How to refactor

Refactoring should be done as a series of small changes, each of which makes the existing code slightly better while still leaving the program in working order.

Checklist of refactoring done *right way*

☑ The code should become cleaner.

If the code remains just as unclean after refactoring… well, I'm sorry, but you've just wasted an hour of your life. Try to figure out why this happened.

It frequently happens when you move away from refactoring with small changes and mix a whole bunch of refactorings into one big change. So it's very easy to lose your mind, especially if you have a time limit.

But it can also happen when working with extremely sloppy code. Whatever you improve, the code as a whole remains a disaster.

In this case, it's worthwhile to think about completely rewriting parts of the code. But before that, you should have written tests and set aside a good chunk of time. Otherwise, you'll end up with the kinds of results we talked about in the first paragraph.

☑ New functionality shouldn't be created during refactoring.

Don't mix refactoring and direct development of new features. Try to separate these processes at least within the confines of individual commits.

☑ All existing tests must pass after refactoring.

There are two cases when tests can break down after refactoring:

- **You made an error during refactoring.** This one is a no-brainer: go ahead and fix the error.
- **Your tests were too low-level.** For example, you were testing private methods of classes.

  In this case, the tests are to blame. You can either refactor the tests themselves or write an entirely new set of higher-level tests. A great way to avoid this kind of a situation is to write **BDD-style** tests.

READ NEXT

Code Smells

Home
Refactoring
Design Patterns