Course review

- Poor software design and engineering are the root causes of most security vulnerabilities in deployed systems today.

- This course takes a close look at software as a mechanism for an attack, a tool for protecting resources, and as a resource to be defended

Measuring system complexity

- **Complexity means structural complexity**, how complicated the program code is, and not **execution complexity**, how much time/memory is required to run the program.
- **Complexity could be measure by <u>two</u> approaches:**
  1. **Lines-of-code metric (LOC):**
     by counting the number of lines in a piece of code. More lines implies more errors.
  2. **McCabe's cyclomatic-complexity metric:**

  measures the complexity of method by counting number of independent paths in method.
  - number of independent paths = number of decision points, each one of the following add 1 to scyclomatic complexity, start counting from 1:
    - Regarding if; each && or || adds one
    - (while, do while, for) loops,
    - Regarding Switch, for each non-default case test,
    - Regarding Try, for each catch block but not the final block,

Measuring system complexity

- Consider the following graph that represents a program, where the letters a through l represent statements in the program:
- To compute the metric, we find the number of independent paths.
- There are 4 independent paths : abdhk, abeik, acfjl, acfgfjl.

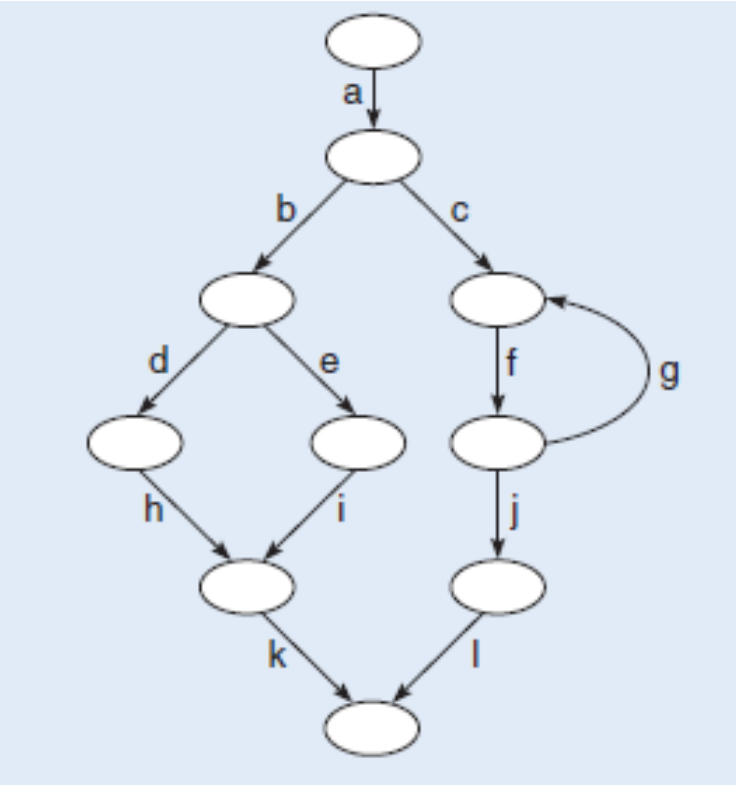

Figure 5 Computing independent paths

Measuring system complexity

SAQ 8

- Compare the complexities of the following two pieces of code using the LOC and cyclomatic-complexity metrics. What conclusions can you draw about the relative complexity of the code?

Code A

```
int i = 1;
while (i <= 5 ) {
   playACard(i);
   if (playerHasWon(i))
       break;
   i++
}
```

Code B

```
int j = 0;
int i = 2;
j = i;
j = j + i;
j++;
System.out.println ( j);
System.out.println (i);
```

Measuring system complexity

ANSWER.....

- Each piece of code has 7 lines.
- The complexity according to the LOC metric is therefore the same.
- Code A has cyclomatic complexity 3.
- Code B has cyclomatic complexity 1.
- Code A is the more complex of the two. This supports an intuitive view that the structure of code A appears more complex than that of code B.

Object-oriented metrics

- **Object-oriented systems require two levels of complexity metric:**
  1. one to measure method complexity,
  2. the other to measure the complexity of the class structure.

Object-Oriented Complexity metrics are:

1. Depth-of-inheritance-tree (DIT) metric:

DIT is the largest number of hops through an object's superclasses, where the starting class is numbered 0.

Object-oriented metrics

2. Coupling-between-objects (CBO) metric:

For a given class, CBO is the number of relationships the class has with other classes.

- The more relationships a class has, and so the higher the value of this metric, the more difficult it is to understand the use of the given class.

3. Number-of-children (NOC) metric:

For a given class, NOC is the number of immediate children for that class. This metric is a measure of the number of classes that will be affected by changes to a given parent class.

Object-oriented metrics

Exercise 3

- Should the cyclomatic-complexity metric be used to measure the complexity of a whole software system?

Solution .....

- Because the cyclomatic-complexity metric is based on decision points, which are present only in methods, it is 'blind' to the class-structuring mechanisms that are available in object-oriented system descriptions. As much of the complexity of an object-oriented system is held in the class structure, applying the cyclomatic complexity metric to a whole system would not, therefore, be appropriate.

Object-oriented metrics

Exercise 4

- The following piece of code is part of TherapyServer (based on an idea from Budd [2000]), an unfaithful translation into Java of Eliza, a program that parodied a therapist to demonstrate some artificial intelligence and natural language processing concepts.
- Calculate the cyclomatic complexity for this method by filling in the right-hand column, cumulatively.

line number	code	cyclomatic complexity
1	`private String response (String text) {`	
2	`  if (text.endsWith ("?"))`	
3	`  return "Why do you want to know?";`	
4	`  Vector words = new Vector ();`	
5	`  StringTokenizer breaker =`	
6	`  new StringTokenizer (text.toLowerCase (), " .,?!");`	
7	`  while (breaker.hasMoreElements ())`	
8	`  words.addElement (breaker.nextElement ());`	
9	`  if ((words.size () > 1) &&`	
10	`  words.elementAt (0).equals ("i") &&`	
11	`  words.elementAt (1).equals ("feel"))`	
12	`  return "So you feel " + words.elementAt (2);`	
13	`  for (int i = 0; i < words.size (); i++) {`	
14	`  String relative = (String) words.elementAt (i);`	
15	`  if (isRelative (relative))`	
16	`  return "Tell me more about your " + relative;`	
17	`  }`	
18	`  return "Tell me more";`	
19	`  }`	

Object-oriented metrics

Solution .....

- The table should have the following entries, giving a cyclomatic complexity of 8.

Line number	Cyclomatic complexity
1	1 (starts at one)
2	2 (if statement)
6	3 (while statement)
8	4, 5 (if statement, &&)
9	6 (&&)
12	7 (for loop)
14	8 (if statement)

Table 4 Calculation of cyclomatic complexity

B. Testing Techniques

- **Black box testing:**
  - Is used to test that each aspect of the customer's requirements is handled correctly by an implementation.
  - **In black box testing** we design **test cases** by looking at the **specification** (that is, requirements and high-level design) of the system to be tested
- **White box testing:**
  - Is used to check that the details of an implementation are correct.
  - **In white box testing** we design **test cases** by looking at the **detail of the implementation** of the system to be tested.

Black box testing techniques

A strategy for black box testing (using partitioning)

Given a class (package, or other subsystem) to test:

1. **Step 1: For each method in the class, determine the input data space** Using use cases or other parts of UML system description.
2. **Step 2: Partition the input data space into subdomains**
3. **Step 3: Test all subdomains given by the case analysis** by choosing test data for each subdomain.

Advantage: it allows all possible user-perceived functions to be tested.

Disadvantage: for many(sub)systems the number of subdomains can be enormous, and hence the effort involved in testing them all can be so large as to be prohibitive.

Black box testing techniques

- **Example 1:** **Step 1 (For each method in the class, determine the input data space)**
  In an airline reservation system, a Booking class allows an operator to update a particular booking using an update method that:
  - has two inputs – command and flightNumber – if a summary of the flight is required,
  - and three inputs – command, flightNumber and seatNumber – if reservation or cancellation of a seat is required.

  The inputs are as follows:
  - command – one of reserve, cancel or summary;
  - flightNumber – an integer in the range 1 to 999;
  - seatNumber – an integer in the range 1 to 450.

  The input data space for the update method thus consists of all possible combinations of three values taken from:
  - {reserve, cancel}, {1, 2, ..., 999}
  - and {1, 2, ..., 450} together with all possible combinations of two values taken from {summary}
  - and {1, 2, ..., 999}.

Black box testing techniques

- **Example 2: Step 2: (Partition the input data space into subdomains)**
  - When the **reserve command** is entered together with a seatNumber and a flightNumber, a seat is reserved.
  - When the **cancel command** is entered together with a seatNumber and a flightNumber, a seat reservation is cancelled.
  - When the **summary command** is entered together with a **flightNumber**, the total number of seats booked is displayed.
- These three cases correspond to three distinct user-perceived functions of the airline reservation system and lead naturally to a partition of the input data space of the update method into three subdomains, as shown in Table 5.

Table 5 The user-perceived functions for the update method

User-perceived function	Subdomain
Reserve a seat	command = reserve 1 < flightNumber < 999 1 < seatNumber < 450
Cancel a seat	command = cancel 1 < flightNumber < 999 1 < seatNumber < 450
Summarise seats booked	command = summary 1 < flightNumber < 999

Black box testing techniques

- Thus, for example, the 'reserve a seat' user-perceived function is exercised by messages of the form (for an object of class Booking, aBooking say):

  aBooking.update (reserve, flightNumber, seatNumber)

- Where flightNumber is between 1 and 999 inclusive and seatNumber is between 1 and 450 inclusive.
- **Example 3: Step 3: (Test all subdomains)**
- Table 5 shows suitable test data for the summary command:

Table 5 Test data for the summary command

Command	Flight number
summary	1
summary	2
summary	500
summary	998
summary	999

- The extreme values for testing the functionality of the summary command are 1 and 999, so we test for the correct processing of these values, along with 2 and 998 (which are values near the extremes) and for 500, the value in the middle of the subdomain.

Black box testing techniques

SAQ 10

- Use the strategy for black box testing described above to choose a good set of test data for the cancel command.

ANSWER.....

- The subdomain for the cancel command is shown in Table 6. We need to test for each flightNumber extreme, near-extreme, and central values of the seatNumber data. Five suitable test values for the seatNumber data are 1, 2, 200, 449 and 450. Thus a suitable set of test data for the cancel command is shown in Table 6.

Black box testing techniques

command	flight Number	seat Number
cancel	1	1
cancel	1	2
cancel	1	200
cancel	1	449
cancel	1	450
cancel	2	1
cancel	2	2
cancel	2	200
cancel	2	449
cancel	2	450

command	flight Number	seat Number
cancel	500	1
cancel	500	2
cancel	500	200
cancel	500	449
cancel	500	450
cancel	998	1
cancel	998	2
cancel	998	200
cancel	998	449
cancel	998	450

command	flight Number	seat Number
cancel	999	1
cancel	999	2
cancel	999	200
cancel	999	449
cancel	999	450

Table 6 Test data for the cancel command

White box testing techniques

1. **Basis-path testing technique**

- Based on the cyclomatic-complexity metric which ensures that all reachable statements in a method are tested at least once.

How to do it?

- Determine the cyclomatic complexity of the flow.
- Count the number of independent paths.
- Prepare test cases that will force execution of each path in the basis set.

2. **Loop testing technique**

- **In Java, there are only three ways in which loops can occur as follows, where each has a different testing strategy:**
  a. simple loops,
  b. nested loops
  c. and concatenated loops.

a. simple loops

```
for (initial; test; increment)
    loop body;
```

```
While (condition)
    loop body;
```

```
Do
    loop body;
while (condition);
```

- **Test data should be found for the following cases:**
  - the loop is skipped entirely.
  - the loop is passed through exactly once;
  - the loop is passed through more than once.
  - the loop body to be executed the maximum n times.

b. nested loops

```
for (int i = 0; i < 10; i++ )
    for (int j = 0; j < 20; j++ )
        for (int k = 0; k < 30; k++ )
            //loop body;
```

- Would generate unmanageable amounts of test data.
- **To cut down the number of cases, The following approach can be used:**

1. Start with the innermost loop, set all other loop counter variables to their minimum values (in the case of the example, set i = 0 and j = 0 and change the limits of the outer and middle loops to i <1 and j < 1, respectively) and conduct a simple loop test on the whole loop.
2. Work outwards, and conduct a simple loop test on the next outermost loop, while keeping all outer loops to 'typical' values (for the example, retain i = 0 and the limit on i of i < 1).
3. Repeat step 2 until all loops are tested.

c. Concatenated loops

```
while (i < 10)
     //loop body;
while (j < 20)
     //loop body;
```

- **Sequential simple loop tests:** can be used to test each one separately

```
while (i < 10)
    total = total + i;
// the next loop uses total, which is set by the previous loop
while (j < total)
    //Do Something
```

- **Nested loop testing:** in case where a later loop uses the termination conditions of an earlier one as its starting values

Problems in Testing Techniques

- **Problems with black box testing**
  - Results indicate poor coverage levels when a black-box testing strategy alone is pursued.
  - A major inadequacy of black-box testing is revealed if a tested method's operation depends for its behavior on the internal state of an object.
- **Problems with white-box testing**
  - The main problem is that white-box testing alone, by concentrating on the code of an implementation, may not reveal customer requirements that have been omitted in an implementation.
- **In general, the solution is to combine black-box and white-box testings.**