Dealing with Bugs

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 5828 — Lecture 27 — 04/21/2009

© University of Colorado, 2009
Goals

- Review material from Chapter 11 of Pilone & Miles
  - Dealing with Bugs
    - Talking with your Customer
    - Scouting out the bug
      - What exactly is not working?
    - Making an estimate
    - Spike Testing
  - Fix the Bugs
In the Chapter example, the situation for our team did NOT look good

- Working to demo integration of Mercury Meal’s code
  - Demo is not working, the system hangs when calling MM code
  - The MM code is a mess!
- The team has three user stories that depend on this new code
- To make matters worse, the CEO and CFO of Orion’s Orbits are expecting to see the demo ASAP
Our process relies on communication, so the first thing we do is talk to the CEO.

He agrees to push back the demo but wants to know how far?

He wants an estimate on how long it will take to fix Mercury Meal’s code.

We need to be confident in the estimate we give him, but we are in a situation where planning poker will not work.

Why not?
Next Steps?

- There are plenty of things we could be doing
  - Get a coverage report on the MM code... how much of this code has been tested?
  - Get a line count on the MM code and use this to make an estimate.
  - Do a security audit on the MM code
  - Use a UML tool to reverse-engineer the code and produce a class diagram
  - etc.
We’re not yet in a situation to do these things; indeed they provide little benefit right now

- We need an estimate of how long we think it will take to track down all the bugs in the MM code that affect our user stories
  - Coverage report? We don’t have tests for this system!
  - Security Audit? Won’t help us with an estimate…
  - UML tool? Could be helpful but it could provide a lot of unnecessary detail and get us off track
  - Line count? We don’t yet know how much of this code we need and whether its missing code that we do need
So, what should we do?

- Fall back on our process and get this code ready to give us the information we need
  - Create an issue for the MM code in your bug tracker
  - Organize the source code into standard directories
  - Write a build script
- Place the code and build script under version control
- Integrate the code into your continuous build system
- Write tests simulating how you need to use the software
  - File bugs as you find them!
Next? Fix code?

- We could immediately start working on the code
  - Here’s an example of what MM gave us
package com.mercurymeals;

//Follows the Singleton design pattern
public class MercuryMeals {

    private Order cO;
    private String qk = "select * from order-table where keywords like %1;";
    private static MercuryMeals instance;

    public MercuryMeals() {
        
    }

    public static MercuryMeals getInstance() {
        instance = new MercuryMeals();
        return instance;
    }

    // TODO Really should document this at some point... TBD
    public Order createOrder() {
        return new Order();
    }

    public MealOption getMealOption(String option) throws MercuryMealsConnectionException {
        if (MM.establish().isAnyOptionsForKey(option)) {
            return MM.establish().getMealOption(option)[0];
        }
        return null;
    }

    public boolean submitOrder(Order cO) {
        try {
            MM mm = MM.establish();
            mm.su(this.cO);
        } catch (Exception e) {
            // write out an error message
            return false;
        }

    }

    public Order[] getOrdersThatMatchKeyword(String qk) throws MercuryMealsConnectionException {
        Order[] o;
        try {
            o = MM.establish().find(qk, qk);
        } catch (Exception e) {
            return null;
        }
        return o;
    }
}
Lizard Brain Response?

Let’s clean this up!
- As shown in next slide
package com.mercurymeals;

public class MercuryMeals {

    private static MercuryMeals instance;

    public MercuryMeals meallythang;
    private Order cO;
    private String qk = "select * from order-table where keywords like %1;";

    public static MercuryMeals getInstance() {
        instance = new MercuryMeals();
        return instance;
    }

    public Order createOrder() {
        return new Order();
    }

    public MealOption getMealOption(String option) throws MercuryMealsConnectionException {
        if (MM.establish().isAnyOptionsForKey(option)) {
            return MM.establish().getMealOption(option)[0];
        }
        return null;
    }

    public boolean submitOrder(Order cO) { try {
        MM mm = MM.establish();
        mm.submit(this.cO);
    } catch (Exception e) {
        // write out an error message
    }
    return false;
    }

    public Order[] getOrdersThatMatchKeyword(String qk) throws MercuryMealsConnectionException {
        Order[] o;
        try {
            o = MM.establish().find(qk, qk);
        } catch (Exception e) {
            return null;
        }
        return o;
    }
}
This code still sucks BUT

- certain problems are now obvious
  - horrible comments (so bad I just deleted them)
  - horrible variable names and lots of potential shadowing
- Use of a package in same name space that appears as if by “magic” in the code (“MM”)
  - Confusingly they define a variable named “mm” that acts as a pointer to this package in one method but use the package name everywhere else
Fixing all the problems you see would represent a waste of time at this moment

- Remember, our focus right now is on getting an estimate

- We don’t want to fix code, we want to fix functionality

- We have three user stories that are not working because of the MM code

- Fixing these user stories is our ultimate goal

- Remember: Simplicity in all things; we’ll fix what we need to make progress on our current tasks; the future will take care of itself
Everything revolves around end-user functionality

We write and fix code to satisfy user stories

- We only fix what is broken
  - We know what’s broken because we have tests that fail

Tests are the ultimate safety net

- They let us know when something is broken and when its fixed again
- If there is no test for a user story, that user story is broken

Functional code trumps beautiful code!
Next Step: Write a Test

- Now we write tests related to our three user stories to see what’s broken
  - See next slide
package com.orionsorbits.solutions;

import com.mercurymeals.*;
import com.orionsorbits.OrderNotAcceptedException;

public class OrionsOrbitsSolution {

    public static void main(String[] args) throws Exception {
        OrionsOrbitsSolution oo = new OrionsOrbitsSolution();
        System.out.println("Adding order...");
        oo.orderMeal(new String[]{"Fish and Chips"}, "VS01");
    }

    public void orderMeal(String[] options, String flightNo) throws Exception {
        MercuryMeals mercuryMeals = MercuryMeals.getInstance();
        Order order = mercuryMeals.createOrder();

        for (int x = 0; x < options.length; x++) {
            MealOption mealOption = mercuryMeals.getMealOption(options[x]);
            if (mealOption != null) {
                order.addMealOption(mealOption);
            } else {
                throw new MealOptionNotFoundException(mealOption);
            }
        }

        order.addKeyword(flightNo);

        if (!mercuryMeals.submitOrder(order)) {
            throw new OrderNotAcceptedException(order);
        }
    }
}
What Next? Spike Test

Now that you’ve written tests and know what’s failing, it’s time to conduct a spike test to create the estimate that we need to provide to the CEO.

- A spike test is a week outside the normal iteration plan in which the focus is on fixing the failing test cases.
- Pick a random sampling of the test cases.
  - But try to avoid the easiest and the hardest test cases.
- After five days, calculate your bug fix rate.
  - Bugs Fixed / Number of Days = Daily bug fix rate.
- Now calculate your estimate.
  - Bug Fix Rate x (Failing Test Cases) = Estimate.
Example

- In the example, the team had 13 failing test cases
  - During a 5 day spike test they fixed 4 of the bugs
  - Bug fix rate: \( \frac{4}{5} = 0.8 \) bugs per day

- They now have 9 failing test cases
- Estimate: \( 0.8 \) bugs per day \( \times \) 9 bugs = 7 days
Accuracy?

Now, temper the estimate with some qualitative data

- You have an estimate but your team might feel that it's not quite right
  - One of the developers might feel like they “grok” the MM code now and so feels like the remaining bugs will fall much more quickly
- Take a survey of the developer’s confidence and factor that into your final estimate
  - \((\text{bug fix rate} \times \text{bugs remaining}) \times \frac{1}{\text{average confidence}}\)
Example

- Three developers are surveyed about their confidence that the remaining bugs will take 7 days to fix
  - One developer says they are 80% confident
  - The other two say they are 60% and 70% confident
- Take the average for the team’s confidence: 70%
- Revise estimate
  - \[(0.8 \times 9) \times \frac{1}{.7} = 10.28 \text{ days}\]
Take this to Customer

- Give the new estimate to the customer
  - And work with them to update the iteration plan
    - Some stories may need to be bumped to the next iteration
- Then… GET TO WORK!
  - At the end of the process, you will have fixed all the bugs needed to fix the three user stories and allow the demo to proceed
  - The problem: there will still be bugs hiding in the MM code
    - Deal with them if and when they affect future user stories
Wrapping Up

- Addressing bugs requires a process
  - When fixing bugs in code that you didn’t write yourself
    - get the code under control before fixing it
      - build scripts, version control, reorganization, but NO FIXING
    - write tests
    - perform a spike test
    - provide estimate to customer and update plan
    - get to work and keep track of all issues in bug tracker
  - Don’t fix bugs just to fix them; let user stories guide you
Coming Up

- Lecture 28: Software Abstractions
  - Overview of the Software Abstractions textbook
- Lecture 29: SE Wrap-Up
  - Chapter 12 of Head First Software Development
  - Review of Class
- Lecture 30: Project Demos