Bugs in the Software Engineering Process and Abuse Cases

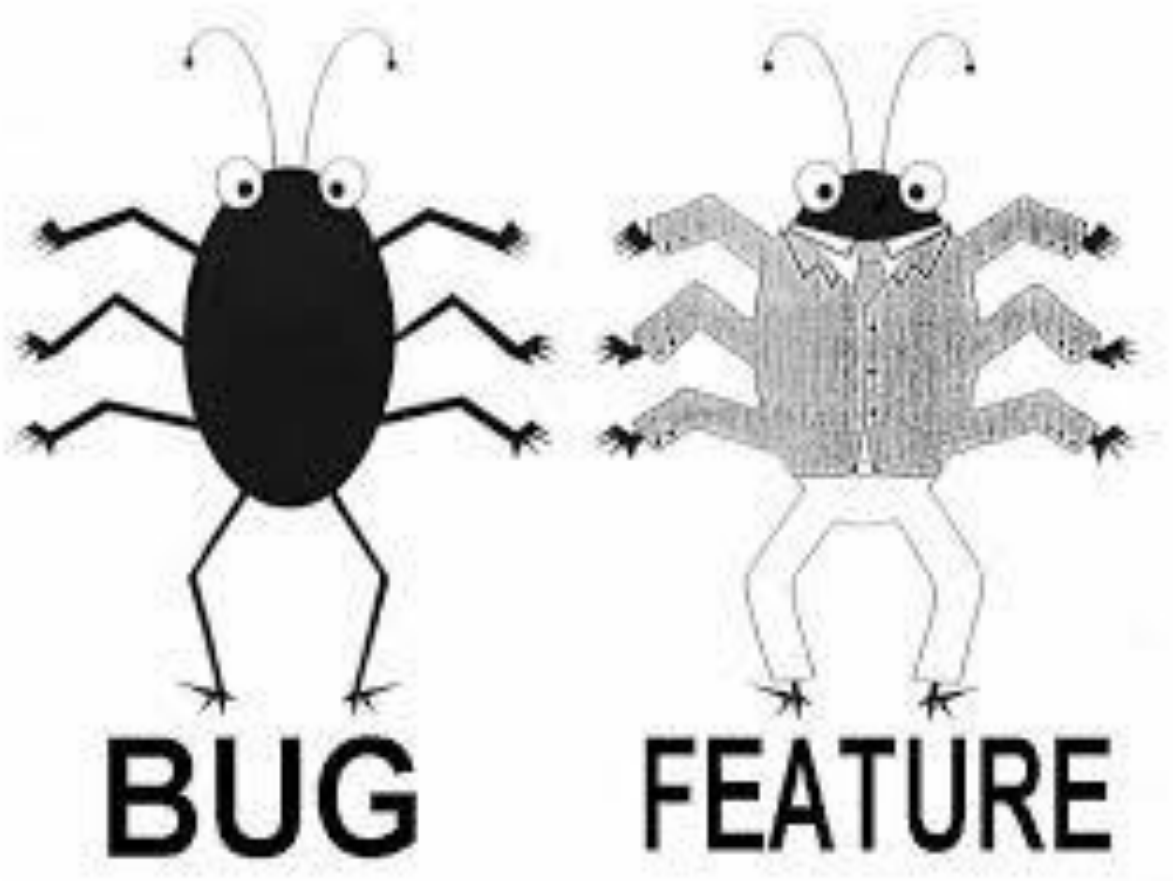
Content from Chapter 11 of "Head First Software Development", Pilone et al.

Miami University Software Technology & Analysis Group (MUSTANG)
Computer Science & Software Engineering
Miami University, Oxford, Ohio, USA

PROJECT'S DUE DATES ?

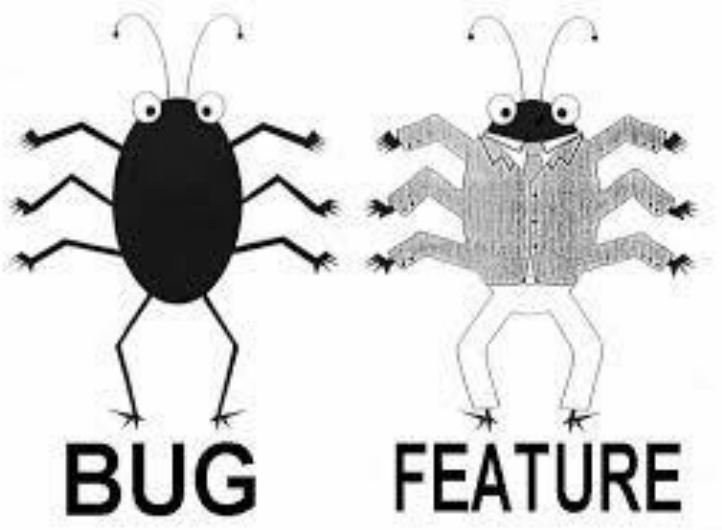
- ~~April 2nd → 5th week deliverables~~
- ~~April 6th → Iteration 1 presentation → weeks 3 + 4 → 90 mints~~
- ~~April 16th → 6th week deliverables~~
- ~~April 20th → Iteration 2 presentation → weeks 5 + 6 → 90 mints~~
- ~~April 23rd → 7th week deliverables~~
- April 30th → 8th week deliverables
- May 4th → Iteration 3 (final) presentation → weeks 7 + 8 → 90 mints

Bugs!



What Do We Do With Them?

- Eventually, your testers are going to find a bug.
- In fact, they'll probably find a lot of them. So what happens then?
  - Do you just fix the bug, and not worry about it?
  - Do you write it down?
  - What really happens to bugs?



LIFE AND DEATH OF A BUG

1 A tester FINDS A BUG
A bug doesn't have to be something that's clearly failing. It could be ambiguity in the documentation, a missing feature, or a break from the style guide for a web site.

2 The tester FILES A BUG REPORT
This is one of the most critical steps: ***you have to track bugs!*** It doesn't matter who reports a bug, but level of detail is crucial. Always record what you were trying to do, and if possible, the steps to re-create the bug, any error messages, what you did immediately before the bug occurred, and what you would have expected to happen.

Some teams work with their customer to prioritize which bugs get fixed using the bug tracker, and don't create stories or tasks for ones that aren't going to get fixed in the current iteration. Other teams create the stories and tasks right away and let the customer prioritize the bugs like any other story. Either way works great as long as things are being prioritized by the customer.

3 CREATE A STORY (or task) to fix the bug
Bugs are just work that has to be done in your system—sometimes in the current iteration, sometimes in a later one. You'll need to capture them and prioritize each bug with the customer. These are tricky to estimate, though, because it's not always clear what's wrong. Some teams have a "Bug Fix" story that they just keep around, and they add tasks to it as needed.

A bug is just like an unplanned task. Once it's on the board, it's handled like any other story and task.

4 FIX THE BUG
The development team works on the bug as part of an iteration. Start by writing a test that exposes the bug (the test should fail before you change any code). Once the team's fixed the bug (and the test lets you know when that is), they should mark it as "Fixed" in the bug tracker. But don't mark it as tested, closed, or verified—that's for the original reporter to take care of. This also helps you get a list of what's ready for turnover to the test team.

5 CHECK THE FIX and verify it works
The tester (or original reporter) verifies the new build and makes sure they're happy with the resolution. *Note* the bug can be marked as closed (or verified).

6 UPDATE the bug report
Once the tester (and original reporter) are happy with the fix, close the bug report. The updated report can be used as a script to retest. Don't delete it...you never know when you might want to refer back to it.

THEY WILL EXIST!

- No matter how hard you work at coding carefully, some bugs are going to slip through.
- Sometimes they're programming errors; sometimes they're just functional issues that no one picked up on when writing the user stories.
- Either way, a bug is an issue that you have to address.

YOUR BUG, YOUR REPUTATION!

TRACK THEM!

- Most important thing about dealing with bugs on a software project is making sure they get
  - Recorded and
  - Tracked.
- For the most part it doesn't matter which bug tracking software you use;
  - There are free ones like **Bugzilla** or
  - Commercial ones like **TestTrackPro** and **ClearQuest**.
- The main thing is to make sure the whole team knows how to use whatever piece of software you choose.

TRACK THEM AND …

- Record and communicate priorities
- Keep track of everything
  - Discussions
  - Code changes
  - Decisions
- Generate metrics (?)
  - Bug submission rate (Up or down)
  - Where they come from
  - Priority

INVOLVE THE CUSTOMER

- Whenever something changes, talk it over with your team.
- If the impact is significant, in terms of functionality or schedule, then you've got to go back to the customer.

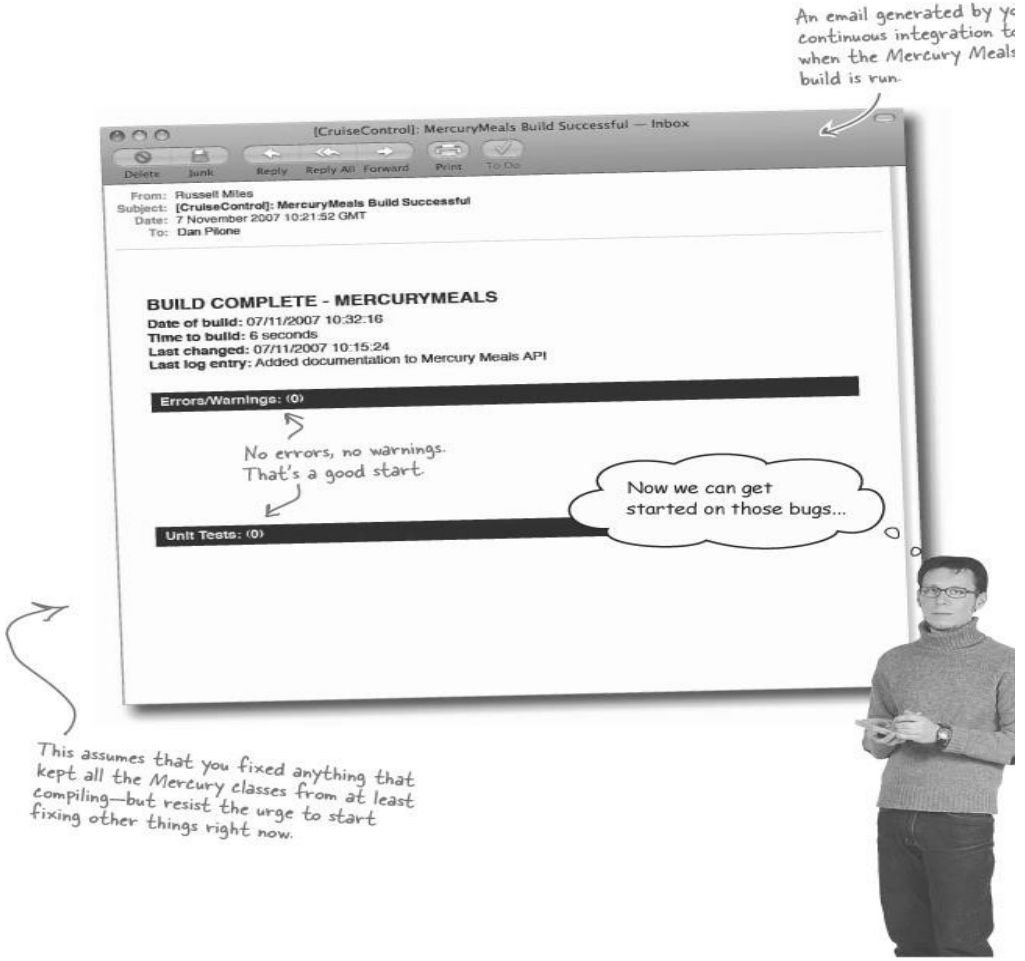
ROLE PLAY – INHERITED CODE

- Often, you inherit and integrate code/libraries (open source, purchased, etc.
  - Examples from your CSE 201 projects

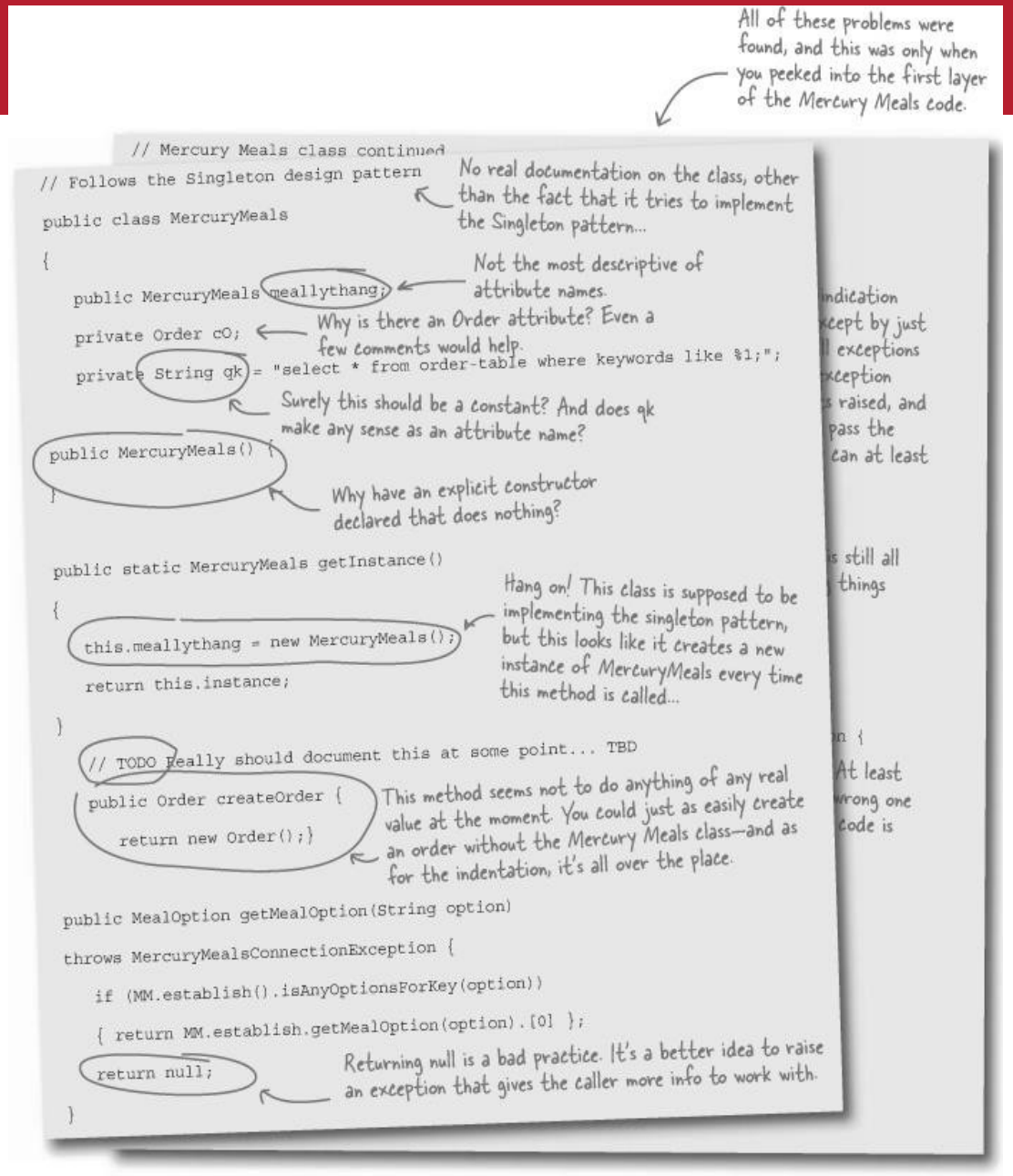
- Following textbook example, for our Orbit (Space Flight Program, we inherit “Mercury Meals” code

What Should Be Our First Priority?

- Get things buildable!
- At least you should have a little bit of control over the code...and that's your first priority.

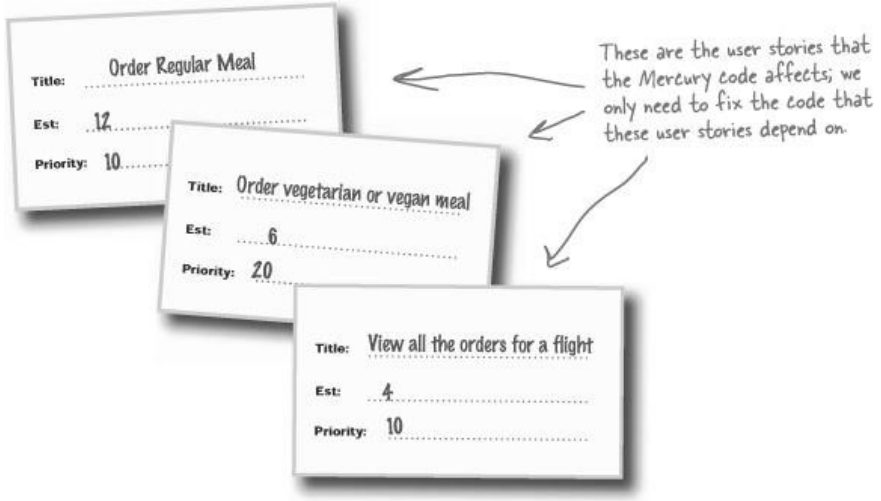


FIX THE CODE?


All of these problems were found, and this was only when you peeked into the first layer of the Mercury Meals code.
// Mercury Meals class continued
// Follows the Singleton design pattern
No real documentation on the class, other than the fact that it tries to implement the Singleton pattern...
public class MercuryMeals
{
public MercuryMeals meallythang;
Not the most descriptive of attribute names.
private Order cO;
Why is there an Order attribute? Even a few comments would help.
private String qk = "select * from order-table where keywords like %1;";
Surely this should be a constant? And does qk make any sense as an attribute name?
public MercuryMeals() {
}
Why have an explicit constructor declared that does nothing?
public static MercuryMeals getInstance()
{
this.meallythang = new MercuryMeals();
Hang on! This class is supposed to be implementing the singleton pattern, but this looks like it creates a new instance of MercuryMeals every time this method is called...
return this.instance;
}
// TODO Really should document this at some point... TBD
public Order createOrder {
return new Order();}
This method seems not to do anything of any real value at the moment. You could just as easily create an order without the Mercury Meals class—and as for the indentation, it's all over the place.
public MealOption getMealOption(String option)
throws MercuryMealsConnectionException {
if (MM.establish().isAnyOptionsForKey(option))
{ return MM.establish.getMealOption(option).[0] };
return null;
Returning null is a bad practice. It's a better idea to raise an exception that gives the caller more info to work with.
}

NO! FIX FUNCTIONALITY!

- You don't have to fix all the bugs in;
- ***You just have to fix the bugs that affect the functionality that you need .***
- Don't worry about the rest of the code—focus just on the functionality in your user stories.

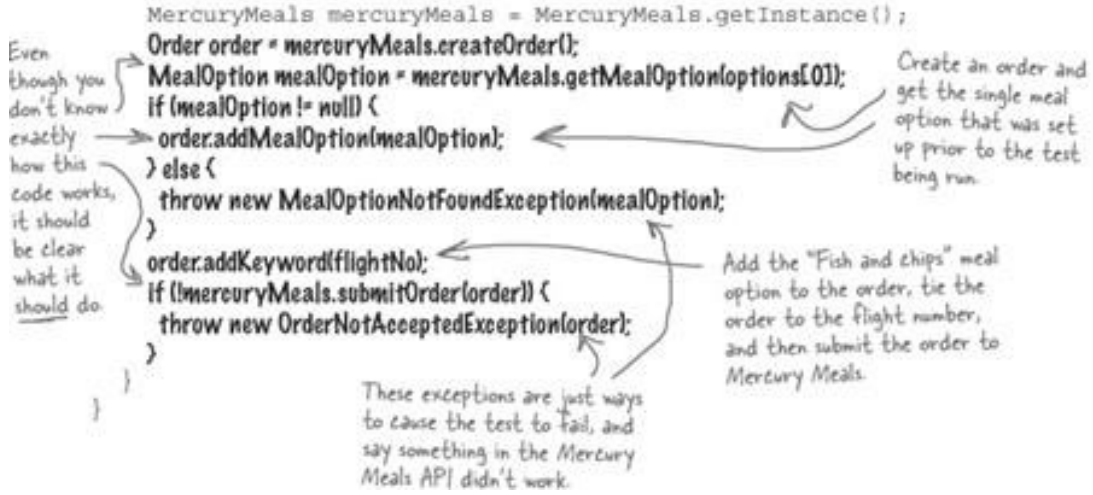


ONLY FIX WHAT IMPACTS YOUR STORIES!

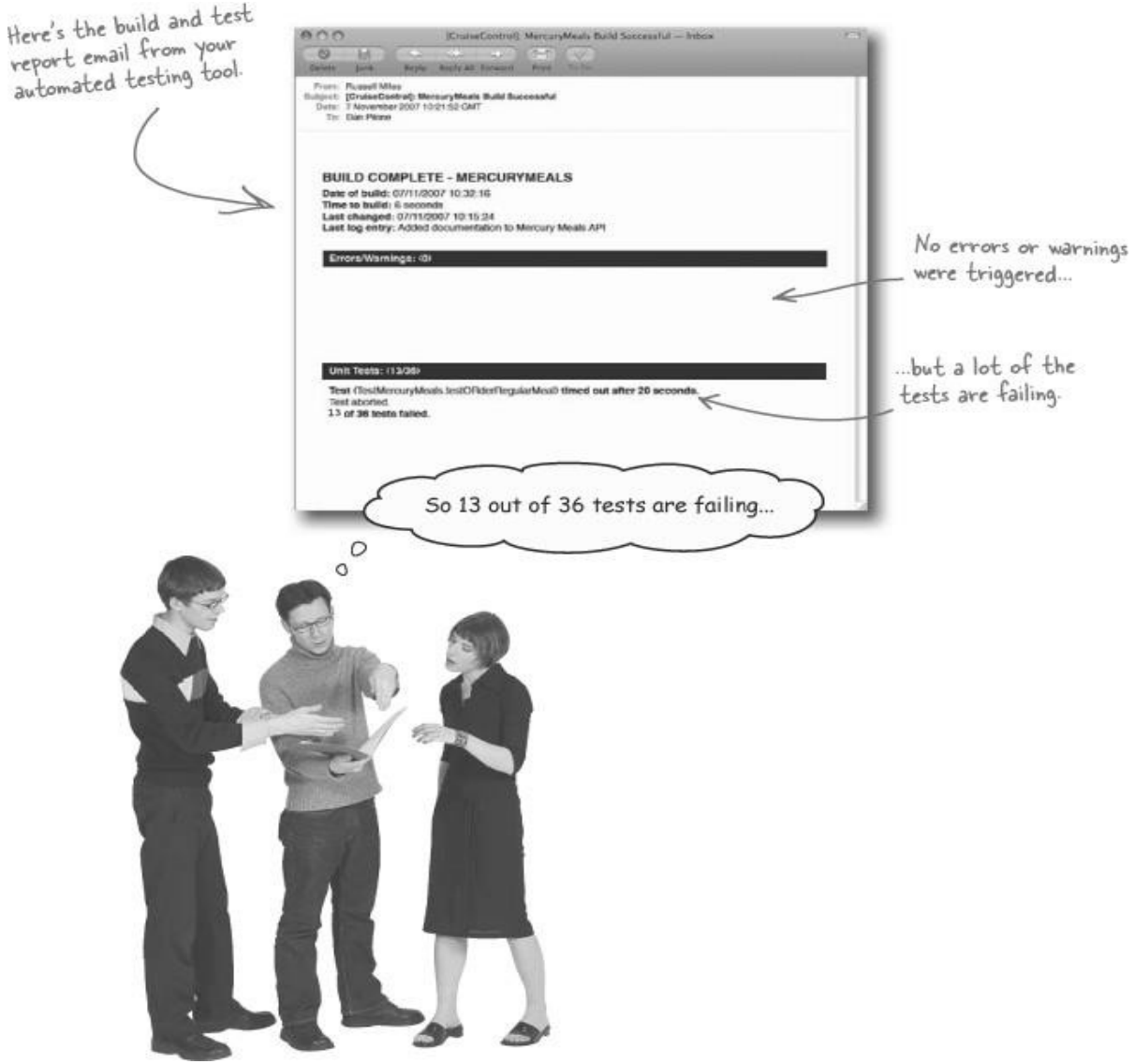
- ***Functionality is the focus. Only fix code to fix user stories.***

DETERMINE WHAT FUNCTIONALITY IS WORKING

- The first step is to find out what's actually working, and that means tests.
- Remember, if it's not testable, assume it's broken.
- Create unit tests for any new integrated parts.



Now You Know What's Not Working



How Do We Estimate Fix Time?

- A certain % of the tests you wrote are failing, but you really have no idea if a single line of code would fix most of that, or if even passing one more test could take new classes and hundreds of lines of code.
  - There's no way to know how big a problem those X test failures really represent.
- Answer: Spike Test
  - Take a little time to work on the code, see what we can get done, and then extrapolate out from that.

SPIKE TESTING

1. You're doing one burst of activity,
2. Seeing what you get done
3. Using that to estimate how much time it will take to get everything else done.

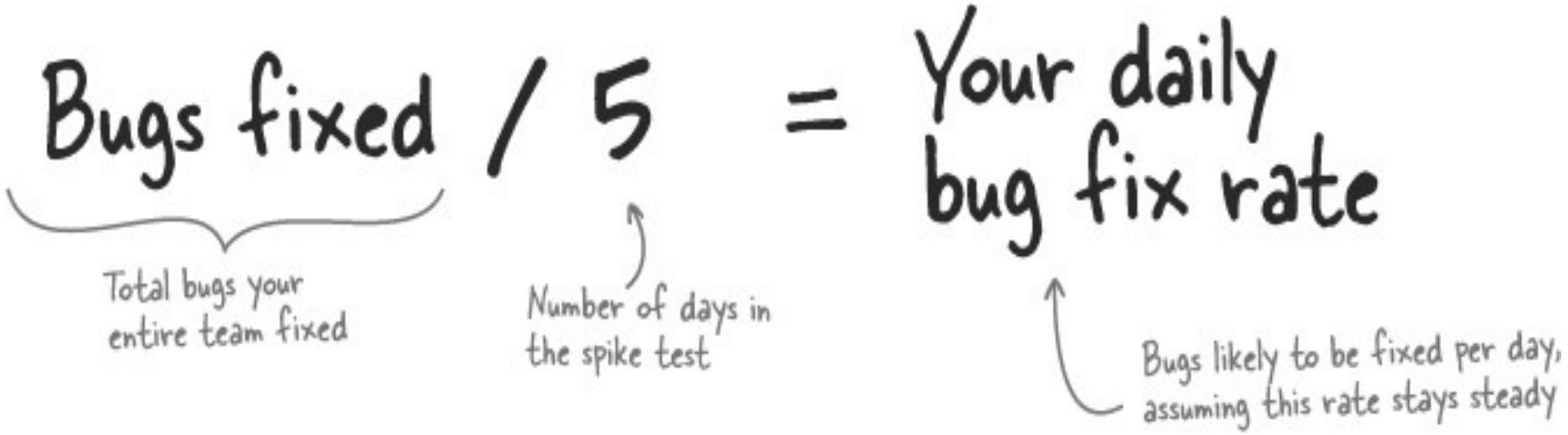
TAKE A WEEK TO CONDUCT YOUR SPIKE TEST

Pick A Random Sampling Of Failed Tests

- Take a random sample of the tests that are failing, and try to fix just those tests.
- But be sure it's random—don't pick just the easy tests to fix, or the really hard ones.
- You want to get a real idea of the work to get things going again.

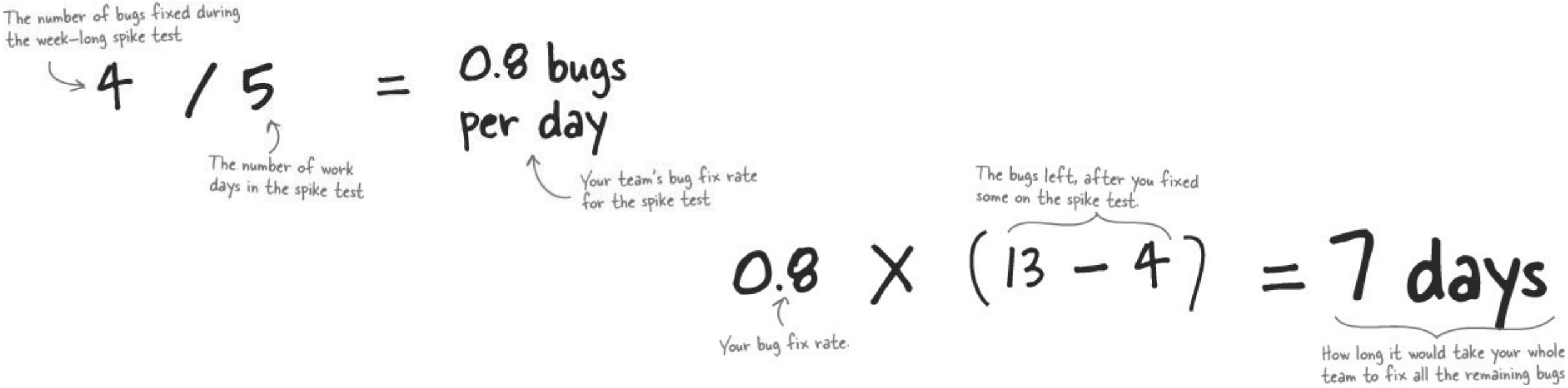
At The End, Calculate Your Bug Fix Rate

- Look at how fast you and your team are knocking off bugs, and come up with a more confident estimate for how long you think it will take to fix all the bugs, based on your current fix rate



What Do Spike Tests Tell Us?

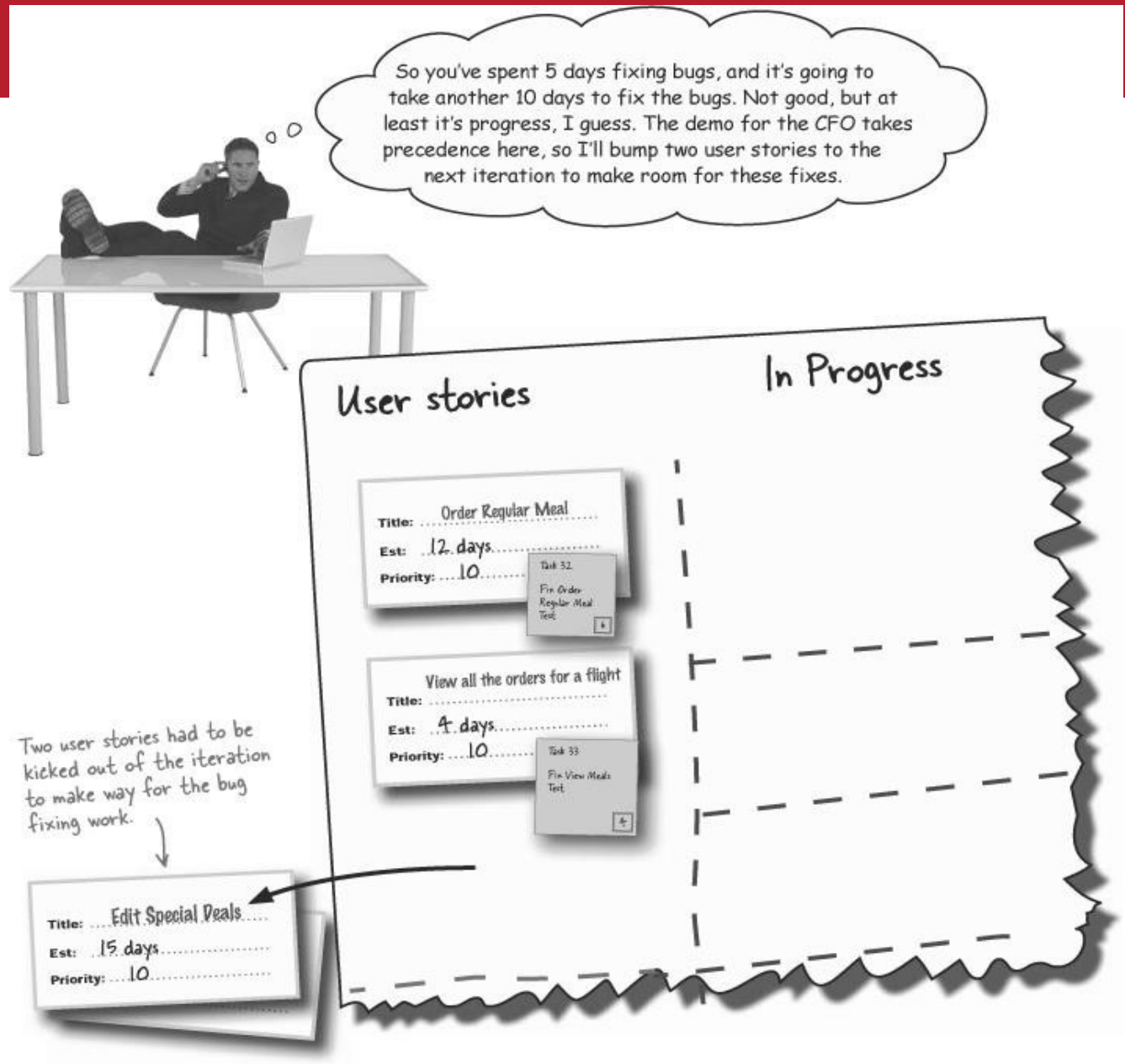
- Your tests gave you an idea as to how much of your code was failing.
- With the results of your spike test, you should have an idea about how long it will take to fix the remaining bugs.



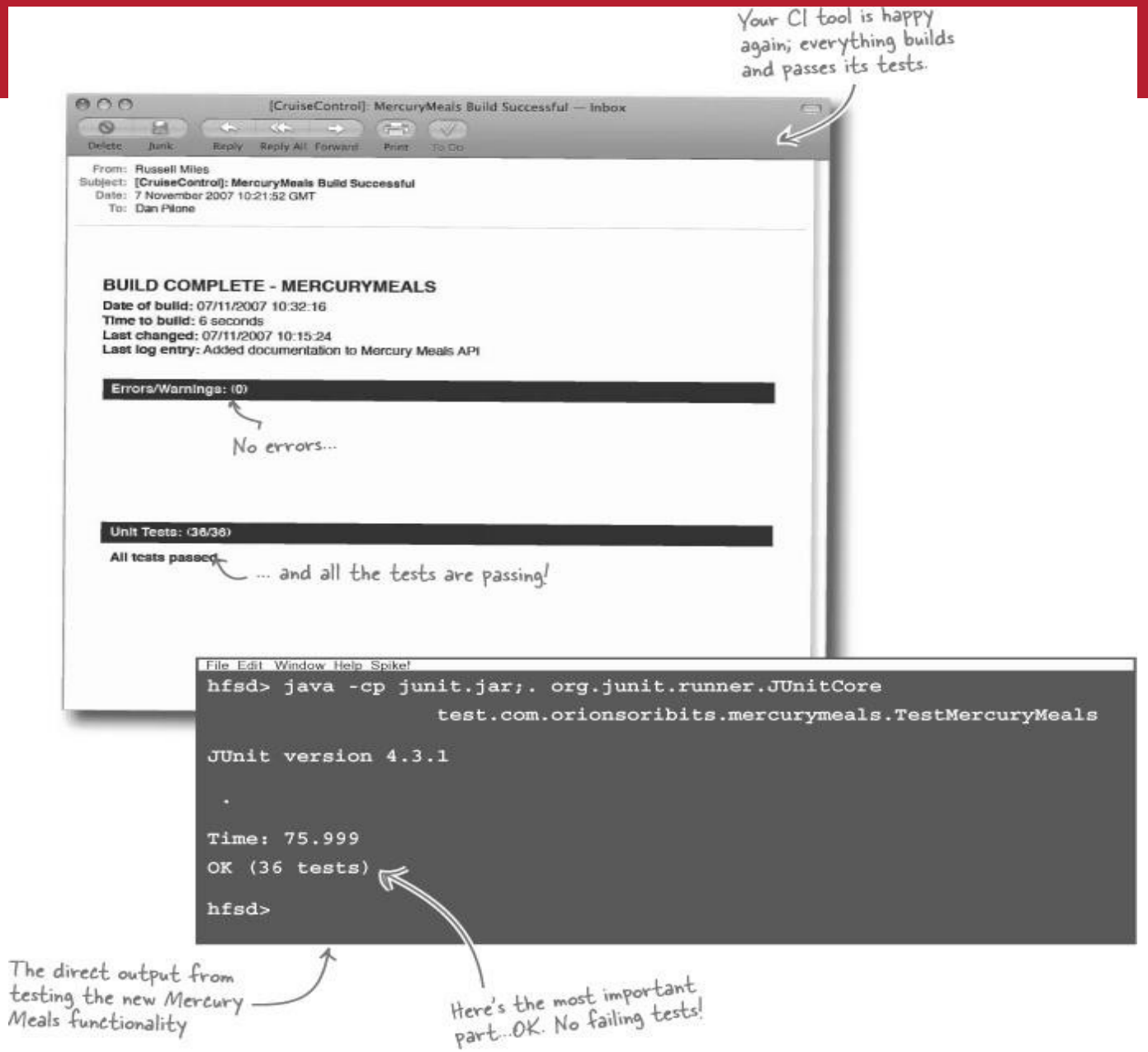
Remember, This is just an Estimate

- When it comes to bug fixing, we really can't be sure
- A spike test really only gives you a more accurate estimate than a pure guess.
  - It's not 100% accurate, and may not even be close.
- But the spike test does give you **quantitative data** upon which you can base your estimates.
- You know
  - how many bugs you fixed, and it was a random sample
    - So you can say with a certain degree of confidence that you should be able to fix the same number of further bugs in roughly the same amount of time.

Give Your Customer The Estimate

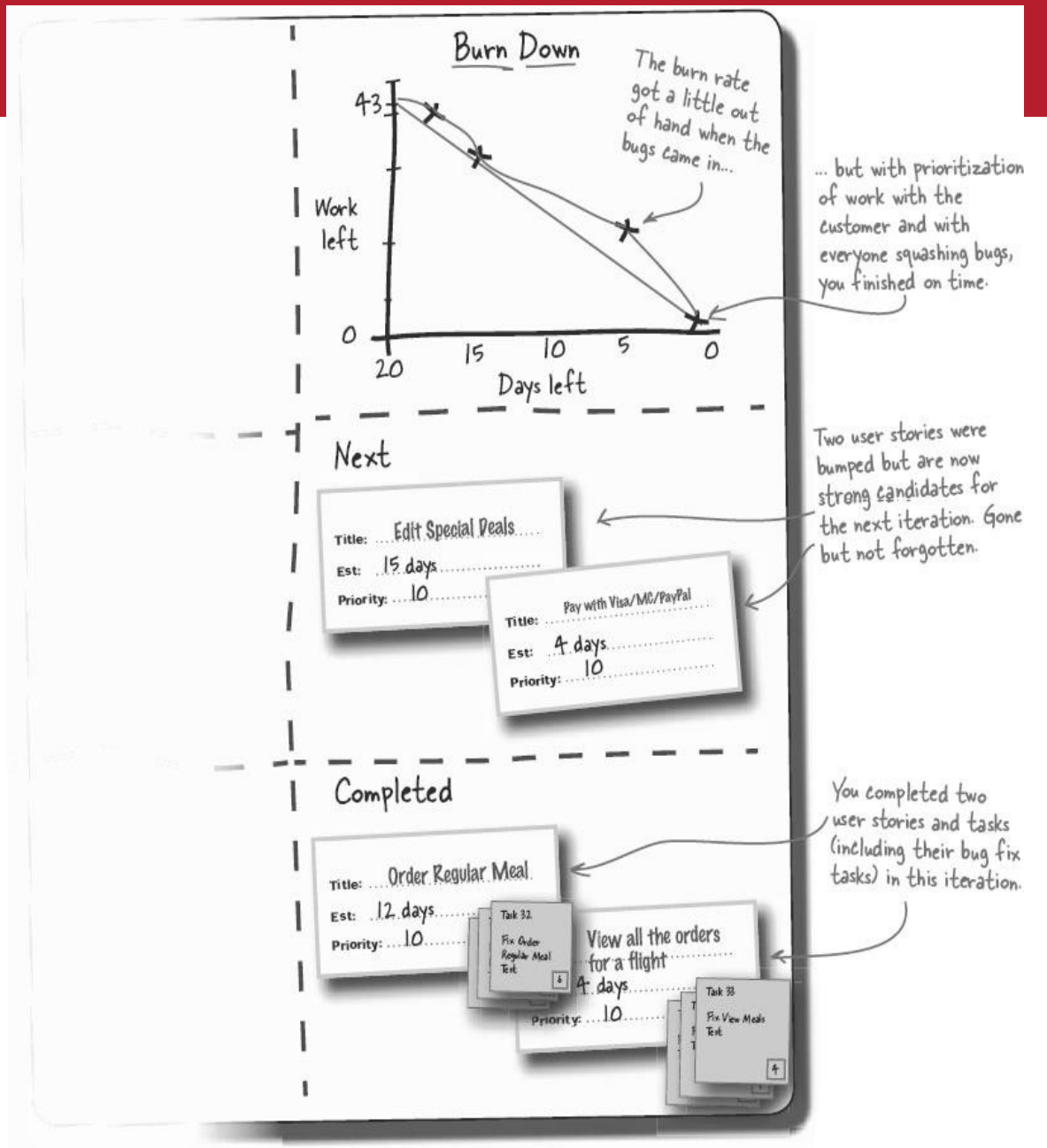


THINGS ARE LOOKING UP



SUCCESS?

- You've reached the end of this iteration and, by managing the work and keeping the customer involved, you've successfully overcome the bugs nightmare.
- Most importantly, you've developed what your customer needed.

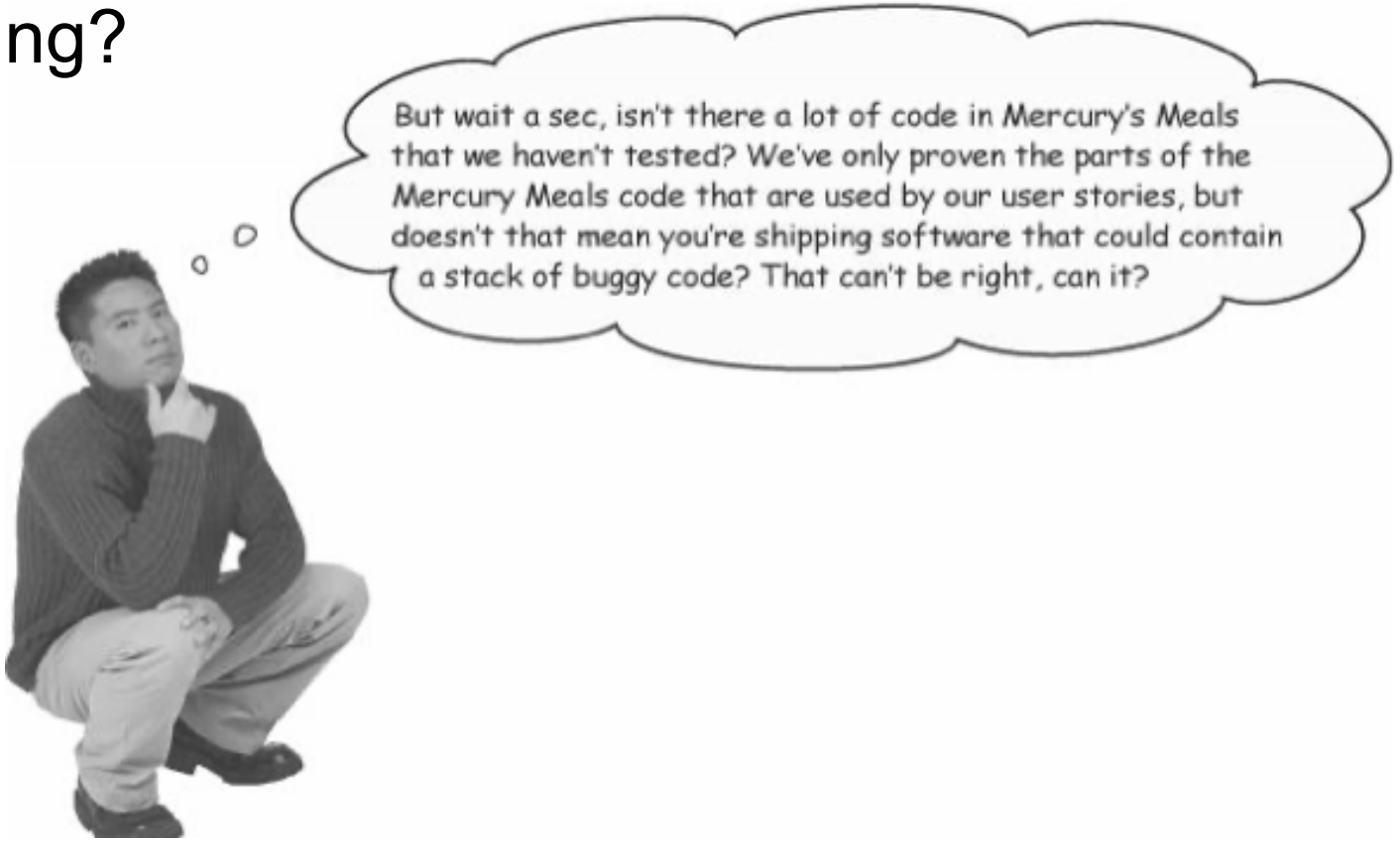


Role Play – Mercury Meals Code From Book

- You inherit and integrate code/libraries (open source, purchased, etc.)
- You write unit tests for that code
- You find bugs
- You fix bugs **related only to functionality**

BUT WE'VE TESTED ONLY (USER STORY) FUNC.

- What might we be missing?



BUT WE'VE TESTED ONLY FUNCTIONALITY

- Yes, there may be bugs in the code, particularly in the Mercury Meals code that you inherited.
  - ***But you delivered code that worked.***
- Yes, there are potentially large pieces of that library that haven't yet been covered by tests.
- ***But you have tested all the code that you actually use to complete your user stories.***

But We've Tested Only Functionality

- The bottom line is that pretty much *all software has some bugs.*
  - However, by applying your process you can avoid those bugs rearing their ugly head in your **software's functionality.**
- Remember, your code doesn't have to be perfect, and often good enough is exactly that: good enough.
  - As long as any problems in the code don't result in bugs (or software bloat), and you deliver the functionality that your customer needs, then you'll be a success, and get paid, every time.

REAL SUCCESS IS ABOUT DELIVERING FUNCTIONALITY

What is an Exception to this?

- Security issues are the one exception.
  - You need to be careful that code that isn't tested isn't available for people to use—either accidentally or deliberately.
  - Your coverage report can help identify which code you're actually using.

SECURITY – ABUSE CASES

- A specification of a type of complete interaction between a system and one or more actors, where the results of the interaction are harmful to the system, one of the actors, or one of the stakeholders in the system.

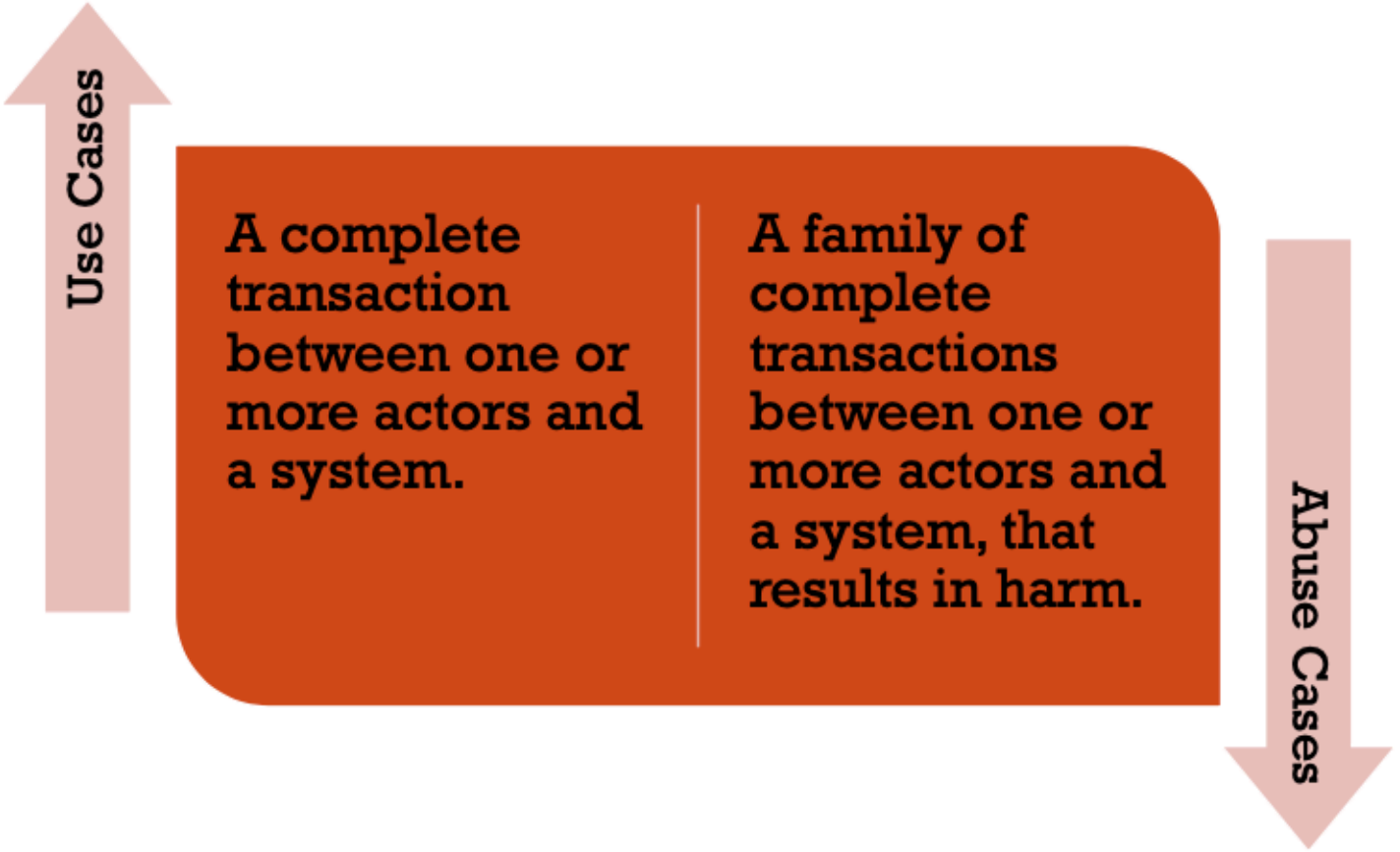
McDermott, John, and Chris Fox. "Using abuse case models for security requirements analysis." *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*. IEEE, 1999.

Abuse Case Purpose

- Identify potential vulnerabilities
- Elicit security requirements
- Created together with a use case diagram

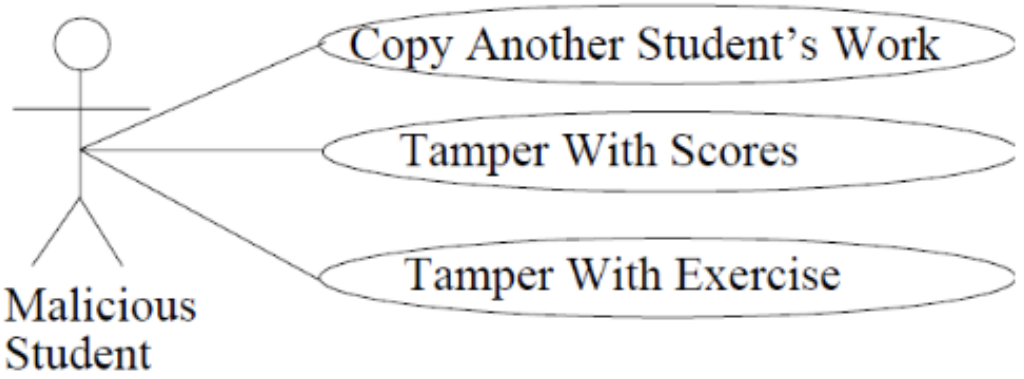
McDermott, John, and Chris Fox. "Using abuse case models for security requirements analysis." *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*. IEEE, 1999.

USE CASE VS. ABUSE CASE

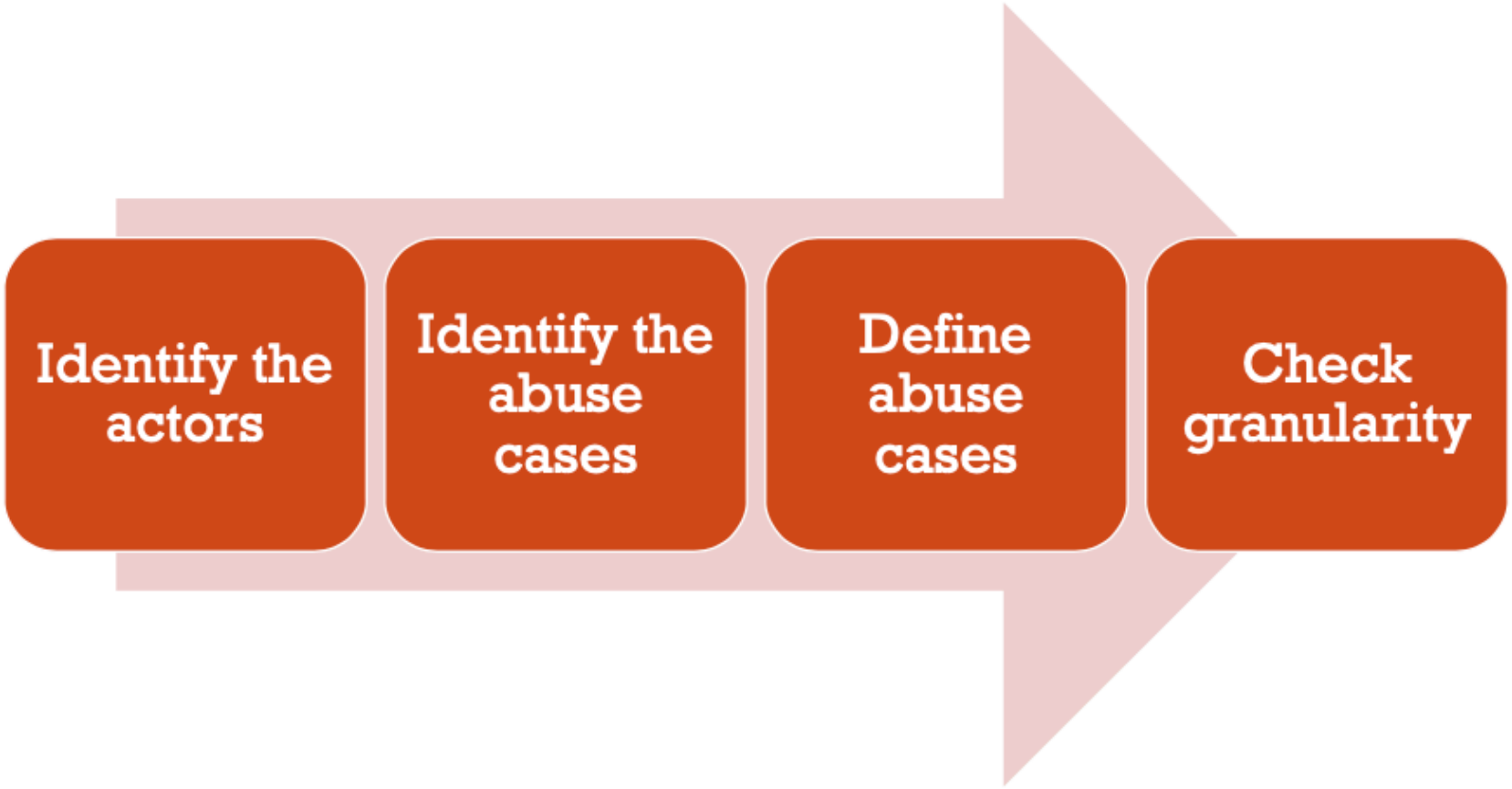


USE CASE VS. ABUSE CASE

- Simply a “negative”/ “harm inducing” version of a Use Case Diagram
- Same symbols as a Use Case Diagram
- Kept separate so as to avoid confusion.



Abuse Case – Process



Costs of Defects

Most defects are introduced in requirements or design activities

- Misinterpreted requirements
- Wrong requirements or recorded incorrectly
- Incorrect design specification or specification incorrectly interpreted

Other sources

- Programming logic or coding error, testing error, bug fix that introduces another error

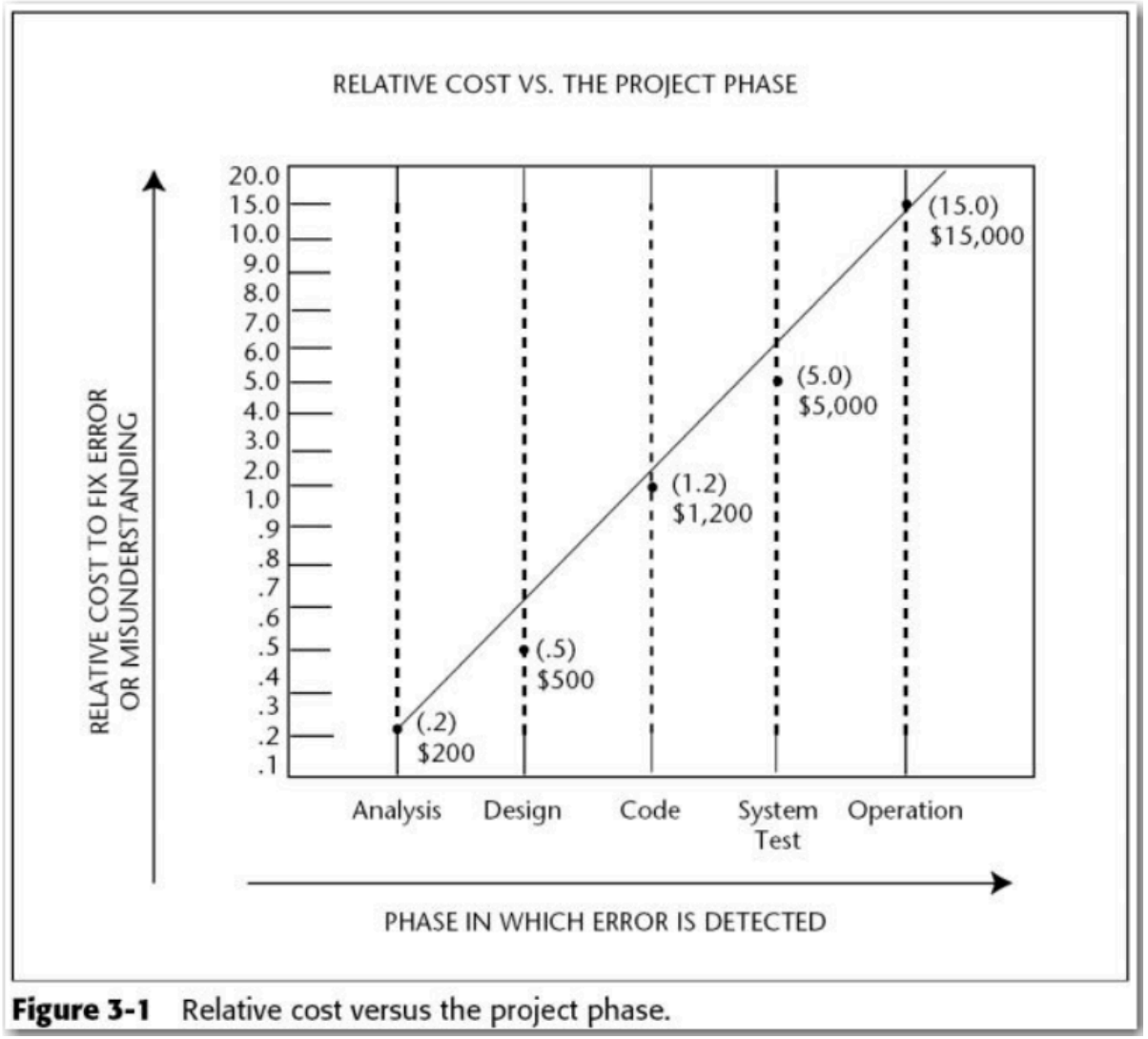


Figure 3-1 Relative cost versus the project phase.

Purpose Of Testing

- Systematically examining design and code to improve quality:
  - Verification: “You built it right” – requirements have been fulfilled
  - Validation: “You built the right product” – fulfills client’s needs
  - Increase confidence
  - Detect failures

- Characteristics of quality
  - Functionality, reliability, usability, efficiency, maintainability, portability

Testing Is Not Debugging

- Testing can identify faults in software (aka bugs or defects)
- Debugging is the task of identifying the cause and correcting faults

General Principles

- Testing shows the presence of defects, not their absence
- Exhaustive testing is not possible
- Testing activities should start as soon as possible
- Defects tend to cluster together
- Tests are dependent on use and environment

Test Types – Review

- Component (aka unit) test – Testing modules, classes, programs, etc., individually.
- Integration test – Assemble groups of modules, classes, etc., and test collaboration
- System Test – Test the system as a whole
  - All modules function together as specified
  - Operating system environment, hardware compatibility, etc.
- Acceptance test – Check if system meets requirements from customer's perspective
  - Install / Uninstall the system
  - Real World user-level testing: test it like a user
  - Get customer's perspective and judgment

Retrospective Questions

- What do you do when you find a bug (most importantly)?
- What is spike testing?
- What does testing show and what does it not show?
- Abuse Cases?
- Test Types?