

Re: TDD and Refactoring

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2003-12/1454.html>

From: Philip (*phlip_cpp_at_yahoo.com*)

Date: 12/24/03

Date: Wed, 24 Dec 2003 14:34:39 GMT

Harry Erwin wrote:

> *I'm running an experiment with test-driven-development and refactoring*
> *as the development methodology for a program that makes use of the swing*
> *GUI and does a lot of complex mathematical modeling with many special*
> *cases in the business logic. Lessons learned so far:*
> ...

7. Low Hanging Fruit. Given a design with more than one poor qualities, don't leap at the biggest refactor that should lead straight to the most design improvement. (Example: Turning a rambling tree-shaped object model into the Composite Design Pattern.) Instead, perform the simplest stoopidest refactor that gives a small but leading benefit. (Example: Extract Method on some dumb sequence of 3 statements with no arguments.) Repeat, under a little guidance and prediction, towards that advanced design, until the refactor producing that design is the only simplest one left.

8. Grind it Till you Find it. If two things look similar, make them more similar, then make them the same. Example:

```
    for (int x = 0; x < 6; ++x)
        for (int y = 0; y < 6; ++y)
            something(x, y);
...
    for (int y = 0; y < 6; ++y)
        for (int x = 0; x < 6; ++x)
            somethingElse(x, y);
```

Something() and SomethingElse() may or may not have a valid reason to go in a particular order – row minor or row major. That reason may or may not show up as a test fault if you arbitrarily switch the order of the two for() statements.

But if switching the order of the second two for() statements passes tests, and passes a code inspection (maybe stepping thru with the debugger), then something() and somethingElse() are candidates for the "Introduce Execute-Around Refactor":

comp.object: Re: TDD and Refactoring

```
for (int x = 0; x < 6; ++x)
  for (int y = 0; y < 6; ++y)
    (this->*pSomethingOrElse)(x, y);
```

That, in turn, implies that new code added inside something(), somethingElse(), or a new somethingNew() in the same position now has one more reason to work in row minor order.

Deprecation Refactor: A variation of the Replace Algorithm Refactor. Given a gap between the model you have and the model you want too large to comfortably cross incrementally, just write a new model in parallel with the old one. Comment the old one "don't use me any more". Grow the new one test-first. Take a single client of that model, write a new model from scratch, and divert that single client to use the new model. Only write the minimal features into the new model that this single client needs. Repeat for each client, growing the new model. When the old model has no clients, retire it. If the new model is indeed better, it will permit simplifications to now ripple out into all its clients. Have an LHR party.

Deprecation Refeaturezation: Like Deprecation Refactor, but behavior drifts a little towards a new feature. Example: You have a dialog box with 5 edit fields. To replace one of them with a combo-box, you start by painting a combo-box next to the edit field. Leave the former online; get the latter to work redundantly. Only when the latter has taken over all abilities of the former do you remove clients of the former, and then retire the former edit field & move the combo box over into its place.

I have found these 3.5 principles guide all design improvements that I encounter. But this has not stopped one Joshua Kerievsky from writing a book /Refactoring to Patterns/ that studies the incremental path between mud some clean pattern. Each to his own, huh?

Happy Boreal-Winter Solstice, everyone!

--
Philip