Software Engineering 301:
Software Analysis and Design

Test case selection
and
test coverage

Agenda

- Overview
- Equivalence-based selection
- Boundary-based selection
- Limitations
- Some practice
- Path-based selection
- State-based selection & testing
- Selection & testing in OO

Remember …

- Exhaustive testing is impractical
- We need to choose our test cases
  - Random choices likely poor
- Let's look at some criteria for selecting test cases

Precondition

- You need to have a purpose
  - an idea of the problems you are looking for
- If you don't have this purpose, the problems will be hard to find
- The purpose ought to help you decide whether it is worthwhile to perform certain tests
  - It takes time to create a test case and run it, so think about whether it makes sense

Black-box vs. white-box test selection

- Broad categories of test selection, not specific techniques
- Black-box:
  - does not consider internal structure of entities
  - test cases based solely on documentation
- White-box:
  - does consider internal structure when selecting test cases
  - expected results still come from documentation
  - impractical if code doesn't exist yet

Techniques

- Black-box
  - Equivalence-based selection
  - Boundary-based selection
- White-box
  - Path-based selection
  - Branch-based selection
- Combinations
  - State-based selection & testing
  - Polymorphism-based selection

Black-box vs. white-box test selection

- In practice, both approaches used in combination
  - black-box analysis used to ensure conformance to interface
  - white-box analysis used to ensure that the code is more fully exercised

Goodness

- Bugs likely "lurk in the corners" where your test suite isn't exercising the program
- In practice, a test suite is considered "good" if it meets some sort of **coverage criterion**
  - This is some measure of how much of the program is being exercised by the test suite
  - There are many different coverage criteria
    - In lab, you will see instruction coverage and branch coverage
- Having good coverage relative to one criterion does not imply good coverage relative to another criterion

Agenda

- ~~Overview~~
- Equivalence-based selection
- Boundary-based selection
- Limitations
- Some practice
- Path-based selection
- State-based selection & testing
- Selection & testing in OO

Equivalence-based selection

- Typically, a black-box technique
- Group all the possible inputs into sets for which the output is identical or “similar”
  - i.e., partition the input into equivalence classes
- For each equivalence class, test one representative member
  - behaviour of other members assumed identical

Equivalence-based selection

- Not always a simple matter to determine useful equivalence classes
- Your test cases should have particular goals
  - these will help you to decide on equivalence classes

Example 1

```
/**
 * Returns the absolute value of the input
 */
public double abs(double num) { … }
```

- What are the equivalence classes for the inputs to abs()?

$$E_1 = \{x \mid x = \text{abs}(x), x \in \texttt{double}\} = [0, \infty)$$

$$E_2 = \{x \mid x = \text{-abs}(x), x \in \texttt{double}\} \setminus \{0\} = (-\infty, 0)$$

- Presumably, two test cases will suffice:
  - input 459.786, expected output 459.786; input -1000, expected output 1000
- Oops! What if num = NaN? For example, num = 0.0/0.0
  - other unusual values: +infinity, -infinity, +0, -0
  - ***WHEN*** would you need to worry about these strange possibilities, and when could you ignore them?

Example 2

```
/**
 * Returns 1 if the input is positive, -1 if it is
 * negative, or 0 if it is 0
 */
public int sgn(double num) { … }
```

- What are the equivalence classes for the inputs to sgn()?

  $E_1 = (-\infty, 0)$, $E_2 = \{0\}$, $E_3 = (0, \infty)$

- Three test cases will suffice for valid inputs:
  - e.g., input 10, expected output 1; input 0, expected output 0; input -100042.3698, expected output -1
- The "weird" values could be tested as well, if they matter in your context

Example 3

```
/**
 * Returns the name of the month, numbered 1-12
 */
public String monthName(int month) { … }
```

- Equivalence classes for the inputs?

$$E_i = \{i\}, \text{ for } 1 \le i \le 12$$

$$E_{13} = \{i \mid i < 1 \text{ or } i > 12\}$$

- Each month, plus one invalid input, must be tested individually
  - expected result for invalid input is undefined: this is a problem!
  - solutions: ignore it, revise the specification, talk to the stakeholders about it

- On the other hand, sometimes you might be satisfied knowing that a non-empty string is returned when the input is between 1 and 12
  - two equivalence classes: 1-12, and any other int
- Sometimes you might split up months on basis of how many days they have
  - four equivalence classes: 30 days, 31 days, 28 or 29 days, and invalid inputs

Example 4

- How about multiple inputs? …

```
/**
 * Returns the string for the date
 * e.g., Monday, 18 October
 * @param month 1-12
 * @param dayOfWeek 1-7
 * @param day 1-31 (day must be possible for month)
 * @exception OutOfRangeException If an invalid input or
 * combination of inputs is given
 */
public String
dateString(int dayOfWeek, int day, int month)
    throws OutOfRangeException { … }
```

- 8 equivalence classes for dayOfWeek
- How many for month?
  - 13 if we want to worry about translation into specific strings
  - 4 if we want to just worry about the check for the number of days
- How many for day?
  - you might think 32, but why bother differentiating all of them?
    - in case of valid input, input day = output day
  - only 2? either valid or invalid?
  - maybe 3: between 1 and maxDay(month), between maxDay(month) + 1 and 31, and outside {1, …, 31}
- Combinations? ≤ 13 x 8 x 3 = 312
  - not all can happen, but that is still a lot
  - this effect is known as **combinatorial explosion**; we'll see more about that in awhile

Agenda

- ~~Overview~~
- ~~Equivalence-based selection~~
- Boundary-based selection
- Limitations
- Some practice
- Path-based selection
- State-based selection & testing
- Selection & testing in OO

Boundary-based selection

- Rather than picking an arbitrary value from an equivalence class, pick ones at the “edges”
- Developers tend to make errors at boundaries of equivalence classes
  - e.g., using “<” instead of “<=”
- At each boundary, test the value on either side of that boundary

Example 1

```
/**
 * Returns the name of the month, numbered 1-12
 */
public String monthName(int month) { … }
```

- We should definitely test 0, 1, 12, and 13
- Whether or not you test all values 1-12 depends on whether the purpose of the test is to check the value of the returned string

Example 2

```
/**
 * Returns 1 if the input is positive, -1 if it is
 * negative, or 0 if it is 0
 */
public int sgn(double num) { … }
```

- What's the largest negative double? or the smallest positive double?
  "-1.0 E-1022 D" and "1.0 E-1022 D" respectively
- Boundary test cases would use these two numbers plus 0 (and maybe the "weird" values, if they matter in your context)

Combining boundary- and equivalence-based selection

- Equivalence-based selection by itself just says that some range of values can be treated equivalently
- Boundary-based selection by itself just says to test values “at boundaries”
- Combining them, we can determine the values to test at the boundaries of equivalence classes

Agenda

- ~~Overview~~
- ~~Equivalence-based selection~~
- ~~Boundary-based selection~~
- Limitations
- Some practice
- Path-based selection
- State-based selection & testing
- Selection & testing in OO

Limitations of equivalence- and boundary-based selection

- Combinations of choices for each input can lead to large number of test cases
  - "combinatorial explosion"
  - lots of test cases will tend either to not be done or not done well
- Some boundaries and equivalence classes may not be apparent from examination of interface and specification
  - white-box methods can help with this

Agenda

- ~~Overview~~
- ~~Equivalence-based selection~~
- ~~Boundary-based selection~~
- ~~Limitations~~
- Some practice
- Path-based selection
- State-based selection & testing
- Selection & testing in OO

Exercise

- Consider black-box testing this:

```
/**
  * Loads an image with the given name.
  */
public Image loadImage(String filename) {
      if (fMap.containsKey(filename)) {
              return (Image) fMap.get(filename);
      }
      Image image = loadImageResource(filename);
      if (image != null) {
              fMap.put(filename, image);
      }
      return image;
}
```

- What would the equivalence classes be? What would the boundary test cases be?

Agenda

- ~~Overview~~
- ~~Equivalence-based selection~~
- ~~Boundary-based selection~~
- ~~Limitations~~
- ~~Some practice~~
- Path-based selection
- State-based selection & testing
- Selection & testing in OO

Path-based selection

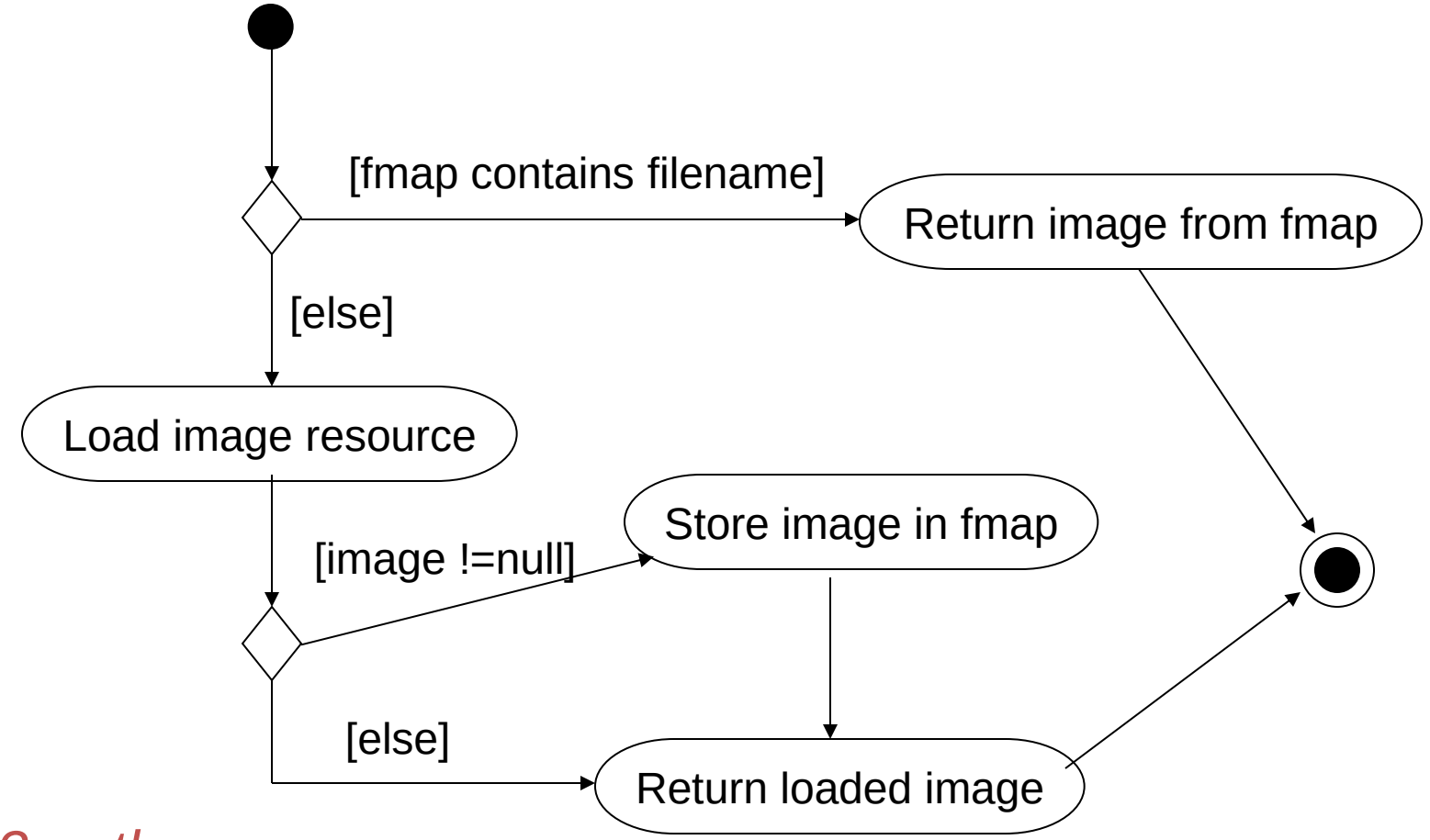
- White-box technique
- Attempts to ensure that every statement is executed by at least one test: *path coverage*
- "Logic errors and incorrect assumptions are inversely proportional to a path's execution probability" [anonymous]
  - This means: It's likely that untested paths will contain some faults

Example 1

- Usually discussed with respect to activity diagram
  - coverage involves traversing each transition at least once

```
/**
 * Loads an image with the given name.
 */
public Image loadImage(String filename) {
      if (fMap.containsKey(filename)) {
            return (Image) fMap.get(filename);
      }
      Image image = loadImageResource(filename);
      if (image != null) {
            fMap.put(filename, image);
      }
      return image;
}
```

Activity diagram



3 paths

Cyclomatic complexity

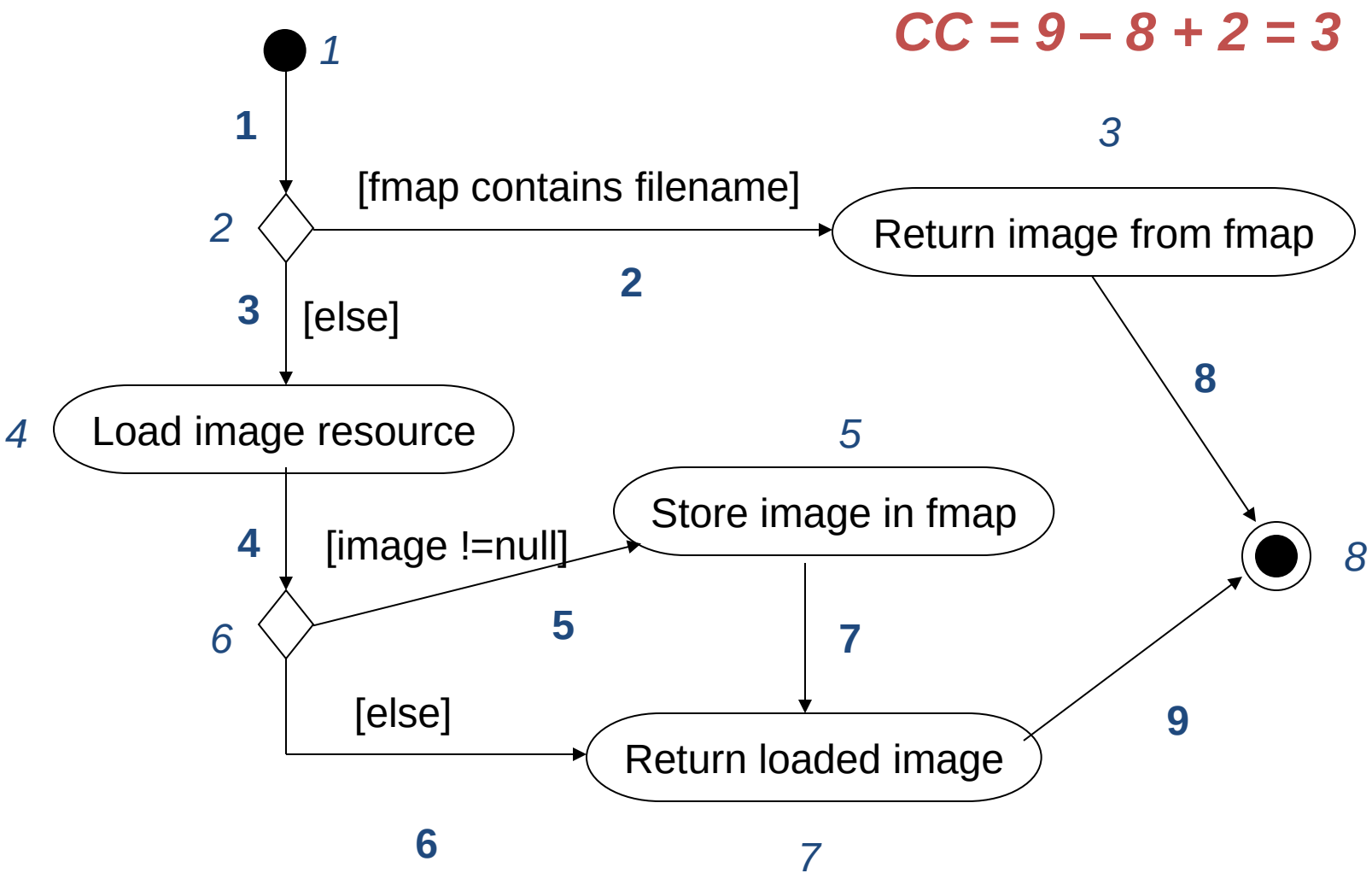
- Minimum number of tests necessary for path coverage:

  cyclomatic complexity =

  # transitions - # nodes + 2

  a *node* is any pseudostate (start or end), action state, branch or join (diamonds)

Example



Example 2

- Does it work with loops?

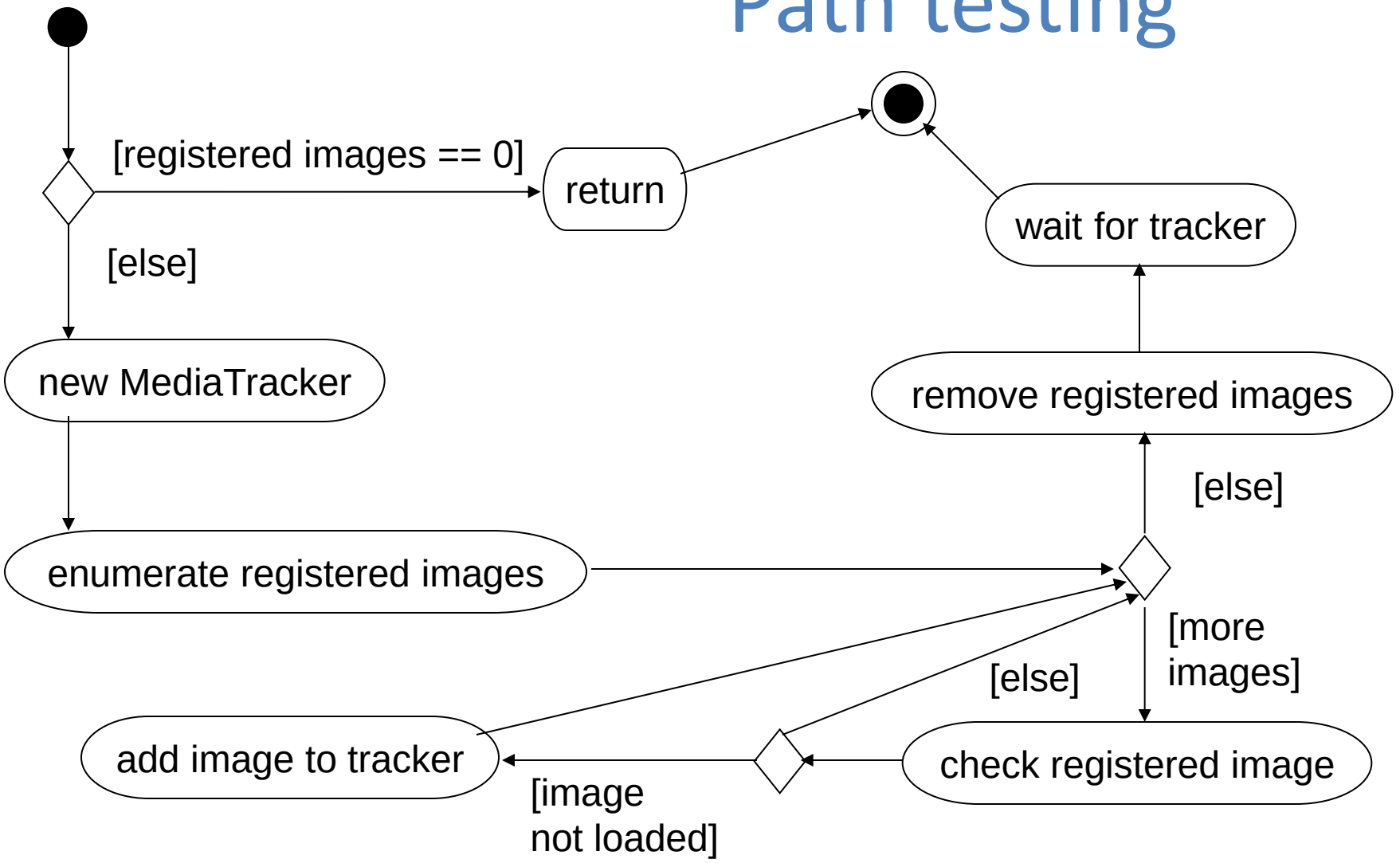
```
public void loadRegisteredImages(Component component) {
        if (fRegisteredImages.size() == 0)  return;

        MediaTracker tracker = new MediaTracker(component);
        Enumeration k = fRegisteredImages.elements();
        while (k.hasMoreElements()) {
                String fileName = (String) k.nextElement();
                if (basicGetImage(fileName) == null) {
                        tracker.addImage(loadImage(fileName), ID);
                }
        }
        fRegisteredImages.removeAllElements();

        try { tracker.waitForAll(); } catch(Exception e) {  }
  }
```

Path testing

CC = 14 – 12 + 2 = 4



Troubles with cyclomatic complexity for test coverage

```
String s = …;
int i = 6;

switch(s) {
    case "One": i = 1; break;
    case "Two": i = 2; break;
    case "Three": i = 3; break;
    case "Four": i = 4; break;
    case "Five": i = 5; break;
}
```

```
String[] arr = {"One", "Two",
    "Three", "Four", "Five"};
String s = …;
int i = 0;

do {
    i++;
} while(arr[i] != s);
```

- CC = 5 and 1 respectively, but same functionality => shouldn't it be tested the same?

Path-based selection

- Summary:
  - exercising some transitions might be difficult (or even impossible)
  - cyclomatic complexity may count paths that are infeasible or it may treat paths through loops as independent that can be covered by a single case
  - careful set-up of objects for input can still be needed
- Path coverage does not imply sufficient testing
- Lack of path coverage *does* imply that some code has not been tested

Agenda

- ~~Overview~~
- ~~Equivalence-based selection~~
- ~~Boundary-based selection~~
- ~~Limitations~~
- ~~Some practice~~
- ~~Path-based selection~~
- State-based selection & testing
- Selection & testing in OO

State-based testing

- Most techniques consider inputs and outputs
- Instead, consider the expected state that should arise from an initial state and some event
  - compare this to the actual state that results
- White-box technique?
  - probably but not necessarily
- Arbitrarily many events might cause a particular transition
  - use equivalence to pick representative ones

State-based testing

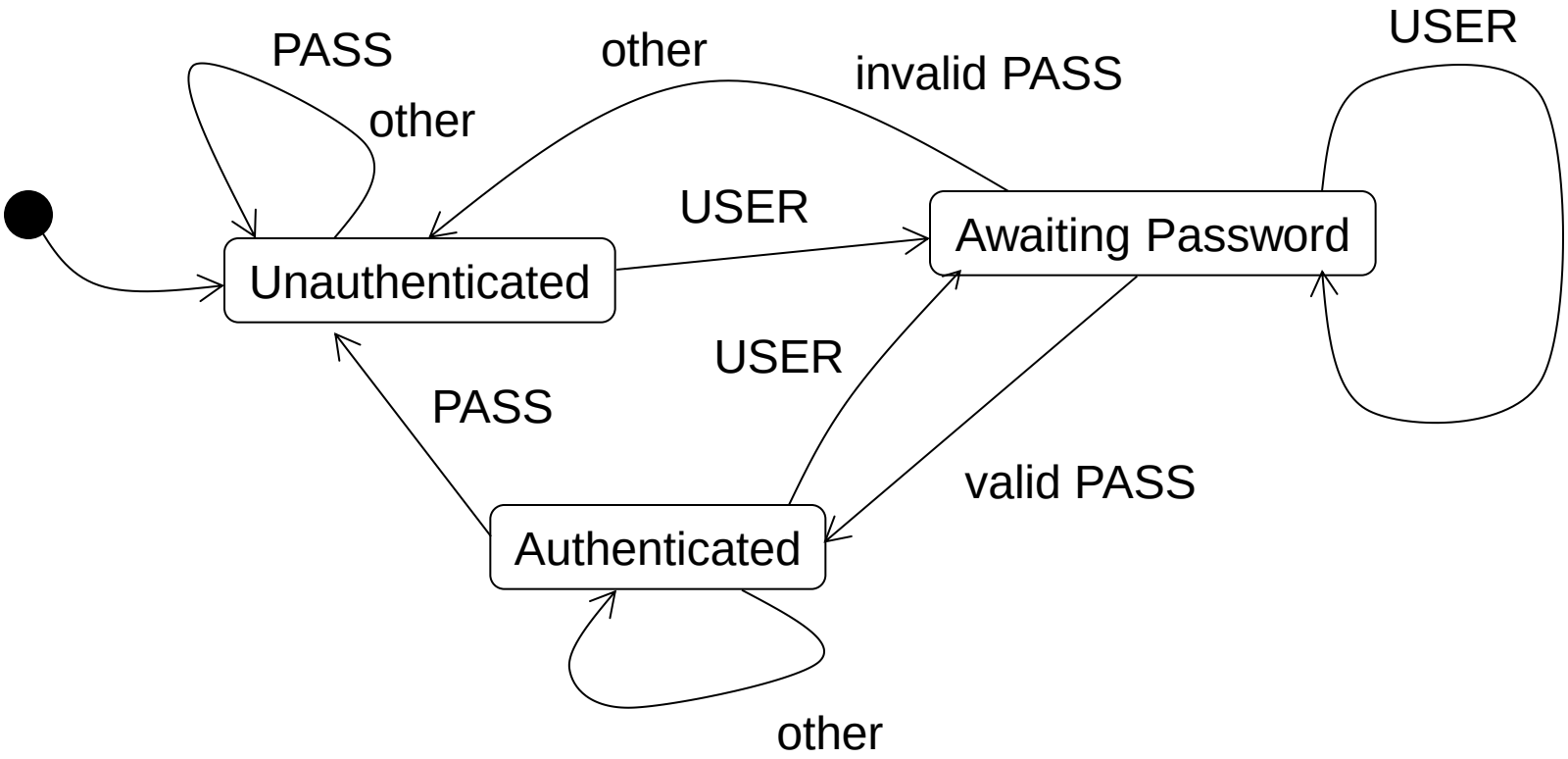
- Problem: How can we observe states?
  - states often represented by private fields
- Indirectly:
  - by observing outputs, some states can be implied
- Directly:
  - some state is public or can be reached through getters
  - we can insert “probes” to determine what values private fields hold

Example: File Transfer Protocol

- FTP defines
  - set of commands to be sent from client to server
    - USER, PASS, STOR, RETR, QUIT, …
  - set of possible responses from server
  - ordering constraints on some commands
    - e.g., to login, USER must be immediately followed by valid PASS
    - e.g., STOR can only succeed if logged in

State-based testing

- File Transfer Protocol (FTP) server



State-based testing

- Indirect approach for testing FTP server:
  - Issue sequence of commands
  - Final command depends on whether authenticated or unauthenticated (e.g., STOR)
  - Select test cases for transition coverage

Exercise

- Determine a set of test cases for transition coverage to indirectly test an FTP server on the basis of its authentication state
  - Use STOR (store a file) as final probe of state
    - "Not logged in" will be sent to the client if the state of the session is Awaiting password or Unauthenticated

Agenda

- ~~Overview~~
- ~~Equivalence-based selection~~
- ~~Boundary-based selection~~
- ~~Limitations~~
- ~~Some practice~~
- ~~Path-based selection~~
- ~~State-based selection & testing~~
- Selection & testing in OO

Object-orientation and equivalence

- Given a formal parameter of a non-primitive type, what are the equivalence classes?
- Clearly null and non-null are significantly different
- Consider an array type of fixed maximum size
  - possible equivalence classes of these objects:
    - empty array
    - full array
    - partially filled array
- More generally, will depend on semantics of objects involved and the context of interest

Polymorphism-based selection

- Parameter of type Collection could hold an object of several concrete types:
  - Vector, Stack, HashSet, LinkedList, …
- Behaviour of method accepting Collection object could be quite different
  - different paths can occur depending on which type is passed as an argument
  - we really ought to test different types of objects separately

Example

```
/**
 * Loads an image with the given name.
 */
 public Image loadImage(String filename) {
      if (fMap.containsKey(filename)) {
              return (Image) fMap.get(filename);
      }
      Image image = loadImageResource(filename);
      if (image != null) {
              fMap.put(filename, image);
      }
      return image;
 }
```

get() returns an Object

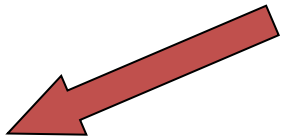
What if get() doesn't return an Image? ClassCastException

Example

```
public void loadRegisteredImages(Component component) {
        if (fRegisteredImages.size() == 0)  return;

        MediaTracker tracker = new MediaTracker(component);
        Enumeration k = fRegisteredImages.elements();
        while (k.hasMoreElements()) {
                String fileName = (String) k.nextElement();
                if (basicGetImage(fileName) == null) {
                        tracker.addImage(loadImage(fileName), ID);
                }
        }
        fRegisteredImages.removeAllElements();

        try { tracker.waitForAll(); } catch(Exception e) {  }
    }
```



MediaTracker might behave differently depending on Component subtype

Polymorphism-based selection

- Some authors claim that you should “expand the source” to include the implicit if-then-else’s buried in polymorphism
  - what if you don’t know all the possible subtypes?
  - a new subtype could break an existing class
  - also, somewhat impractical as code will expand significantly
  - often, sufficient to treat subclasses as equivalence classes

Polymorphism-based selection

- But this is implicitly what you would be doing if you stuck something other than an Image in the fmap
- fmap could be a variety of types; worthwhile to expand?

Next time

- Midterm review