CS 3250: Software Testing (Fall 2025)

Activity: Control Flow Graph (CFG) – IfYouAreHappy

(no submission)

Purpose: Create graph representation for source code; apply structural graph coverage to source code; get ready for assignment 4; prepare for quiz 3 and the final exam

You may make a copy of a worksheet and complete this activity, or type your answers in any text editor. You may work alone or with at most two other students in this course.

For this activity, we will create graph models representing the following Java methods. We will stop periodically, and solve the problem one step at a time with discussion between steps.

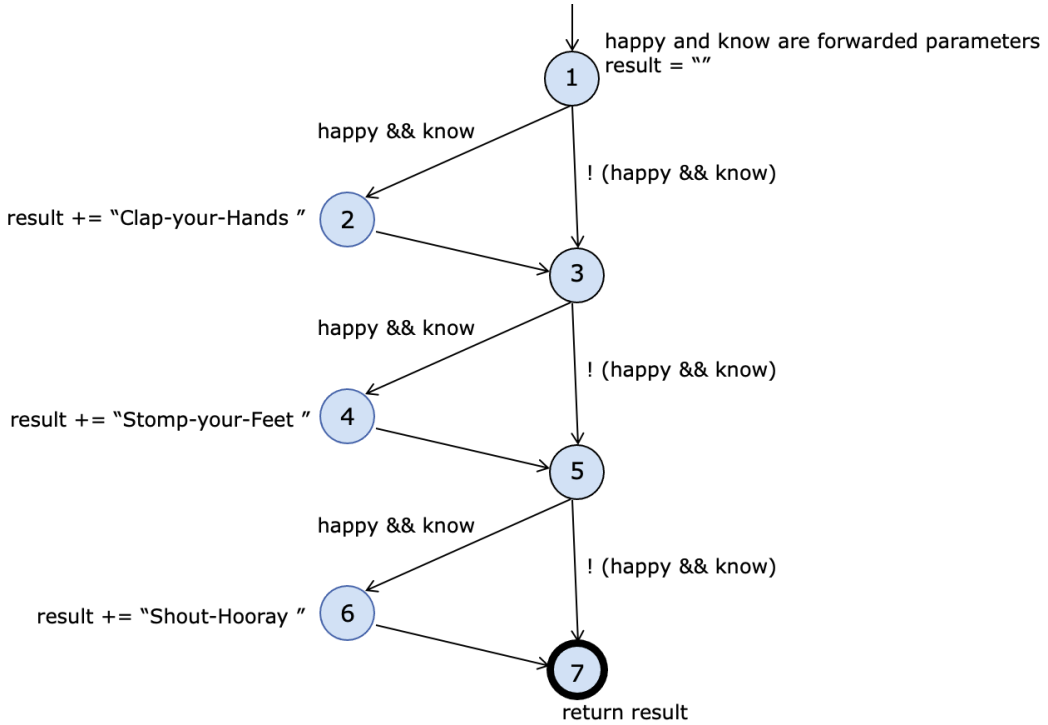
Alternatively, you may transform the "`If You're Happy and You Know It`" song [*https://www.youtube.com/watch?v=Im5i7EqZE1A*] into an `IfYouAreHappy()` method using any program control structures and then generate a Control Flow Graph (CFG) representing the method.

Part 1: Sequential If-statements

Consider the following Java method

```
public static String IfYouAreHappy_v1(boolean happy, boolean know)
{
   String result = "";

   if (happy && know)
      result += "Clap-Your-Hands ";

   if (happy && know)
      result += "Stomp-Your-Feet ";

   if (happy && know)
      result += "Shout-Hooray ";

   return result;
}
```

Draw a **Control Flow Graph (CFG)** for the `IfYouAreHappy_v1()` method. Annotate **all** information (i.e., source code)

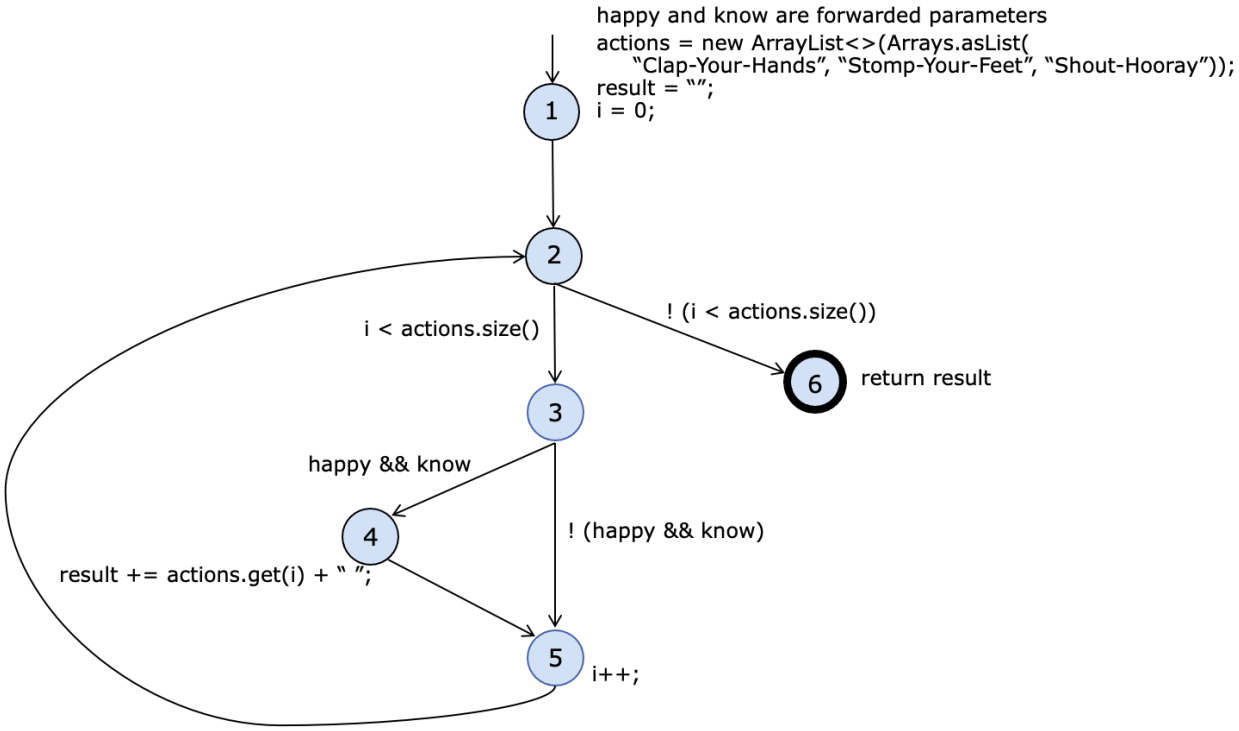


Part 2: For-loop and If-statement, with fixed iterations

Consider the following Java method

```
public static String IfYouAreHappy_v2(boolean happy, boolean know)
{
   List<String> actions = new ArrayList<>(Arrays.asList("Clap-Your-Hands",
                                                        "Stomp-Your-Feet",
                                                        "Shout-Hooray"));
   String result = "";

   for (int i=0; i<actions.size(); i++)
   {
      if (happy && know)
         result += actions.get(i) + " ";
   }

   return result;
}
```

Draw a **Control Flow Graph (CFG)** for the `IfYouAreHappy_v2()` method. Annotate **all** information (i.e., source code)

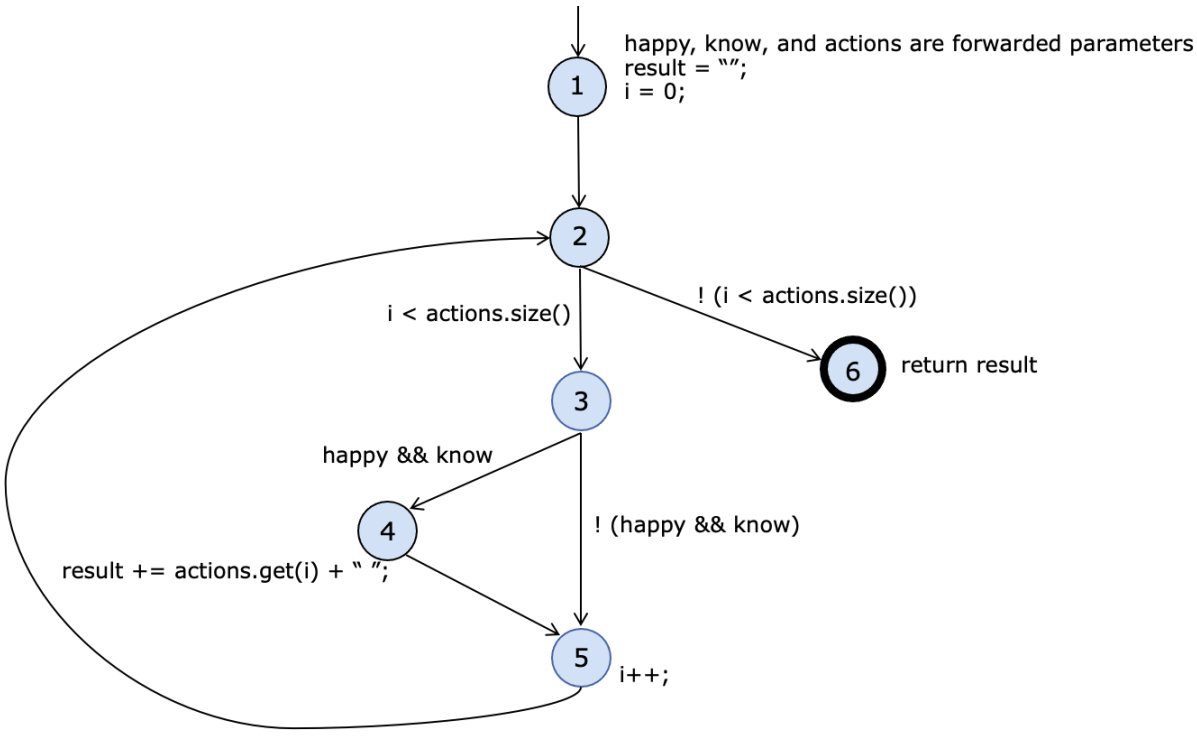


Part 3: For-loop and If-statement

Consider the following Java method

```
public static String IfYouAreHappy_v3(boolean happy, boolean know,
                                      List<String> actions)
{
   String result = "";

   for (int i=0; i<actions.size(); i++)
   {
      if (happy && know)
         result += actions.get(i) + " ";
   }

   return result;
}
```

Draw a **Control Flow Graph (CFG)** for the `IfYouAreHappy_v3()` method. Annotate **all** information (i.e., source code)



Part 4: Reflect on the graphs

Consider the graphs from parts 1, 2, and 3. Reflect on how the graphs are generated for the different program structures and how the graphs represent the execution flows of the programs.

1. How do you decide on the code associated with nodes?

2. How do you decide on the code associated with edges?

3. How do you decide on the initial nodes and the final nodes?

4. How do you avoid an infinite loop?

5. *[Thought question]* What might the test paths look like?

[Optional] Additional practice (do at least one of your graphs) – (Part 3: V3)

1. Apply **Node Coverage (NC)** to design tests

Test requirements	Test paths	Test cases (input values and expected output)
{1, 2, 3, 4, 5, 6}	[1,2,3,4,5,2,6]	Input: happy=true, know=true, actions[“Clap”] Expected output: “Clap ”

2. Apply **Edge Coverage** (EC) to design tests

Test requirements	Test paths	Test cases (input values and expected output)
{ (1,2), (2,3), (2,6), (3,4), (3,5), (4,5), (5,2) }	[1,2,3,4,5,2,6] [1,2,3,5,2,6]	Input: happy=true, know=true, actions[“Clap”] Expected output: “Clap ” Input: happy=true, know=false, actions[“Clap”] Expected output: “”

3. Apply **Edge-Pair Coverage** (EPC) to design tests

Test requirements	Test paths	Test cases (input values and expected output)
{ (1,2,3), (1,2,6), (2,3,4), (2,3,5), (3,4,5), (3,5,2), (4,5,2), (5,2,3), (5,2,6) }	[1,2,3,4,5,2,6] [1,2,3,5,2,3,5,2,6] [1,2,6]	Input: happy=true, know=true, actions[“Clap”] Expected output: “Clap ” Input: happy=true, know=false, actions[“Clap”, “Stomp”] Expected output: “” Input: happy=true, know=false, actions[] Expected output: “”

4. Apply **Prime Path Coverage** (PPC) to design tests

Test requirements	Test paths	Test cases (input values and expected output)
{ [1,2,3,4,5], [2,3,4,5,2], [3,4,5,2,3], [3,4,5,2,6], [4,5,2,3,4], [5,2,3,4,5], [1,2,3,5], [2,3,5,2], [3,5,2,3], [3,5,2,6], [5,2,3,5], [1,2,6] }	[1,2,3,4,5,2,3,5,2,6] [1,2,3,5,2,3,4,5,2,6] [1,2,3,4,5,2,3,4,5,2,6] [1,2,3,5,2,3,5,2,6] [1,2,6]	*infeasible* *infeasible* Input: happy=true, know=true, actions[“Clap”, “Stomp”] Expected output: “Clap Stomp” Input: happy=true, know=false, actions[“Clap”, “Stomp”] Expected output: “” Input: happy=true, know=false, actions[] Expected output: “”

Last updated 2025-10-03 10:12