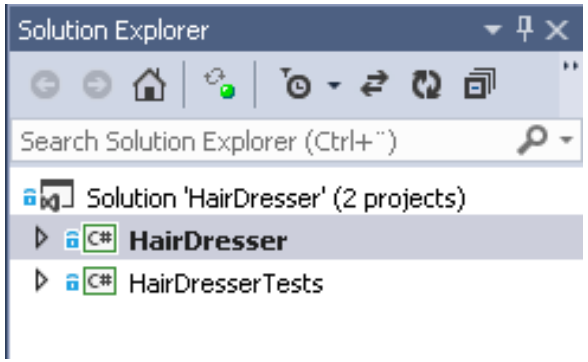
A testing mentality: Fitting tests into the process puzzle

Solution structure

Put tests in a **different project** than the code they're testing.

That way, you don't have to ship your test code off into production. (such as a frontend) will.



Continuous Integration

A principle that extreme programming teaches us is this: **make common workflows cheap**. Make them super-easy. Whenever possible, make the common operations *recurring*, daily, weekly, or monthly.

Continuous integration aims to make merging cheap.

Continuous Merge

More to the point, the whole team **constantly merges** into the main branch, usually several times a day. Automated tests run for each merge, either locally on people's computers, or on a build server.

A bit like with **optimistic concurrency**, the idea here is to make each merge small, minimizing its surface area against other "parallel" work by other developers.

Commit early and often

EDUMENT

Reduce the cost of committing by committing often. Make commits as small as possible, but no smaller.

Basically, this is the **Single Responsibility Principle**, but for commits. You want one commit per task. This makes commit messages easier to write.

Good commit messages lead to commits that are easier to understand and review, and so make it easier to reason about changes that happened in the past.

Commit early and often

Basically, the more you make your commit history focus on cohesive, logical chunks of work, the more valuable that history will be.

In more modern version control systems such as Git, commits can even be used to store away *partial* work, since it's possible to come back to a commit later and amend it, squash it into some other commit, or simply edit it out of the history.

Continuous review

Continuing the theme of lowering friction and reducing barriers, is there any way we can reduce the friction of reviews?

The idea of **continuous review** is to review the code *as it is written*. This is the minimal distance, in time and in space, that we can have between the act of writing code and reviewing it. Feedback flows immediately into the ongoing work.

Pair programming is a very common name for *continuous review*.

There are plenty of variants, including: Distributed pair programming, ping-pong pair programming…

Pair programming; research

EDUMENT

- **Williams 2001**: Improvement in correctness by ~15%, 20%-40% decrease in delivery time (though 15%-60% increase in total programmer-time.)
- **Lui 2006**: When everyone's a novice, pairing shows bigger gains than when everyone's an expert. *(Highlighting the exploration/invention aspect of things.)*
- **Arisholm et al. 2007:** Complex systems more correct; simple systems developed faster. Increase in effort.
- **Lui, Chan, and Nosek 2008**: Pair programming outperforms in design tasks.
- Meta-analysis, **Hannay et al. 2009**: You cannot expect faster and better and cheaper. Suggests publication bias. The benefit of pairing is greatest when the task isn't yet fully understood.

Frequent releases

EDUMENT

When continuous integration is the norm, and the **master** branch always passes the tests, the barrier lowers to cutting a release.

Some projects switch to a **time-based release schedule** rather than a feature-based release schedule. Branches with new features/fixes/refactors are integrated into `master` as they are mature enough, and the release happens from `master` at fixed, predictable dates.

The simplifying part here is that the continuous integration makes sure that a release can be cut from `master` at any time. Done correctly, it can also lead to less release manager burnout.