Checksims Developer Guide

Matthew Heon Dolan Murvihill

September 10, 2015

Contents

1 Developer Guide

This guide is intended to serve as an introduction to *Checksims* to introduce new developers to the codebase. It is hoped that this will make contributing to the codebase easier and more accessible.

1.1 Obtaining the Source

The source code for *Checksims* is available on *Github*, at the following URL:

`https://github.com/mheon/checksims/`.

The source can be checked out using Git via the instructions located on the Github webpage, which are duplicated here for convenience:

```
git clone https://github.com/mheon/checksims.git
```

1.2 Building Checksims

Building *Checksims* has two requirements: a Java 8 JDK, and version 3 of the *Apache Maven* build system. The following assumes familiarity with Maven and its capabilities. The commands below must be run in the root of the cloned repository. All paths are relative to this directory.

`https://maven.apache.org/download.cgi`

Maven itself is written in Java, and the provided Jar files should be cross-platform.

Note that JDK 8 or later is a buildtime and runtime requirement. Earlier versions of the JDK will not be able to build and run Checksims.

Most testing was performed using the Oracle JDK across Linux and OS X, but other configurations (Linux/OS X + OpenJDK 8, Windows) should be supported without issue.

Checksims uses the typical Maven lifecycle commands for building. Again, a full review of Maven's capabilities is outside the scope of this document, but critical commands for building and testing will be briefly reviewed. Please note that all commands shown below are assumed to be run in the root of the cloned repository, and all paths given are relative to this directory.

To run unit tests:

```
mvn clean test
```

To build an executable `.jar` file:

```
mvn clean compile package
```

Build artifacts (executable `.jar` files) will be placed in the `target/` directory. By default, two `.jar` files will be produced: one with, and one without, library dependencies. They are easily differentiated — the `.jar` with dependencies included will be named as follows:

`checksims-VERSION-with-dependencies.jar`

Where VERSION is the current version number (1.1.1 at time of this writing).

1.3 Contributing: Modifying the Source

Checksims is fairly well documented and ships with unit tests, to enable ease of modifications. Furthermore, measures have been taken to make the addition of new algorithms, preprocessors, and output strategies especially easy. This section details the project structure to ease understanding of the source code and provide the location of critical project components.

1.3.1 Directory Structure

Checksims is structured as a typical Maven project; again, full description of this format is beyond the scope of the document, but important locations will be summarized.

All source code (production and test) is contained in the `src/` directory. Test-only code is found in `src/test/java/`, while main code is located in `src/main/java`. Runtime resources (non-Java source files, for example the template for generating HTML output) are located in the `src/main/resources/` directory.

Source files are contained in package-appropriate subdirectories of these main directories. In the case of a source file named `ChecksimRunner.java` in package `edu.wpi.checksims`, the location of this file would be:

`src/main/java/edu/wpi/checksims/ChecksimRunner.java`

Most Java IDEs should have the ability to import a Maven project; this should pick up all source directories automatically.

1.3.2 Source Structure

This section contains a description of the package and source structure of the project.

1.3.2.1 Package Structure The root package for *Checksims* is `edu.wpi.checksims`, with a number of subpackages, described below.

The `algorithm` subpackage contains core similarity detection functionality. All similarity detection algorithms (and the root interface `SimilarityDetector`, which all detection algorithms implement) are contained within `algorithm`. Algorithms at time of writing (*Line Similarity* and *Smith-Waterman*) are contained within appropriately-named subpackages (for example, `algorithm/smithwaterman` for the *Smith-Waterman* algorithm). It is intended that the same scheme will be used for future algorithms. The overall `algorithm` package also contains the `output` subpackage, containing all valid output strategies and the `SimilarityMatrixPrinter` interface, which all output strategies implement. Finally, the `algorithm` package also contains the `preprocessor` subpackage, containing all supporting preprocessor algorithms for token lists.

The `submission` subpackage contains code relating to the creation of submissions from directories and files on disk. All concrete classes within are `final`; this part of the interface is considered stable, and should not require much modification or extension.

The `token` subpackage contains code relating to tokenization of files and tokens generated. Several data structures for tokens, including `TokenList` (implementing `List<Token>`) and trees of arbitrary arity (contained in the `tree` subpackage), are included here. Concrete implementations of tokens themselves are `final`; again, this is considered a stable part of the interface.

The `util` subpackage contains general utility code.

1.3.2.2 Code Structure The entry point of *Checksims* is `ChecksimRunner` in the root package. This class contains both argument parsing and the `runChecksims` method, which controls the core functionality of the program.

`runChecksims` begins by initializing a list of submissions from the given directory, generating one submission per subdirectory of the given assignment directory(s). After the list of all submissions has been built, common code detection and removal is performed (if requested by the user). Preprocessors are then applied, sequentially, on each submission. Finally, `SimilarityMatrix.generate()` is called, creating a results matrix.

`SimilarityMatrix.generate()` generates a list of all unique unordered pairs of student submissions, then applies a given pairwise similarity detection algorithm to each submission. The results are then collated to produce the *similarity matrix*: a square array of floats, representing the similarity of each assignment when compared with each other assignment.

The generated similarity matrix is then passed to an output strategy to generate human-readable results. These are then printed (to `STDOUT` or, if requested, to a file), and the program exits.

Checksims includes a number of library dependencies. The most important of these are the *Apache Commons* and *Google Guava* libraries. These do overlap to an extent, but utilities from both are used throughout. In cases where both libraries provide overlapping functionality, the version provided by Apache Commons was preferred, though this is not a hard rule — if the API offered by Guava was felt to be superior, it was used instead. Finally, the *Apache Velocity* templating library is including to make the creation of complex output possible (for example, the included HTML output strategy). As in all Maven projects, the build process and dependencies are controlled by the `pom.xml` file in the root of the repository.

1.3.3 Adding an Algorithm

This section details the process of adding a similarity detection algorithm, though the process is very similar for a preprocessor or output strategy.

All similarity detection algorithms must be contained in package `algorithm` or a subpackage thereof. The core interface is `SimilarityDetector`, requiring three methods:

- `getName()`
- `getDefaultTokenType()`
- `detectSimilarity`

The `getName()` method returns the name of the algorithm, as the user will type it in on the command line. This must be a unique string (no two algorithms may have the same name), and it is highly recommended that it not contain spaces.

The `getDefaultTokenType()` method returns the default tokenization used by the algorithm. There are three supported tokenizations in *Checksims* at the time of this writing: `Line`, `Whitespace`, and `Character`. Each breaks submitted files up differently (by newline, by whitespace character, and into individual characters) to produce tokens. The returned token type from this method is the default for this algorithm (this can be overridden by the user at runtime, however).

The `detectSimilarity()` method accepts a pair of student submissions, performs similarity detection, and returns an `AlgorithmResults` object representing the results of the detection. Exceptions are permitted to be thrown on internal algorithm errors.

Algorithms must also implement a static `getInstance()` method taking no arguments and returning an instance (preferably a singleton) of the algorithm. All detection algorithms are automatically loaded at runtime using reflection using this method. As a result of this, no work is required to make an algorithm available outside of writing it and placing it in the `algorithm` package; all implementors of `SimilarityDetector` will be scanned via reflection and loaded automatically.

Similarity detection is multithreaded by default, and is performed using a single instance of the similarity detector. Consequently, it is not advised to have class-level mutable state or `synchronized` methods within a similarity detector. Instead, mutable state should be confined to the `detectSimilarity()` method and any helpers. The overall implementation of a similarity detector must be thread-safe.