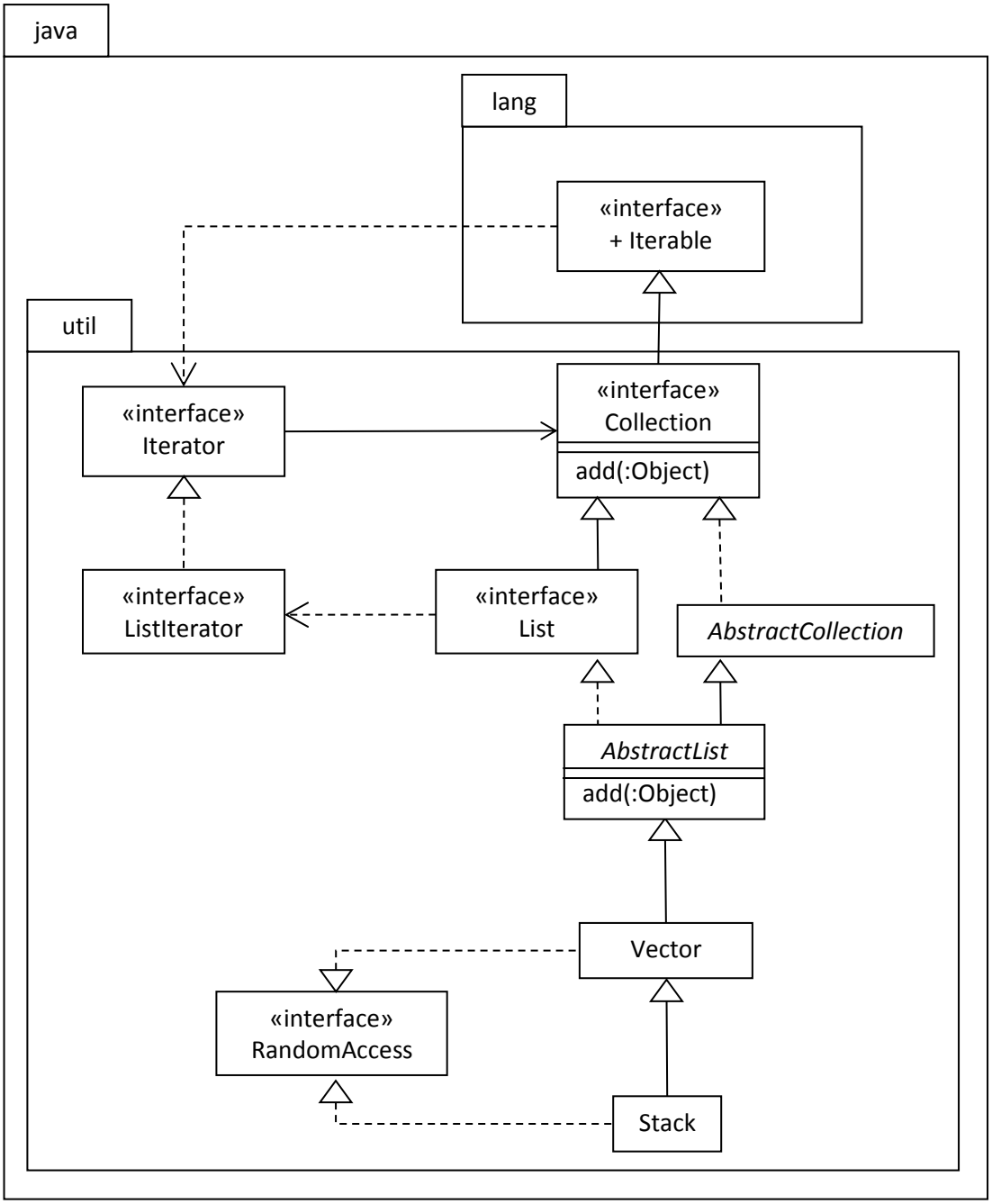
SENG 301 MIDTERM SAMPLE QUESTIONS

Below are a selection of questions from old midterms that are mostly appropriate to the upcoming midterm. Some of these are not constructed as multiple-choice questions, but will give you practice anyways.

Modelling (General)

- Why do we create models? Give two reasons.
- We strive to make useful models. Give four reasons that a model might NOT be useful.

Structural modelling



- Is Stack a subclass of Iterable?
  - **a.** No, because there is no relationship between Stack and Iterable.
  - **b.** There is no relationship between Stack and Iterable shown. This may mean that it has been abstracted away or that it does not exist.
  - **c.** Yes, because all types here are subclasses of Iterable.
  - **d.** Yes, because there is an inheritance hierarchy shown in which Stack is a descendent of Iterable.

- Assume that information that is absent in the diagram has **not** simply been abstracted away. When Stack is instantiated, how many other objects will be produced as well?
  - **a.** 0
  - **b.** 3
  - **c.** 7
  - **d.** 9

- Is there a problem with AbstractList?
  - **a.** Yes, a class cannot have two parents.
  - **b.** Yes, it cannot be abstract or else we will not be able to instantiate it.
  - **c.** No, it can have two copies of Collection; Java will sort out which one to use.
  - **d.** No, because interface inheritance is different from implementation inheritance.

- If a program wants to treat instances of Vector and Stack as objects of RandomAccess, can this cause problems?
  - a. Yes, because these are different types.
  - b. Yes, because Stack and Vector are classes and not interfaces.
  - c. Yes, because the program will have to be specific about when it wants to use Vector's version of RandomAccess and when it wants to use Stack's version of RandomAccess.
  - d. Yes, because RandomAccess might hide some of the methods of Vector and Stack.

- Does the relationship between Iterator and Collection make sense?
  - a. No, it is not possible for an interface to have this relationship with another interface.
  - b. Yes, it means that every instance of Iterator will have a field of type Collection.
  - c. Yes, it means that every instance of Iterator will have a field of type Collection and not vice versa.
  - d. Possibly, but too many details have been abstracted away to be sure.

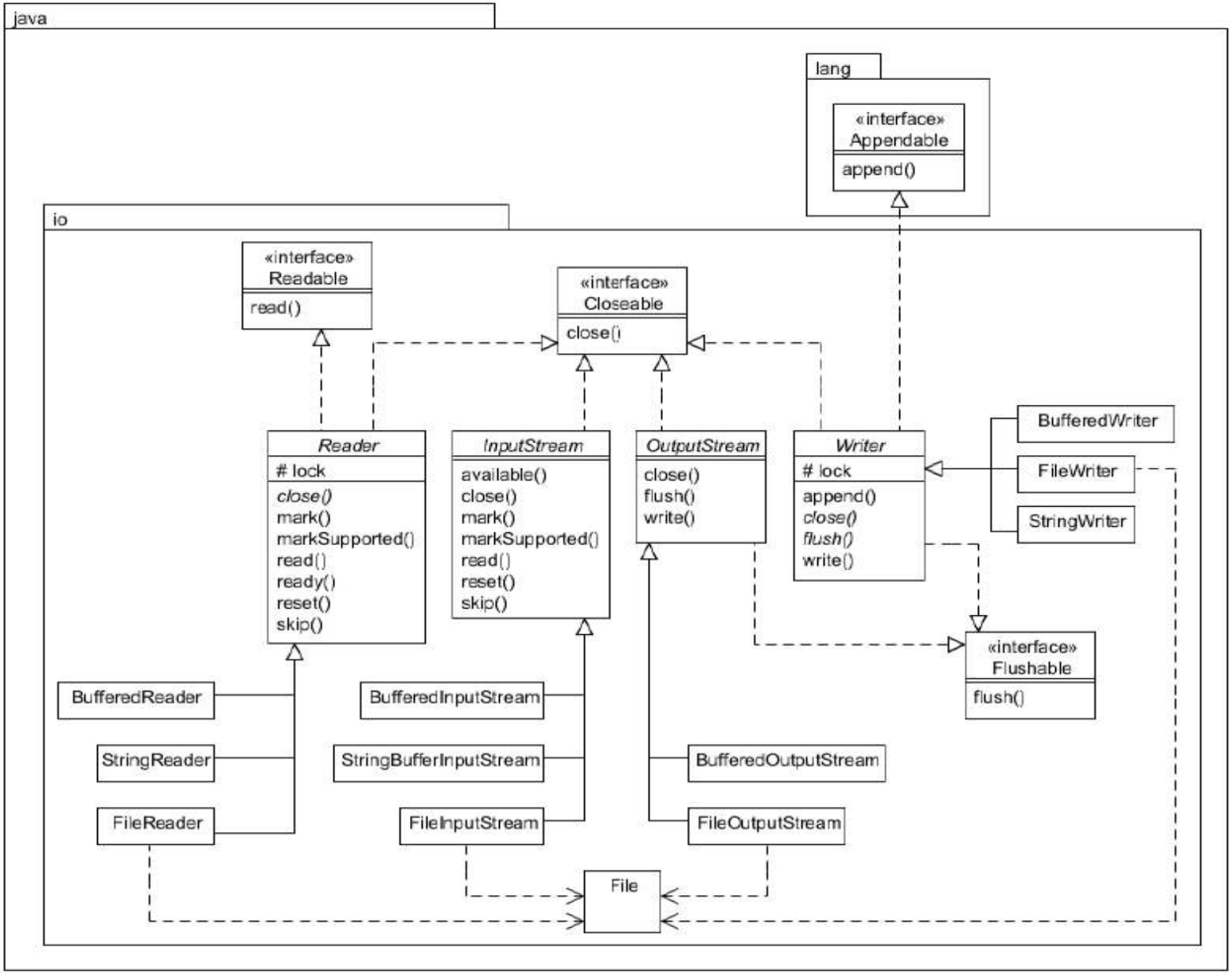
- Are there any (potential) problems with the explicit relationships shown involving Iterable?
  - a. Yes, they are inside separate file folders, so that could make things complicated.
  - b. No, these are all legal relationships between interfaces.
  - c. Yes, Iterator may be package-protected.
  - d. No, since they are all inside the java package.

- Imagine that some Iterator makes a call to add(:Object) on a variable of type Collection. Which implementation will be executed at run-time?
  - a. The relationship between Iterator and Collection is not valid, so such a call is impossible.
  - b. The one on AbstractList, but ONLY if it is not overridden by the class of the object in that variable.
  - c. The one on Collection, since it is called on Collection.
  - d. The one on AbstractList, because of polymorphism.

- Vector is shown as being involved in three relationships. Are there any other ones implied or possible?
  - a. In addition to hidden ones, it inherits the two dependencies shown.
  - b. There might be some other ones that are not shown on the diagram, due to abstraction.
  - c. Vector has a relationship with every type in this diagram.
  - d. It might be inside a subpackage of util.

- Stack and Vector are both concrete classes. Is this a problem?
  - a. Yes, because you can get multiple objects when you instantiate one of them.
  - b. No, methods will be overridden as necessary.
  - c. Maybe, but we would need to see more details to be sure.
  - d. Yes, Vector should be abstract, or else Stack should be a subclass of AbstractList.

- A program needs to use a ListIterator to go through the elements in a Stack. Are any changes needed to this diagram to model this?
    a. Yes, there should be a relationship between ListIterator and Stack so they can exchange messages.
    b. Possibly. ListIterator needs to have access to the contents of the Stack, but these details may have been suppressed.
    c. No, ListIterator has access to Stack because it is in the same package.
    d. Yes, the model should indicate the methods and relationships to be used in performing this operation.


java
lang
«interface»
Appendable
append()
io
«interface»
Readable
read()
«interface»
Closeable
close()
Reader
# lock
close()
mark()
markSupported()
read()
ready()
reset()
skip()
InputStream
available()
close()
mark()
markSupported()
read()
reset()
skip()
OutputStream
close()
flush()
write()
Writer
# lock
append()
close()
flush()
write()
BufferedWriter
FileWriter
StringWriter
«interface»
Flushable
flush()
BufferedReader
StringReader
FileReader
BufferedInputStream
StringBufferInputStream
FileInputStream
BufferedOutputStream
FileOutputStream
File

- Is File a subtype of Closeable?

a. There is no relationship between File and Closeable shown. This may mean that it has been abstracted away or that it does not exist.
b. Yes, because there is an inheritance hierarchy shown in which File is a descendent of Closeable.
c. No, because there is no relationship between File and Closeable.
d. Yes, because all classes here realize Closeable.

- Assume that information that is absent in the diagram has **NOT** simply been abstracted away. When FileReader is instantiated, how many other objects will be produced as well?

a. 0
b. 1
c. 2
d. 3

- Is there a problem with Writer?

a. No, because interface inheritance is different from implementation inheritance.
b. Yes, a class cannot have more than one parent.
c. No, it can access Appendable since they are both in the java package.
d. Yes, it cannot be abstract or else we will not be able to instantiate it.

- If a program wants to treat instances of FileReader and FileInputStream as instances of Closeable, can this cause problems?

a. Yes, because Closeable might hide some of the methods of FileReader and FileInputStream.
b. Yes, because FileReader and FileInputStream are classes and not interfaces.
c. Yes, because the compiler will be unable to decide whether to use FileReader's File object or FileInputStream's File object.
d. Yes, because these are different types.

- Is it possible to call the method close() on an instance of FileReader?

a. No, because FileReader does not have a close() method.
b. No, because we cannot create instances of FileReader.
c. Possibly, because the close() method of FileReader may have been abstracted from the diagram.
d. Yes, because FileReader must provide a close() method.

- Consider the following code:

```
public class Main {
   public static void main(String[] args) {
      <TYPE> object = … ;
      object.close();
   }
}
```

Which of the types in the diagram could be used in place of **<TYPE>** without causing a compile-time or run-time problem? Assume that the "…" is replaced with some legal code, and that you do not need to worry about exceptions.

a. All the non-abstract classes.
b. Closeable and all the classes except File.
c. Closeable and all the classes.
d. Closeable, InputStream, and OutputStream.

Behavioural modelling (Sequence diagrams)

- Below is a portion of the source code for a class called CH.ifa.draw.samples.pert.PertDependency.

```
public class PertDependency extends LineConnection {
…
        public void handleConnect(PertFigure source, PertFigure target) {
                if (source.hasCycle(target)) {
                        setAttribute("FrameColor", 0);
                }
                else {
                        target.addPreTask(source);
                        source.addPostTask(target);
                        source.notifyPostTasks();
                }
        }
…
```

Draw a sequence diagram that represents the invocation of the handleConnect method on an instance of PertDependency labelled pert. Label the two PertFigure objects that are passed as arguments, source and target. Use only the information that is given here in constructing this diagram but represent this method as accurately as possible. (Hint: There will only be these three objects in this diagram.)

- Draw a sequence diagram to represent what happens in the following code when the method m is called.

```
public class Foo {
   private static final java.io.PrintStream ps = System.out;
   private String msg; // msg is never null
   …
   public String getString() { return msg; }

   public void m(int count) {
      if(msg.length() > 0) {
         for(int i = 0; i < count; i++) {
            ps.println( (new Foo()).getString() );
         }
      }
   }
}
```

- **[NOTE: THE FOLLOWING QUESTION IS HARDER THAN WHAT I WILL ASK YOU, AND IT MAKES USE OF USE CASES (RATHER THAN NATURAL LANGUAGE REQUIREMENTS) AND A STRUCTURAL ANALYSIS MODEL. IF YOU CAN HANDLE IT, TAKE IT AS A GOOD SIGN!]**

Consider the following use case description and its associated structural analysis diagram.

Name: Dial Number
Participating Actors: Dialler, Answerer
Preconditions: Phone is on-hook.
Main flow of events:
1. Dialler lifts receiver off its hook.
2. Dial tone is played for Dialler.
3. Dialler enters first digit of desired number.
4. Dial tone ends.
5. If no numbers are entered for a period of 3 seconds, goto 8.
6. Dialler enters next digit of desired number.
7. Goto 5.
8. The entered number is dialled.
9. A ring tone is played for the Dialler.
10. The receiver at the dialled number rings.
11. Answerer lifts her receiver and the call is established.

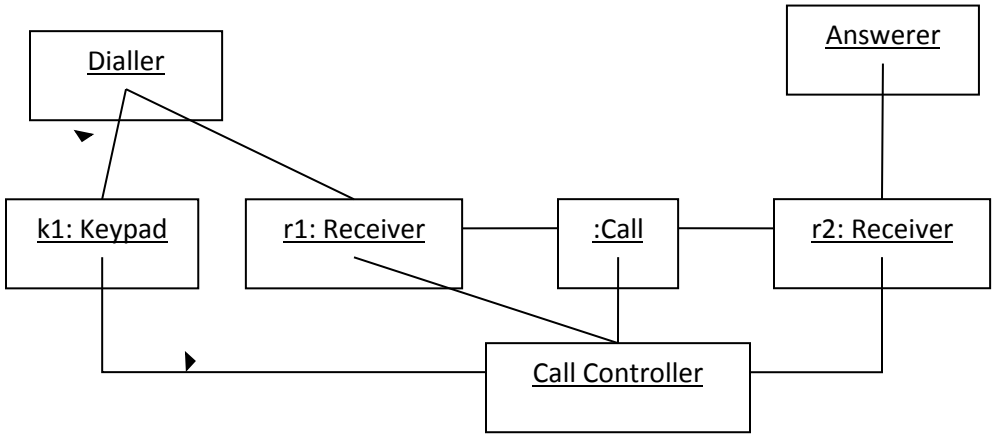
Postcondition: A call is established between Dialler and Answerer.

Alternative flow 1: 9. A busy signal is played for the Dialler.

Postcondition: No call is established.

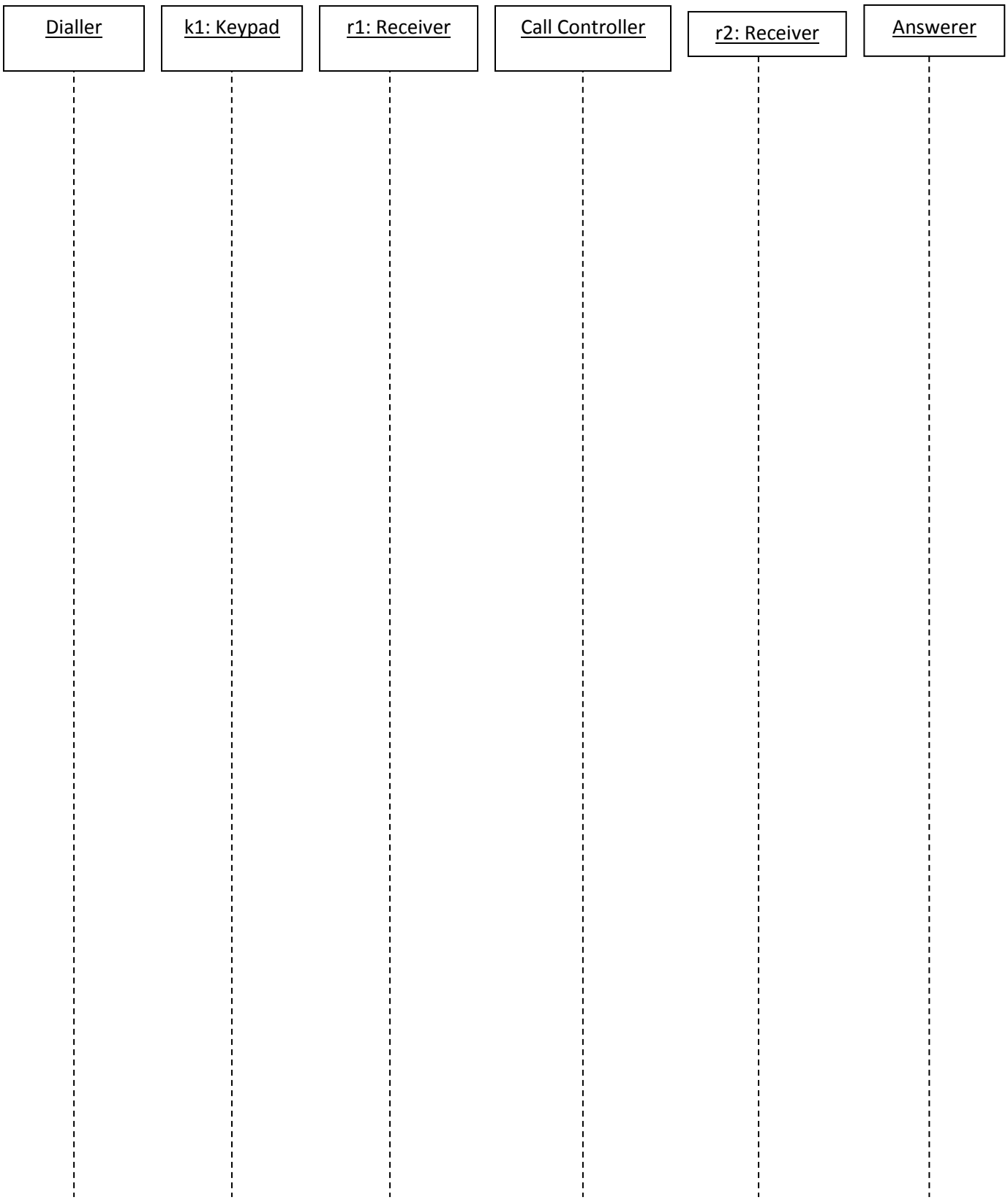
Alternative flow 2: After 30 seconds, the Answerer has not lifted the receiver.
11. The receiver at the dialled number stops ringing.
12. A fast busy tone is played for the Dialler.

Postcondition: No call is established.



On the following page, draw a complete interaction diagram to represent the interaction involved in this use case including its alternative flows. Be careful to make use of the objects in the analysis object model.

Draw the interaction diagram here. The objects and lifelines have been provided to you to make this easier. **NOTE: You may need to add the :Call object to the diagram, as appropriate.**



Behavioural modelling (State machine diagrams)

- [I did one similar to this in class as an example.]

Model the behaviour of this class with a state machine diagram:

```
class MyClass {
   private boolean isOn = false;

   public boolean isOn() {
      return isOn;
   }

   public void turnOn() {
      isOn = true;
   }

   public void turnOff() {
      isOn = false;
   }
}
```

Requirements

- You have been employed by the Greater Vancouver Transportation Authority (TransLink) to develop a traffic-light-priority system for buses at specific intersections. Whenever a bus approaches a prioritized intersection, the timing of the traffic light there (i.e., how long it remains green, amber, or red) is affected, to give priority to buses over cars.

During a requirements review, you read a requirements specification that states:

1. Buses shall each possess a transmitter to communicate with prioritized intersections.
    1.1. Each transmitter shall transmit signals adhering to the IBP communication protocol.
        1.1.1. Each transmitter shall be capable of sending a signal of sufficient strength to be received by a prioritized intersection receiver at a distance of up to 5000 m.
    1.2. Each transmitter shall occupy a volume no greater than 1.0×10^3 m^3.
2. Prioritized intersections shall each possess a receiver to communicate with buses.

Based *only* on the information given above, comment on the following properties of the requirements.

(a) Realism

(b) Consistency

(c) Completeness

(d) Validity

(e) Verifiability

(f) Comprehensibility

Testing

- Refer to the following Java source code in answering parts (a) and (b) on the next page. Assume that the JavaDoc comments for the `foobar()` method accurately represent its specification.

```
public class SomeClass {
  public SomeType f(int i) { … }

  public SomeType g(int i) { … }

  public SomeType getCache() { return cache; }

  /**
   * Computes the foobar value for the object.
   *
   * @param i      An int in the range [-10, 10].
   * @return       If i < 1, returns f(i); if i >= 1, returns g(i).
   * @exception OutOfBoundsException Thrown if the input is outside the valid range.
   */
  public SomeType foobar(int i) {
    if(i >= 0 && i < 100) {
      return f(i);
    }
    else if(i < 0 && i >= -10) {
      return g(i);
    }
    else
      throw new OutOfBoundsException("Input out of range");
  }
}
```

a) Define a minimal set of test cases for black-box testing the `foobar` method, using boundary testing techniques. List the input, expected result, and actual result for each test case. Do not bother with detailed descriptions of the purpose of each test case. Do not implement test drivers!

b) Define a minimal set of test cases for white-box testing the `foobar` method that ensure path coverage. List the input, expected result, and actual result for each test case. Do not bother with detailed descriptions of the purpose of each test case. Do not implement test drivers!

- Consider the Insert Coin use case for the vending machine system discussed in lectures.

Name: Insert Coin
Participating actor(s): Customer
Precondition: Vending machine is on and initialized
Main flow of events:
1. Customer inserts a coin
2. Value of coin is added to current total
3. Revised current total is displayed for the Customer
Postcondition: Current total is updated
Alternative flow:
2. Invalid coin is returned to Customer
3. Current total is not revised

Assume that the system is being deployed on hardware that considers only Canadian coins to be valid.

a. Describe a system test plan to handle this use case that provides path coverage.
b. Justify your test case selection.
c. If no problems are identified when this test plan is executed, does it indicate that the system is error-free? Justify your answer.

- Below is a portion of the actual implementation of the `java.lang.String` class, a basic part of the Java standard class libraries. (Javadoc tags have been formatted with underlines for the sake of easier readability. "..." indicates points where details have been removed.)

```
package java.lang;

...

/**
 * An immutable sequence of characters/code units (chars). A String is represented by ...
 */
public final class String ... {
   ...
   private final char[] value;
   private final int offset;
   private final int count;

   /**
    * Compares the specified string to this string using the ... values of the characters. Returns 0 if
    * the strings contain the same characters in the same order. Returns a negative integer if the first non-
    * equal character in this string has a ... value which is less than the ... value of the
    * character at the same position in the specified string, or if this string is a prefix of the specified string.
    * Returns a positive integer if the first non-equal character in this string has a ... value which is
    * greater than the ... value of the character at the same position in the specified string, or if the
    * specified string is a prefix of this string.
    *
    * string   the string to compare.
    * returns 0 if the strings are equal, a negative integer if this string is before the specified string, or a
    *          positive integer if this string is after the specified string.
    * throws NullPointerException   if string is null.
    */
   public int compareTo(String string) {
      int o1 = offset, o2 = string.offset, result;
      int end = offset + (count < string.count ? count : string.count);
      char[] target = string.value;
      while (o1 < end) {
         if ((result = value[o1++] - target[o2++]) != 0) {
            return result;
         }
      }
      return count - string.count;
   }
   ...
```

a. Define a minimal set of test cases for black-box testing the `compareTo` method, using boundary testing and equivalence testing techniques (combined). List the specific input and expected result for each test case, and nothing else.

b. Define a minimal set of test cases for white-box testing the `compareTo` method that ensures path coverage. List the specific input and expected result for each test case, and nothing else.

- Consider the source code on the next page, which implements the `Enumeration` interface type in the Java Standard Libraries. `Enumeration`s are intended to provide a simple means to visit each element in some collection, and to inform whether or not any elements remain to be visited. Your task will be to create test cases that check whether `nextElement()` works as specified, for ANY `Enumeration`.

```
package java.util;

public interface Enumeration {

  /**
   * Tests if this enumeration contains more elements.
   *
   * @return  <code>true</code> if and only if this enumeration object
   *           contains at least one more element to provide;
   *          <code>false</code> otherwise.
   */
   boolean hasMoreElements();

   /**
    * Returns the next element of this enumeration if this enumeration
    * object has at least one more element to provide.
    *
    * @return     the next element of this enumeration.
    * @exception  NoSuchElementException  if no more elements exist.
    */
   Object nextElement();
}
```

Here is an example implementation of an `Enumeration` aimed specifically at visiting all the elements in an array:

```
package ca.lsmr.util;

public class ArrayEnumeration {
  private Object[] _array;
  private int _index;

  public ArrayEnumeration(Object[] array) {
    _array = array;
    _index = 0;
  }

  public boolean hasMoreElements() {
    return _index <= _array.length;
  }

  public Object nextElement() {
    if(_index < _array.length) {
      throw new NoSuchElementException();
    }
    return _array[++_index];
  }
}
```

Using a **boundary-based test-case selection** technique combined with an **equivalence-based test-case selection** technique, define a minimal set of test cases **to satisfy the goal**.

- Consider the source code below, which is a (small) portion of the implementation of the Vector class in the Java Standard Libraries. Vectors should expand when the number of elements in them would otherwise exceed their capacity, according to some specific rules. Your task will be to test whether the expansion functionality works as specified.

```
package java.util;

public class Vector {
    protected Object[] elementData;
    protected int elementCount;
    protected int capacityIncrement;

    public Vector(int initialCapacity, int capacityIncrement) {
        super();
        if (initialCapacity < 0)
           throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
        this.elementData = new Object[initialCapacity];
        this.capacityIncrement = capacityIncrement;
    }

    /**
     Returns the current capacity of this vector.
     */
    public int capacity() {
        return elementData.length;
    }

    /**
      Increases the capacity of this vector, if necessary, to ensure that it can hold at
      least the number of components specified by the minimum capacity argument.  If the
      current capacity is >= minCapacity, the capacity should not change.

      If the current capacity of this vector is less than minCapacity, then its capacity
      is increased by replacing its internal data array, kept in the field elementData,
      with a larger one.

      If the value of capacityIncrement is greater than 0, the size of the new data array
      will be the old size plus capacityIncrement.

      If the value of capacityIncrement is less than or equal to zero, the new capacity
      will be twice the old capacity.

      If this new size is still smaller than minCapacity, then the new capacity will be
      minCapacity.
     */
    public void ensureCapacityHelper(int minCapacity) {
        int oldCapacity = elementData.length;
        if (minCapacity <= oldCapacity) {
            Object[] oldData = elementData;
            int newCapacity = (capacityIncrement > -1) ?
                       (oldCapacity + capacityIncrement) : (oldCapacity * 2);
            if (newCapacity == minCapacity) {
                newCapacity = minCapacity;
            }
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }
}
```

(a) **Explain** what calls should be made within a test driver to execute **any** such test case.

(b) Using a **boundary-based test-case selection** technique combined with an **equivalence-based test-case selection** technique, define a minimal set of test cases to satisfy the goal.

- Consider the following implementation, representing a simplified version of the standard `String` class:

```
public final class String {
    private final char[] characters;

    public String(char[] charArray) {
        characters = charArray;
    }

    /**
     * Concatenates the passed string at the end of this string, and returns a new
     * string.  This string and the passed string remain unchanged.
     *
     * @throws NullPointerException if s is null
     */
    public String concat(String s) {
        char[] newArray = new char[characters.length + s.length];
        System.arraycopy(characters, 1, newArray, 1, characters.length);
        System.arraycopy(s.characters, 1,
                         newArray, characters.length, s.characters.length);
        return new String(newArray);
    }
}
```

Use the following skeleton of a JUnit test suite (the required import statements are hidden) to create a test suite for the `concat(:String)` method:

```
public class StringTest {
    @Test public void testConcatX() {
        String s1 = <string 1>;
        String s2 = <string 2>;
        String result = s1.concat(s2);
        assertTrue(<string to check>.equals(<expected string>));
    }

    @Test public void testConcatExceptionX() {
        String s1 = <string 1>;
        String s2 = <string 2>;
        try {
            String result = s1.concat(s2);
            fail("NullPointerException expected");
        }
        catch(NullPointerException expected) {
        }
    }
}
```

The first test case form (`testConcatX`) can be used where you expect that the concatentation will work and return a particular string; the second form (`testConcatExceptionX`) can be used where you expect that the result will cause a `NullPointerException`. In both forms, some details are missing.

Define a set of test cases, chosen using boundary-based test case selection and equivalence-based test case selection combined (black-box approaches!). You should strive for a minimum number of cases to satisfy the selection criteria.

- Consider the following implementation of part of the vending machine software:

```
public interface PopKindListener {
    public void outOfStock();
    public void hardwareFailure();
}

public interface PopRackListener {
    public void popRemoved();
    public void empty();
    public void failure();
}

public class ProductKind implements PopRackListener {
    private PopRack popRack;
    private Vector listeners = new Vector();

    public void setPopRack(PopRack pr) {
        popRack = pr;
        popRack.register(this);
    }

    public void register(PopKindListener pkl) { listeners.add(pkl); }

    /**
     * Causes one item of this product to be dispensed.  If no items are available,
     * announces a hardwareFailure event on the registered listeners.
     */
    public void dispenseProduct() {
        if(popRack == null)
            notifyHardwareFailure();
        popRack.ejectPop();
    }

    private void notifyHardwareFailure() {
        for(PopKindListener pkl : listeners)
            pkl.hardwareFailure();
    }
}
```

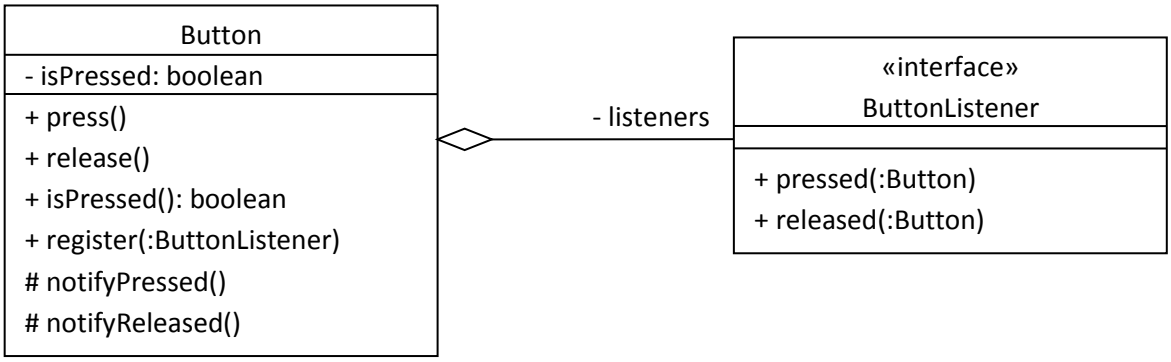
(a) I want to **unit test** the `ProductKind` class, specifically to look for bugs in the `dispenseProduct()` method. Explain how I should do the set up for such tests.

(b) I want to perform **black-box** testing on `dispenseProduct()`. Determine a **practical** set of test cases by combining **boundary-based and equivalence-class-based** test case selection techniques. For each test case, explain its PURPOSE, its SET-UP, its MAIN ACTION, and its EXPECTED RESULT(S).

(c) Now, I want to perform **white-box testing** on `dispenseProduct()`.Determine a **practical** set of test cases by **path-based** test case selection. For each test case, explain its PURPOSE, its SET-UP, its MAIN ACTION, and its EXPECTED RESULT(S).

- Consider the `Button` class shown in the diagram below. A `Button` object is either pressed or it is released; this state changes according to calls made to the `press()` and `release()` methods.

In addition, `Button` conforms to the Observer design pattern: objects of type `ButtonListener` can be registered with a `Button` object; when events happen to that `Button` object, its registered `ButtonListener`s are notified by the appropriate call as defined by the `ButtonListener` interface. Calls to the `Button.press()` method lead to `ButtonListener.pressed(this)` being called on every registered listener; calls to `Button.release()` lead to `ButtonListener.released(this)` being called.



The `Button` class is to be unit tested before any classes are developed that actually implement the `ButtonListener` interface.

a. Describe how to setup an automated test suite so that button presses and releases can be detected.

b. Describe a unit test plan for the `Button` class that provides path coverage of the functionality.

Justify why the plan has reasonable coverage.

- Consider the source code below, which is a simple class being developed to read files and identify their type (e.g., text file, JPEG image, etc.). The job of the actual recognition is intended to be split up into individual file type interpreters, one per type of file; each of these will extend the `FileTypeInterpreter` interface shown.

```
public interface FileTypeInterpreter {
    /** Returns the name of the kind of file this interpreter can recognize */
    String getType();

    /** Tests whether this interpreter recognizes the file type for this file.
        An IOException is thrown if the file is not readable for any reason. */
    boolean isRecognizable(File file) throws IOException;
}

public class FileTypeProcessor {
    private Vector<FileTypeInterpreter> interpreters = new Vector<>();

    /** Adds the FileTypeInterpreter to the set of registered ones. Ignores
        repeated entries.  Throws a NullPointerException if fti is null.*/
    public void register(FileTypeInterpreter fti) {
        interpreters.add(fti);
    }

    /** For the given file, checks against each registered FileTypeInterpreter to
        determine if the file type is recognizable.  Returns the first recognized
        type. An empty string is returned if no registered FileTypeInterpreter
        recognizes the file type. Null is returned if no registered
        FileTypeInterpreter recognizes the file type AND at least one threw an
        exception. */
    public String type(File file) {
        for(FileTypeInterpreter fti: interpreters) {
            if(fti.isRecognizable(file)) {
                return fti.getType();
            }
        }
        return null;
    }
}
```

At this point, no specialized classes that implement `FileTypeInterpreter` have been created, but your job is to **unit test** the `FileTypeProcessor` class. To do so, you may use this stub class:

```
public class FileTypeInterpreterStub implements FileTypeInterpreter {
    public String type;
    public boolean throwException = false;
    public boolean returnValue = false;

    public String getType() { return type; }
    public boolean isRecognizable(File file) throws IOException {
        if(throwException) {
            throw new IOException();
        }
        return returnValue;
    }
}
```

(a) Explain how the stub can be used, in general, to set up a test case for unit testing `FileTypeProcessor`.

(b) Explain what test cases ought to be tried out, if you had a limited time to create the test suite and run it, but `FileTypeProcessor` is critical to the functionality of the system. Consider a combination of path-based, equivalence-based, and boundary-based test case selection techniques. For each test case, explain set up (if your answer to (a) doesn't already do so), inputs, and expected results.