CS1632, Lecture 7: Smoke, Exploratory, Path-based Testing

Wonsun Ahn

Smoke Testing

Smoke Testing (plumbing)

- Send smoke down the pipes to find leaks BEFORE sending water or other fluids
- Why?
  - Won't waste effort: If there is a leak, nothing to clean up
  - Won't cause further damage: high pressure water going through a hole means a bigger hole will be formed

Smoke Testing (software)

- Minimal testing to ensure that the system is, in fact, ready for serious testing
- Why?
  - No need to test system that can’t perform minimal acceptable functionality
  - Setting up test harnesses / installing software may be non-trivial
  - Avoid wasting testers’ time

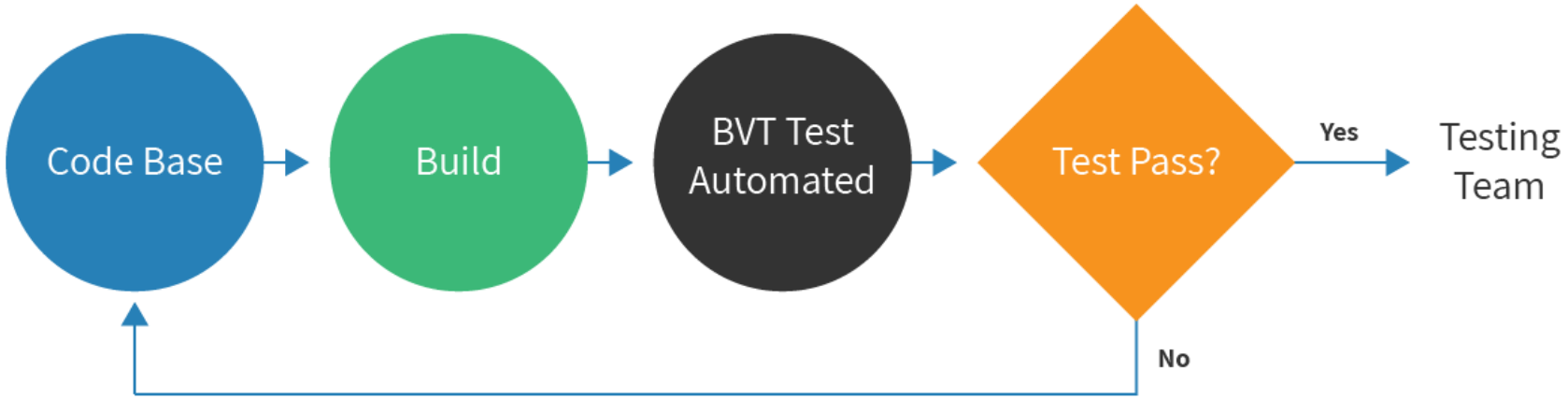
Smoke Testing can be:

- **Scripted**: A few small but important test cases are run before the software is ready to be tested. These can be automated or manual.
- **Unscripted**: An experienced tester does exploratory testing for a small amount of time to ensure that it meets minimum standards.

Other names for Smoke Testing

- Confidence Testing
  - Because it’s intended to inspire enough confidence to pass to the QA team
- Sanity Testing
  - Because it’s intended to check that developer was fully awake when coding
- Build Verification Testing (BVT)
  - Because it’s intended to be performed after every build before further testing

Smoke Testing is a GATEWAY



Exploratory Testing

So far …

- We have developed a very formal manner of testing
  - Develop requirements
  - Write test plan
  - Create and check traceability matrix
  - Run tests

So far …

- But we assumed that we know the EXACT expected behavior, EXACTLY how to cause it, and it is necessary to DEFINE all of these behaviors
  - Works fine in some circumstances!
  - But not others!

- If I asked you to “test a poker program”, what would you do?

Sometimes, we don’t know exactly what the expected behavior is! Why not?

- Uncertain of exact reproduction steps
- Uncertain of interface
- Unfamiliarity with general interaction
- Implicit requirements
- Domain-specific
- Subjective

Exploratory Testing

- Definition: testing without a specific test plan, in which the goals are:
  - To learn more about the system
  - To guide development by finding defects and possible enhancements

Sometimes called *“ad hoc” testing*

- Personally, I don’t like this term
- It implies carelessness
- Less rigid != more careless
- Faith in the testers is required
  - To not go down blind alleys
  - To use their best judgment

How To Do It

1. Use your best judgment
2. If in doubt about next step, see Step 1.

Faith in Testers

Exploratory testing has faith that you instinctively "know" that there's a defect, or at least that you know something doesn't seem quite right.

Tips:

1. Try to accomplish important tasks
2. Think of edge cases on the fly
3. Try doing different things together
4. If I were the programmer, what wouldn't I have thought of?
5. Write down defects IMMEDIATELY
6. You can record your steps and write them down later as formal tests

Benefits of Exploratory Testing

1. Fast
2. Flexible
3. Relies on testers' knowledge, and helps improve it
4. Very easy to update!

Drawbacks to Exploratory Testing

1. Unregulated
2. Possibly unrepeatable
3. Hard to say how much coverage there is
4. Difficult to automate

Path-Based Testing

Possible paths in a method

```
// How many paths?

public int somethingElse(boolean a, boolean b) {
  int toReturn = 5;
  toReturn += (int) Math.cos(100);
  toReturn *= 3;
  return toReturn;
}
```

Possible paths in a method

```
// How many paths?

public int doSomething(boolean a, boolean b) {
  int toReturn = -1;
  if (a || b) {
    toReturn = 5;
  } else {
    toReturn = 97;
  }
  return toReturn;
}
```

Possible paths in a method

```
// How many paths?

public int somethingElse(boolean a, boolean b) {
  int toReturn = 0;
  if (a) {
    toReturn = 5;
  } else if (b) {
    toReturn = 97;
  } else {
    toReturn = 6;
  }
  return toReturn;
}
```

Path-Based Testing

- What are all the possible paths through a program or method?
- Then test all of the paths
- Similar to equivalence class partitioning
  - Just as you need to test only one (or a few) values to test an equivalence class
  - You can test only one (or a few) values to test each path
  - Just like for equivalence classes, you want to cover all paths for good coverage

Path-Based Testing Example

- Racing game: user can select Red Car (fast acceleration, low top speed) or Blue Car (slow acceleration, high top speed). One or the other car always wins.
- Possible paths:
  - Red Car -> Win -> “You win, Blue Car loses”
  - Red Car -> Lose -> “You lose, Blue Car wins”
  - Blue Car -> Win -> “You win, Red Car loses”
  - Blue Car -> Lose -> “You lose, Red Car wins”

Complexity Increases Superlinearly As We Add Variables / Paths

- Add “Easy / Hard” modes to previous game
- Hard mode rewards you with an exclamation point
- Now there are EIGHT paths to test:
  - Easy -> Red Car -> Win -> “You win, Blue Car loses”
  - Easy -> Red Car -> Lose -> “You lose, Blue Car wins”
  - Easy -> Blue Car -> Win -> “You win, Red Car loses”
  - Easy -> Blue Car -> Lose -> “You lose, Red Car wins”
  - Hard -> Red Car -> Win -> “You win, Blue Car loses!”
  - Hard -> Red Car -> Lose -> “You lose, Blue Car wins!”
  - Hard -> Blue Car -> Win -> “You win, Red Car loses!”
  - Hard -> Blue Car -> Lose -> “You lose, Red Car wins!”
- One Boolean variable doubles the number of paths/tests