A reading path through `docxwatermarker`

Fabio F.G. Buono

2026

Version 1

This guide proposes an order for reading the repository, for a student meeting code and formal specification together for the first time, or for a teacher looking for a worked example. It points to the other documents and leaves the detail to them. The Italian edition is *Un percorso di lettura attraverso `docxwatermarker`*.

1 Why this order

Most courses teach Hoare logic on a three-line of `while` loop and teach real code in projects that have no formal specification anywhere. The two never meet. This repository puts them in one place, so the thing worth following is not either one alone but the thread that ties them together. That thread runs through a piece of code, the property it is supposed to have, and the test that fails when the two drift apart. The path below follows it three times, at three levels.

2 First pass: one operation, end to end

Start with a single library operation and see it from all three sides before widening out.

1. **The code.** Open `src/docxwatermarker/core.py` and read `Template.replace_image`. It takes a template and an image and returns a new template with one image swapped. Notice that it returns a new object and never mutates the original, and that near the end it calls `ensure(...)` with a `spec="I2"` argument.

2. **The specification.** Open `SEMANTICS.md` (or the typeset `SEMANTICS.pdf`) and find the triple for `replace_image` and the invariant I2, nameset preservation. Read the Hoare triple as a contract. Given these preconditions, the returned template satisfies these postconditions. The invariant I2 is the property that the set of archive entries is unchanged by the swap.

3. **The link.** Look again at the `ensure(spec="I2")` call in the code. That `spec="I2"` is the same I2 you just read in the document. The two are not connected by a comment that can rot. They share an identifier. The *Axiomatic specification* section of the README explains the mechanism.

4. **The guardian.** Open `tests/test_spec_crossref.py`. It reads every `spec=` out of the code and every invariant out of the document and checks that the two sets agree. Rename I2 in one place and this test goes red. Run the suite once so you have seen it pass, then you know what its failing would mean.

3 Second pass: the design that makes the contracts simple

With one operation understood, read why the whole library is shaped the way it is, because the shape is what makes the specification short.

1. The *No XML manipulation* and *Template is immutable* notes in the README. The library treats the `.docx` as a ZIP and swaps one entry's bytes, and every operation returns a new template. These two choices are why the specification can speak of pure functions on archives, with no mutable state to reason about.

2. `src/docxwatermarker/_invariants.py`, the `require` and `ensure` primitives. These are the design-by-contract checks the code carries at runtime, and the `spec=` argument is what ties a check to a clause. The companion reading is Meyer on design by contract, cited in the specification.

3. The preliminaries of `SEMANTICS.md`, the sets and functions the triples are written over. Read them now, after the code, so the abstract names (Archive, Template) attach to the concrete objects you have already seen.

4 Third pass: the command, as a process

The library is pure functions. The command-line tool is a process, and it needs a different kind of semantics. This is the part a course rarely shows, two styles of formal specification on one codebase, each where it fits.

1. `src/docxwatermarker/cli.py`, the function `cmd_stamp`. Read it as a sequence of phases, resolve the source, open the template, build the image, swap, save, and optionally produce a PDF, each able to stop with a numeric exit code.

2. `OPERATIONAL.md` (or `OPERATIONAL.pdf`), which models that function as a transition system. The configurations pair a phase with a state, the terminal configurations are the exit codes, and the rules are the phases you just read in the code. Compare the rule for each phase with the corresponding span of `cmd_stamp`.

3. The exit-code guardian, again in `tests/test_spec_crossref.py`, in the operational part. It extracts the codes the command returns and checks them against the codes the operational document tabulates. This is the operational counterpart of the `spec=` link, the same idea applied to a different kind of specification.

4. The section *Command syntax and format recognition* in the same operational document. It describes the command line with a BNF grammar and the library's format check with a finite automaton, written as a transition table. Read it for a second reason as well. It states plainly why the library's pure functions are not drawn as automata, and why one internal recognizer is. The choice of which tool fits which object is part of the lesson.

5 What to take from it

By the end the three files this guide has walked through, the code, the specification, and the tests, should read as one connected piece. The code does the work, the specification says what the work guarantees, and the tests refuse to let the two disagree. The lesson is not any single triple or rule. It is that a specification kept honest by a test is a living description, where a comment or a line-number reference would have rotted the first time the code moved. A small piece of engineering can carry that, and this one was built to show it.