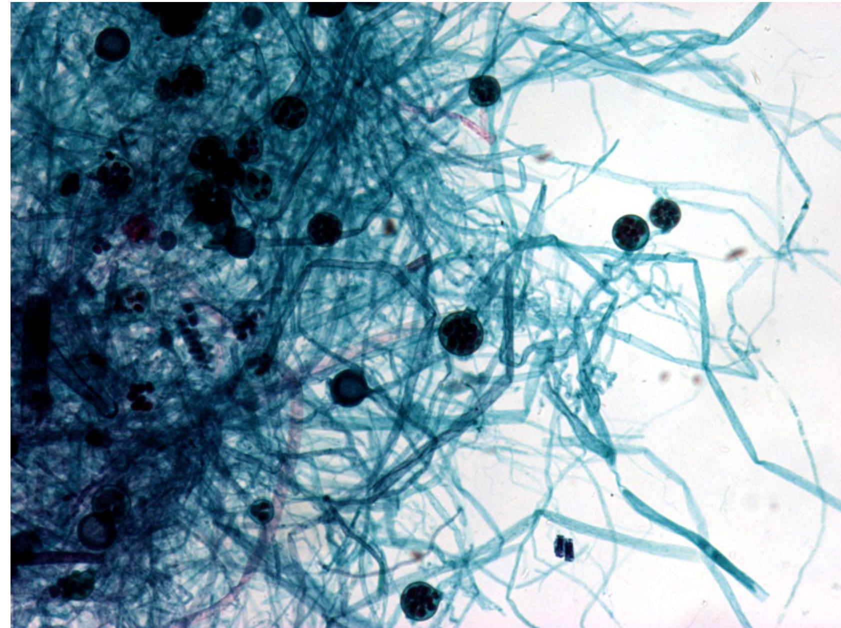
The Return of the Son of ‘Working Effectively with Legacy Code’

Michael Feathers
mfeathers @ objectmentor.com

Topics

- Global Mud
- Componentization
- Scopes of Replacement
- Explicitness of Seams
- Type Cruft
- ‘Tell, Don’t Ask’ and Testable Design
- FP and Legacy Code
- Resurrecting Code
- Testability and Language Design (TUC vs. TUF)
- Recoverability and Dynamic Languages
- Salvage-ability
- The Joy of Legacy Code



© 2006-2007 Object Mentor Incorporated. All rights reserved.

Global Mud

- Once a large system gets too many global variables, it is hard to get rid of them
- The points of use for singletons are too scattered

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Componentization

- Repository Hubs
- Factory Hubs

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Scopes of Replacement

- In any large existing system you have to make pragmatic decisions about where you will break dependencies:
  - System
  - Component
  - Class
  - Method
- Heuristic:
  - Wide for coverage, Close for progress

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Seams

A Seam is a place where you can alter behavior in your program without editing it in that place.

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Seams

- Seeing the seams

```
double perimeter(Point *polygon, int size)
{
    double result = 0;
    for (int n = 0; n < size; n++) {
        Point next = polygon [(n + 1) % size];
        result += distance (polygon [n], next);
    }
    return result;
}
```

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Explicit Seams

- Favor explicit factoring for testing
- You may not be able to avoid hacks when first getting a system under test, but you are better off when you eventually refactor to make your test seams explicit

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Synergy Between Testability & Good Design

- Excessive setup indicates excessive coupling
- Slow tests indicate insufficient granularity or coupling to I/O
- The urge to test private methods indicates granularity issues
- Why
  - Tests are a way of understanding code in a documentary fashion.
  - Understandability is the essence of good design.

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Type Cruft

- A system is only as testable as its linkage with its base types
- Pervasive problem in C++, not quite so much in other languages. Everyone wants to redefine the base types.
- Valuable system asset:
  - Separation of “plain code” from frameworks and libraries.
  - Hard to achieve

© 2006-2007 Object Mentor Incorporated. All rights reserved.

‘Tell, Don’t Ask’ and Testable Design

- ‘Tell, Don’t Ask’ minimizes coupling
- It is often far easier to mock outward interfaces than inward interfaces

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Functional Programming and Testability

- There is an argument that you really don't need unit testing in FP
  - Pure code has no IO to mock
- Mocking can be useful for replacing computationally intensive bits or providing access to a place where the effect of some code can be better sensed.
- Polymorphic calls are perfect for system recovery
  - The functional alternative is parameterization

```
pageWith :: (ListBoxModel -> ListBoxModel) -> (ListBoxModel -> ListBoxModel)
                -> ListBoxModel -> ListBoxModel

pageWith step select m@(Model _ w) = select $ (iterate step (select m)) !! windowSize w
```

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Resurrecting Code

- Refactoring tools help
- Wide disparity across the languages
  - C#, Java - easy
  - C++ - many issues
  - C – easier than C++
  - Niche static languages – insufficient tool support
- Extract Method and Extract Interface are key

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Testability and Language Design

- Historically, language designers have not thought about the recovery case:
  - Programmers will make mistakes.
  - Entropy happens ☺
  - Recovery is an important language design consideration
- What is needed:
  - Language level support for dependency injection
  - Special access for tests (even intra-method)
  - Awareness of TUFs and TUCs

© 2006-2007 Object Mentor Incorporated. All rights reserved.

The Cardinal Rule of Testability

“Never Hide a TUF within a TUC”

- TUF = Test Unfriendly Feature
  - File IO, database access, long computation, message sink to external lib, etc
- TUC = Test Unfriendly Construct
  - Static method, non-virtual function, constructor, static initializer blocks, new expressions, singletons, special generics cases

Recoverability and Dynamic Languages

- Will we have less of a problem with dynamically typed languages?
- Explicitness
- The “No Lie” Principle – “Code should never lie to you”
- Ways that code can lie
    - People can dynamically replace code in the source
    - Addition isn’t a problem
    - System behavior should be “what I see in the code plus something else” never “what I see in the source minus something”
    - Weaving and aspects
    - Impact on the use of inheritance
- The Fallacy of Restricted Languages

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Salvage-ability of Systems

- How far can we go?
- The organic growth metaphor
  - Architecture is more fixed than we expect
  - Business logic is often “glued to the edges”
- Selective rewrite of logic is often easier than replacing architecture
- Technologies do make a difference (type cruft, build issues)
- The challenge is in making work within existing systems faster and more deterministic

© 2006-2007 Object Mentor Incorporated. All rights reserved.

Reframing Legacy Code

- What should our stance be?

© 2006-2007 Object Mentor Incorporated. All rights reserved.