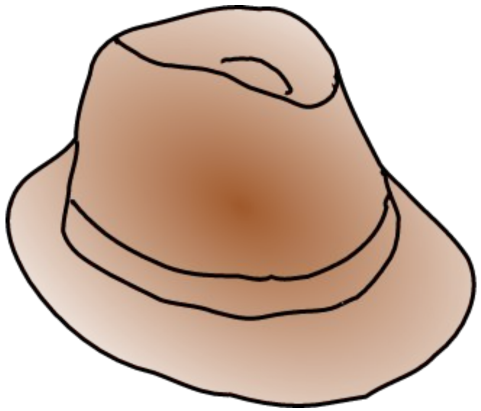
Workflows & Economics of Refactoring

CODESAI

Let’s consider two different thinking modes when you're programming

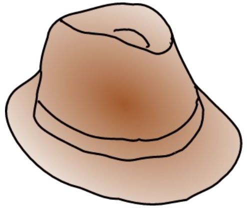
The metaphor of Two Hats



Refactoring

Adding Function

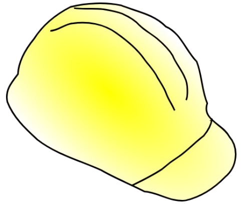
You can only wear one hat at a time



Refactoring hat

When refactoring, every change you make is a small behavior-preserving change. You only refactor with green tests.

Focus on good design



Adding functionality hat

Any other change to the code is adding function. You will add new tests and break existing tests.

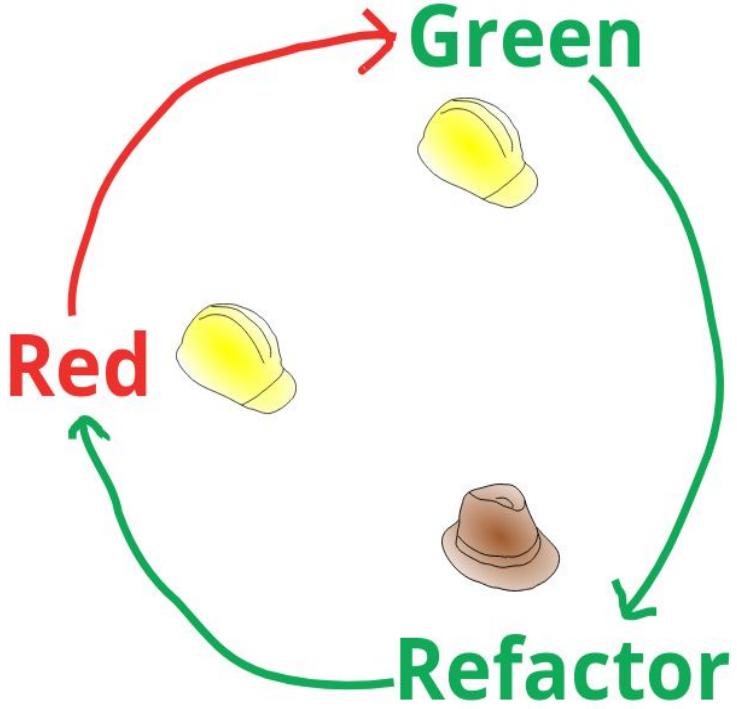
Focus on the problem

Workflows of Refactoring

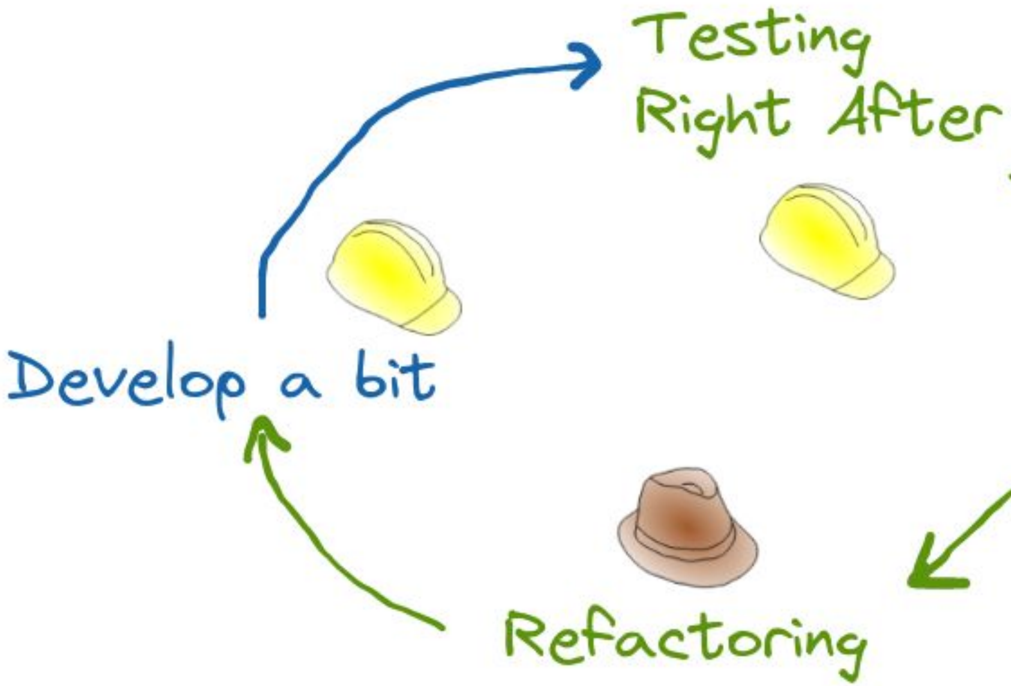
Incremental Refactoring

- Code is continuously refactor while it’s being developed.
- **Small refactorings are applied “mercilessly”** enough, so that large refactorings are rarely needed.
- Refactoring produces **better results and is cheaper if done regularly.**

Incremental Refactoring using TDD



Incremental Refactoring not using TDD



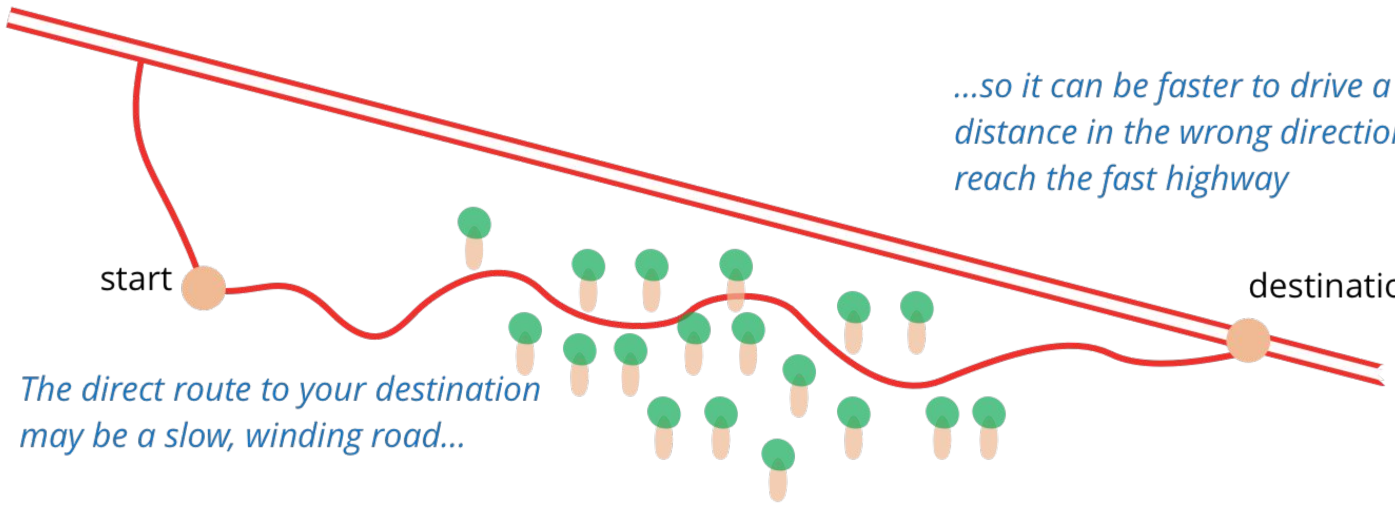
Preparatory Refactoring

Sometimes **adding a new functionality is hard**, so **you begin by refactoring existing code**:

That preparatory refactoring, can make adding the new functionality much easier.

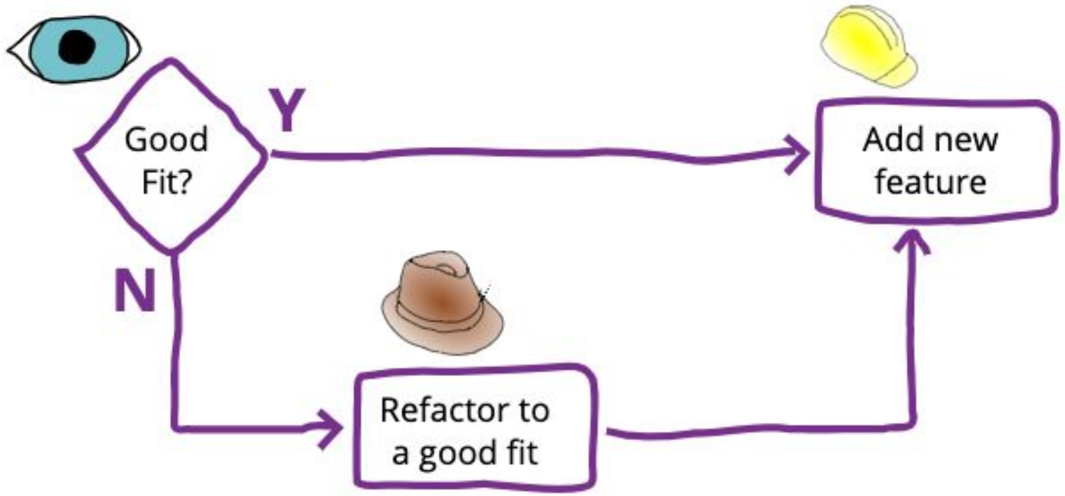
Often this has the benefit of making the overall change faster than trying to add the change without the preparation.

Preparatory Refactoring metaphor


...so it can be faster to drive a short distance in the wrong direction to reach the fast highway
start
destination
The direct route to your destination may be a slow, winding road...

Jessica Kerr's metaphor in An example of preparatory refactoring post

Preparatory Refactoring can be done when starting a new task



Comprehension Refactoring

Ward Cunningham explained it like this:

Whenever you **have to figure out** what the code is doing, you are building some understanding in your head. Once you've built it, **you should move that understanding into the code** so nobody has to build it from scratch in their head again.

Litter-Pickup Refactoring

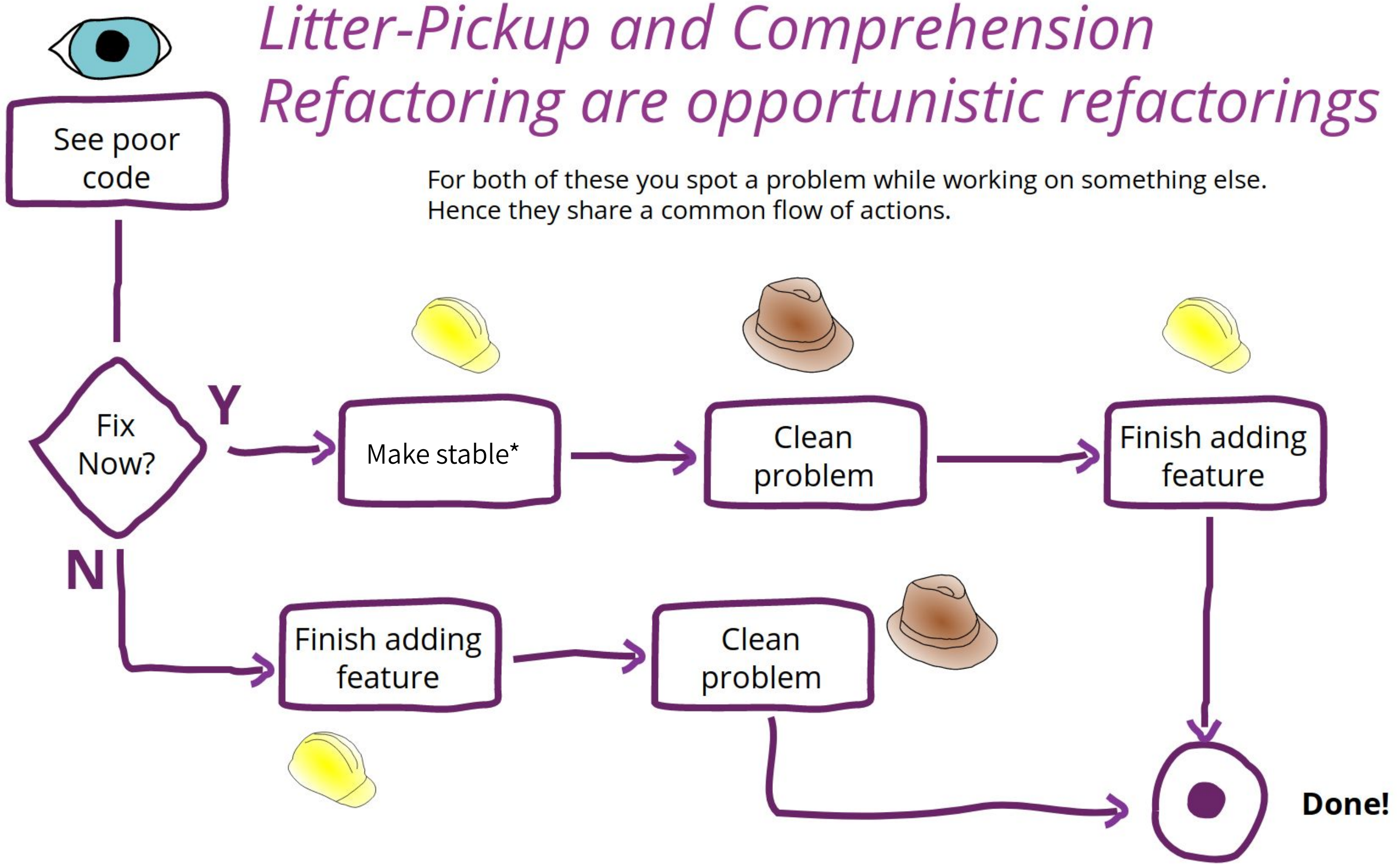
- This is often referred to as the **boy-scout rule:**

  “Always leave the code better than when you found it.”

- By cleaning up code as we work in it, we make things quicker for us the next time we need to work with it.

Litter-Pickup and Comprehension Refactoring are opportunistic refactorings

For both of these you spot a problem while working on something else.
Hence they share a common flow of actions.



* Make stable **using TDD** means: Get to green.

* Make stable **not using TDD** means: Get to green and add tests for new code.

Incremental Refactoring,
Preparatory Refactoring,
Comprehension Refactoring and
Litter-Pickup Refactoring

are **opportunistic refactorings**.

Planned Refactoring

- Planned refactoring (scheduled fix in large areas) is a **necessary element of most teams' approach.**
- However it's also a **sign that the team hasn't done enough refactoring using the other workflows.**

Long-Term Refactoring (I)

- Some restructuring requires large-scale changes. This can still be done using refactoring.
- **The team needs to agree** not only on a **rough end-state** but also on a **rough plan** to get there.

Long-Term Refactoring (II)

- During their regular work, developers move the architecture **towards** the desired direction.
- Since all changes are refactorings, the code base can **remain in a working state** even as features are added, (see Parallel Change, Narrowed Change, etc).

Is refactoring wasteful rework?

Economic justification

Refactoring makes it easier to **understand** the code - which makes subsequent **changes quicker and cheaper.**

Don't refactor unless you think you will recoup your investment later by quicker work

Balance with feature delivery

Refactoring should be done in conjunction with adding new features.

It is a gradual **improvement through many passes** through the codebase.

Summary: workflows

- Opportunistic refactorings
  - Incremental Refactoring
  - Litter-Pickup Refactoring
  - Comprehension Refactoring
  - Preparatory Refactoring

- Planned Refactorings
  - Short and Long-Term Refactorings

Summary: main ideas

To use refactoring in its most effective way, you need to **combine all the refactoring workflows,** into your development work.

If your code base is **difficult to understand and modify**, that's a sign that **you need to do more refactoring.**