CS1530, Lecture 13: Working With Legacy Code

Bill Laboon

What Is Legacy Code?

- No official definition, but code that is already in use.
- Kind of like a heap of sand - one grain is not a heap, is two? three? four? etc.

You Will Work With Legacy Code

- Software development techniques and tools move quickly!
- Projects sometimes last for decades
- Many common tools and techniques did not exist 5-10 years ago
- You seldom have the opportunity to start a “greenfield” project

Implications (*NOT* Rules)

- User interfaces are unfriendly and error prone
- Little to no documentation
- Written without benefit of modern software practices
- Written in rarely-used or obsolete language
- Programs not organized and difficult to understand
- Lots of inefficiencies, inconsistencies, redundancies, etc.
- Expensive and difficult to modify

Our Definition

- Defined by Michael Feathers as "code without tests" (or sufficient test coverage)
- Why? Sufficient test coverage protects you from product-breaking changes
- Other issues can be dealt with more easily if you have a testing "safety net"

- The Social Network
- Apollo 13
- Hackers
- TRON
- WarGames
- Revolution OS
- Sneakers
- Mr. Robot

None of these shows deal with a programmer spending five days updating sales tax calculations in 35-year-old inventory software.

Dealing With Legacy Code Is Not Glamorous

- It's often hard, it’s often thankless…
- … but it is important.
- I guarantee you that you will deal with it at some point in your career.

Why Do It?

- Changes necessary (e.g. sales tax changes)
- No longer meeting performance/security/other requirements
- Add functionality
- Fix defect

Reasons Not On That List

- Code is ugly
- Code written in old language
- Programmers bored

Let Sleeping Dogs Lie

- If it ain't broke, don't fix it.
- Avoid pre-emptive work on legacy code.
- Re-writing or modifying can often *introduce* more failures/defects than it fixes.

The Problem With Re-Writing

- Remember that most approaches work with small programs! Maybe the best case is a rewrite in those circumstances.
- For larger programs, rewrites often re-make the same mistakes. The reason that method is 500 lines long is that it probably fixed lots of edge cases.

Three Questions Before Modifying Legacy Code

- What *exactly* needs to be changed?
- How do you check if the changes are correct?
- How do you ensure that there are no regression failures from this change?

Answers To First Two Are Simple

- 1 - Changes should be spelled out in detail and verified with stakeholders
- 2 - A proper test plan (including unit tests and other tests) should be considered before work starts

Answer To Question 3 - Pinning Tests

- Pinning tests (also called "characterization tests" or "test covering") check the EXISTING behavior of a program
- Not *expected* behavior, *existing*. Very different!
- Adds an invariant to the system (should be the same results before/after)

Why Not Check Expected Behavior?

- Users may depend on these “glitches"
- You may not understand ALL of the requirements for a large project, especially if you are making a small change to a large program
- Understand what you're changing, even if it's a fix

General Strategy When Dealing With Legacy Code

- Get to the point where you can make small changes - one bit at a time
- Look for seams (places where you can modify functionality without modifying code)
- Add pinning tests as you go
- Soon you find yourself with a little, but ever-growing, test suite
- Add traditional unit tests to it, can transition project to a more modern software development paradigm

Example of No Seam

```
ArrayList<Cat> _cs = new ArrayList<Cat>();

public int getNumFluffyCatsAnimals() {
    int toReturn = 0;
    Attribute f = new Attribute(“fluffy”);
    for cat in _cs {
        if (a.hasAttribute(f)) {
            toReturn++;
        }
    }
}
```

Example of a Seam

```
public int getNumAnimals(List<Animal> as,
                         Attribute attr) {
    int toReturn = 0;
    for animal in as {
        if (a.hasAttribute(attr)) {
            toReturn++;
        }
    }
}
```

Tactics

1. Identify change points
2. Find an inflection point
3. Cover the inflection point with pinning tests
    1. *Break external dependencies to it*
    2. *Break internal dependencies to it*
    3. *Write tests*
4. Make Changes
5. Refactor Code

Change Points

- What part of the code needs to be modified in order to add the feature or fix the defect
- This could be something localized, it could be a set of classes, it could be all over the codebase!

Inflection Points

- A narrow interface to a set of classes
- Any change to any place "behind" the interface can be "discovered" by the interface, or is inconsequential

Example Inflection Point - WolfPackSimulator

- Rest of code interacts with class WolfPack.
- WolfPack composed of objects of class Wolf, subclasses AlphaWolf, WereWolf, and CookieWolf, along with WolfPackHierarchy. No other code references these other classes.
  - Note I am not a wolfologist, these may not be actual wolf titles
- WolfPack can be an inflection point for the whole set of Wolf-related classes (WolfPack, AlphaWolf, WereWolf, CookieWolf, WolfPackHierarchy)
- Any relevant changes can be detected via WolfPack

Cover Inflection Point With Pinning Tests

- Determine current behavior at the inflection point level
- Write automated tests for this behavior checking the _existing_ behavior
- These should be marked in your test suite as pinning tests!

Break External Dependencies

```
class Bird {
   // External dependency - class Nest

   private _nest = new Nest();

   public void goHome()  {

      flyTo(_nest);

   }

}
```

Break External Dependencies

```
class Bird {

  private Nest _nest = null;

  // Better - external dependency can be
  // doubled (fake test object)

  public Bird(Nest nest) {
    _nest = nest;
  }

  public void goHome() {
    flyTo(_nest);
  }
}
```

Break External Dependencies

```
class Bird {

  // Even better if you can -dependency
  // injection right into method

  public void goHome(Nest n) {
    flyTo(n);
  }
}
```

Break Internal Dependencies

- Provide seams (see sprout and wrap methods)
- Break methods up so that each method does one thing
- Provide good testing surface

Write Tests

- Add additional tests to check the functionality you have just added
- Note: these are not pinning tests - you are looking for expected behavior, not existing!

Refactor

- Start making the code better
  - Better algorithms
  - Better documentation
  - Better design
- Only under areas of code covered by tests at inflection point!
- That is your “safety net”

Sprout Method

- A new method "sprouts" out of the old one, like a plant coming out of the ground
- Take functionality done as part of a larger method, make it independent
- Provides a new seam and localizes testing

Sprout Method

```
public void reset() {
    this.textBox = “”;
    this.numCats = 0;
    this.numDogs = 0;
    this.numBirds = 0;
    if (_databaseConnection != null) {
        _databaseConnection.terminate();
    }
}
```

Sprout Method

```
public void resetVals() {
    this.textBox = “”;
    this.numCats = 0;
    this.numDogs = 0;
    this.numBirds = 0;
}
public void terminateDbConnection() {
    if (_databaseConnection != null) {
        _databaseConnection.terminate();
    }
}
public void reset() {
    resetVals();
    terminateDb();
}
```

Wrap Method

- Rename existing method, then have code call your new method, which delegates old work to the old method

Wrap Method

```
public void reset() {
    if (this.numMonkeys < 0) {
        this.monkeyBox = "MONKEY ERROR";
    }
    this.monkeyBox = "";
    this.textBox = "";
    this.numCats = 0;
    this.numDogs = 0;
    if (_databaseConnection != null) {
        _databaseConnection.terminate();
    }
}
```

Wrap Method

```
public void legacyReset() {
  this.textBox = “”;
   this.numCats = 0;
   this.numDogs = 0;
   if (_databaseConnection != null) {
       databaseConnection.terminate();
   }
}
public void reset() {
   if (this.numMonkeys < 0) {
      this.monkeyBox = “MONKEY ERROR”;
   }
   this.monkeyBox = “”;
   legacyReset();
}
```

Applying To Classes

- You can sprout or wrap classes along with methods!
  - Sprout - split small sections of class off into their own classes
  - Wrap - Replace class with new class which delegates old functionality to old class, adds new functionality into it

Local Minima

- Note that by using methods such as sprouting and wrapping, you may actually put your code in a worse place than before!
- However, the goal is to use these methods to eventually make it better, even at the expense of short-term losses
- Adding the ability to test more easily will help increase the quality of your codebase

No TUFs Inside TUCs

- TUF = Test-Unfriendly Feature (i.e., a feature that is hard to test)
- TUC = Test-Unfriendly Construct (i.e., a code construct which makes it hard to test the code inside)

Example TUFs

- Long-running methods
- Network access (incl. third-party API access)
- Reading/writing to database
- Reading/writing to file
- Static variable modification

Example TUCs

- Constructors / destructors
- Final methods / classes
- Static methods / initialization blocks
- Private methods

Move TUFs Out of TUCs

- It is possible to overcome problems with TUFs or TUCs, but it becomes more-than-linearly difficult when dealing with both at the same time
- Also makes tests more verbose, difficult to modify, difficult to understand
- Use techniques mentioned above (sprout, wrap, dependency injection) to move TUFs out of TUCs

Further Reading

- Essay, "Working Effectively with Legacy Code" by Michael Feathers. http://www.netobjectives.com/system/files/WorkingEffectivelyWithLegacyCode.pdf
- He also wrote a book of the same name, which goes into much more detail
- Testable Java, https://drive.google.com/file/d/0B8ZX1RoWHuiJWHI1Q3JOc21qS00/view?usp=drive_web