The maddi Manual

Bart Naudts

Version 1, 2026-07-20

***This is the** operator's manual **for** maddi **: how to run the analyzer on your** own project — through the Gradle plugin, the Maven plugin, or the command line — how to give it the right inputs, how to read its output, and how to supply analysis hints for libraries. For the* **concepts** *behind the annotations it produces (immutability, modification, independence, linking), see the companion document The Road to Immutability.*

Table of Contents

Chapter 1. Overview

maddi is a whole-program static analyzer for Java. It reads your source code and its compile classpath, works out how each type behaves — is it modified, is it immutable, does it leak its internals — and records the result as a set of annotations (`@Immutable`, `@Container`, `@NotModified`, `@Independent`, …) together with a machine-readable **analysis result** per package.

This manual is about **operating** the tool. The meaning of the annotations, and the theory that produces them, are described in the companion document *The Road to Immutability*.

1.1. What it does, concretely

Given a set of source directories and a classpath, a *maddi* run performs, in order:

1. **parse** — build a concrete syntax tree (CST) of your sources;
2. **prep** — resolve the types, build the call graph, and compute the analysis order;
3. **modification analysis** — the actual immutability / modification computation (iterative);
4. **write results** — one analysis-result `.json` file per package, under the results directory (default `<build>/maddi`).

You can stop after any step with the `analysis-steps` option (see The command line); running `prep` only is a fast way to check that a project parses and resolves cleanly.

1.2. The three ways to run it

There is one analysis engine, reached through three front doors:

Entry point	Use when
The Gradle plugin	Your project builds with Gradle. The plugin derives the source sets and classpath from the Gradle model for you and runs the analyzer as a task.
The Maven plugin	Your project builds with Maven. Mojos derive the input from the Maven project model.
The command line	You want to drive the analyzer directly — from a captured build log (`--compile-log`), a saved input configuration, or explicit source/classpath flags. Works for any build system.

All three assemble the same object — an **input configuration** (source sets + classpath + JDK modules) — and hand it to the analyzer. That object is the subject of The input configuration.

1.3. Prerequisites

- **A recent JDK.** *maddi* is built and run on current Java (the modules target Java 25/26). Run it on a JDK at least as new as the one that built it.
- `--add-exports` **for the openjdk front-end.** The default (openjdk) parser reaches into

`jdk.compiler`'s `com.sun.tools.javac.*` internals, so the JVM that runs the analyzer must be started with:

```
--add-exports jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED
```

The Gradle plugin injects these into a forked worker automatically. For the Maven plugin and the CLI you supply them yourself (see the relevant sections).

maddi deliberately analyzes with a *partial* classpath — it does not need every transitive dependency to be present. Types it cannot resolve are reported as **warnings**, not fatal errors (see Reading the output: errors, warnings, exit codes).

Chapter 2. Getting started

Two quick paths, depending on how your project is built.

2.1. A Gradle project

Apply the plugin and configure the `maddi` extension:

```
plugins {
    java
    id("io.codelaser.maddi.analyzer")
}

maddi {
    analysisSteps = "modification"   // parse -> prep -> modification -> write results
    // analysisResultsDir defaults to "$buildDir/maddi"
}
```

Then run:

```
$ ./gradlew maddi-analyzer
```

The plugin computes the input configuration from your source sets and compile classpath, forks a worker JVM with the required `--add-exports`, runs the analyzer, and writes one result `.json` per package under `build/maddi`. A non-zero exit fails the task; see Reading the output: errors, warnings, exit codes. The plugin is described in full in The Gradle plugin.

2.2. Any project, from a build log

If you are not on Gradle — or you want to analyze a project exactly as its build compiles it — capture the compiler invocation from a verbose build and feed it to the CLI. This is the **compile-log** path: *maddi* reconstructs the input configuration from the `javac` (and, for Kotlin, `kotlinc`) command lines the build actually issued.

```
# 1. capture the compiler arguments from a Gradle build (Maven and raw javac also
work)
$ ./gradlew :your-module:compileJava --debug 2>&1 | grep 'Compiler arguments:' >
build.log

# 2. derive the input configuration and run the prep analysis
$ maddi --compile-log build.log --analysis-steps prep --analysis-results-dir out
```

Here `maddi` is the CLI launcher for the openjdk runner (started with the `--add-exports` from Overview). You can also derive the configuration once and reuse it:

```
$ maddi --compile-log build.log --write-input-configuration project.json
$ maddi --input-configuration project.json --analysis-steps modification --analysis
-results-dir out
```

The command line is covered in The command line; the input-configuration object it produces in The input configuration.

2.3. What you get

- A results directory (default `build/maddi`) containing one `.json` per analyzed package. These hold the computed annotations and are what downstream tooling — and subsequent *maddi* runs — consume.
- A summary printed to the log, including any errors and warnings with their file/line/column.
- An exit code that reflects the outcome (`0` OK, non-zero on parse/analysis failure — see Exit codes).

Chapter 3. The Gradle plugin

The Gradle plugin (id `io.codelaser.maddi.analyzer`) derives the input configuration from your project's source sets and resolved classpath, then runs the analyzer in a forked worker JVM (so the required `--add-exports` are injected without touching your build's daemon).

3.1. Applying it and the tasks

```
plugins {
    java                                // or java-library
    id("io.codelaser.maddi.analyzer")
}
```

It registers two tasks (group `maddi`):

`maddi-analyzer`

: Run the analyzer over the project and write results. Depends on `compileJava` (and `compileKotlin` when the Kotlin JVM plugin is applied). Declares its source files and classpath as inputs and the results directory as its output, so Gradle skips it when nothing relevant changed.

`maddi-write-input-configuration`

: Write the computed input configuration to `build/inputConfiguration.json` without running the analysis — useful for inspection, or to hand to the CLI later (see The command line).

3.2. The `maddi` extension

Configure the run through the `maddi` extension block. All options are optional.

Option	Type	Meaning
`analysisSteps`	String	Comma-separated steps to run: `none`, `prep`, `modification`, `rewire-tests`. Absent runs the default pipeline.
`analysisResultsDir`	String	Where result `.json` files are written. Default `<build>/maddi`.
`incrementalAnalysis`	boolean	Build on results already present in the results directory.
`parallel`	boolean	Parallelize the analysis where possible.
`quiet`	boolean	Suppress the end-of-run summary (the exit code still reflects the outcome).
`debugTargets`	String	Debug channels to enable (e.g. `memory`).
`sourcePackages`	String	Restrict analysis to these packages (dot-suffix includes sub-packages).
`testSourcePackages`	String	As above, for test sources.
`jmods`	String	JDK modules to add to the classpath (e.g. `java.base`).

Option	Type	Meaning
`jre`	String	Alternative JRE location.
`excludeFromClasspath`	String	Jar names/coordinates to drop from the classpath.
`workingDirectory`	String	Base directory the analyzer resolves relative paths against.
`preloadAnalysisResultsDirs`	String	Directories of pre-computed analysis results to load for library types (see Analysis hints, use case 1).
`analysisResultsTargetDir`	String	Compile analysis-hint sources into results here (use case 2).
`updatedHintsDir`, `updatedHintsPackage`, `hintsPackages`	String	Write updated hint files (use case 3).

Example:

```
maddi {
    analysisSteps = "modification"
    sourcePackages = "com.example."           // this package and below
    jmods = "java.base"
    excludeFromClasspath = "some-noisy-lib.jar"
}
```

3.3. Configuration cache and up-to-date behaviour

The plugin is built on lazily-configured tasks and holds no `Project` reference at execution time, so it is compatible with Gradle's configuration cache. Because `maddi-analyzer` declares its inputs and output, a second run with unchanged sources and classpath is `UP-TO-DATE`.

3.4. Kotlin-aware source-set collection

When the Kotlin JVM plugin is applied, the plugin also collects each source set's Kotlin source directories (and depends on `compileKotlin`), so `maddi-write-input-configuration` produces a correct mixed Java+Kotlin configuration.

The `maddi-analyzer` task itself currently drives the **openjdk (Java)** analyzer, which parses only `.java`; Kotlin sources in the configuration are recorded but not yet analyzed by this task. See Kotlin and mixed projects for the state of Kotlin/mixed analysis.

Chapter 4. The Maven plugin

The Maven plugin (`maddi-mvnplugin`) derives the input configuration from the Maven project model (compile and test source roots, and the dependency classpath resolved via Maven's own resolver) and, like the Gradle plugin, hands it to the openjdk `RunAnalyzer`.

4.1. Goals

Goal	Does
`run`	Parse, prep- and modification-analyze the project; write results.
`compile-analysis-hints`	Compile hand-written analysis-hint sources into result `.json` files (Analysis hints, use case 2).
`write-analysis-hints`	Generate first-cut analysis-hint skeletons for the library types the project calls into (use case 3).
`write-input-configuration`	Write the computed input configuration to JSON.
`statistics`	Method-call-frequency histograms.

4.2. The `--add-exports` requirement

Unlike Gradle (which forks a worker), a Maven mojo runs in-process, so the Maven JVM itself must be started with the javac `--add-exports` from Overview. Put them in `.mvn/jvm.config` or `MAVEN_OPTS`:

```
# .mvn/jvm.config
--add-exports jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED
```

Status. The plugin's sources build in-tree and are aligned with the Gradle plugin, but the Maven plugin **descriptor** is not yet generated, so it cannot yet be invoked as a normal Maven plugin. This section will be completed once packaging is in place.

Chapter 5. The command line

There are two command-line entry points, both requiring the javac `--add-exports` from Overview on the launching JVM:

- the **openjdk runner** (`io.codelaser.maddi.run.openjdkmain.Main`) — the primary CLI, for Java projects;
- the **Kotlin/mixed runner** (`io.codelaser.maddi.run.kotlinmain.Main`) — for mixed Java+Kotlin, prep only (see Kotlin and mixed projects).

In the examples below, `maddi` stands for a launcher that starts the appropriate main class with the `--add-exports` already applied (the `run-openjdk` / `run-kotlin` modules apply Gradle's `application` plugin, so `./gradlew :maddi-run-openjdk:run --args="…"` works too).

5.1. Providing the input

The openjdk CLI accepts the input configuration in three mutually-exclusive ways (checked in this order):

`--input-configuration <file.json>`

Read a previously written input configuration.

`--compile-log <file>`

Derive it from a build/compiler log — raw `javac …` lines, Gradle `Compiler arguments: …` lines, or Maven `[DEBUG] -d …` lines; `.gz` is accepted. Add `--extra-jmod <module>` (repeatable) to widen the JDK-module closure.

Explicit flags

`--source <dirs>`, `--test-source <dirs>`, `--classpath <cp>`, `--jmod <module>`, `--source-packages <pkgs>`, `--test-source-packages`, `--exclude-from-classpath`.

See The input configuration for what these build.

5.2. Choosing what to run

`--analysis-steps <steps>`

`none`, `prep`, `modification`, `rewire-tests` (comma-separated).

`--analysis-results-dir <dir>`

Where results are written.

`--preload-analysis-results-dirs <dirs>`

Load pre-computed analysis results for library types before modification analysis (see Analysis hints). `-p`/`--parallel`, `--quiet`, `-d`/`--debug <targets>`, `--jre <dir>`, `--source-encoding <enc>`.

5.3. Two workflows

Derive once, run many. `--write-input-configuration <file>` writes the (derived) input configuration and exits without analyzing — capture a project's configuration, then re-run against it:

```
$ maddi --compile-log build.log --extra-jmod java.sql --write-input-configuration
project.json
$ maddi --input-configuration project.json --analysis-steps modification --analysis
-results-dir out
```

Straight run from a log.

```
$ maddi --compile-log build.log --analysis-steps prep --analysis-results-dir out
```

5.4. The Kotlin/mixed runner

The `run-kotlin` CLI mirrors the openjdk one for mixed projects. It takes `--compile-log <mixed log>` (a log containing both `javac` and `kotlinc` invocations, parsed into one configuration) or `--input-configuration <json>`, runs the **prep** analysis, and prints a summary:

```
$ ./gradlew :maddi-run-kotlin:run --args="--compile-log mixed-build.log"
```

5.5. Exit codes

Every run returns a code: `0` on success, and distinct non-zero codes for parser / IO / analyzer failures. See Exit codes. A parse error never returns `0`.

Chapter 6. The input configuration

Whatever front door you use, the analyzer ultimately consumes one object: the **input configuration**. It is worth understanding, because it is the thing you serialize (`--write-input-configuration`), inspect, and hand around.

6.1. What it contains

An input configuration is:

- a set of **source sets** — each a named group of source directories (e.g. `main`, `test`), with an encoding, a test flag, an optional package restriction, and **dependencies** on other source sets;
- a set of **class-path parts** — the compiled libraries (jars/directories) and JDK modules the sources are analyzed against;
- a working directory (relative paths are resolved against it), and an optional alternative JRE.

Source sets carry a dependency graph, which the analyzer linearizes so that a set is analyzed after the sets it depends on. Class-path parts marked as JDK modules use the `jmod:` URI scheme.

6.2. The three ways to build one

Way	How it is derived
Build-tool plugins	The The Gradle plugin and The Maven plugin read their build model — source sets and the **resolved** compile classpath — directly. This is the most faithful for a project you own.
Compile-log capture	`--compile-log` (The command line) reconstructs the configuration from the compiler command lines a build actually issued (`javac`/`kotlinc`). Because it consumes what the compiler was told, it works for **any** build system, not just Gradle/Maven.
Explicit / serialized	The CLI `--source`/`--classpath`/`--jmod` flags build one by hand; `--input-configuration` reads a previously written one.

6.3. Partial classpaths are fine

The analyzer does not require every transitive dependency. A reference to a type that is not on the classpath is reported as a **warning** and the run continues (see Reading the output: errors, warnings, exit codes). In practice you add just enough of the classpath for the analysis to be meaningful; library types you care about are better supplied as pre-computed **analysis hints** (Analysis hints) than as full jars.

Chapter 7. Analysis hints

maddi cannot analyze the source of the libraries you depend on — you rarely have it, and even the JDK is only available as bytecode. **Analysis hints** fill that gap: hand-written `.java` files that annotate a library's public API surface (`@Immutable`, `@Container`, `@NotModified`, `@NotNull`, …) with the conclusions the analyzer would have drawn had it seen the source.

Compiling those hint sources produces **analysis results** (`.json`) — the same format the analyzer emits for your own code — which are then loaded before your project is analyzed, so calls into those libraries resolve to real annotations instead of unknowns.

The JDK and a set of common libraries ship pre-annotated in the `maddi-aapi-archive` module.

7.1. The three use cases

Use case 1 — load pre-computed results

Point `preloadAnalysisResultsDirs` (extension / `--preload-analysis-results-dirs`) at directories of analysis-result `.json` files. Loaded before modification analysis so library types have annotations.

Use case 2 — compile hints into results

Turn a directory of hand-written hint sources into result `.json` files. Gradle: `analysisResultsTargetDir`; Maven: the `compile-analysis-hints` goal; CLI: an `--analysis-results-target-dir`.

Use case 3 — write first-cut hints

Generate skeleton hint sources for the library types your project actually calls into (annotated with call-frequency comments), as a starting point to curate. Gradle: `updatedHintsDir` / `updatedHintsPackage`; Maven: the `write-analysis-hints` goal.

Status. Out of the box the runner only preloads results when you **explicitly** point it at them; an automatic "default location" preload of the shipped JDK/library results is not yet wired into the openjdk runner. Until then, supply `preloadAnalysisResultsDirs` yourself to get library annotations.

Chapter 8. Reading the output: errors, warnings, exit codes

A run reports what happened in two channels: a human-readable **summary** logged at the end, and a **process exit code**.

8.1. Errors versus warnings

maddi distinguishes two severities, because it analyzes with a partial classpath and must tolerate the gaps:

Warnings are non-fatal. The most common is a reference to a type that is not on the classpath — from either front-end (the openjdk parser surfaces javac's diagnostics; the in-house parser surfaces its unresolved-type resolutions). Warnings do not stop the run and do not, on their own, make it fail. They are listed in the summary with their file, line and column, capped to keep the output readable.

Errors are genuine failures: a syntax error, an IO failure setting up a source set, or an exception during analysis. Errors are listed in the summary and drive the exit code.

8.2. The summary

Unless `--quiet` (CLI) / `quiet = true` (Gradle) is set, the run prints, at the end, the collected errors and warnings — each with its location and message. This is the first thing to read when a run does not produce the results you expected.

8.3. Exit codes

The exit code reflects the outcome; a parse error never yields `0`. The full table is in Exit codes. In short:

- `0` — success;
- `2` — a parser/inspection error (a source failed to parse);
- `4` — an IO failure;
- `5` — an analyzer error (the modification analysis threw);
- `1` — an unexpected internal error.

The Gradle plugin turns any non-zero code into a task failure; the Maven plugin fails the build; the CLI returns the code from `main` (a launcher will `System.exit` with it).

8.4. Fail-fast versus accumulate

The parse can run in **fail-fast** mode (stop and report at the first error) or **accumulate** mode (collect all errors, then report). The openjdk runner accumulates by default; a single syntax error still yields

a parser exit code, but you see every problem the parse found rather than only the first.

Chapter 9. Kotlin and mixed projects

maddi has a Kotlin front-end (built on the Kotlin K2 analysis API) and a **mixed inspector** that parses Java and Kotlin over one shared core, so a cross-language reference (a Java class using a Kotlin type, or vice versa) resolves to a single type.

9.1. What works today

- **Source-set collection.** The Gradle plugin recognizes Kotlin source directories and can emit a mixed input configuration (see The Gradle plugin).
- **Mixed input-configuration derivation.** The compile-log path reconstructs a mixed configuration from a build log containing both `javac` and `kotlinc` invocations (see The command line).
- **Prep analysis.** The `run-kotlin` runner parses a mixed project (each type placed in its own source set, honouring the classpath) and runs the **prep** analysis — call graph and analysis order — over the combined Java+Kotlin types.

```
$ ./gradlew :maddi-run-kotlin:run --args="--compile-log mixed-build.log"
```

9.2. Current limits

Kotlin/mixed support is **prep-only** for now:

- the **modification** analysis is not yet run on mixed projects;
- the Gradle `maddi-analyzer` task drives only the Java (openjdk) analyzer, so it does not yet analyze `.kt` (Kotlin sources are recorded in the configuration but skipped by that task);
- the mixed inspector's cross-language scope has known follow-ups (some Java□Java and both-direction cross-language cases);
- the Kotlin front-end's own language coverage is still growing.

This section will grow as the Kotlin/mixed pipeline moves beyond prep.

Appendix A: Exit codes

Every runner returns one of these process exit codes (shared across the openjdk, in-house and Kotlin runners). A parse error never returns `0`.

Code	Name	Meaning
`0`	OK	The run completed; results (if any step wrote them) are in the results directory.
`1`	Internal exception	An unexpected error not attributable to a specific phase.
`2`	Parser error	One or more sources failed to parse (or a fail-fast parse aborted on the first).
`3`	Inspection error	An error during type inspection. (Reserved; parse-phase failures currently map to `2`.)
`4`	IO exception	An IO failure — e.g. a source directory or classpath entry could not be read.
`5`	Analyzer error	The modification analysis threw.

Unresolved references to types not on the (partial) classpath are **warnings**, not errors — they do not change the exit code. See Reading the output: errors, warnings, exit codes.