Lecture Notes
November 16, 2017

Second Iteration Development and Code Inspection due November 30
(includes preliminary demo for second iteration)

Panel with professional software engineers tomorrow
Friday November 17, 11:45-12:45, Davis Auditorium
Refreshments afterwards in my lab (CEPSR 6LE1)

White Box Testing

White box testing can be static or dynamic - recall dynamic means execute the code, static means do not execute the code, just examine

White box = clear box = glass box

What does *static* mean here? Code review (or code inspection)

What does *dynamic* mean here? Running the code with sample inputs and checking the results - What we normally think of as “testing”, but…

Main purpose of white-box testing is *coverage*: exercise “all” the code

Black-box testing more concerned with exercising all the features

Coverage usually targets *control-flow* testing, also known as structural testing

- Run all black box and grey box tests first, then construct additional inputs to force reaching any code not already covered
- Possible to automate test case generation to reach coverage goals, by finding inputs that will force a particular branch or path not already covered
- Test case generators can also supply random inputs, or random changes to existing inputs - fuzz testing

Cannot catch errors of omission, unlikely to catch errors in design

How would you test this function so that every statement is covered?

```
template <typename T>
int seqSearch(const T list[], int listLength,
       T searchItem)
{
  int loc;

  for (loc = 0; loc < listLength; loc++)
    if (list[loc] == searchItem)
      return loc;

  return -1;
}
```

Can we cover this statement with a test?

```
int BidderCollection::add (const Bidder& value)
//  Adds this bidder
//Pre: getSize() < getMaxSize()
{
  if (size < MaxSize)
    {
     addToEnd (elements, size, value);
     return size - 1;
    }
  else
    {
     cerr << "BidderCollection::add - collection is full" << endl;
     exit(1);
    }
}
```

- During unit test, certainly.
- During an integration or system test, probably not.
  - If the program calling this function allows execution of that statement, it’s almost certainly a bug in that program.
  - Typical of code inserted as “defensive programming”

Branch coverage and condition coverage are better than statement coverage

```
boolean purchaseAlcohol (int buyerAge, int ageFriend)
{
        boolean allow = False;
        if ((buyerAge>=21) or (ageFriend >= 21))
        {
        allow = True
        }
        return allow;
}
```

Rule is that both people need to be over 21. **Bug** in the function: *OR* vs. *AND*

Assert purchaseAlcohol(25,20) == False
will FAIL (one of friends is over 21 and the ‘or’ is the bug)

Assert purchaseAlcohol(25,25) == True
will PASS has **100% statement** coverage

Assert purchaseAlcohol(25,25) and Assert purchaseAlcohol(20,20)
will satisfy **branch coverage**

Assert purchaseAlcohol(25,20) and Assert purchaseAlcohol(20,25)
will satisfy **condition coverage**

Or will it? Condition coverage where all conditions must evaluate to T or F

purchaseAlcohol(25,25) T, T
purchaseAlcohol(20,20) F, F

Will satisfy condition coverage as will

purchaseAlcohol(25,20) T, F
purchaseAlcohol(20,25) F, T

But only the latter option will trigger the bug, coverage != effectiveness
Multiple condition coverage demands *all* permutations of conditions executed - truth table

Monitoring Statement Coverage

We can check whether we have covered statements by adding debugging output.

add a unique output to each block of “straight line” code

```
#define COVER cerr << "Block " << __FILE__ << ":" << __LINE__ << endl
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
    COVER;
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            {
             COVER;
             return loc;
            }

    COVER;
    return -1;
}
```

However, most languages have tools that monitor statement and branch coverage, possibly other coverages

- Coverage tools instrument code, and then records when exercise (e.g., Coverage.py)

Loop coverage is rarely supported by coverage tools

- if in at least one test the body was executed 0 times, and
- if in some test the body was executed exactly once, and
- if in some test the body was executed more than once.

Another approach to coverage is *state-based* testing

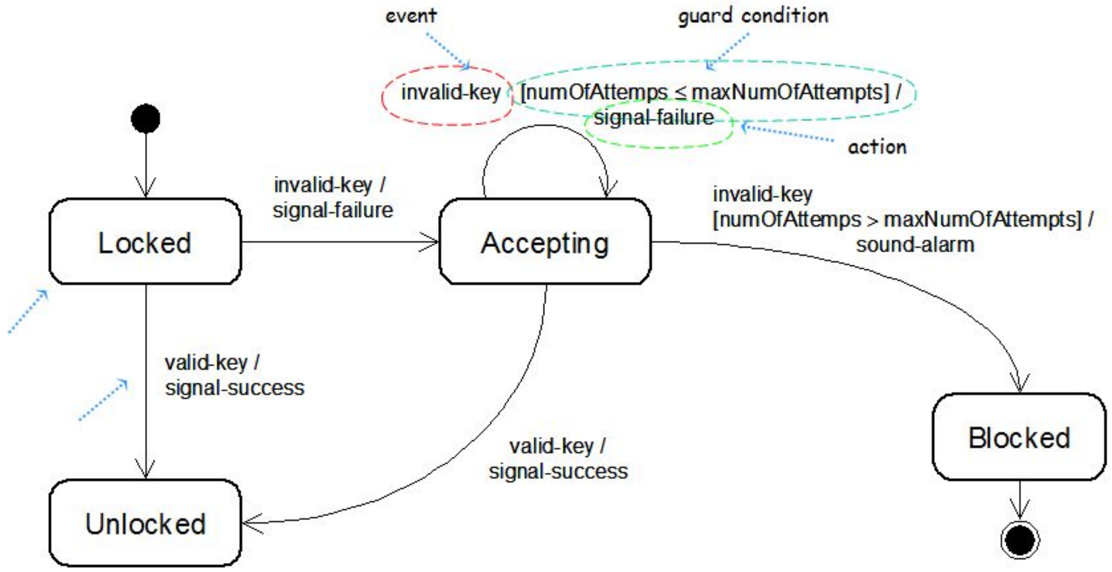
Define a set of abstract states that software can be in and transition between

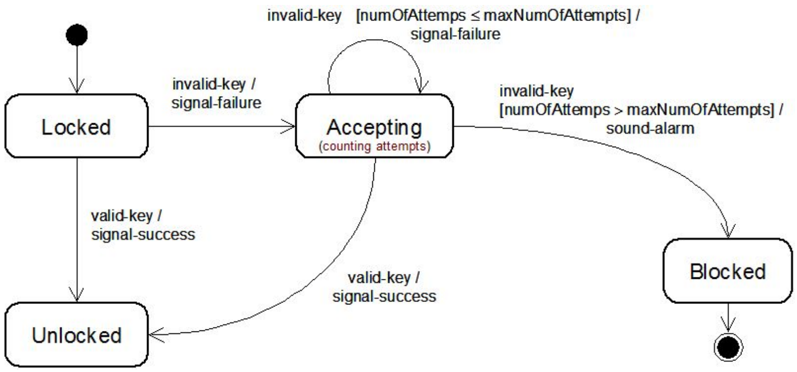
- Cover every state and every transition

- Is it possible to move between states via undefined/not-allowed transitions?

In some web and mobile applications, different pages or screens correspond to different states

- Less common with single-page applications and other cases where multiple states reflected by small changes to same page/screen





Stress Testing

May be black, grey, white box

Check every potentially shared code or resource for thread-safe

- Force simultaneous execution in multiple threads, every possible read/write interleaving

Check all components with “right” and “wrong” security roles

Check resource constraints, need to fail gracefully

- Memory, disk or file system, network connections, database connections

“Model checking” - Check every call to third-party APIs, standard libraries, system calls

- Null return value, every possible error code

“Quick tests” - relatively simple actions that break almost anything

Shoe test - put cursor on text input field, put shoe or other heavy objection on auto-repeat keyboard (or put keyboard in cage with small active animal), go to lunch

GUI Interference tests

- Force screen to refresh
- Change video resolution
- Change device orientation for mobile apps
- Click mouse or use touch gestures randomly all over screen
- Toggle “accessibility” options back and forth
- Change focus to another application, do something, return - or leave in background for long period of time

Non-GUI Interference tests

Leave application running for long period of time

Set timer to go off that causes program to do something

Change system date/time to near/far past/future

Load enough other applications to force out of memory

Lock necessary database records by another program

Pause/kill component processes (in multi-process applications)

Pause/kill client or server (in distributed applications)

File system interference tests

Remove DVD-ROM, flash drive or other media while in use

Fill file system to capacity (example resource constraint)

Reserved file name in use or locked for some other purpose

Vary file names and access permissions while program running and/or between runs

Change or corrupt contents of file during or between runs

Scalability tests

Connect too many clients

Connect/disconnect/reconnect repeatedly

Bombard with requests (DDOS attack)

Class exercise if time permits