Timefold · Separating timefold-solver-core into nine modules using CodeLaser

`timefold-solver-core` is a single artifact in which most types depend on most other types. This is an account of separating it into nine modules that build in a fixed order, using the CodeLaser refactoring engine, and measured as the work happened: what was done and in what order, what had to change in the published API and why, and how the result was checked.

SUBJECT
`timefold-solver-core`

CONTACT
piet.vanremortel@codelaser.io

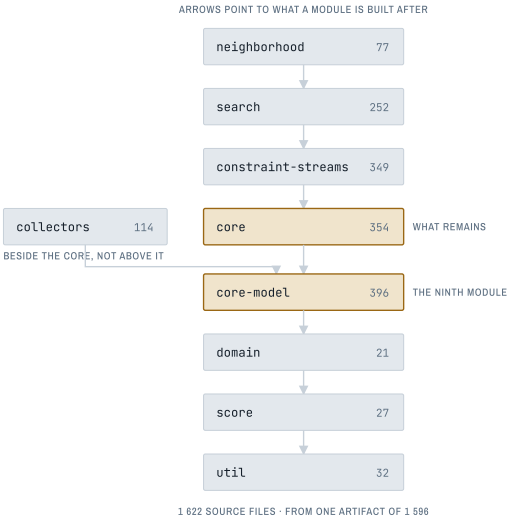
TABLE OF CONTENTS

IN SHORT

1. Nine modules, four changes to the published API, and no change to the tests

- **The goal.** `timefold-solver-core` — **the core** throughout this document — is one artifact. Anything that uses any part of it compiles against all of it. The goal was to separate it into modules that build in a fixed order, so a consumer takes only the part it needs — without changing what the library does.
- **The obstacle.** 978 classes in `timefold-solver-core` depend on each other in a loop. A module has to be built before whatever uses it. Inside a loop there is no first, so no boundary can be drawn anywhere in it.
- **What was done.** The loop was reduced first. A dependency-injection container was introduced into the core, so that classes are given what they need instead of creating it. Nine modules were then separated out, one at a time.
- **Two modules cost nothing at all.** `util` and `neighborhood` came out with no change to the published API.
- **The rest needed four changes to the published API.** Four types change package or declaration. Section 8 describes each with what forces it and what it costs a user. Three are forced by the module system. One was a judgement call, and was put to the operator rather than taken.
- **No regression.** The full test suite at the end is identical to a control build, module for module and not merely on totals: the same suites, the same tests, the same outcomes. It was run and compared after every step, and twice it disagreed and the work stopped until it agreed again. Section 10 gives the control and what it covers.
- **What a consumer's build has to pull in.** Twenty-four modules in the `timefold-solver` repository build against the core — adapters, integrations, tools and benchmarks. **Eleven** of them declare a planning model and never construct a solver: those used to resolve **45%** of the core's compiled classes in order to compile, and now resolve **19%**. The rest are unchanged, because code that runs a solve needs the engine either way. Section 9 has the table.
- **How it was done.** The CodeLaser engine holds a model of the source. It answers questions about it with counts, prices a proposed change before it is made, and ranks the alternatives. Claude works from those measurements — not from reading the source — chooses the next step, and issues it as a single operation that CodeLaser applies across the whole project. Sections 2 and 3 describe the setup.
- **Where it ends up.** Nine modules with a fixed build order, the shape of which is below and again in section 9.

- **Two observations about the build.** The API-compatibility filter matches classes only, so interfaces, enums and annotation types are not compared. And `dependency:analyze` reports a test-scoped dependency as unused when the tests still need it. Section 11.



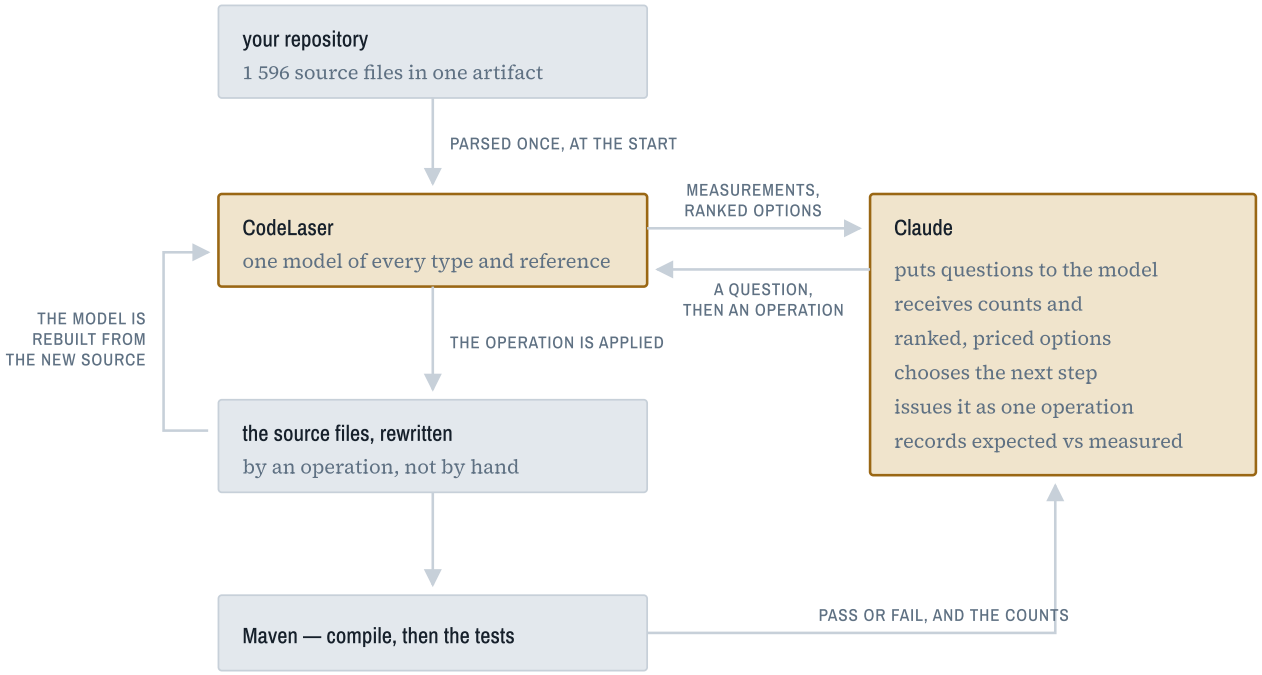
THE NINE FINAL MODULES AFTER THE SPLIT, AND WHICH IS BUILT AFTER WHICH

THE SETUP

2. The project setup

Four components were involved: the Timefold source, the CodeLaser engine, Claude Code, and the Maven build.

> In the rest of this document, **the engine** means the solver engine — the part of `timefold-solver-core` that runs a solve. CodeLaser is always called CodeLaser.



HOW THE PIECES WERE CONNECTED

The Timefold source. A fresh clone of `timefold-solver-core` at upstream commit `9adba5acf5`, on its own branch. The Timefold repository itself was never written to.

The CodeLaser engine. A refactoring and modernization engine. It reads the whole project once and builds a model of it — every type, every member, every reference from one to another. It answers questions from that model, and it changes code through operations rather than by editing text.

Claude Code. Drives the work. It puts questions to the model, receives measurements and ranked options back, chooses the next step from them, and issues it as one operation. It records what it expected each step to produce, and then what was actually measured.

The Maven build. Timefold's own build, unchanged. After each step it compiles the project and runs the whole test suite.

Nothing in this work is decided from a description of the code. Every question is answered from the CodeLaser model as it stands at that moment, and every result is measured again after the change has landed.

THE TOOL

3. How the CodeLaser engine works

This section describes what CodeLaser does that an editor and a compiler do not.

It holds a model of the code, not the text. The project is parsed once. What CodeLaser then holds is its structure: the types, what each one declares, and how the type hierarchies fit together. It also holds every reference from one place to another — calls, constructions, annotations, imports, and the javadoc links between them. The dependency graph follows from that.

So do the questions worth asking of it. "How many places create this class by name?" comes back as a number, not as a text search somebody has to read and judge. So does "what can this type still reach", and "what would this boundary cost".

It changes code as whole operations. Moving a set of classes is one instruction. It is one kind among many: this run used 59 different operations. Others extract a supertype, pull a member up a hierarchy, relocate static members, rewrite a declaration across every implementation of it, replace an expression wherever it occurs, or write a module descriptor.

Whichever it is, CodeLaser finds every place the change reaches and rewrites each one — imports, module declarations and build files included. One instruction in this run rewrote 5 603 places.

It refuses what it cannot do correctly. An operation that would leave the project not compiling is declined before anything is written. Almost nothing in this run was typed by hand. Where CodeLaser had no operation for a shape, that was recorded as a gap in CodeLaser rather than patched around.

It shows which types belong together. This is what decided where the boundaries went. From the type hierarchies and the call chains, CodeLaser reports which types form a group that reaches only inward. Such a group is a candidate module. For any set someone proposes, it reports how many references would still cross the line, and in which direction. Sets can then be compared by number instead of argued over.

The fifth separation was chosen that way. Three candidate sets were scored: 112 classes with one reference still crossing, 113 with none, and 140 with one again. The middle one was taken. It was the only one that left nothing behind.

It computes and ranks options. Given a proposed boundary it reports what that boundary would cost and what it would free, before anything is changed. In this run it also proposed relocations of its own and priced each one; ten of sixteen were taken.

It rebuilds its model after every change it makes. A change that leaves the project not compiling stops there, and the model is never out of step with the files on disk.

This run used 59 different operations across 548 calls. Some examples, from simple to more advanced:

OPERATION	WHAT IT DOES	WHERE IT WAS USED HERE
`query.dependentTypes`	lists every type that depends on a given one	to confirm a converted class was no longer named anywhere in production code
`graph.giantComponent`	reports the largest group of classes that all reach each other	measured the loop at the start, and after every step that was meant to shrink it
`remove.method`	deletes a method and every call to it	one edit, inside one of the nine moves of step 6
`rename.moveType`	moves one class to another package and rewrites every reference	`ScoreDirector`, 149 edits
`structure.moveTypesToSubProject`	moves a set of classes into a module, rewriting references, imports, module declarations and build files	349 classes into `constraint-streams`, 5 534 edits
`graph.splitReadiness`	reports, before anything is written, whether a proposed module can be separated and what stands in the way	run before each separation; it appears in 26 of the 276 scripts

SEVEN OF THE OPERATIONS USED IN THIS RUN

The first two and the last change nothing; they answer a question. Most of the API is of that kind — querying the dependency graph, measuring a proposed boundary, scoring one arrangement against another. Of the 548 calls in this run, 301 changed nothing.

THE STARTING POSITION

4. What the module system requires, and why the core cannot simply be cut

Throughout this document, **the core** means `timefold-solver-core` — the single artifact this work started from — and **the engine** means the part of it that runs a solve.

The target is the Java Platform Module System — JPMS — which arrived in Java 9. Three of its rules decide everything that follows.

1. **A module declares what it needs and what it offers.** Each module carries a `module-info.java` naming the modules it requires and the packages it exports. Both are enforced by the compiler. Code that uses a type from a package another module has not exported does not compile, even though the class is on the path.
2. **Modules are built in an order.** Module B can use module A if A is built first. Two modules cannot require each other, in either direction, at any remove.
3. **A package belongs to one module only.** The same package name cannot appear in two of them. A candidate module cannot take three classes out of a five-class package and leave two behind. Packages move whole, or not at all.

Rule 2 is what makes `timefold-solver-core` hard to cut. **978** of its classes reach each other in a loop: A uses B, B uses C, and C uses A again. No order puts any one of them first, so no boundary can be drawn between any two of them. In a loop of 978 classes that applies to every pair in it.

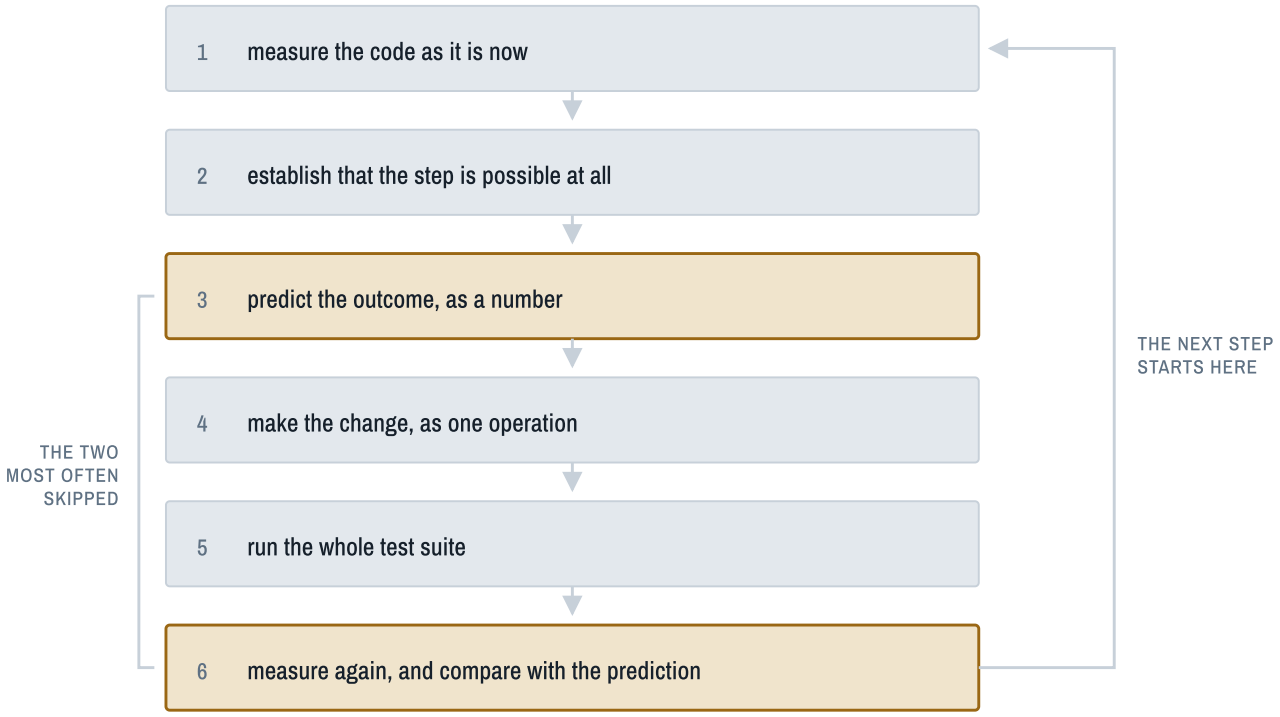
Moving files does not help. A class moved from one side of a proposed boundary to the other takes its dependencies with it, so the loop is the same size afterwards. The loop shrinks only when a dependency is removed, which is what the first three steps of the work did.

Rule 3 decided more outcomes than the dependency graph did. More than once a boundary the dependencies allowed was still unavailable, because taking it would have split a package across two modules.

METHOD

5. The loop each step of the work went through

The work followed the same loop at every step: measure the code as it stands, get back a ranked set of possible moves, take one, and check the result against what was predicted.



THE LOOP EACH STEP WENT THROUGH

1. **Measure the source itself.** Every figure in this document was derived from the Timefold source, on the branch the work ran on, at the moment it was needed. Nothing was carried in from elsewhere and nothing was assumed to still hold.
2. **Establish that a step is possible before pricing it.** Before any cost is estimated, CodeLaser is asked whether the boundary holds: `graph.splitReadiness` reports whether any package would end up split across two modules, and what the proposed module would still need from the one it is leaving. Nothing crossing means the step can go ahead. Operations can also be dry-run, and were, but a dry run checks one edit rather than deciding whether a boundary is sound.
3. **Predict the outcome as a number, then compare.** Each step was priced before it was made and scored afterwards. The tally is in section 12.
4. **Make the change with an operation, not by hand.** Every change to Java source goes through CodeLaser, which rewrites every place it reaches. Section 12 lists the few exceptions and what they were.
5. **Gate on the tests, not only on the compiler.** Everything is compiled, of course, and a change that does not compile never lands. But a clean compile says nothing about behaviour, so the standard is the whole test suite: no new failure, and a per-suite result identical to that step's own starting point.

6. **Score the step by measuring again.** A result was never taken from what an operation reported having done. It was taken from a fresh measurement afterwards.

From the complete chain, steps 3 and 6 are the two most often skipped. They are cheap to run and they are what makes a claim checkable afterwards: without a prediction there is nothing for the measurement to disagree with. Step 5 is the expensive one — a full install is about nine minutes and the acceptance suite about fifty.

WHAT WAS DONE

6. Stepwise overview of the split

	STEP	WHAT IT PRODUCED
0	Measure the starting point	The loop at 978 classes, the groups inside it, the references that close them, and a 344-suite test baseline to compare every later step against
1	Price the change that reduces the loop	Thirteen predictions of how much of the loop a container would release. All thirteen exact
2	Reduce the loop, with a dependency-injection container	Jakarta CDI introduced into the core, in two passes. The loop falls from 978 to 851
3	Straighten what still pointed the wrong way	851 to 814. The loop stops shrinking here; from this point on the work is separation
4	Separate five modules that use the engine, so they build after it	`util`, `neighborhood`, `search`, `constraint-streams`, `collectors` — 824 classes. Each takes its own tests with it
5	Separate two modules the engine uses, so they build before it	`score` and `domain` — 48 classes. Five serialization adapters rewire onto them and stop pulling in the solver
6	Clear the engine's own remaining wrong-direction dependencies	48 references from lower-level code up into the engine, reduced to zero. Without this the last separation is impossible
7	Separate the largest module	`core-model`, 396 classes, out from under the engine. No class a consumer writes changes name
8	Verify	Whole build green including dependency analysis and the API comparison; whole test suite matching the control; the module graph itself enforced by the compiler

THE RUN, STEP BY STEP

Reducing the loop, with a dependency-injection container

Much of the loop came from classes creating the objects they needed — either naming the class directly, or looking a name up at run time. The first is a dependency the module system has to honour. The second is a dependency no static check can see at all, which does not make it go away.

A dependency-injection container was introduced into the core to remove them. It is Jakarta CDI, with Weld as the implementation, declared `provided` so that it does not reach consumers of the library. Classes that used to create their collaborators now declare what they need and are given it. The dependency stops being a compile-time reference and becomes something the container satisfies when the solver starts.

This was done in two passes, then afterwards the remaining wrong-direction dependencies were straightened:

SIZE OF THE LOOP, AS EACH STEP CLOSED

The first pass converted five factory classes and deleted a block of static creation code — 5 files, 35 insertions, and 24 deletions. The second did the same for move selectors, deleting 87 lines. The straightening step was 21 edits across 3 moves.

At every one of those steps the test result was identical to the baseline.

Separating the modules

In a next phase five modules were taken off the top — parts that use the engine, so they can be built after it. Then two were taken from underneath — parts the engine uses, built before it. Then `core-model`, the largest of the nine at 396 classes: the types that describe a planning problem, lifted out from under the engine so that everything else can be built on them.

	MODULE	CLASSES	NOTES
1	`util`	32	no change to the published API
2	`neighborhood`	77	no change to the published API
3	`search`	252	first API change — nine extension points relocated
4	`constraint-streams`	349	49 test files relocated
5	`collectors`	114	`ConstraintCollectors` repackaged. Zero test files relocated by hand
6	`score`	27	taken from underneath. No test cost
7	`domain`	21	taken from underneath. `ConstraintRef` relocated
—	the engine's remaining dependencies	—	`ScoreDirector` relocated; wrong-direction dependencies reach zero
8	`core-model`	396	the largest single separation

THE NINE MODULES, IN THE ORDER THEY WERE SEPARATED

> The two taken from underneath cost far less than the five taken off the top. A module built **before** the engine is still visible to the engine's tests, so no tests have to move. The five taken off the top had to take their tests with them.

PHASE BY PHASE

7. The nine steps, in more detail

The same nine steps as the table above, with what each was for and what it took. A step ended only when the whole test suite matched its own starting point.

Step 0 — measure the starting point

Goal. To know exactly what we were starting from, before changing anything.

In numbers. The project parsed to 4 213 types, and six quantities were measured:

1. the size of the loop — **978** classes;
2. how many separate groups of mutually-reaching classes exist — 128;
3. how many types those groups hold in total — 1 369;
4. how many references close them — 898;
5. how many pairs point the wrong way — 212;
6. the test suite — 344 suites, which became the baseline every later step was compared against.

Step 1 — price the change that reduces the loop

Goal. Most of the loop is caused by classes creating other classes by name at run time. Before changing any of it, we wanted a number: how much of the loop would that actually release?

In numbers. Thirteen predictions of what the change would release, made before any of it was written, and all thirteen exact.

Step 2 — reduce the loop, by introducing a container

Goal. To stop classes creating the objects they need, and have a dependency-injection container supply them instead. Jakarta CDI, with Weld, declared so that it does not reach consumers. A class that receives its collaborators no longer names them, and the compile-time dependency goes away with the naming. CodeLaser primitives were used that facilitate introduction of container construction and all downstream effects of it.

In numbers. Two passes. The first converted five factory classes and removed a block of static creation code — 5 files, 35 insertions, 24 deletions. The second did the same for move selectors and removed 87 lines.

Step 3 — straighten what still pointed the wrong way

Goal. A module has to be built before anything that uses it. So for a set of classes to become a module, every reference must run one way: from the rest of the core into that set, never back out of it. The container change removed most of the references running the wrong way. The ones left had to be turned round one at a time — by moving a type, by inverting a call, or by giving the caller an interface to name instead.

In numbers. Thirteen steps, each gated, with the test result unchanged at every checkpoint.

Step 4 — separate five modules that sit above the engine module

Goal. To take out the parts that **use** the engine, which can therefore be built after it. These are the easier direction, but each one has to take its own tests with it. Which types went into which module was decided the way section 3 describes. CodeLaser was asked, for each candidate set, how many references would still cross the boundary. The sets that left none were the ones taken.

In numbers. `util`, then `neighborhood`, `search`, `constraint-streams` and `collectors` — 824 classes between them. The first moved 32 classes and their 8 test classes as a single `structure.moveTypesToSubProject` call, which made 2 052 edits across the project to keep it compiling.

Two of the four published API changes are here: the nine extension points that leave `search`, and the repackaging of `ConstraintCollectors` that lets `collectors` stand apart from the core. Both are in section 8.

Step 5 — separate two modules from underneath

Goal. Code that only needs to talk about a score, or about the shape of a planning problem, was compiling against the entire engine. The serialization adapters are the clearest case. Two modules were taken out **below** the engine, one after the other: the score types, then the domain types. A module built before the engine is still visible to the engine's tests, so no tests move.

In numbers. 27 classes, then 21. The first of the two was a single operation that made 5 603 edits. **The third published API change is here**, in the domain carve: `ConstraintRef` moves to a package that can be separated. Afterwards the JPA adapter compiles against one module of 63 KB. It previously needed two artifacts of about 1 690 KB. The jaxb and jackson adapters end up at 81 KB, roughly 1.5% of the stack, against about 58% before. The second published API change is here.

Step 6 — clear the engine's own remaining dependencies

Goal. The last separation is impossible while lower-level parts of the core still reach up into the engine. Those references had to reach zero. This is the least visible phase and one of the most important.

In numbers. It started at 48 such references across 31 targets and ended at **zero**, in nine separate moves. **The fourth and last published API change is here**: `ScoreDirector` returns to `api.score.director`, which took 149 edits.

Step 7 — the largest separation

Goal. To take the model of a planning problem out from under the engine.

In numbers. 396 classes moved, spread over 79 packages — the largest separation of the nine, and about a quarter of the core. They are the types that describe a planning problem: the model a user writes against, and the machinery that reads it. Everything else in the core is built on top of them, which is why they come out underneath rather than above.

Before anything was written, CodeLaser primitive `graph.splitReadiness` was asked whether the boundary held: whether any package would end up split across two modules, and what the proposed module would still need from the one it was leaving. It reported neither problem, and the move went ahead.

⭐ **Nearly four hundred classes changed module, and no consumer has to change a line.** A class keeps its package name when it moves to another module, so every import stays valid. What could have broken is visibility: a module only sees packages that a module it requires has exported. The core declares the new module as `requires transitive`, which passes that visibility straight through, so anything that could be named before can still be named.

Step 8 — verify

Goal. To check the finished state properly rather than infer it from the last step having gone well.

In numbers. The whole build, including dependency analysis, the API comparison and the framework's build-time step: no errors. The whole test suite against the control: identical, module for module. The module graph itself is checked by the compiler, because each module now declares what it is allowed to depend on. **This is the slowest part of the whole exercise.** A full install takes about nine minutes and the acceptance suite about fifty. It was run at every step that could have moved it, not only here at the end.

THE TRADE-OFF

8. Four changes to the published API, and what each unblocks

Four types changed package or were re-declared. Two happened in step 4, one in step 5 and one in step 6. Each is described below with what forced it.

Three of the four are consequences of the module system rather than choices. A package cannot live in two modules. So a type that has to end up in a different module from its neighbours must first leave the package it shares with them. The fourth was a judgement call.

Every one of them removed an obstacle that was blocking a separation. That is the case for making them, and it is a case that can be argued with.

One — nine extension points, out of `search`

Nine public extension points move out of `impl.heuristic.selector` and `impl.partitionedsearch.partitioner` into the configuration layer.

What it costs a user. The relocation changes the signature of every `*Config` accessor that returns one of them. Code that calls those accessors, or implements one of the extension points, is affected.

How it is recorded. As one entry in `revapi-differences.json`, naming all nine types. The entry matches the old package only. A later break somewhere else will therefore still be reported, rather than being

covered by this entry.

Two other options were considered and dropped. Reverting the relocation would undo the separation. Raising the API baseline would silence every other difference at the same time, not only this one.

Two — `ConstraintCollectors` gets its own package

`ConstraintCollectors` moves from `api.score.stream` to `api.score.stream.collector`.

Why it is forced. The `collectors` module needs 113 classes. 112 of them can move. The 113th is `ConstraintCollectors`, and it holds the last dependency. But it sat in `api.score.stream`, whose other fourteen classes stay in the engine — and a package cannot exist in two modules. The class had to be given a package of its own before anything could move.

What it costs a user. `ConstraintCollectors` is the class whose static methods are handed to `groupBy` — `count()`, `sum()`, `toList()` and the rest. Any constraint that groups names it, usually through a static import. The class, its methods and their behaviour are unchanged. What changes is the package in the import line, and the migration rule that shipped with the move rewrites it.

Three — `ConstraintRef` moves, and a smaller option was chosen

`ConstraintRef` moves to `api.score.stream.common`, and one static method moves into it.

Why. A single static call inside `ConstraintRef's constructor was dragging the constraint-building API across the boundary.

The choice. Two alternative moves were considered. The other one needed no code change at all, but relocated five public types including `Constraint` — which appears in every user's `defineConstraints` method. The path chosed now costs one internal edit and moves one data type instead.

Four — `ScoreDirector` returns to `api.score.director`

What it is. A restoration rather than a new location. Timefold v2 moved this type out of `api.score.director`; this puts it back. The existing migration rule is therefore inverted rather than deleted: code written against v1 is correct again and is left alone, and v2-era code is migrated.

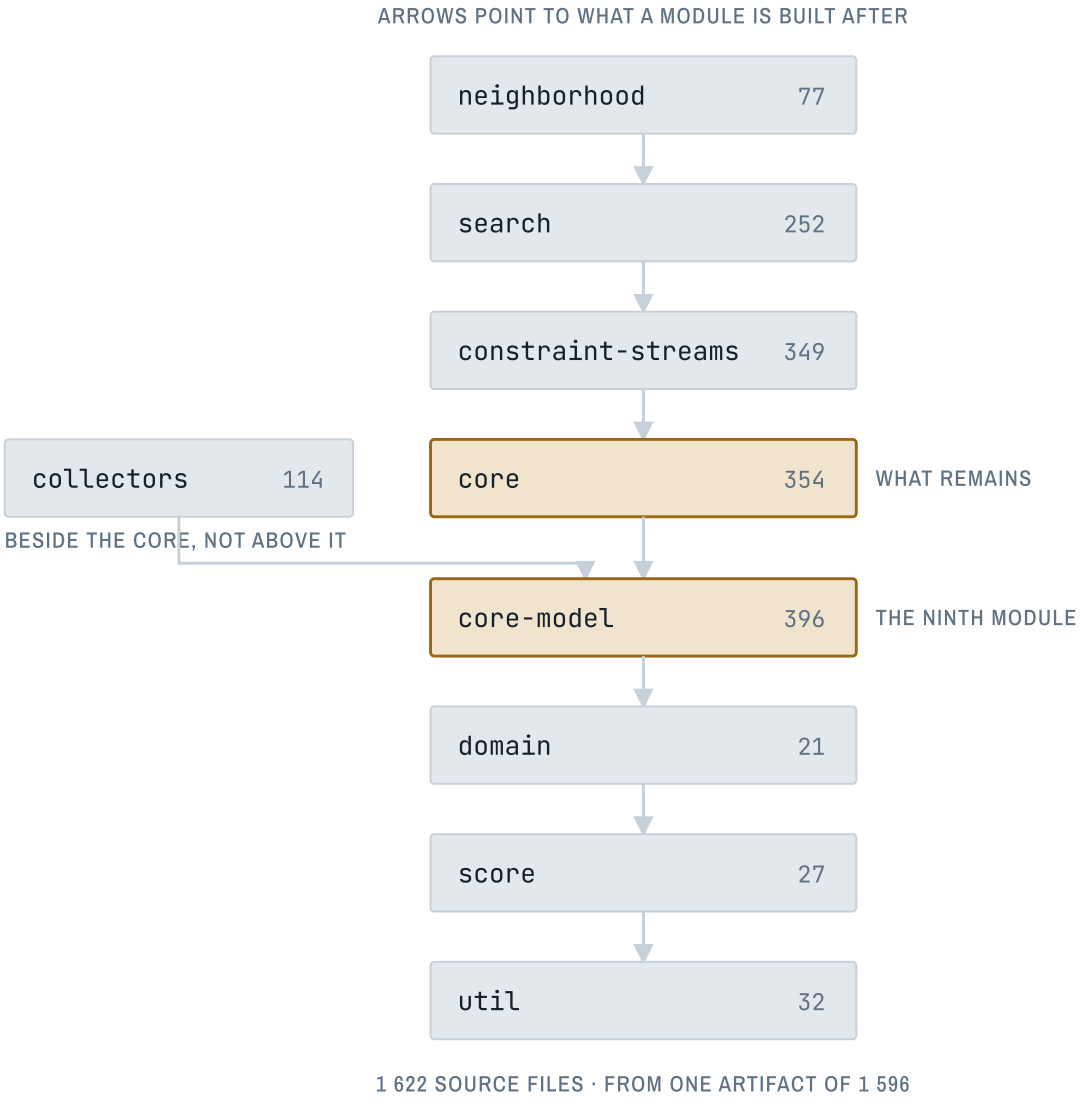
Why it was worth doing. The last four wrong-direction dependencies in the engine were all held by this one class, named by four configuration extension points. Those four are interfaces that users implement. Moving one class is one change instead of four.

⚠ **This one was put to the operator rather than decided in the run.** Whether a break in a published API is acceptable depends on the release, the users and the timing. None of that is visible from the code. The alternative — leaving four dependencies unresolved and stopping there — was measured and offered alongside it.

> Each of the four shipped with its migration rule, written in the same commit. Downstream code is moved by tooling rather than by a release note.

THE RESULT

9. Nine modules, built in a fixed order



THE NINE MODULES AND THEIR BUILD ORDER

What it buys, and for whom

The `timefold-solver` repository holds twenty-four modules besides the solver itself: persistence adapters, framework integrations, tools, benchmarks and integration tests. Each of them builds against the solver.

What each one has to pull in, to compile. Take one of them — the JPA adapter, say. Look at every class its own code mentions. Then every class those classes mention. Keep going until nothing new turns up. That whole set has to be present for it to compile. Add up the size of those class files, and divide by the size of all nine solver modules together, which is 9 934 KB. That is the percentage below.

MODULES, GROUPED BY WHAT THEIR CODE NEEDS	HOW MANY	BEFORE	AFTER
name only a `Score` type	5	1%	1%
persistence adapters: name a score and a planning model	3	2%	2%
declare a planning model and a configuration, but never construct a solver	**11**	**45%**	**19%**
construct and run a solver	2	45%	45%
run benchmarks	1	62%	62%
extend the framework integrations	2	78%	78%

WHAT EACH OF THE TWENTY-FOUR MODULES HAS TO PULL IN, BEFORE AND AFTER

⭐ **The third row is the result.** Eleven of the twenty-four modules describe a planning problem and hand it over. They never construct a solver, so they never needed the engine — but before the split there was nothing smaller to depend on than the whole artifact. They now build against a quarter of what they used to.

⚠ **And the honest half: thirteen of the twenty-four do not move at all.** Eight of those were already at 1–2% and had nothing to gain. The other five construct a solver or extend the framework integrations, and they need the engine — the engine is most of the bytes, and no arrangement of modules changes that. The gain is real, and it is for one kind of consumer. The table shows which.

Every module in the build with compiled code that is not part of the solver is a row here. Nothing was left out to make the table read better.

1 622 source files across nine modules, from one artifact of 1 596. Every module carries a declared list of what it may depend on, and the compiler enforces it.

ASSURANCE

10. How we know nothing broke

A control build was held at a fixed commit and never written to. After each significant step the whole project was tested and compared against that control, module by module rather than on totals.

The control sits partway through the work, at the commit immediately before the fourth separation. It already carries the container change and the first three separations. So the comparison covers everything from that point on — the remaining separations and both carves, which is where most of the file movement happened. The original starting point was never run through the full suite, so we make no claim about the phases before the control.

	SUITES	TESTS	FAILURES	ERRORS
the control, at the fixed commit	595	5 944	91	72
after the first separation from underneath	595	5 944	91	72
a later step, first attempt	595	5 944	**92**	72
the same step, after the fix	595	5 944	**91**	72
the largest separation, first attempt	595	5 944	91	**117**
the same separation, after the fix	595	5 944	91	**72**

THE CONTROL, AND EACH STEP MEASURED AGAINST IT

⭐ The two marked rows show the comparison doing its job. A comparison that never disagrees is not evidence that anything is being checked. This one disagreed twice: one extra failure, then 45 extra errors. Each time the work stopped until the numbers returned to the control.

⚠ **About the 91 failures and 72 errors.** They are present in the control, so nothing measured here introduced them. We are not claiming they were present before this work started, because that was never measured. The working record puts most of them down to two causes: the container change made earlier in this work, and a third-party tool used by one module. We did not fix them. They are quoted only because the claim is a comparison, and the numbers mean nothing without one.

The finished state also builds clean: the whole project installs with no errors, including dependency analysis, the API comparison and the framework's build-time step.

ASIDE

11. Remark — two build checks cover less than they appear to

Separating a module means moving types between artifacts. Two checks in the build exist to catch what that can break. `revapi` compares the published API against the previous release. `dependency:analyze` reports declared dependencies that are not used. Both were relied on during this work, so both were examined.

Each turns out to cover less than its name suggests. Neither affects any result above, and both predate this work. They are written down because they are useful to know about.

The API-compatibility filter matches classes only

`core/src/build/revapi-filter.json` matches `class ai.timefold.solver.core.api.*`. The tool spells an interface `interface X`, an enum `enum X`, and an annotation type `@interface X`. **None of those have ever matched.**

Measured against the 2.1.0 baseline: `api.solver` holds 13 types and the check reported 2 — the two that are classes. `api.domain.solution` holds 9, none of them a class, and the check reported nothing at all.

Widening the filter and re-running surfaced **92 further removals** — 66 interfaces, 21 annotation types, 5 enums — and no other kind of difference. Declared removals go from 55 to **147**.

⚠ Nothing shipped that should not have. The changes were deliberate and are on the record. The point is only that this category of type was not being compared.

⚠ There is a second half to it: the check only runs at `install`. Neither a compile nor a test run reaches it, so a change can be green everywhere a normal build looks and still fail.

`dependency:analyze` reads main code and test code differently

`dependency:analyze` correctly reported that `collectors` no longer needs the engine, and the dependency was removed. The test suite then produced 45 errors. The tests drive a class from the engine, and a test-scoped dependency does not bring in the artifact it was built from.

Both tools were right about what they measure. Neither measures the other.

EFFORT

12. Effort required to compute and perform the split

Included because it bears on whether the exercise is repeatable, not as a comparison with what it would take by hand.

	VALUE
runs submitted	369, of 276 distinct scripts
commits	50
files changed	1 861, +8 015 / −4 167
predictions made and scored	102 — 84 correct, 6 partial, 12 wrong

THE RUN

One instruction is not one edit. A single operation moving a set of classes rewrites every reference to them across the project.

ONE OPERATION	EDITS	WHAT IT MOVED
move a set into a module	**5 603**	27 classes
move a set into a module	**5 534**	349 classes
move a set into a module	**4 166**	252 classes
relocate one class	353	the repackaging in section 8
relocate one class	149	`ScoreDirector`

WHAT SINGLE OPERATIONS PRODUCED IN THIS RUN

⭐ The `score` module is the clearest case. 27 classes moved, and the operation made **5 603 edits** to keep the project compiling.

Almost nothing was typed by hand. Every Java edit went through a refactoring operation. The exceptions are enumerated: two one-line edits, five module descriptors, fourteen export grants, three whole-file rewrites, and eleven repairs of CodeLaser's own output. That is 35 files, in a diff of 1 861.

⚠ **These are counts of work, not of time.** How long the exercise took is not reported here: most of it is the build waiting, and it says more about the machine than about the method.

APPENDIX

A. What was not done, and why

- **No tenth module.** A user-facing API split fails because `api.solver` genuinely mixes model and configuration types. A separate descriptor module has no consumer that wants descriptors without also building a solver. Both were measured and refused rather than attempted.
- **206 documentation links were downgraded, and left that way.** This one needs explaining, because it is a defect in CodeLaser rather than a decision.
  During the largest separation, the move operation rewrote 206 `{@link}` references into `{@code}` in 114 files that did not move, across ten modules. In generated javadoc a `{@link}` is a hyperlink and `{@code}` is plain text, so the name is still written out and nothing fails to compile. It is a loss of navigation in the documentation, and no check reports it.
  It should not have done this. At the moment the move runs, the new module has no descriptor yet. So the operation cannot see that the reference will still resolve, and it takes the safe option. The reference does resolve: `core` passes the new module through to everything that reads it, and all ten of those modules read `core`.
  We left them alone. Re-linking 206 references across 114 files is a larger and riskier edit than the loss it repairs, and the fix belongs in the operation rather than in this codebase.

APPENDIX

B. Why the test suite, and not the compiler, was the gate

Section 5 gives running the whole test suite as the standard at every step, rather than a clean compile. A clean compile is faster and it is what most refactoring tools stop at. Three problems in this work show why it is not enough: each one compiled, and each was found by running.

Code that names a class only when it runs. A test that loads a class by its name in a string has no visible dependency on it. Nothing in the source says the two are connected, so no analysis of the source can say so either. The largest separation was expected to break tests of this kind, and the number was predicted before the move: 151. The first suite run afterwards produced 23 failures and 128 errors across 22 test classes — 151.

A framework that is told which artifacts to scan, by name. Quarkus builds an index at build time from a list of artifacts. When annotated classes moved into a new artifact, that artifact was not on the list, so the index no longer covered them. The compile was clean and the build-time step failed. The remedy was one line naming the new artifact.

CodeLaser changing files it was not moving. Six annotated varargs declarations and five imports were printed back wrongly in files outside the move, and one `provides` declaration was deleted where a `requires` should have been added. The compiler found all of them and each was repaired. The 206 documentation links in appendix A are the same class of problem, and are the one case nothing reported.

⚠ **And a weakness in the procedure, from the same record.** For three steps the routine check ran one module's tests rather than the whole project's, because it is much faster. Problems accumulated unseen during that window and had to be paid off later, all at once. The check was widened to the whole project afterwards, and it is the reason the acceptance figures in section 10 are whole-reactor runs rather than one module's.

APPENDIX

C. How the figures here were checked

Every figure was verified against the file it is attributed to, mechanically rather than by reading back: 22 of 22 present in every source claimed.

The record is checkable in a second way. Runs are numbered as they are submitted, and the archive holds numbers 1 to 369 with no gaps, so no run is missing and every claim traces to one.

Where two sources disagreed, the underlying register settled it rather than a preference. One count of decisions differed between a summary and a narrative; counting the register directly gives 48.