Lecture 25
Software Testing

Overview

Aims of Testing

Basic Concepts

Black box Testing

White box Testing

Categorising Defects

Testing Integration Strategies

Exercises

Workshop: Developing Test Harnesses with JUnit

Workshop: Agile Practices for Testing

- Test First Programming
- Pair Programming
- Test Cases
- Implementation

Recommended reading

Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engineering*. McGraw Hill, Shoppenhangers Road, Maidenhead, Berkshire, SL62QL, second edition, 2005.
Practical Software Development using UML and Java. chapter 10

Aims of testing

Defect detection

Failure documentation

Acceptance demonstration

Trade-offs in testing

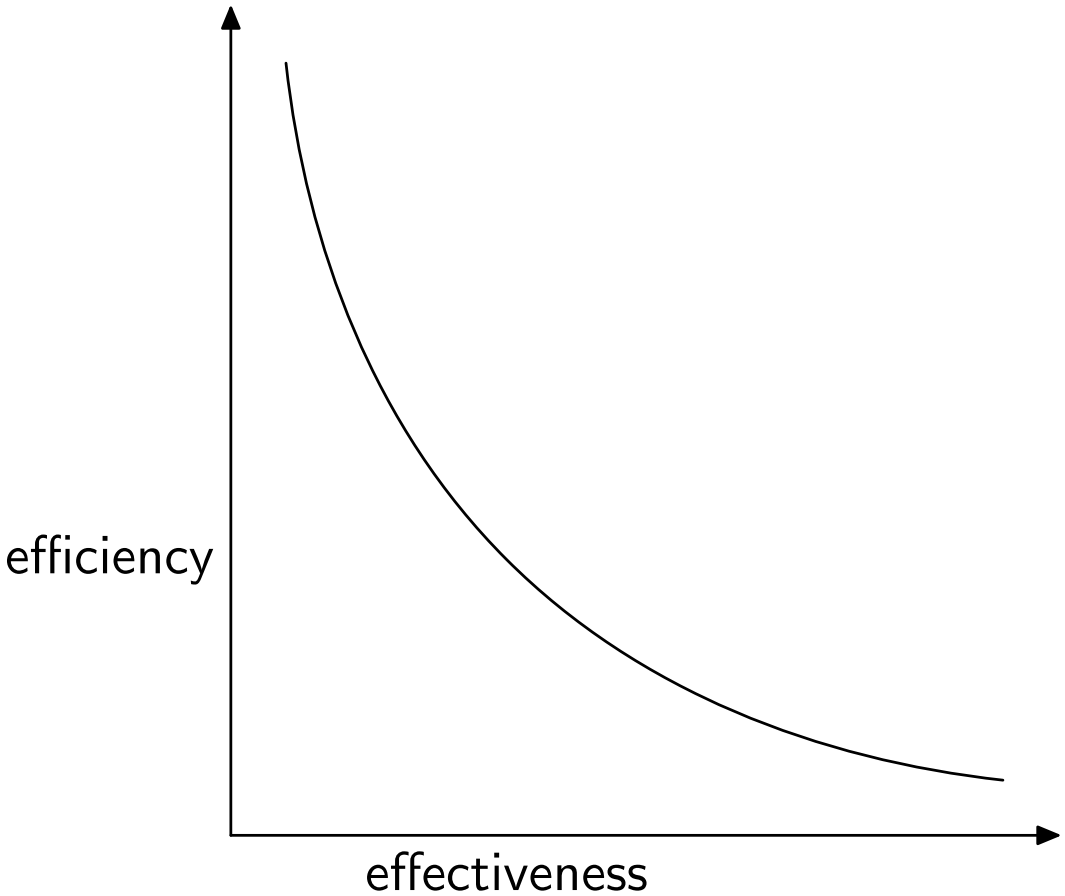

Figure: effectiveness vs. efficiency in testing

Key definitions in testing

Interfaces :

- inputs,
- outputs, and
- non-functional characteristics;

failures ;

Defects ; and

Slips .

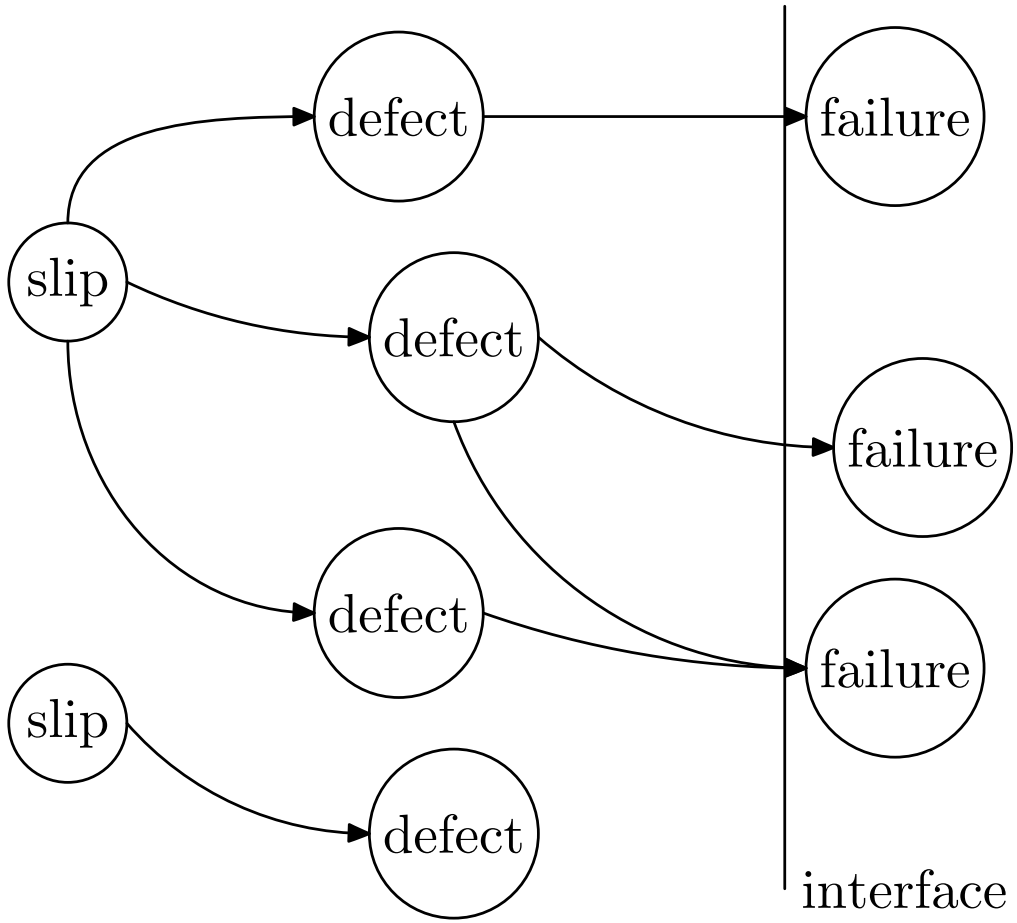

Figure: slip, defect, failure model

Testing the interface, not the implementation

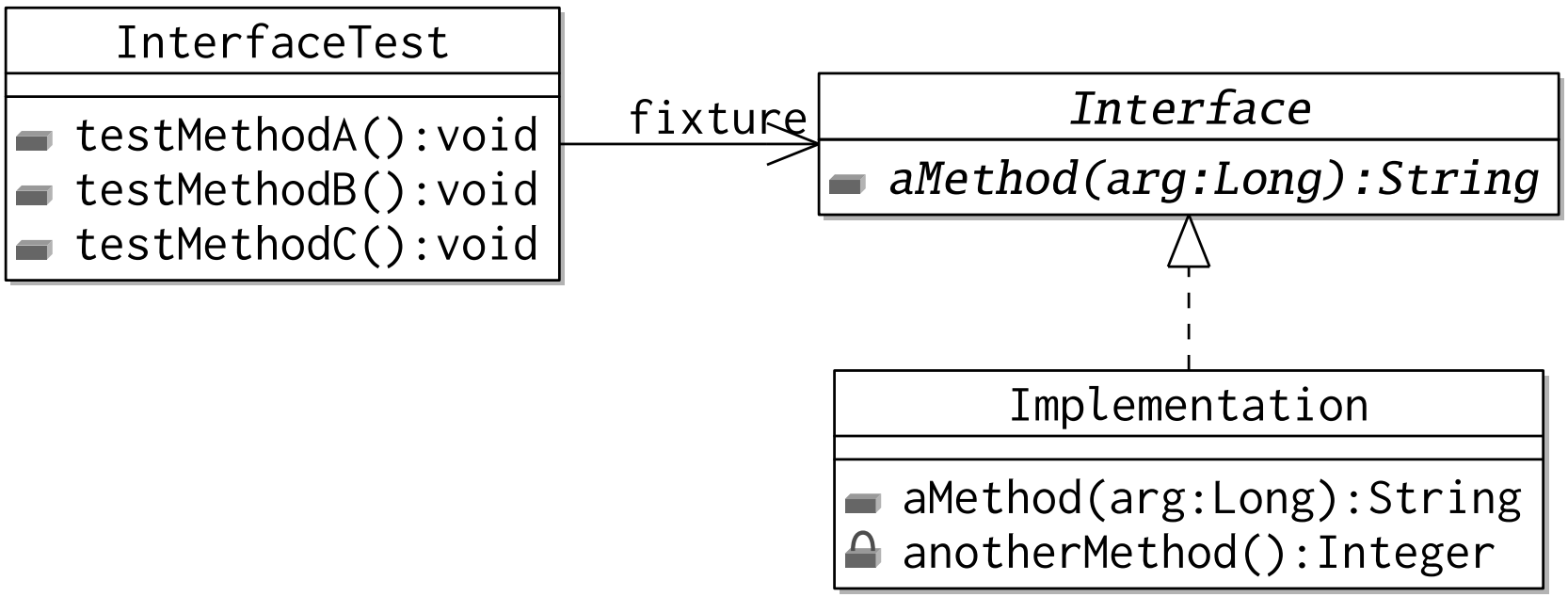

Figure: arranging tests, interfaces and implementations

The subjective nature of testing

The testing process

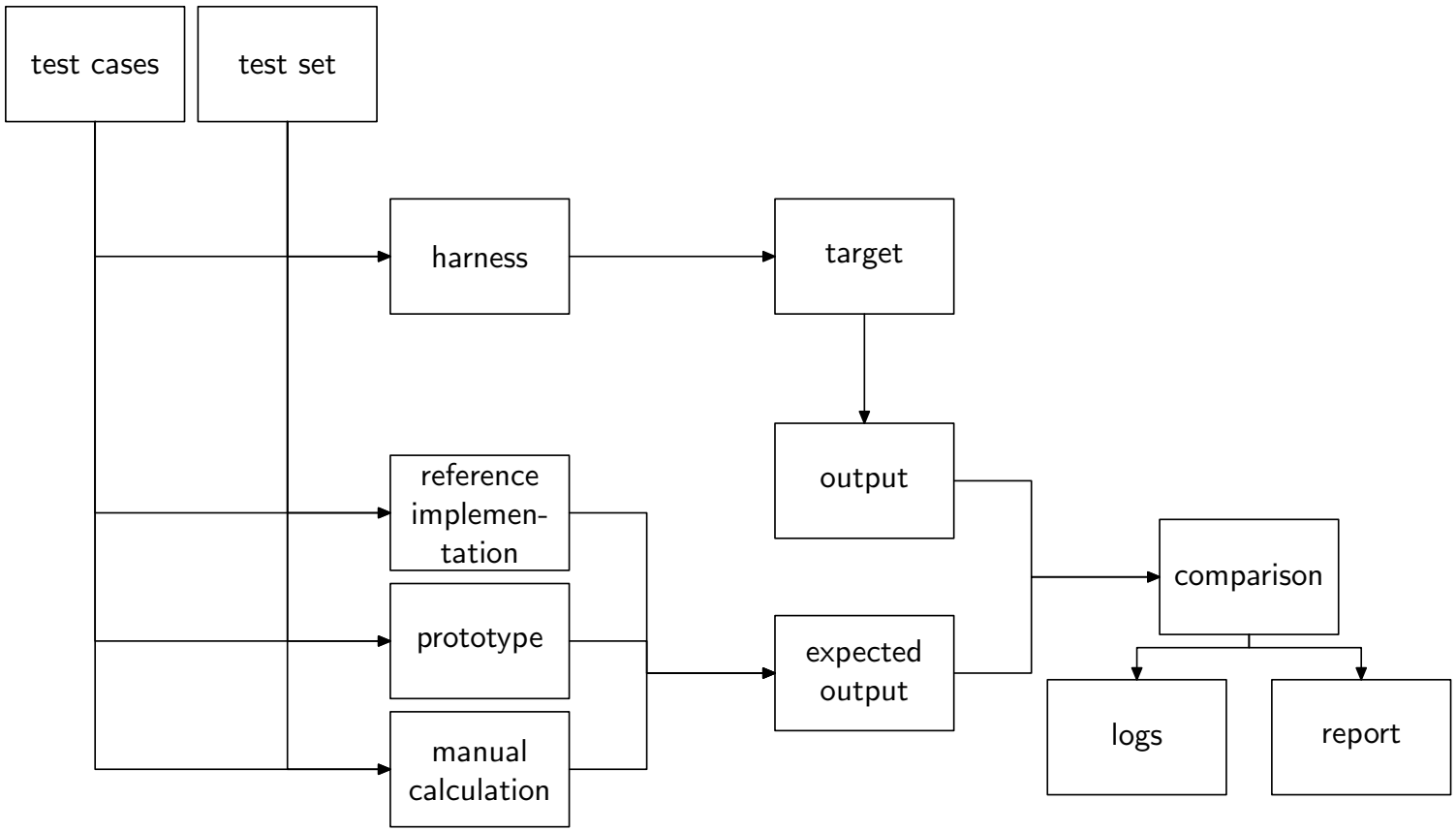

Figure: the software testing process

Testing process terminology

targets];

Test cases ;

Test sets ;

Test harnesses ;

Test oracle ; and

Test report .

Test oracles

Reference implementations ;

Prototype ; and

Manual calculation .

Defining tests

- $\{$black, opaque$\}$ box; and
- $\{$white, clear, glass$\}$ box.

Black box testing

types of specification:

- formal specifications stating pre and post conditions and invariants;
- syntactic formal specification with informal semantic documentation; or
- informal description only.

search for:

- invalid parameters or ordering;
- out of range arguments to parameters;
- extreme (legal) arguments; and
- invalid sequences of operations.

Example Specification

```
/**
 * Converts a String of binary symbols (0,1)* in twos
 * complement format, into the equivalent integer value.
 *
 * @param binString
 *            the string of binary symbols (0,1)*
 * @param isTwosComplement
 *            specifies whether the string of binary symbols
 *            is to be treated as a twos complement signed
 *            integer or as an absolute value
 * @returns a decimal representation of the binary input.
 * @throws NumberFormatException
 *             if the input string contains any character
 *             other than 0 or 1
 */
public int convert(String binString,
  boolean isTwosComplement)
```

Equivalence partitions

examples in Java, there are:

- 2^{32} **int** values
- 2^{16} **char** values
- $(2^{16})^{2^{31}} = (2^{16})^{2147483648} = 2^{34359738368}$ `String` values

equivalence class testing:

- at and near the class boundary (if ordered); and
- randomly within the class.

Example Equivalence partitions: calculating income tax

```
public int calculateTax(double income);
```

- income is not taxed up to £10000.00 inclusive;
- income between £10000.01 and £24999.55 inclusive is taxed at 20%; and
- income over £24999.55 is taxed at 40%.

Example Equivalence partitions: leap years

```
public boolean isLeapYear(int year);
```

- all years divisible by 4 are leap years; except
- all years divisible by 100 are not leap years; except
- all years divisible by 400 are leap years.

Combining and prioritising equivalence partitions

Example:

```
public abstract Double convertTemperature(
  Metric from, Metric to, Double temperature);
```

conversion	**equivalence class boundaries**
Kelvin $\rightarrow$ Celcius	$-\infty$, 0.0, 273.15, ∞
Celcius $\rightarrow$ Kelvin	$-\infty$, -273.15,0.0, ∞
Kelvin $\rightarrow$ Fahrenheit	$-\infty$,0.0, 255.37,∞
Fahrenheit $\rightarrow$ Kelvin	$-\infty$, -459.67, 0.0, ∞
Fahrenheit $\rightarrow$ Celcius	$-\infty$, -459.67, 0.0, 32.0, ∞
Celcius $\rightarrow$ Fahrenheit	$-\infty$,-273.15, -17.7, 0, ∞

Table: equivalence class boundaries for temperature conversion

Sequencing operations

- Primitive operation arity
- Floating point precision problems
- out of memory defects
- Shared memory defects
- Timing defects

example

```
int x = (1+2/3*4-5)%6+3/2+1;
```

White box testing

purpose:

- documentation of defects identified during an inspection; and
- prevention of defect introduction during system evolution.

steps:

1. identify the steps, conditions and transitions of the algorithm;
2. map flows through algorithms as a flow chart or activity diagram;
3. label each transition of the algorithm;
4. express the desired test case as a sequence of transitions through the algorithm;
5. determine the inputs to the algorithm that will result in the paths being executed; and
6. estimate code coverage of test suite in terms of *paths*, *edges* or *nodes*.

```
for (char c: binaryString.toCharArray())
 if (c != '0' && c != '1')
  throw new NumberFormatException(
    "Only binary symbols [0,1] permitted");
int result = 0;
int twos = 1;
StringBuffer buf =
 new StringBuffer(binaryString).reverse();
String revBinString = buf.toString();
for (char c: revBinString.toCharArray()){
 if (c == '1')
  result += twos;
 twos *= 2;
}
if (isTwosComplement){
 //apply two's complement rule
 char twosComplement =
  revBinString.charAt(revBinString.length()-1);

 if (twosComplement == '1') result = result - twos;
}
return result;
```

Activity diagrams

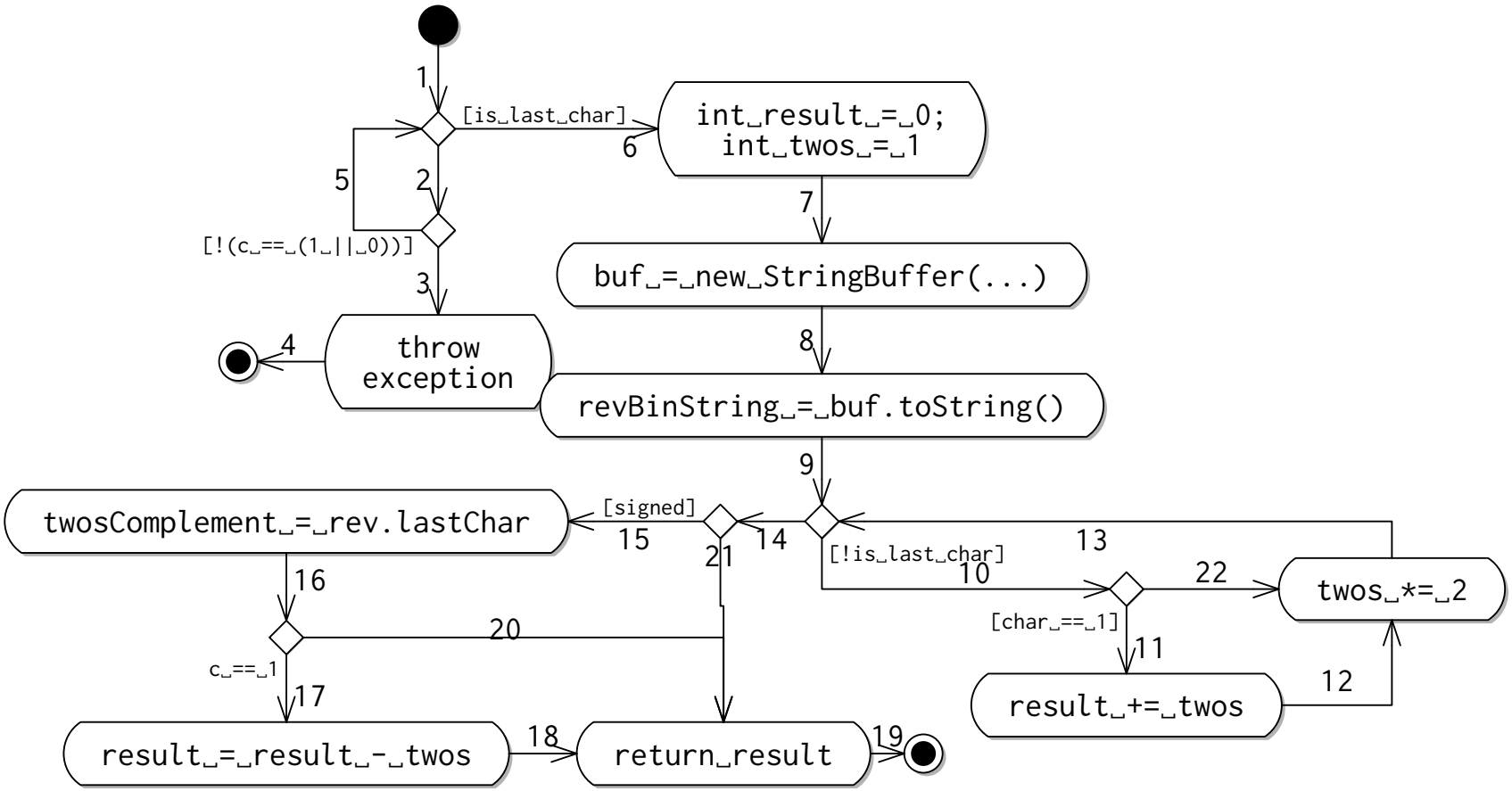

Figure: an implmentation of the BinaryToDecimal operation

Specifying test cases

In terms of the edges to be transitioned during execution:

- 1,2,3,4 (e.g. ”a”)
- 1,2,5,2,5,3,4 (e.g. ”0a”);
- 1,2,5,2,5,6,7,8,9,10,11,12,13,10,11,12,13,14,21 (???);
- ...

Determine inputs afterwards.

Estimating coverage: nodes, paths and edges

levels of rigour:

- nodes;
- edges;
- paths

where to stop?

- automatic test case generation; and
- automated measurement of coverage.

Categorising defects

by specification:

- functional
- non-functional

or by occurrence:

- deterministic
- non-deterministic

or by error [Lethbridge and Laganière, 2005]:

- control
- numerical
- timing and coordination
- external dependencies
- documentation

Control condition defects

Example:

> *The power supply must be off if the emergency stop has been pressed, or if the load exceeds 90% of capacity, except if the normal capacity limits are overridden, in which case, the load may not exceed 99% of capacity.*

```
if (powered
  && (capacity - load) / capacity > .9
  || (override &&
    (capacity - load) / capacity > .99 ||
    stopPressed))
 throw new PowerControlException();
```

exercise all combinations of input equivalence classes in the conditions

Control construct scope defects

example:

```
public List<String> controlDefect(
  List<String> input, int seed){

 List<String> results = new ArrayList<String>();
 results.addAll(input);

 Random r = new Random(seed);
 int shuffles = r.nextInt();

 for (int i = 0; i < shuffles; i++);
  Collections.shuffle(results);

 return results;
}
```

Infinite loops defects

example:

```
private double water = 0.0;
private double capacity = 100.0;

public void fillTheBath(double rate){
 while (water != capacity){
  water += rate;
 }
}

public void pullThePlug(){
 water = 0.0;
}
```

Off-by-one defects

example:

```
public int factorial(int i){

 int result = 1;
 int j = 1;
 while (j < i){
  result *= j;
  j++;
 }
 return result;
}
```

Pre-condition defects

example:

```
public class CarControl {

 private double max_speed;
 private double speed;

 public void accelerate(double rate, int duration){
  if (speed == max_speed) return;
  else speed += rate*duration;
 }
}
```

Null and (Non)-Singleton defects

example 1: the `BinaryToDecimal` application we have already seen.

```
"1"
"0"
""
```

example 2: `Collections.max()`

Precedence defects

example:

```
int z = 0;
int i = 1;
int y = (1+2/(z*4-5))%6+32/i;
System.out.println(y);
```

Memory overflow defects

examples:

```
int x = Integer.MAX_VALUE+1;
int y = Integer.MAX_VALUE*Integer.MAX_VALUE;
//etc.
```

Precision defects

```
Double op1 = 22.0;
Double op2 = 7.0;
Double val = op1/op2;

val = val +1;
val = val -1;
val = val*op2;

assertEquals(22.0, val,0.0);
```

Symbolically:

$$val = (op1/op2 + 1 - 1) * op2 = 22/7 + 1 - 1 * 7 = 22 * 7/7 = 22$$

output:

```
java.lang.AssertionError: expected:<22.0> but was
    :<21.999999999999996>
```

Documentation and External dependency defects

- absent;
- incorrect (e.g. out of date); or
- inconsistent.

- specification;
- implementation; or
- dependency descriptions.

More definitions

a test suite ;

a test integration strategy ; and

a test plan .

Unit testing and Stubs

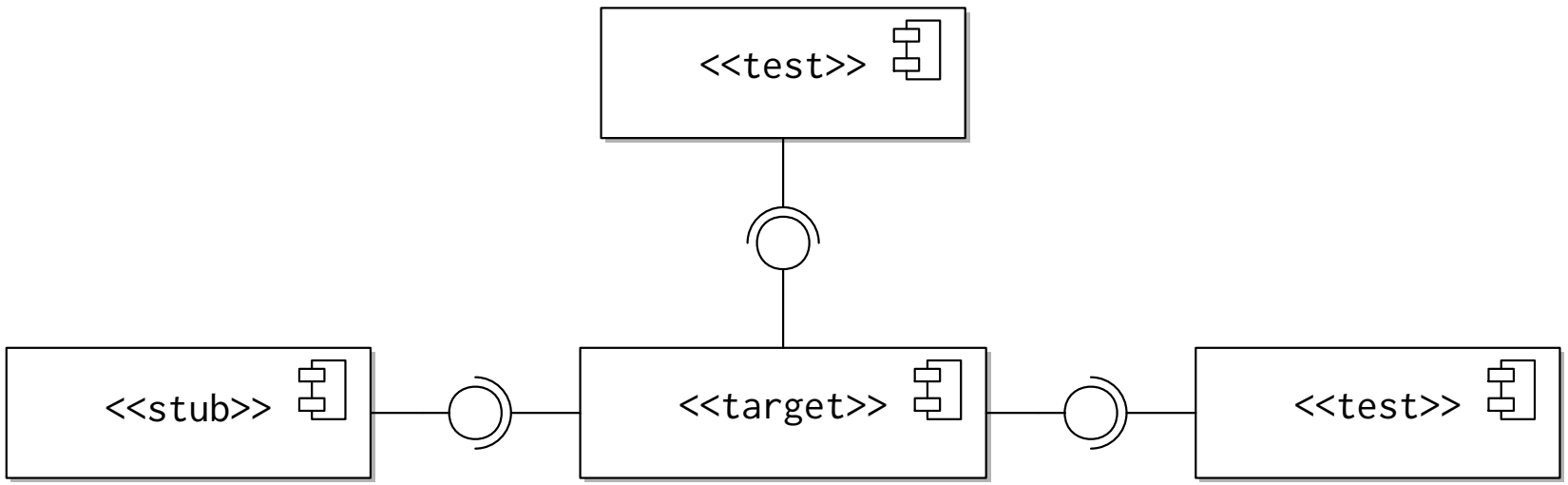

Figure: unit testing and stubs

Integration testing

guidelines:

- components with no internal application dependencies can be subjected to unit testing on their provided interfaces first. These are often at the boundary or bottom of a system architecture;
- optional components can be added later in the integration process;
- components that mainly provide presentational capability for data provided by other components can be tested using stubs.

common strategies:

top down ;
bottom up ;
middle out ;
outside in ; and
a mixture of all of the above .

Example integration

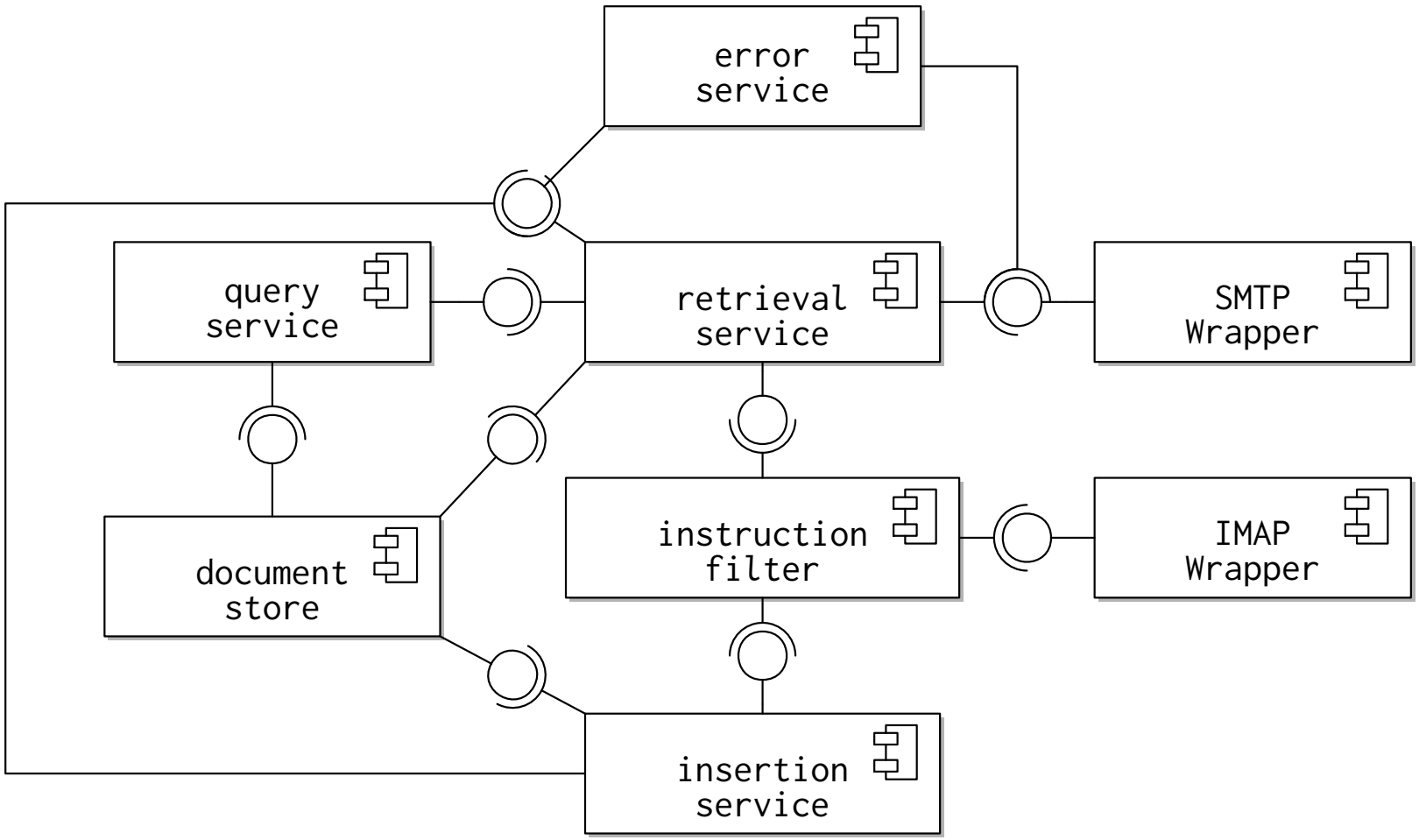

Figure: example integration sequence

Example integration

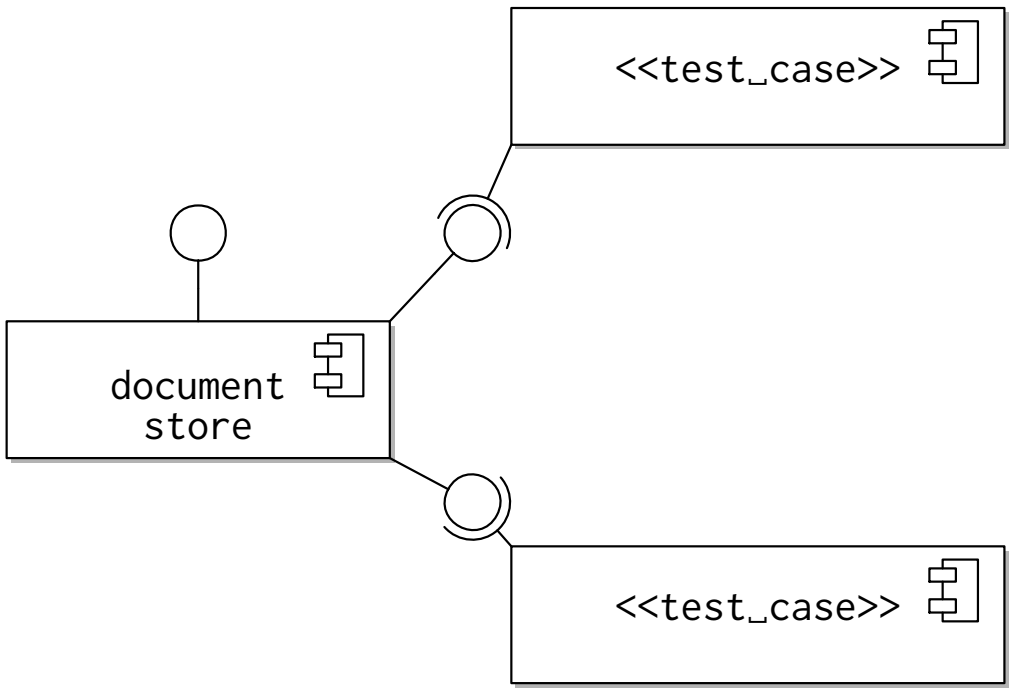

Figure: example integration sequence

Example integration

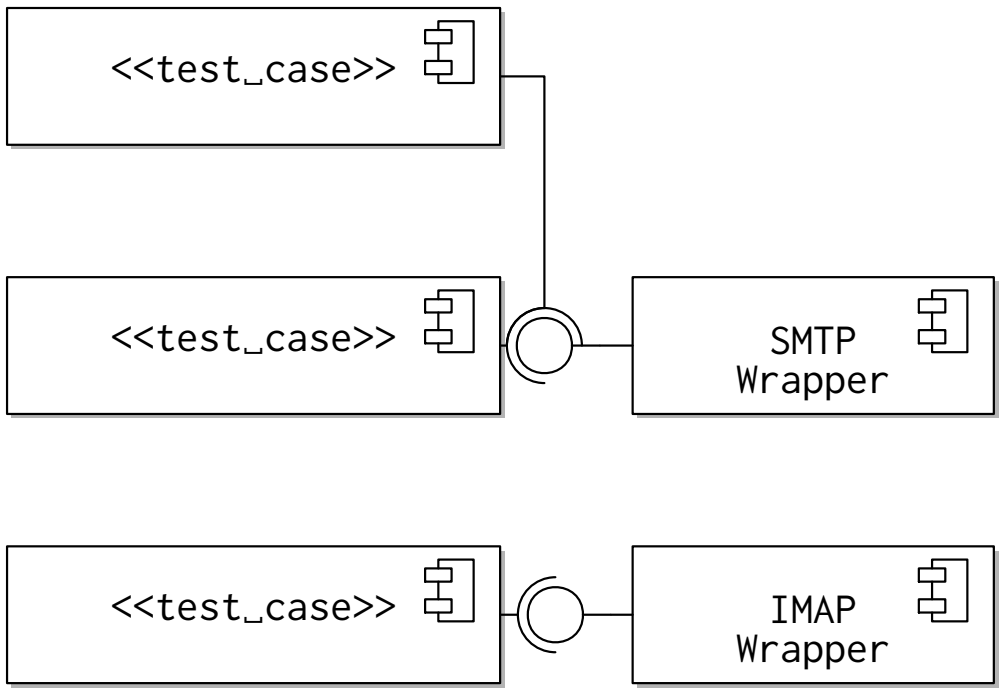

Figure: example integration sequence

Example integration

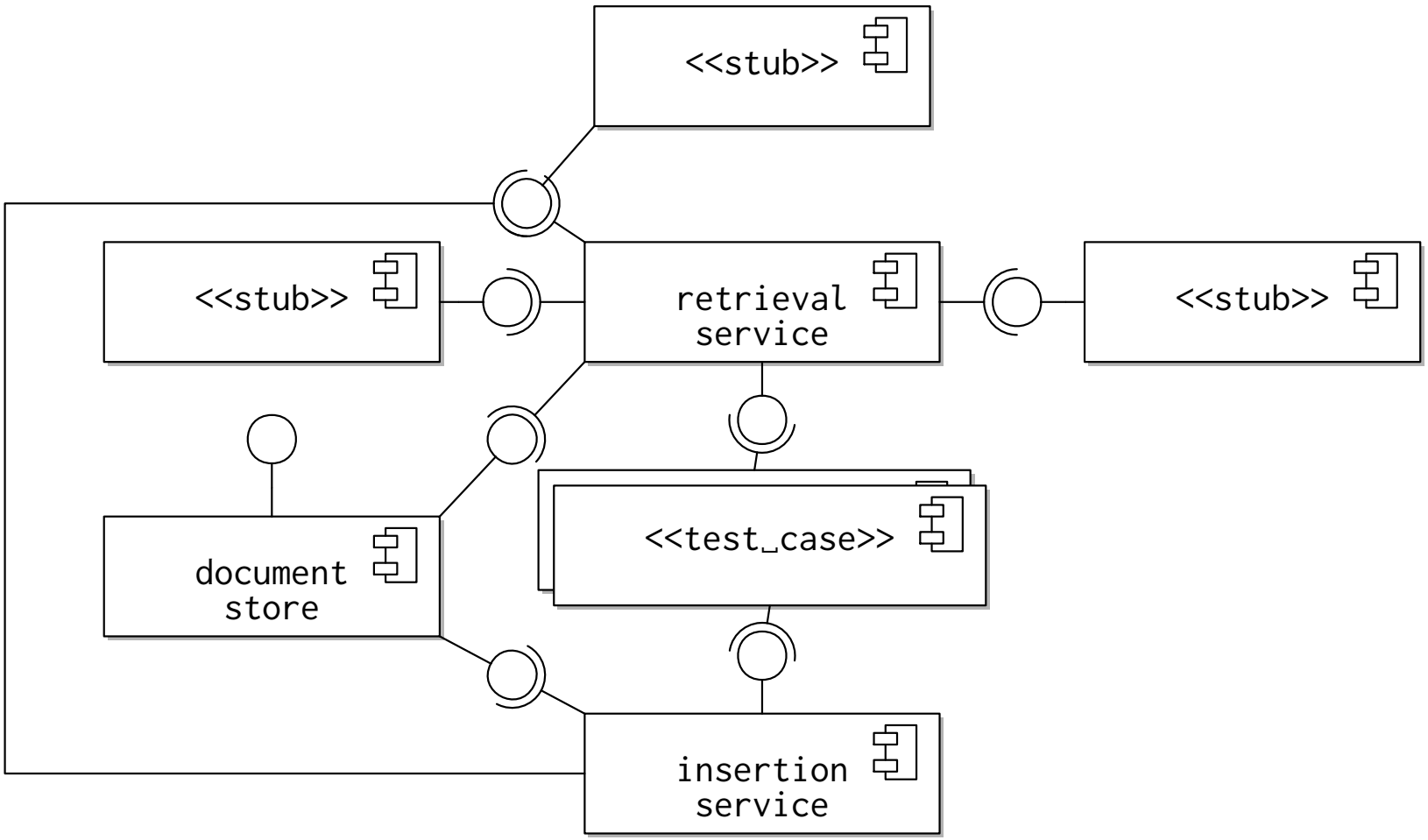

Figure: example integration sequence

Example integration

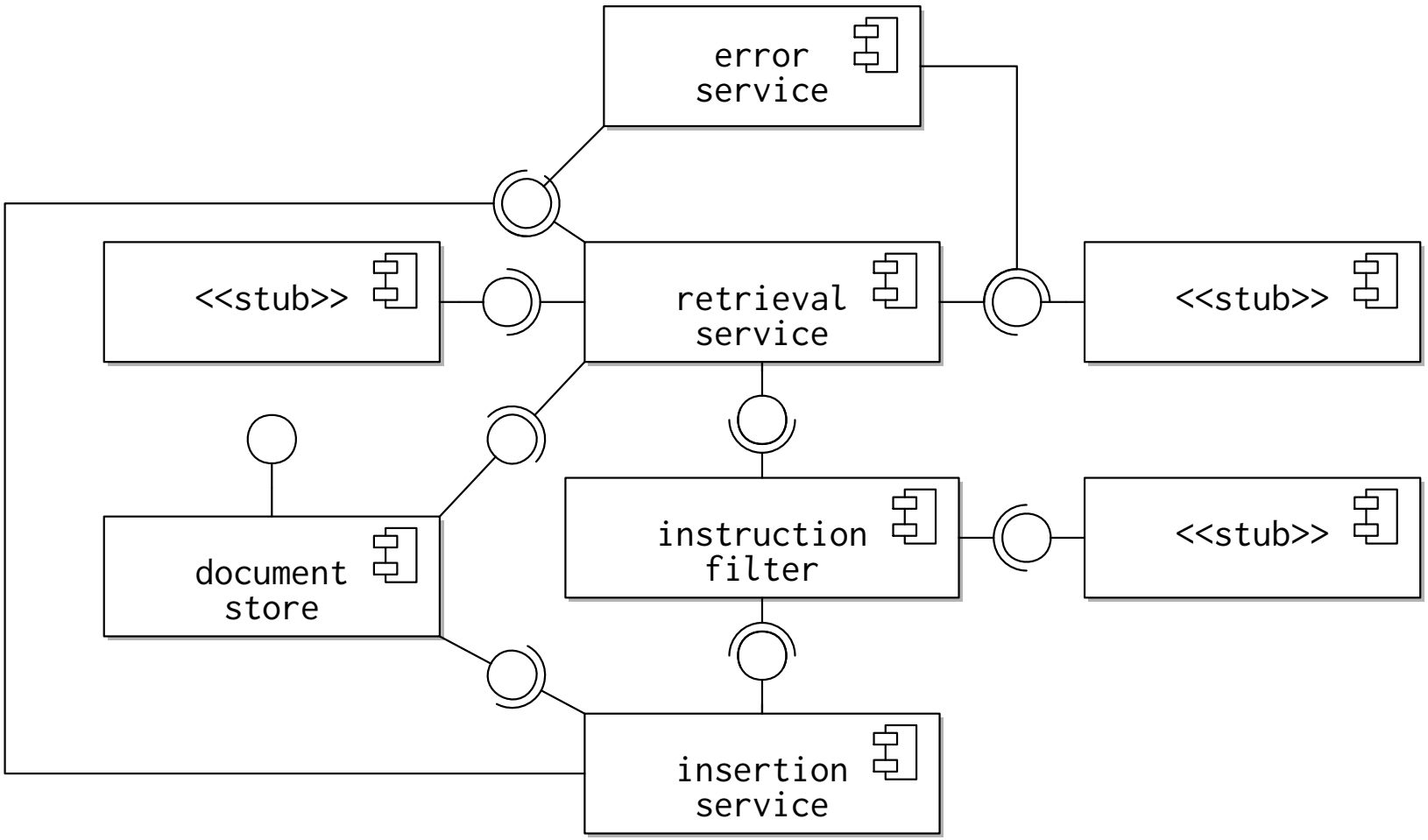

Figure: example integration sequence

Example integration

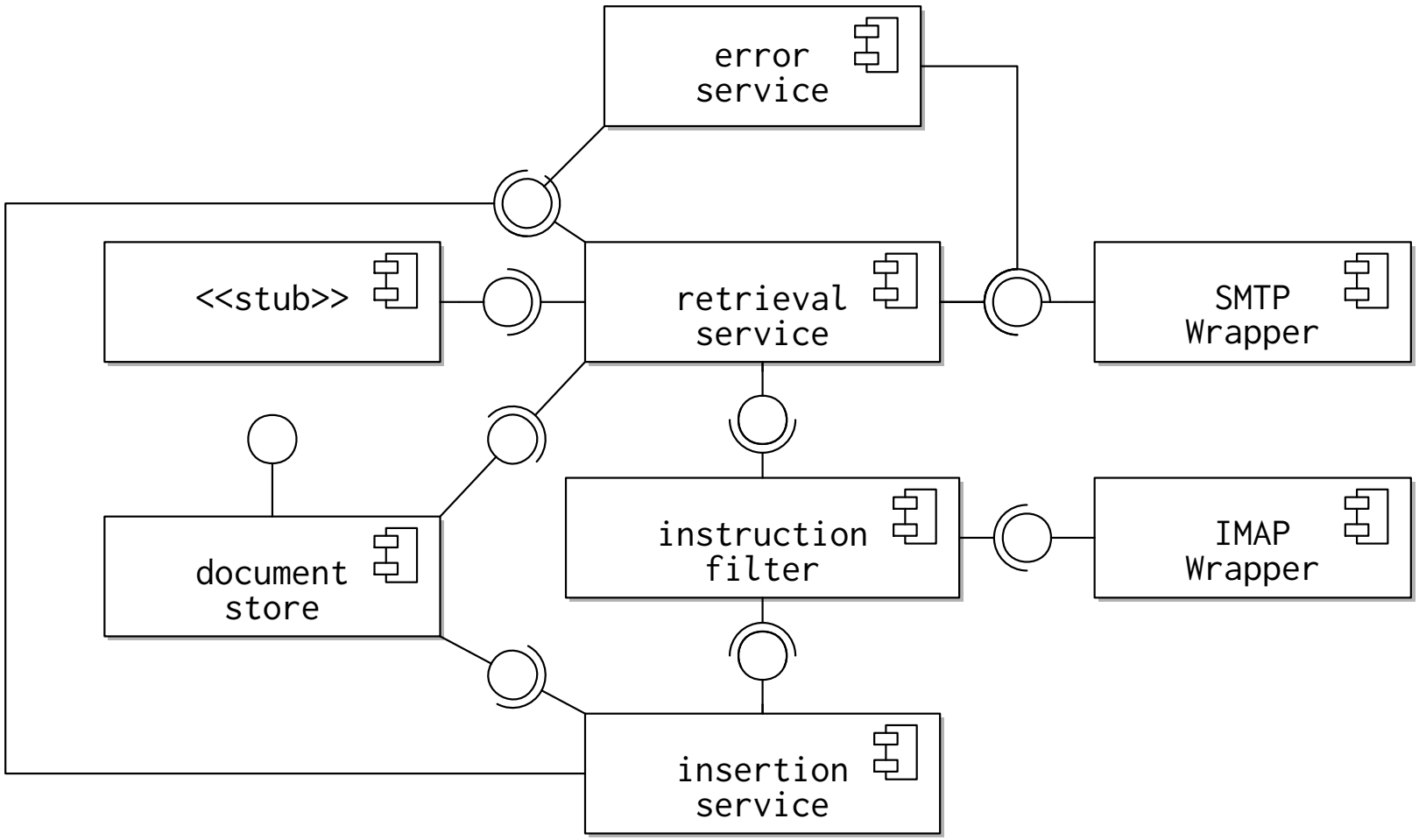

Figure: example integration sequence

Example integration

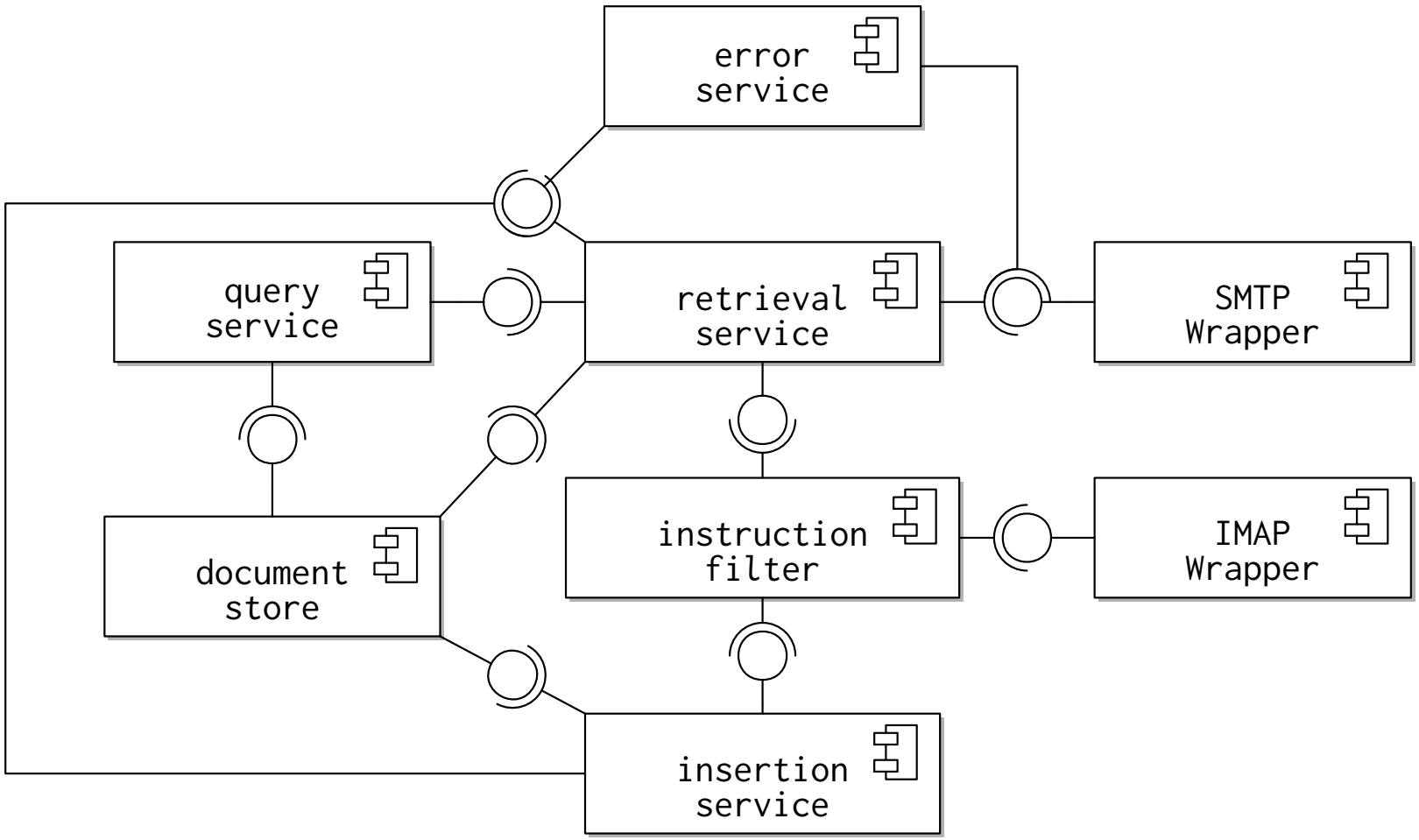

Figure: example integration sequence

The Test-Fix Cycle

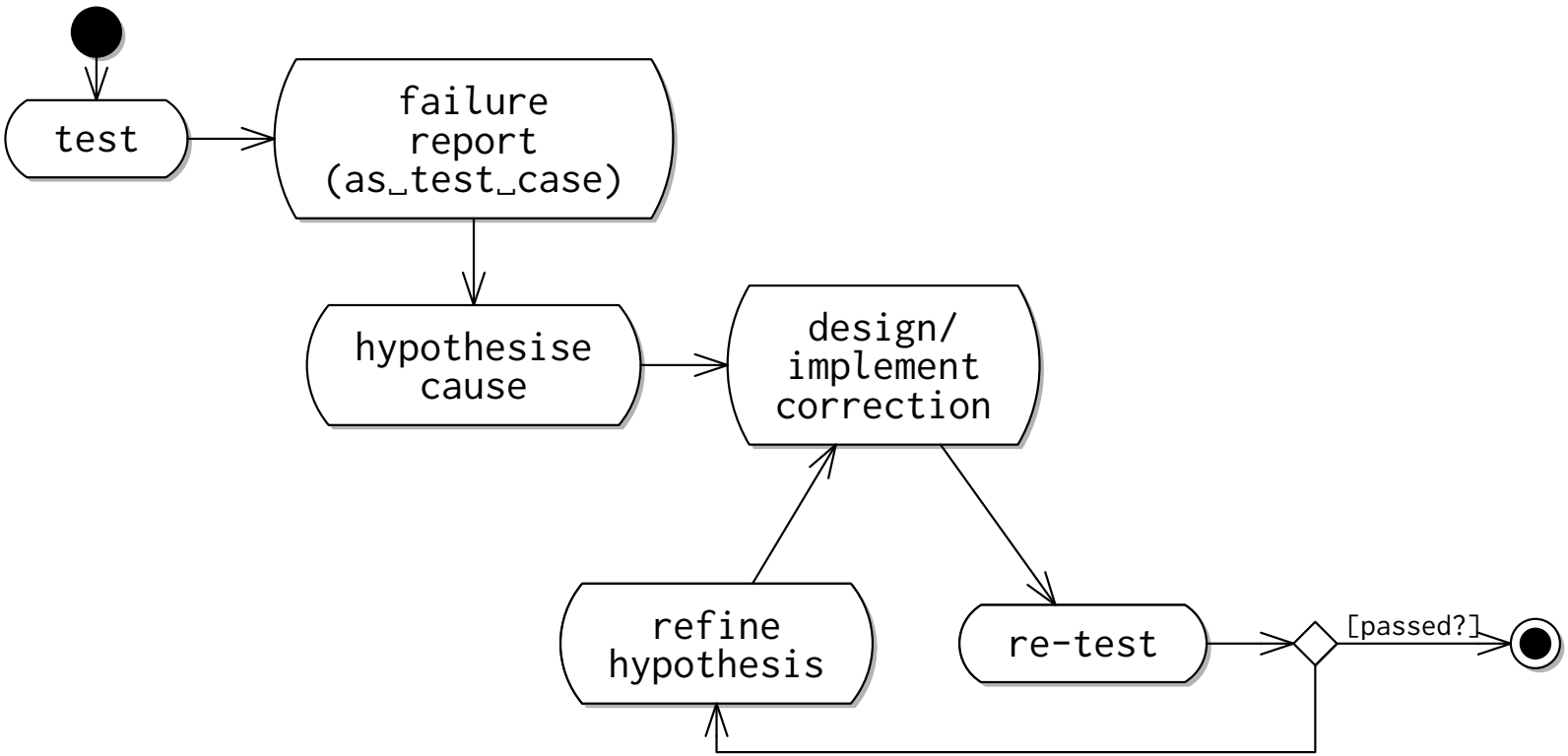

Figure: the test-fix cycle

Testing and software evolution

- release testing
  - alpha
  - beta
- ripple effect
- regression testing

Key point

Testing must be optimised to be both efficient and effective within the resources available

Consider the operation specifications shown below:

```
/**
 * A palindrome is a string that reads the same backwards as
 * well as forwards.
 * @return true, if the candidate is a palindrome
 */
public boolean isPalindrome(String candidate);
```

```
/**
 * Returns a new string, substituting all occurrences of
 * oldChar in source with newChar. If matchCase is false,
 * every occurrence of oldChar in the range a-z,A-Z is
 * substituted for newChar in the same case as the occurrence
 * of oldChar. Other characters are unaffected.
 */
public String substitute(String source, char oldChar,
  char newChar, boolean matchCase);
```

Problems

- identify the independent input variables and their equivalence partitions;
- specify the operational equivalence classes by combining the input variable classes; and
- propose test cases (with example data values) for the operational equivalence classes.

Equivalence classes from natural language specification from Lethbridge and Laganière [2005, pp. 382]:

> *"The landing gear must be deployed whenever the plane is within 2 minutes from landing or takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet."*

- what is the method signature for `deployLandingGear()` ?
- what are the equivalence classes for the parameters?
- what are the dependencies between parameters?
- what are the operation equivalence classes?

Construct an activity diagram for the `isPrime()` function:

```
public static boolean isPrime(Integer candidate) {
 // naive implementation, for kicks.

 if (candidate  < 0) candidate = -candidate;
 if (candidate == 0) return false;
 if (candidate == 1) return false;

 for (int i = 2; i < candidate; i++)
  if (candidate % i == 0) return false;

 return true;
}
```

- label all the edges in the diagram;
- specify a testing strategy and identify high priority paths;
- propose test cases to exercise the identified paths; and

How would you test this system architecture?

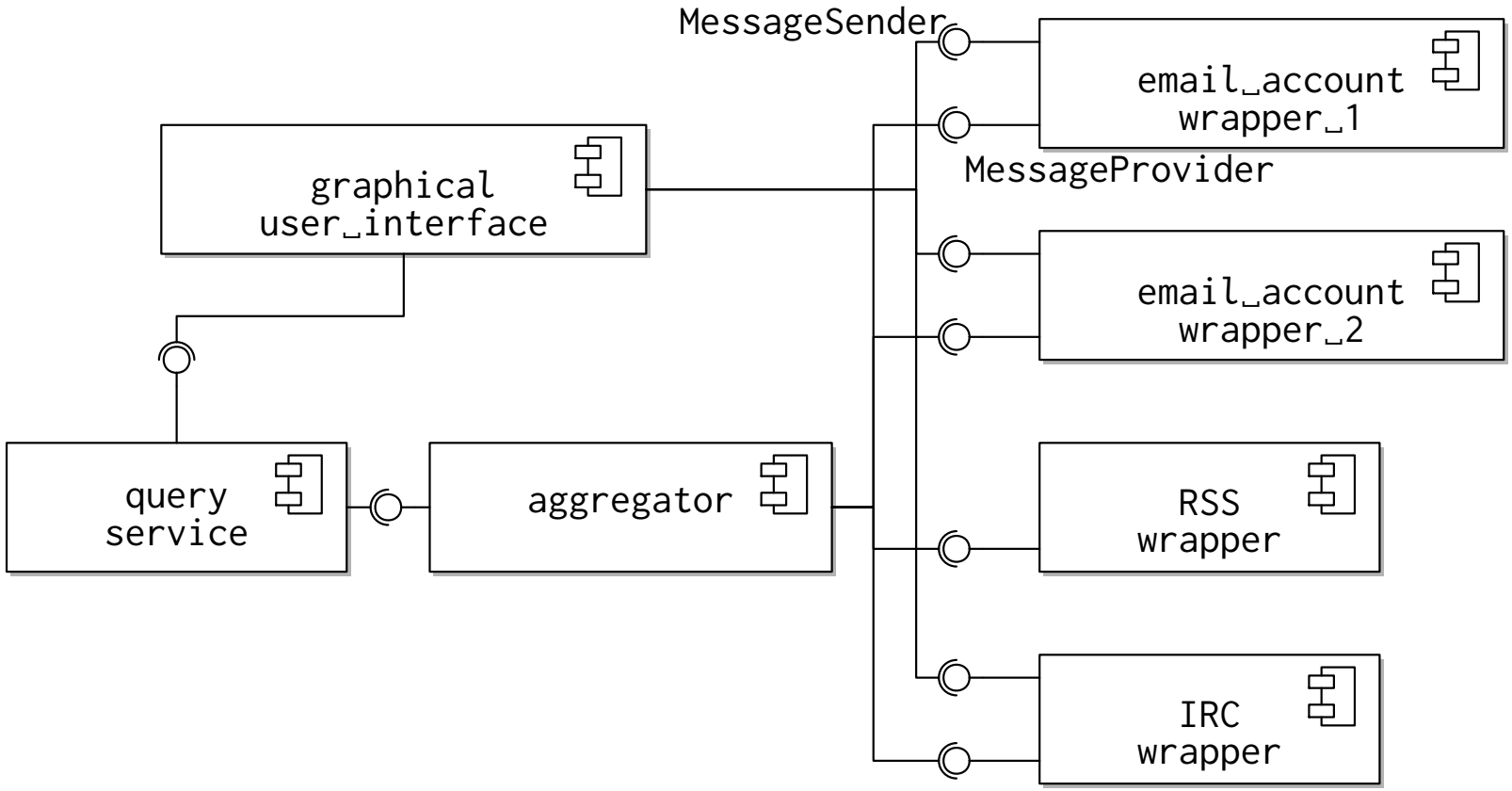

Figure: integration testing exercise

- Which components would you test first?
- Where are the boundary components?
- What stubs would you need to create?
- Which are the presentation components, how would you test them?
- How would you integrate your remaining tests, and in what order?
- How would you refine the architecture to make testing easier?

Consider the following Java method

```
public String caesarCipher(String plaintext) {
 if (plaintext == null) return null;

 String result = "";
 int shift = 3;

 // now encrypt the whole message
 for (int i = 0; i < plaintext.length(); i++) {
  int plainChar = plaintext.charAt(i);

  // set all out-of-bound characters to 'x';
  if (plainChar < 97 || plainChar > 122)
   plainChar = 23;
  else plainChar -= 97;

  // cyclically shift the char
  int cipherChar = (plainChar + shift) % 25;
  if (cipherChar < 0) cipherChar += 25;

  // get the character equivalent and add it to the result
  result += Character.toString((char) (cipherChar + 97));
 }
 return result;
}
```

Solution

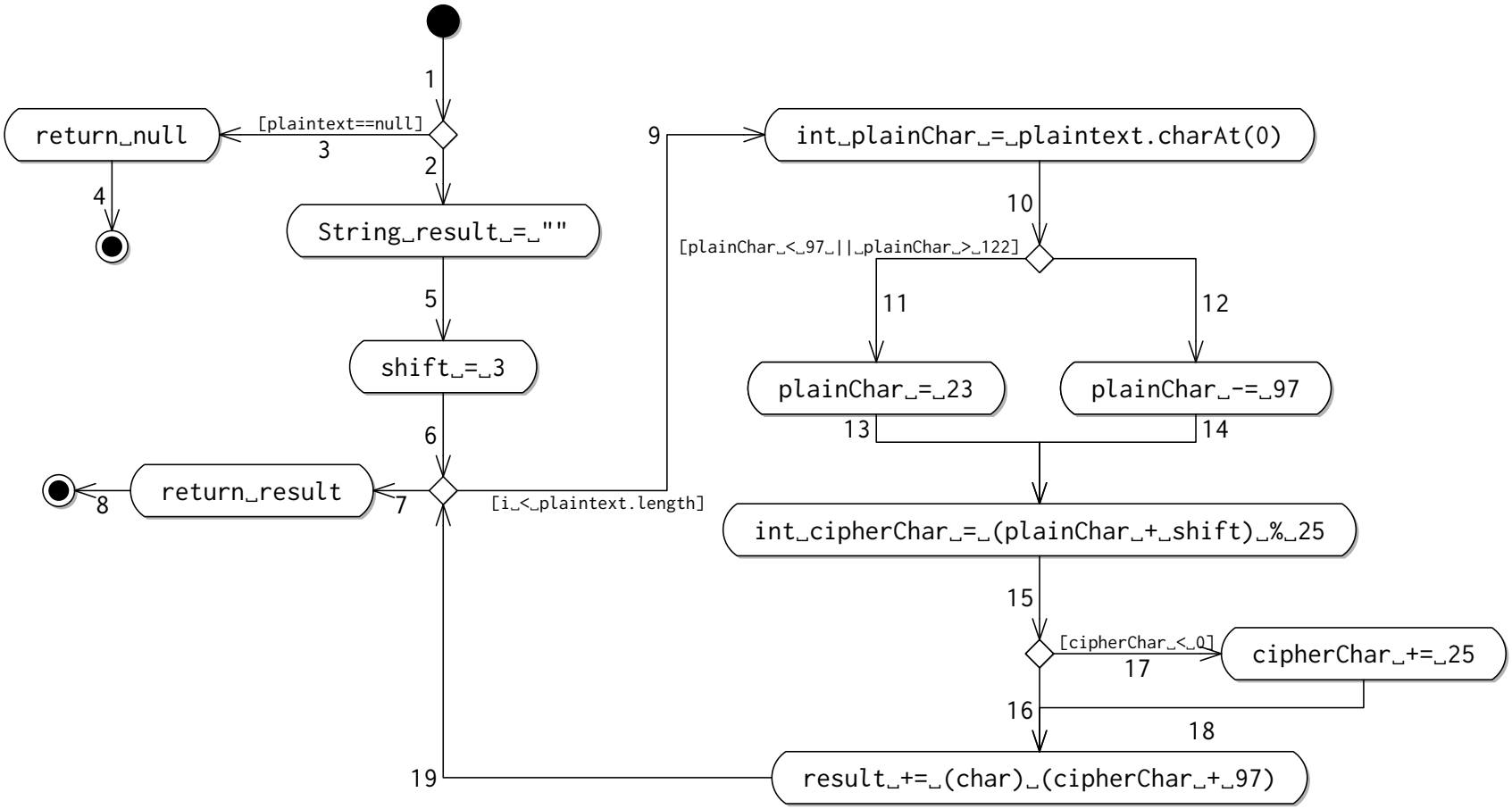

Figure: flow chart of the Caesar-cipher program

Propose three test cases suitable for the method

based on your activity diagram. For each test case, state the

- initial input,
- expected output, and
- the list (in order of use) of the edges covered during execution.

Give a short justification of the test cases you have proposed.

Example interface

```
/**
 * Converts a String of binary symbols (0,1)* in twos
 * complement format, into the equivalent integer value.
 *
 * @param binString
 *            the string of binary symbols (0,1)*
 * @param isTwosComplement
 *            specifies whether the string of binary symbols
 *            is to be treated as a twos complement signed
 *            integer or as an absolute value
 * @returns a decimal representation of the binary input.
 * @throws NumberFormatException
 *             if the input string contains any character
 *             other than 0 or 1
 */
public int convert(String binString,
  boolean isTwosComplement)
```

Setup the test case class

```
package uk.ac.glasgow.senotes.binarytodecimal.test;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class BinaryToDecimalImplTest {
}
```

Populate the test with methods

```
public class BinaryToDecimalImplTest {

 @Before
 public void setUp() throws Exception {
 }

 @After
 public void tearDown() throws Exception {
 }

 @Test
 public void testBinaryToDecimalString() {
  fail("Not yet implemented");
 }

 @Test
 public void testBinaryToDecimalStringBoolean() {
  fail("Not yet implemented");
 }
}
```

Creating and removing fixtures

```
private BinaryToDecimal binaryToDecimal;

@Before
public void setUp() throws Exception {
 binaryToDecimal = new BinaryToDecimalImpl();
}

@After
public void tearDown() throws Exception {
 binaryToDecimal = null;
}
```

Creating tests I

```
/**
 * defects: off-by-one in loop, wrong handling
 * of 2s complement.
 */
@Test
public void testBinaryToDecimalStringPositiveEven() {
 assertEquals(
   "Positive even conversion",
   4,
   binaryToDecimal.convert("0100"));
}
```

Creating tests II

```
/**
 * defects: off-by-one in loop, wrong handling
 * of 2s complement.
 */
@Test
public void testBinaryToDecimalStringPositiveOdd() {
 assertEquals(
   "Positive even conversion",
   11,
   binaryToDecimal.convert("01011"));
}
```

Testing for exceptions I

```
/**
 * defect: null condition
 */
@Test(expected=NullPointerException.class)
public void testBinaryToDecimalStringNull() {
 b2dUtil.convert(null);
}
```

Testing for exceptions II

```
/**
 * Test empty string
 */
@Test(expected=NumberFormatException.class)
public void testBinaryToDecimalStringEmpty() {
 b2dUtil.convert("ab");
}
```

Creating test suites

```
@RunWith(Suite.class)
@Suite.SuiteClasses(
  {
   BinaryToDecimalImplTestPositive.class,
   BinaryToDecimalImplTestIllegal.class
  })
public class AllTests {
 public AllTests(){
  System.out.println(
    "Beginning tests for BinaryToDecimal class...")
        ;
 }
}
```

Parameterised test cases

```
@RunWith(Parameterized.class)
public class BinaryToDecimalImplBulkTest {
}
```

Generating data for parameterised tests I

```
@Parameters
public static List<Object[]> createTestSet (){

 List<Object[]> result = new ArrayList<Object[]>();

 while (result.size() < testsToRun){

  String binaryString = createRandomBinaryString();

  BigInteger bigInteger = new BigInteger(binaryString,2);

  Integer expected = bigInteger.intValue();

  result.add(new Object[]{expected, binaryString});
 }

 return result;
}
```

Generating data for parameterised tests II

```
private static String createRandomBinaryString() {
 Random random = new Random();

 Integer inputLength = random.nextInt(31)+1;

 String binaryString = "";

 while(binaryString.length() < inputLength)
  binaryString += random.nextBoolean()?"1":"0";

 return binaryString;
}
```

Test case class constructor

```
private Integer expected;
private String binaryString;

public BinaryToDecimalImplBulkTest(
 Integer expected, String binaryString) {

 this.expected = expected;
 this.binaryString = binaryString;
}
```

Using test case parameters

```
@Test
public void test() {
 Integer actual =
  binaryToDecimal.convert(binaryString, false);

 String messageTemplate =
  "Conversion of [%s]";

 String message =
  String.format(messageTemplate, binaryString);

 assertEquals(message, expected, actual);
}
```