Mistaeks I Hav Made

"Good judgement is the result of experience ... Experience is the result of bad judgement." — Fred Brooks

Main | All articles about Test-Driven Design

State vs Interaction Based Testing

Martin Fowler has recently written an article comparing what he calls "state-based" and "interaction-based" unit testing. The article doesn't really cover the subject in much depth but one statement in particular surprised me: "interaction-based tests are ... more coupled to the implementation of a method." I think that Martin is spot on when he says "one of the hardest things for people to understand in OO design is the 'Tell Don't Ask' principle," but that principle has a big influence on how you write tests and is exactly what makes interaction-based testing necessary. In an object-oriented design, an object's *state* is an implementation detail that should be properly encapsulated and its interactions with its environment should be its only visible behaviour. If you follow the "Tell, Don't Ask" style, objects have very little visible state to assert about.

When writing a program, I care only about what that program does, not the internal state that the program uses to control what it does. The only visible behaviour that a program has is its interactions with external entities, such as I/O devices or remote processes. When a program is divided into modules, likewise the internal state of a module is unimportant; it is only how a module interacts with other modules that matters.

In a procedural program, modules interact by reading and writing state that is stored in shared data structures, and so changes to that state should be tested. In an object oriented program, on the other hand, a program is modularised as collaborating objects that perform actions by sending messages to other objects. To effect changes in the program's environment, application objects send messages to objects that represent entities in the environment. The behaviour of a program, and that of the objects within it, is defined solely in terms of message sending, and those messaging interactions are what should be tested.

In software that has a pure object-oriented design, in which logic operating upon state is defined only in the objects that hold that state and objects interact in a "Tell, Don't Ask" style, objects expose next to no visible state that can be used for state based testing. Making assertions about state and state changes therefore requires objects to provide access to their internals that is not neccessary for the normal execution of the software, and ties the tests to implementation details that should be properly encapsulated.

Blaming "brittleness" of tests upon interaction-based testing is a red herring. Both interaction-based tests and state-based tests become brittle if they make assertions upon implementation details and overly constrain the interfaces between modules. Whether you prefer a procedural style in which you test the changes of visible state in data structures, or an object-oriented style in which you test the coordination of actions between objects, you need to carefully choose what your tests specify to keep them from being brittle. In state based tests you have to be careful that you don't test for inconsequential state changes or make too tight assertions about state values. In interaction tests you have to be careful not to test for inconsequential interactions and make too tight assertions about parameter values. This is why jMock provides so much flexibility in the way a test can define constraints upon method signature, parameter values, invocation ordering, expected invocation counts, etc.

I think that the most important benefit of interaction based testing is that it helps *reduce* the amount of mutable state in a program. Mutable state makes a program harder to understand and maintain because the behaviour of a piece of code cannnot be easily predicted merely by reading the text but depends on the sequence of events that put it and its environment into signigicant states. By concentrating on the interactions between objects instead of state changes, interaction-based testing guides the design towards objects that transform data as they pass it around rather than store data and perform logic on the state of other objects.

I know this makes me sound like a functional programming zealot instead of an object-oriented programming zealot, so I've

dug up this quote by Alan Kay to reestablish my OO purist credentials:

> *"Doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state oriented metaphors from programming."* — Alan Kay, Early History of Smaltalk.

I have come to think of object oriented programming as an inversion of functional programming. In a lazy functional language data is pulled through functions that transform the data and combine it into a single result. In an object oriented program, data is pushed out in messages to objects that transform the data and push it out to other objects for further processing.

Update: example and code.

Posted on July 14, 2004 [Permalink]

Comments

This "interaction-based tests are ... more coupled to the implementation of a method." rings true to me so I'm assuming that there's misunderstanding somewhere here. Perhaps an example would be useful?

Posted by: Jason Yip at August 4, 2004 12:54 AM

I've written an example at http://nat.truemesh.com/archives/000356.html

Posted by: Nat Pryce at August 6, 2004 05:12 PM