestTGDuration` already covers core conversions, precise-time mapping, and several historical split cases. The five new tests are **additive**, and are designed specifically to exercise partition boundaries and invalid/robustness scenarios without modifying or replacing the original test logic.

Representative values (one per partition):

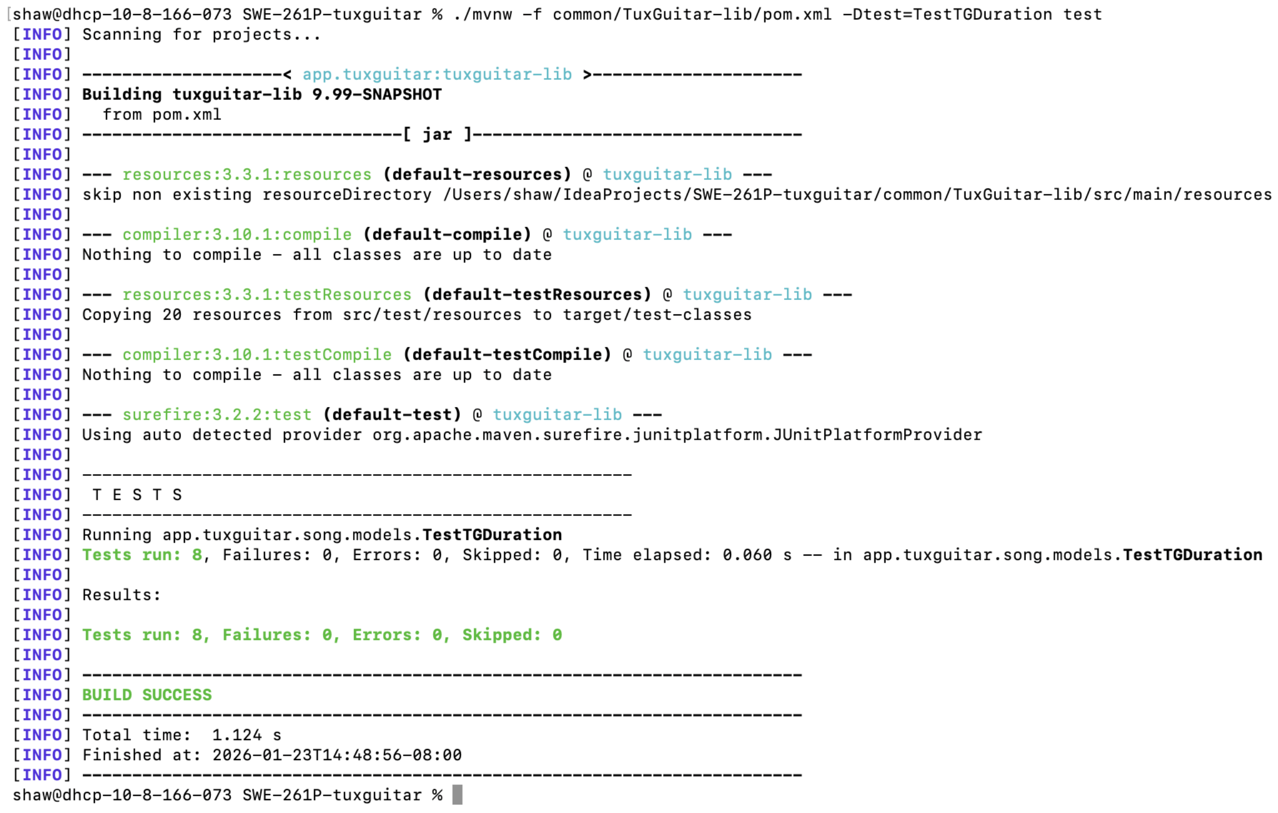
- **P1 - testSplitDurationValidSimple():** `total = WHOLE_PRECISE_DURATION / 2`, `max = WHOLE_PRECISE_DURATION / 8`
- **P2 - testSplitDurationFineSubdivision():** `total = WHOLE_PRECISE_DURATION * 3 / 64`, `max = WHOLE_PRECISE_DURATION`
- **P3 - testSplitDurationImpossibleReturnsNull():** `total = WHOLE_PRECISE_DURATION / 19`, `max = WHOLE_PRECISE_DURATION`
- **P4 - testSplitDurationMaxBoundary():** `total = WHOLE_PRECISE_DURATION`, `max = WHOLE_PRECISE_DURATION * 3 / 8`
- **P5 - testSplitDurationLargeTotalNoCrash():** `total = 5 * WHOLE_PRECISE_DURATION`, `max = 2 * WHOLE_PRECISE_DURATION`

How representative values were chosen:
We selected values that map directly to each partition’s behavior, with explicit **boundary cases** (non-power-of-two max and fine subdivisions) and an **invalid case** (non-representable total).

JUnit tests added:
`testSplitDurationValidSimple`, `testSplitDurationFineSubdivision`, `testSplitDurationImpossibleReturnsNull`, `testSplitDurationMaxBoundary`, `testSplitDurationLargeTotalNoCrash`

How to run (command):

```
cd /YourPathTo/SWE-261P-tuxguitar
./mvnw -f common/TuxGuitar-lib/pom.xml -Dtest=TestTGDuration test
```



```
shaw@dhcp-10-8-166-073 SWE-261P-tuxguitar % ./mvnw -f common/TuxGuitar-lib/pom.xml -Dtest=TestTGDuration test
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< app.tuxguitar:tuxguitar-lib >---------------------
[INFO] Building tuxguitar-lib 9.99-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ tuxguitar-lib ---
[INFO] skip non existing resourceDirectory /Users/shaw/IdeaProjects/SWE-261P-tuxguitar/common/TuxGuitar-lib/src/main/resources
[INFO]
[INFO] --- compiler:3.10.1:compile (default-compile) @ tuxguitar-lib ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ tuxguitar-lib ---
[INFO] Copying 20 resources from src/test/resources to target/test-classes
[INFO]
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ tuxguitar-lib ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- surefire:3.2.2:test (default-test) @ tuxguitar-lib ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running app.tuxguitar.song.models.TestTGDuration
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.060 s -- in app.tuxguitar.song.models.TestTGDuration
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.124 s
[INFO] Finished at: 2026-01-23T14:48:56-08:00
[INFO] ------------------------------------------------------------------------
shaw@dhcp-10-8-166-073 SWE-261P-tuxguitar % 
```

4.2.2: Music Key Utilities (TestMusicKeyUtils.java)

Feature chosen: `TGMusicKeyUtils.noteName / noteFullName / sharpNoteFullName`
This feature is suitable for partitioning because the output depends on **MIDI note range** and **key signature validity**, which define clear validity boundaries.

Partitioning scheme and partitions:

- **Q1 Valid MIDI range (boundary):** MIDI notes at the minimum and maximum valid values.
- **Q2 Invalid MIDI range:** MIDI notes below minimum or above maximum should return `null`.
- **Q3 Invalid key signature:** Key signature outside the valid range should return `null`.

How partitions differ:
Q1 validates correct boundary behavior within the accepted range; Q2 validates input rejection for out-of-range notes; Q3 validates rejection for invalid key signatures.

Relationship to existing tests:
The original `TestMusicKeyUtils` already verifies many note-naming and accidental cases. The new tests are **additive**, focusing specifically on boundary and invalid-input partitions (MIDI range and keySignature validity) without replacing existing coverage.

Representative values (one per partition):

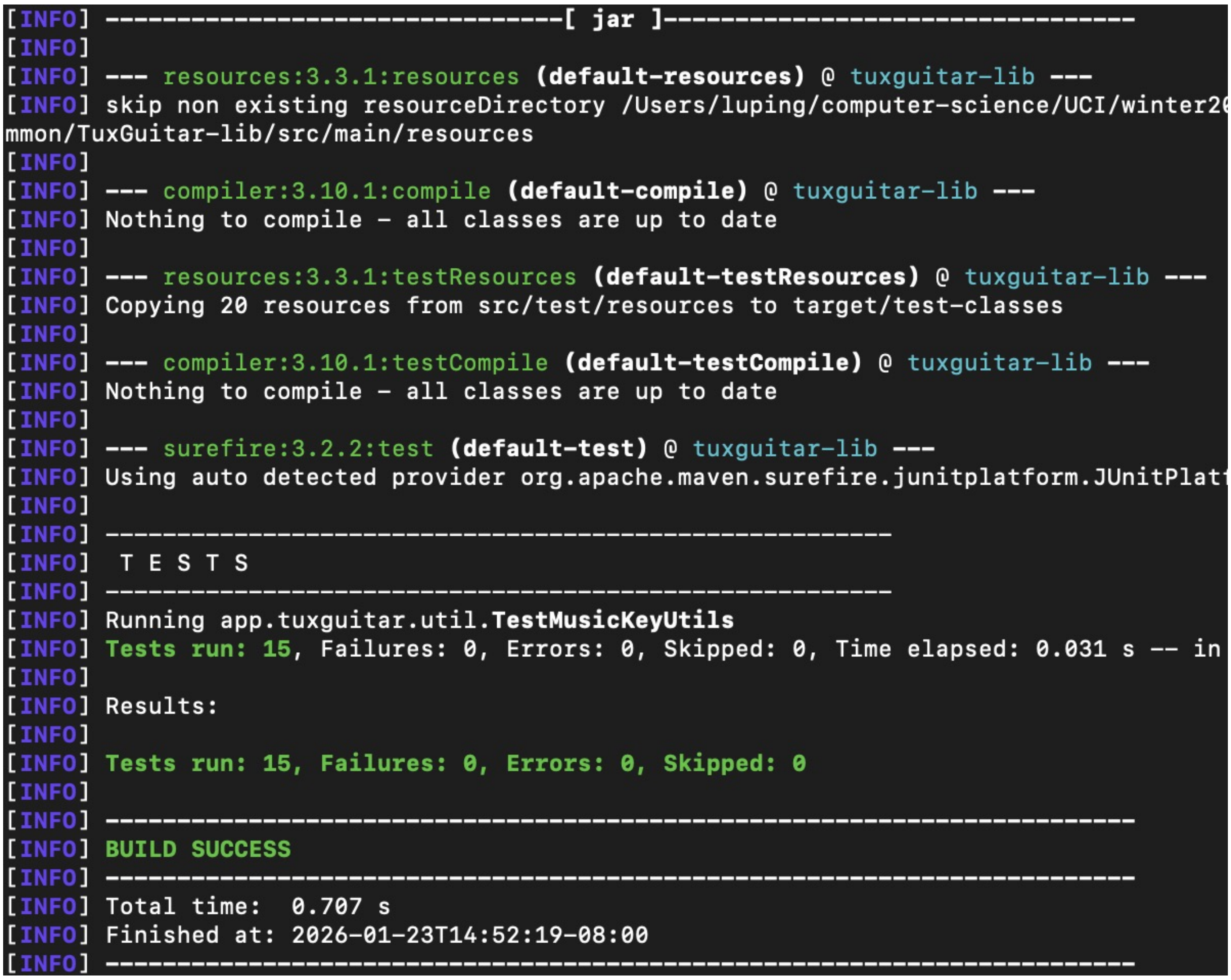
- **Q1 - testNoteNameMidiRangeBoundaries():** `MIN_MIDI_NOTE = 12`, `MAX_MIDI_NOTE = 127`
- **Q2 - testNoteNameInvalidMidiRangeReturnsNull():** `midiNote = 0`, `midiNote = 200`
- **Q3 - testNoteNameInvalidKeySignatureReturnsNull():** `keySignature = -1`, `keySignature = 15`

How representative values were chosen:
We used explicit boundary values for valid input and clearly invalid values to verify rejection logic.

JUnit tests added:
`testNoteNameMidiRangeBoundaries`, `testNoteNameInvalidMidiRangeReturnsNull`, `testNoteNameInvalidKeySignatureReturnsNull`

How to run (command):

```
cd /YourPathTo/SWE-261P-tuxguitar
./mvnw -f common/TuxGuitar-lib/pom.xml -Dtest=TestMusicKeyUtils test
```



```
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ tuxguitar-lib ---
[INFO] skip non existing resourceDirectory /Users/luping/computer-science/UCI/winter2
mmon/TuxGuitar-lib/src/main/resources
[INFO]
[INFO] --- compiler:3.10.1:compile (default-compile) @ tuxguitar-lib ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ tuxguitar-lib ---
[INFO] Copying 20 resources from src/test/resources to target/test-classes
[INFO]
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ tuxguitar-lib ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- surefire:3.2.2:test (default-test) @ tuxguitar-lib ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlat
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running app.tuxguitar.util.TestMusicKeyUtils
[INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.031 s -- in
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.707 s
[INFO] Finished at: 2026-01-23T14:52:19-08:00
[INFO] ------------------------------------------------------------------------
```

---

5. Functional Models (Finite State Machines) - Part 2. Functional Testing and Finite State Machines.

5.1 Why Finite Models Are Useful for Testing

Finite models (especially finite state machines) make behavior **explicit and enumerable**. By listing states and transitions, we can:

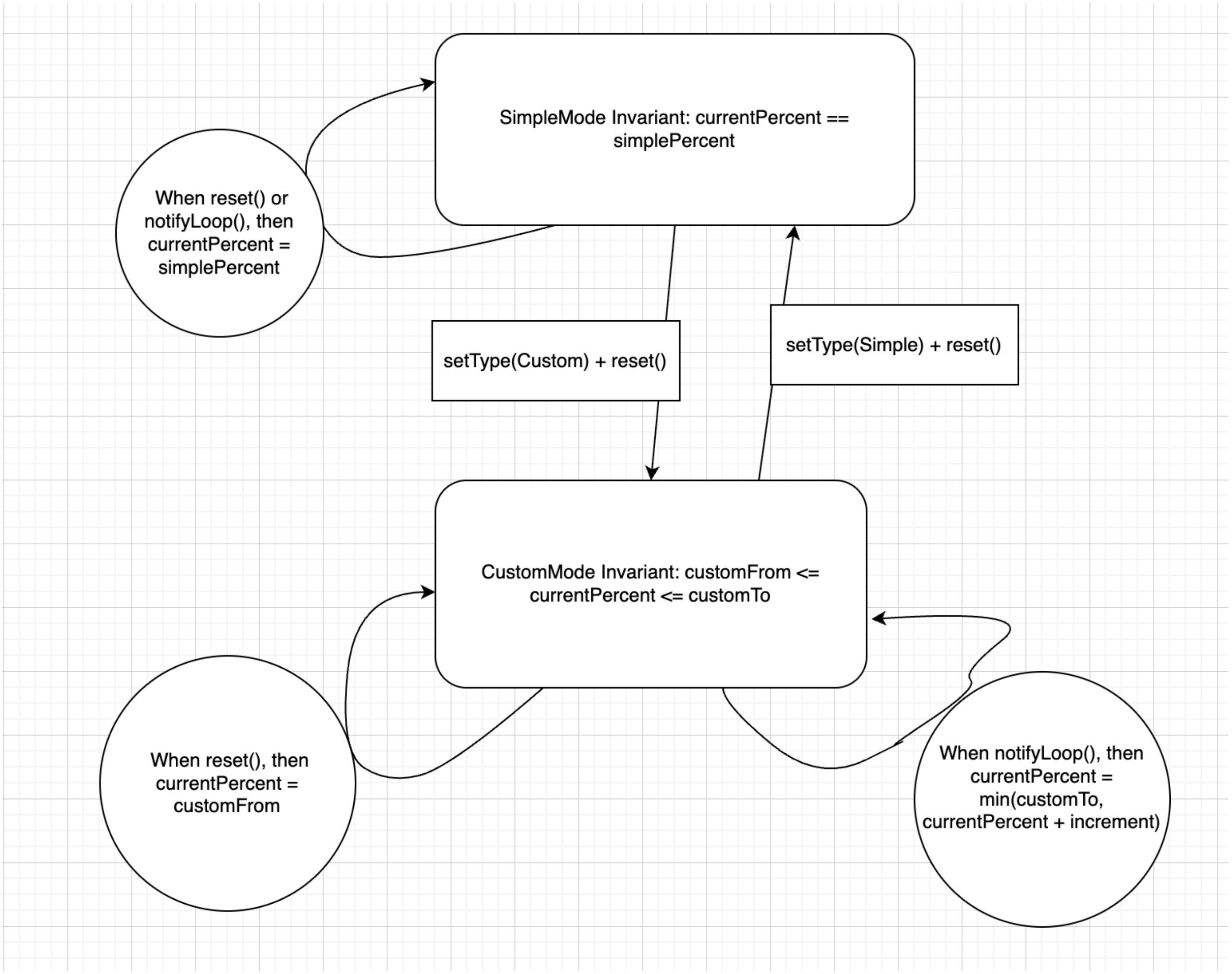
- Systematically cover **all valid transitions** and **self-loops**.
- Expose **illegal or missing transitions** (e.g., surprising states reachable by certain operations).

- Validate **sequences of operations**, not just single input/output pairs. This yields higher confidence with a bounded set of tests.

5.2.1: `MidiPlayerMode` (Tempo Mode Progression)

Feature chosen: `app.tuxguitar.player.base.MidiPlayerMode`
This component governs how playback tempo is computed across loops. Its behavior depends on the **mode** (`TYPE_SIMPLE` vs `TYPE_CUSTOM`) and the **currentPercent** value, making it well-suited to a finite state model.

Finite state model (FSM): States are defined by `type` plus the invariant for `currentPercent`.


SimpleMode Invariant: currentPercent == simplePercent
When reset() or notifyLoop(), then currentPercent = simplePercent
setType(Custom) + reset()
setType(Simple) + reset()
CustomMode Invariant: customFrom <= currentPercent <= customTo
When reset(), then currentPercent = customFrom
When notifyLoop(), then currentPercent = min(customTo, currentPercent + increment)

How the model works:
`reset()` initializes `currentPercent` according to the mode.
`notifyLoop()` is the loop-transition: in `SimpleMode` it remains constant, while in `CustomMode` it increments and caps at `customTo`.

JUnit tests added (covering all transitions and self-loops):

- `testDefaultSimpleResetSetsCurrentPercent`
- `testCustomResetStartsFromCustomFrom`
- `testCustomNotifyLoopIncrementsAndCaps`
- `testSimpleNotifyLoopResetsToSimplePercent`

Test file:
`common/TuxGuitar-lib/src/test/java/app/tuxguitar/player/base/TestMidiPlayerMode.java`

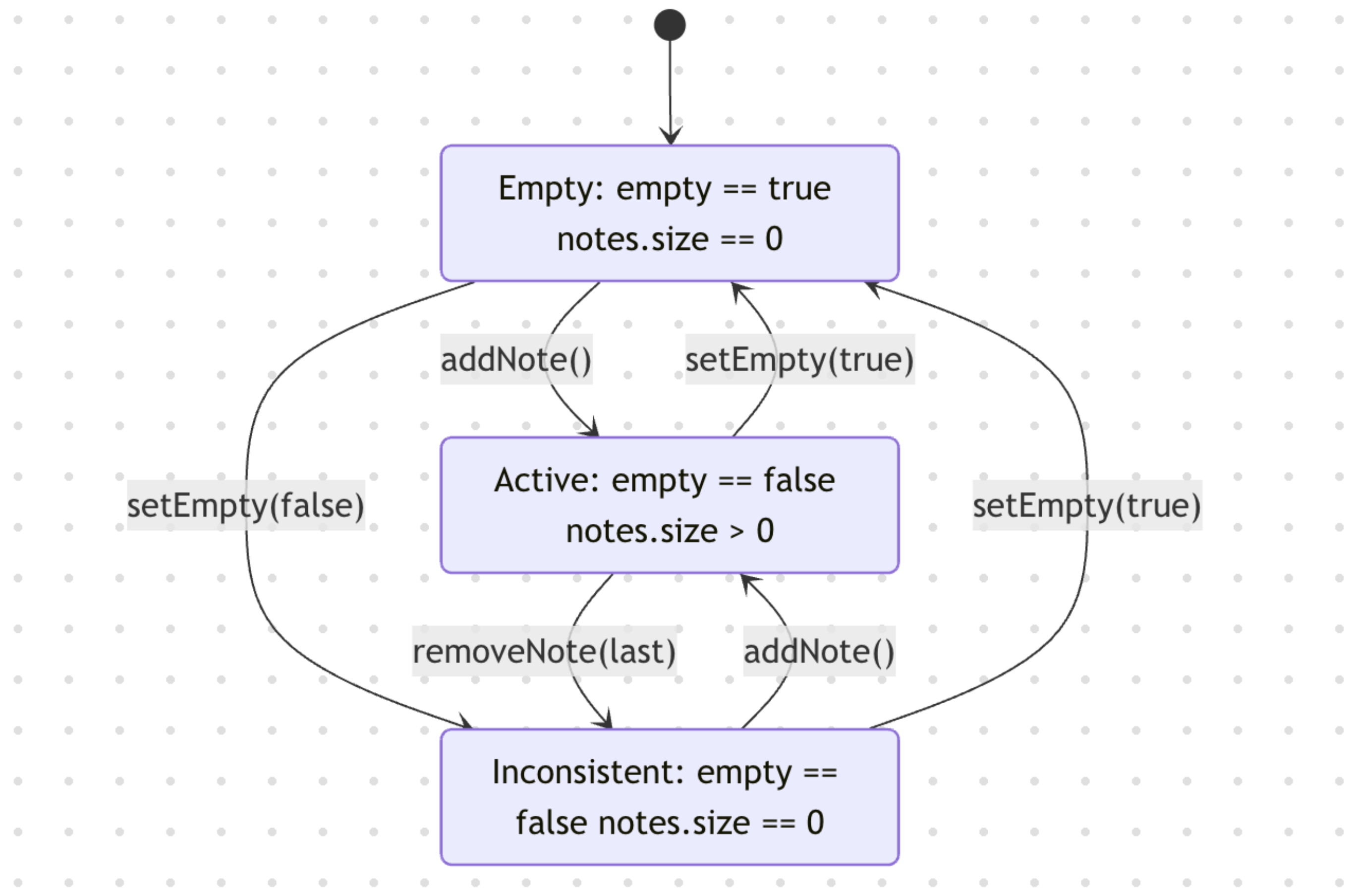
How to run:

```
cd /YourPathTo/SWE-261P-tuxguitar
./mvnw -f common/TuxGuitar-lib/pom.xml -Dtest=TestMidiPlayerMode test
```

5.2.2: `TGVoice` (Voice Content State)

Feature chosen: `app.tuxguitar.song.models.TGVoice`
`TGVoice` tracks musical content using two observable aspects: an `empty` flag and the `notes` list. The interactions between `addNote`, `removeNote`, and `setEmpty` create a **finite set of reachable states**, including an interesting "inconsistent" state that is reachable by design.

Finite state model (FSM):



How the model works:
`addNote()` always sets `empty=false` and adds to the list.
`removeNote()` does **not** flip `empty` back to `true`, so removing the last note creates state `[V2]`.
`setEmpty(true)` clears the list and returns to `[V0]`.

JUnit tests added (covering all states and transitions):

- `testAddNoteTransitionsToActive`

- `testRemoveLastNoteLeavesInconsistentState`
- `testSetEmptyTrueClearsNotesAndSetsEmpty`
- `testSetEmptyFalseFromEmptyCreatesInconsistentState`

Test file:
`common/TuxGuitar-lib/src/test/java/app/tuxguitar/song/models/TestTGVoiceStateMachine.java`

How to run:

```
cd /YourPathTo/SWE-261P-tuxguitar
./mvnw -f common/TuxGuitar-lib/pom.xml -Dtest=TestTGVoiceStateMachine test
```