

CS1632, Lecture 5:  
Traceability Matrices + Smoke ,  
Exploratory Testing, and Path-Based  
Testing

Bill Laboon



# Traceability Matrices

- ▶ Answer the questions - how do I know that my tests are checking all of the requirements, AND/OR that all of my requirements are being checked by tests?
- ▶ Simply a list of requirements and the associated test case



# Example - Good Matrix

**LOGIN-REQ:** GOOD-LOGIN-TEST, BAD-LOGIN-TEST, THREE-TIMES-ERROR-TEST

**DATABASE-REQ:** VALID-QUERY-TEST, INVALID-QUERY-TEST, DB-DOWN-TEST

**LOGIN-SCREEN-REQ:** LOW-BANDWIDTH-TEST, HIGH-BANDWIDTH-TEST

**CALC-REQ:** ADD1-TEST, ADD2-TEST, ADD3-TEST, SUBTRACT1-TEST



# Example - Requirement Not Tested!

**LOGIN-REQ:**

**DATABASE-REQ:** VALID-QUERY-TEST, INVALID-QUERY-TEST, DB-DOWN-TEST

**LOGIN-SCREEN-REQ:** LOW-BANDWIDTH-TEST, HIGH-BANDWIDTH-TEST

**CALC-REQ:** ADD1-TEST, ADD2-TEST, ADD3-TEST, SUBTRACT1-TEST



# Example - Tests Not Checking Any Reqs

**LOGIN-REQ:** GOOD-LOGIN-TEST, BAD-LOGIN-TEST, THREE-TIMES-ERROR-TEST

**DATABASE-REQ:** VALID-QUERY-TEST, INVALID-QUERY-TEST, DB-DOWN-TEST

**LOGIN-SCREEN-REQ:** LOW-BANDWIDTH-TEST, HIGH-BANDWIDTH-TEST

**CALC-REQ:** ADD1-TEST, ADD2-TEST, ADD3-TEST, SUBTRACT1-TEST

: DEFROBALIZE-TEST, ANTI-DEFROBALIZE-TEST




# Exploratory Testing

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



# Exploratory Testing

- But we have 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?

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





# Exploratory Testing

- Definition: testing without a specific test plan, in which the goals are to both learn more about the system and inform the development of system 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 keep track of your steps and write them down later as formal tests.




# Benefits of Exploratory Testing

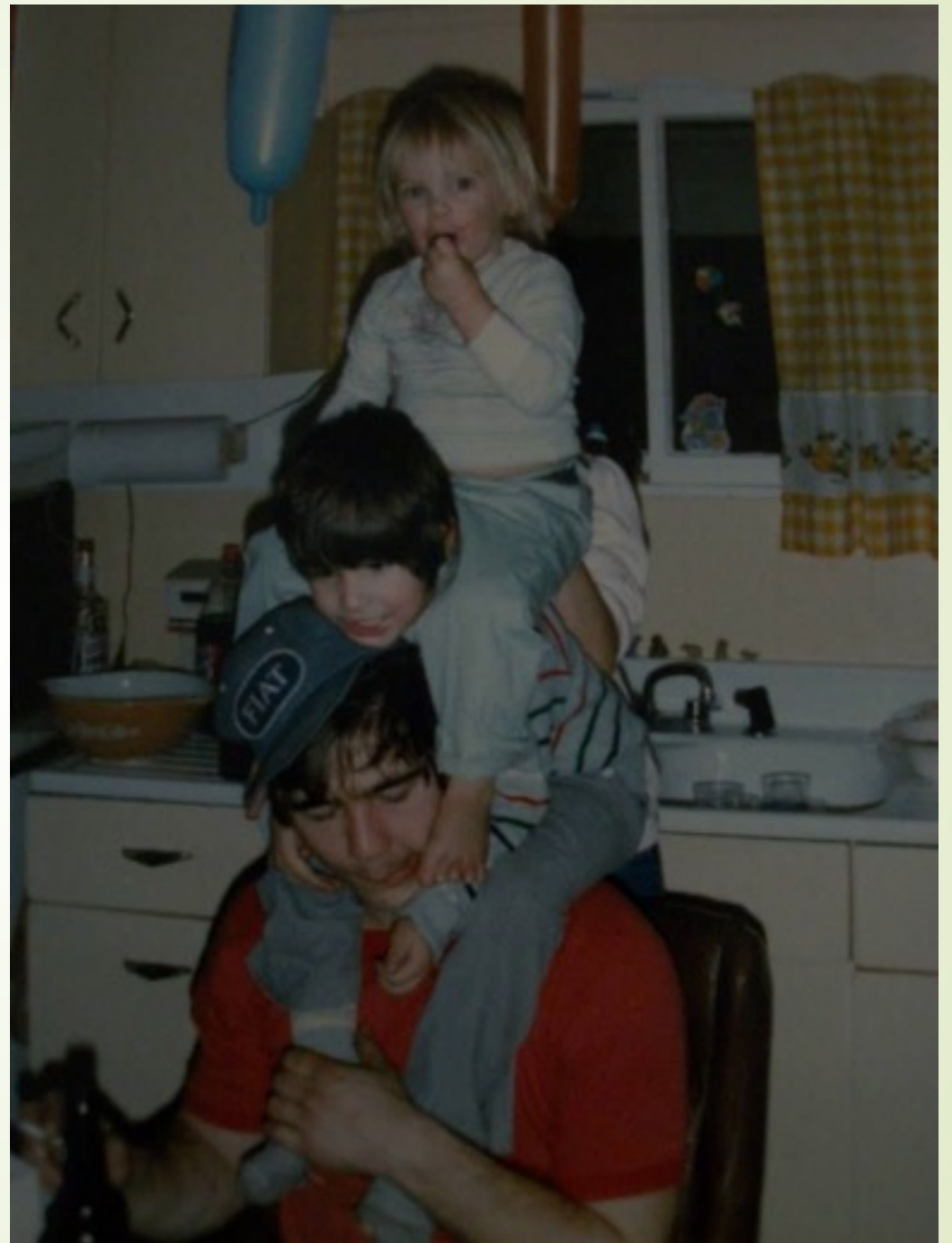
1. Fast
  2. Flexible
  3. Relies on testers' knowledge, and helps improve it
  4. Very easy to update!
- 



# Drawbacks to Exploratory Testing

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

# Smoke Testing







# Smoke Testing (plumbing)

- Send smoke down the pipes to find leaks BEFORE sending water or other fluids
- Why?
  - If there is a leak, much easier to clean up / find smoke
  - Won't waste effort
  - Won't cause further damage (high pressure water going through a hole means a bigger hole will be formed)




# Smoke Testing (software)

- ▶ Do some minimal testing to ensure that the system is, in fact, testable or ready to be released
- ▶ 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.
- 

# Smoke Testing is a GATEWAY

## FULL TEST PLAN





# Media Check

- ▶ A really, really basic smoke test
  - ▶ Can the CD be read?
  - ▶ Do files exist on server?
  - ▶ Etc.




# A Note on “Sanity Testing”

- ▶ Note: Some texts use the term “sanity testing” for “smoke testing”. I avoid this because:
  - ▶ It could be offensive
  - ▶ I think the parallel with smoke testing in plumbing is much more apt
- ▶ However, you may come across the term so I wanted to cover it



# Path-Based Testing

- What are all the possible paths through a program/method/etc.?
  - Then test all of the paths
  - Similar to equivalence class partitioning
- 



# 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 / Pathways

- ▶ Add “Easy / Hard” modes to previous game
- ▶ Hard mode rewards you with an exclamation point
- ▶ Now there are EIGHT paths to test
- ▶ One Boolean variable doubles the number of paths/tests



# Possible Paths

- 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!"



# Possible paths in a method

// How many paths?

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



# 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;  
}
```




# Possible paths in a method

```
// How many paths?  
  
public int somethingElse(int x) {  
    if (x < 0) {  
        return -1;  
    } else {  
        if (x % 2 == 0) {  
            return 3;  
        } else {  
            if (x > 100) {  
                x++;  
            }  
            return x;  
        }  
    }  
}
```



# McCabe Cyclomatic Complexity

- ▶ A measure of the number of paths through a method, function, or other unit of control flow
  - ▶ Analysis of method from the perspective of graph theory
  - ▶ Higher complexity -> more chance of defects
  - ▶ More details: <http://www.mccabe.com/pdf/mccabe-nist235r.pdf>
- 

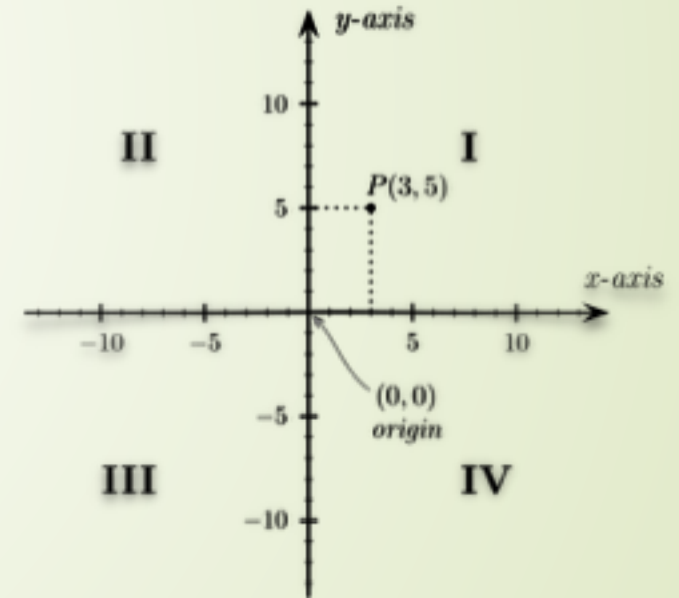


# McCabe cyclomatic complexity

- ▶ Views a program's control flow through the lens of graph theory
- ▶ Given a method's control flow, calculate:
  - ▶  $E$  = number of edges of graph
  - ▶  $N$  = number of nodes of graph
  - ▶  $p$  = number of connected components (usually 1)
  - ▶ Cyclomatic complexity =  $E - N + 2p$
  - ▶ Also equal to the number of possible paths through a method

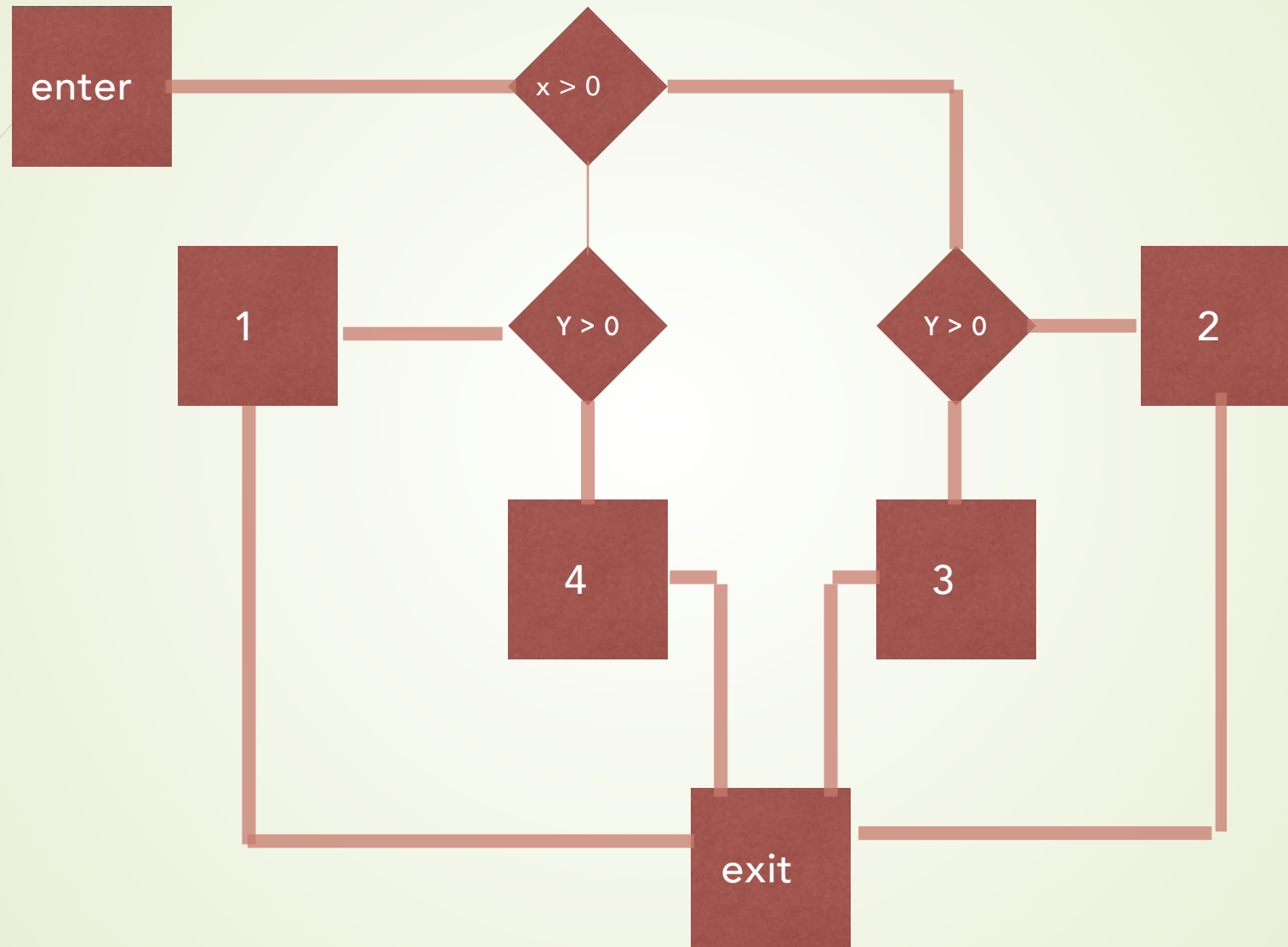
# Cyclomatic Complexity Example

```
public int whichQuadrant(int x, int y) {
    int toReturn = -1;
    if (x > 0) {
        if (y > 0) {
            toReturn = 1;
        } else {
            toReturn = 4;
        }
    } else {
        if (y > 0) {
            toReturn = 2;
        } else {
            toReturn = 3;
        }
    }
    return toReturn;
}
```

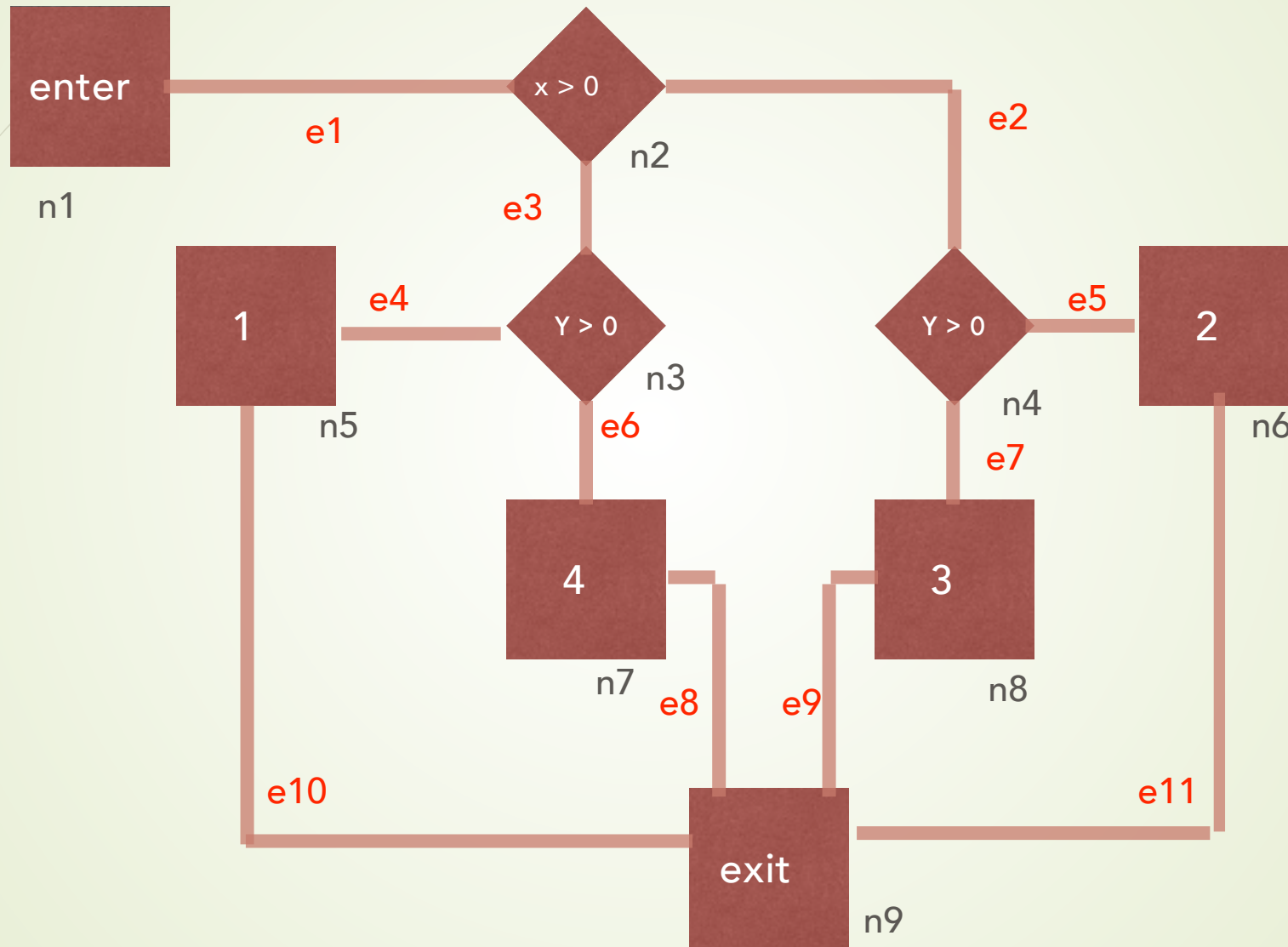




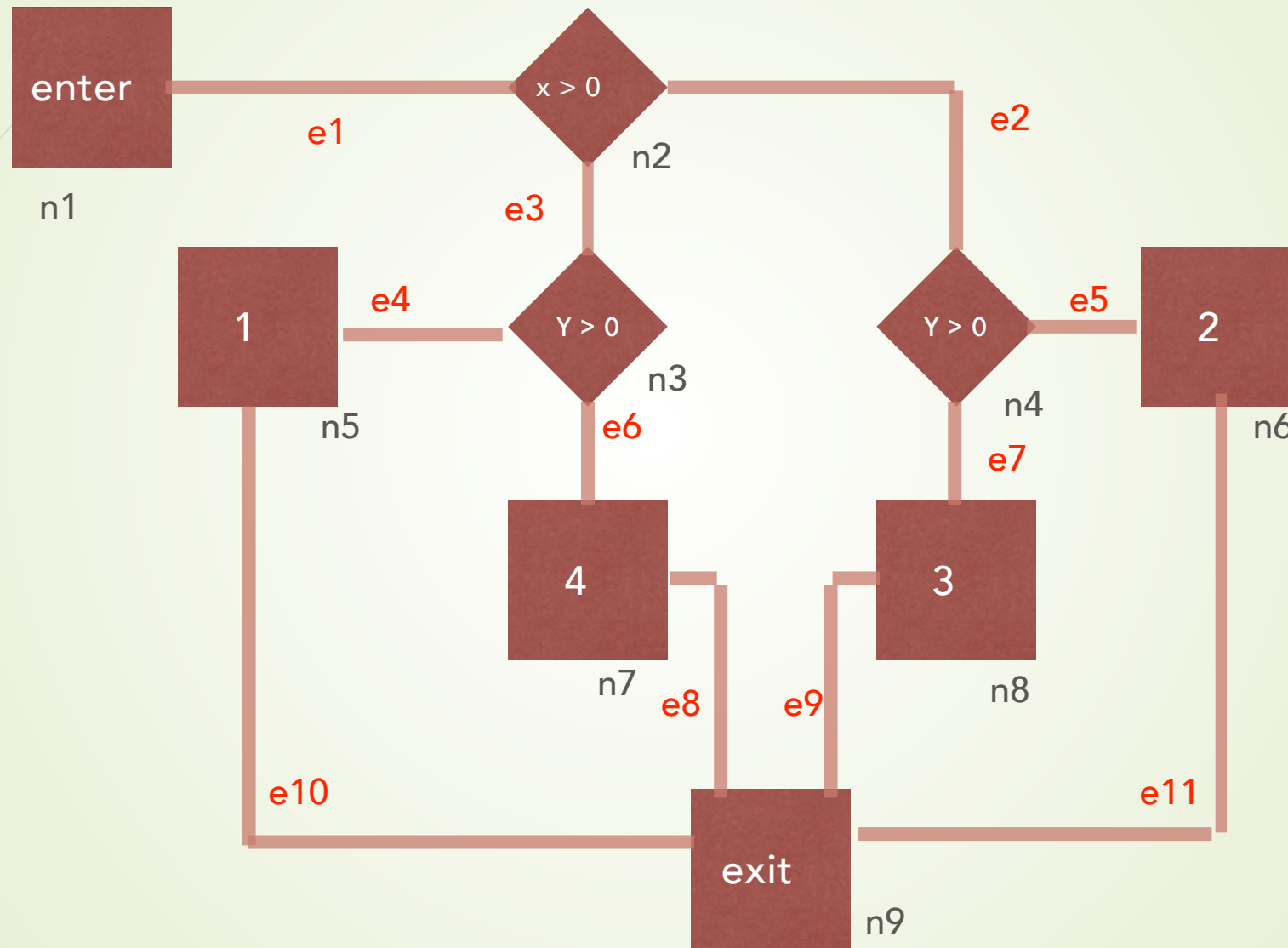
# Cyclomatic Complexity Example



# Cyclomatic Complexity Example



# Cyclomatic Complexity Example



Edges = 11

Nodes = 9

$p = 1$

$E - N + 2p$

$11 - 9 + 2 * 1$

CC = 4



# Understanding Cyclomatic Complexity

- The maximum number of linearly independent paths through the control flow of the method
- Lower cyclomatic complexity = lower risk, easier to understand
- $< 10$  = very simple, low risk
- $> 50$  = very complex, high risk