```
54        int testDay = day - DAYS_FOR_RESULT;
55        if (testDay < 0 || testDay >= dropsByDay.size()) {
56           return false;
57        }
58        for (int d = 0; d <= testDay; d++) {
59           ArrayList<Bottle> bottles = dropsByDay.get(d);
60           if (hasPoison(bottles)) {
61              return true;
62           }
63        }
64        return false;
65     }
66  }
```

This is just one way of simulating the behavior of the bottles and test strips, and each has its pros and cons.

With this infrastructure built, we can now implement code to test our approach.

```
1   int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2      int today = 0;
3
4      while (bottles.size() > 1 && strips.size() > 0) {
5         /* Run tests. */
6         runTestSet(bottles, strips, today);
7
8         /* Wait for results. */
9         today += TestStrip.DAYS_FOR_RESULT;
10
11        /* Check results. */
12        for (TestStrip strip : strips) {
13           if (strip.isPositiveOnDay(today)) {
14              bottles = strip.getLastWeeksBottles(today);
15              strips.remove(strip);
16              break;
17           }
18        }
19     }
20
21     if (bottles.size() == 1) {
22        return bottles.get(0).getId();
23     }
24     return -1;
25  }
26
27  /* Distribute bottles across test strips evenly. */
28  void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
29     int index = 0;
30     for (Bottle bottle : bottles) {
31        TestStrip strip = strips.get(index);
32        strip.addDropOnDay(day, bottle);
33        index = (index + 1) % strips.size();
34     }
35  }
36
37  /* The complete code can be found in the downloadable code attachment. */
```

Note that this approach makes the assumption that there will always be multiple test strips at each round. This assumption is valid for 1000 bottles and 10 test strips.

If we can't assume this, we can implement a fail-safe. If we have just one test strip remaining, we start doing one bottle at a time: test a bottle, wait a week, test another bottle. This approach will take at most 28 days.

Optimized Approach (10 days)

As noted in the beginning of the solution, it might be more optimal to run multiple tests at once.

If we divide the bottles up into 10 groups (with bottles 0 - 99 going to strip 0, bottles 100 - 199 going to strip 1, bottles 200 - 299 going to strip 2, and so on), then day 7 will reveal the first digit of the bottle number. A positive result on strip `i` at day 7 shows that the first digit (100's digit) of the bottle number is `i`.

Dividing the bottles in a different way can reveal the second or third digit. We just need to run these tests on different days so that we don't confuse the results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9
Strip 0	0xx	x0x	xx0
Strip 1	1xx	x1x	xx1
Strip 2	2xx	x2x	xx2
Strip 3	3xx	x3x	xx3
Strip 4	4xx	x4x	xx4
Strip 5	5xx	x5x	xx5
Strip 6	6xx	x6x	xx6
Strip 7	7xx	x7x	xx7
Strip 8	8xx	x8x	xx8
Strip 9	9xx	x9x	xx9

For example, if day 7 showed a positive result on strip 4, day 8 showed a positive result on strip 3, and day 9 showed a positive result on strip 8, then this would map to bottle #438.

This mostly works, except for one edge case: what happens if the poisoned bottle has a duplicate digit? For example, bottle #882 or bottle #383.

In fact, these cases are quite different. If day 8 doesn't have any "new" positive results, then we can conclude that digit 2 equals digit 1.

The bigger issue is what happens if day 9 doesn't have any new positive results. In this case, all we know is that digit 3 equals either digit 1 or digit 2. We could not distinguish between bottle #383 and bottle #388. They will both have the same pattern of test results.

We will need to run one additional test. We could run this at the end to clear up ambiguity, but we can also run it at day 3, just in case there's any ambiguity. All we need to do is shift the final digit so that it winds up in a different place than day 2's results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 0	0xx	x0x	xx0	xx9
Strip 1	1xx	x1x	xx1	xx0
Strip 2	2xx	x2x	xx2	xx1
Strip 3	3xx	x3x	xx3	xx2
Strip 4	4xx	x4x	xx4	xx3
Strip 5	5xx	x5x	xx5	xx4

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 6	6xx	x6x	xx6	xx5
Strip 7	7xx	x7x	xx7	xx6
Strip 8	8xx	x8x	xx8	xx7
Strip 9	9xx	x9x	xx9	xx8

Now, bottle #383 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #4), while bottle #388 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #9). We can distinguish between these by "reversing" the shifting on day 10's results.

What happens, though, if day 10 still doesn't see any new results? Could this happen?

Actually, yes. Bottle #898 would see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> [NONE]). That's okay, though. We just need to distinguish bottle #898 from #899. Bottle #899 will see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> #0).

The "ambiguous" bottles from day 9 will always map to different values on day 10. The logic is:

- If Day 3->10's test reveals a new test result, "unshift" this value to derive the third digit.
- Otherwise, we know that the third digit equals either the first digit or the second digit *and* that the third digit, when shifted, still equals either the first digit or the second digit. Therefore, we just need to figure out whether the first digit "shifts" into the second digit or the other way around. In the former case, the third digit equals the first digit. In the latter case, the third digit equals the second digit.

Implementing this requires some careful work to prevent bugs.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2     if (bottles.size() > 1000 || strips.size() < 10) return -1;
3
4     int tests = 4; // three digits, plus one extra
5     int nTestStrips = strips.size();
6
7     /* Run tests. */
8     for (int day = 0; day < tests; day++) {
9        runTestSet(bottles, strips, day);
10    }
11
12    /* Get results. */
13    HashSet<Integer> previousResults = new HashSet<Integer>();
14    int[] digits = new int[tests];
15    for (int day = 0; day < tests; day++) {
16       int resultDay = day + TestStrip.DAYS_FOR_RESULT;
17       digits[day] = getPositiveOnDay(strips, resultDay, previousResults);
18       previousResults.add(digits[day]);
19    }
20
21    /* If day 1's results matched day 0's, update the digit. */
22    if (digits[1] == -1) {
23       digits[1] = digits[0];
24    }
25
26    /* If day 2 matched day 0 or day 1, check day 3. Day 3 is the same as day 2, but
27     * incremented by 1. */
28    if (digits[2] == -1) {
```

```
    if (digits[3] == -1) { /* Day 3 didn't give new result */
      /* Digit 2 equals digit 0 or digit 1. But, digit 2, when incremented also
       * matches digit 0 or digit 1. This means that digit 0 incremented matches
       * digit 1, or the other way around. */
      digits[2] = ((digits[0] + 1) % nTestStrips) == digits[1] ?
                  digits[0] : digits[1];
    } else {
        digits[2] = (digits[3] - 1 + nTestStrips) % nTestStrips;
    }
  }

  return digits[0] * 100 + digits[1] * 10 + digits[2];
}

/* Run set of tests for this day. */
void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
  if (day > 3) return; // only works for 3 days (digits) + one extra

  for (Bottle bottle : bottles) {
    int index = getTestStripIndexForDay(bottle, day, strips.size());
    TestStrip testStrip = strips.get(index);
    testStrip.addDropOnDay(day, bottle);
  }
}

/* Get strip that should be used on this bottle on this day. */
int getTestStripIndexForDay(Bottle bottle, int day, int nTestStrips) {
  int id = bottle.getId();
  switch (day) {
    case 0: return id /100;
    case 1: return (id % 100) / 10;
    case 2: return id % 10;
    case 3: return (id % 10 + 1) % nTestStrips;
    default: return -1;
  }
}

/* Get results that are positive for a particular day, excluding prior results. */
int getPositiveOnDay(ArrayList<TestStrip> testStrips, int day,
                     HashSet<Integer> previousResults) {
  for (TestStrip testStrip : testStrips) {
    int id = testStrip.getId();
    if (testStrip.isPositiveOnDay(day) && !previousResults.contains(id)) {
      return testStrip.getId();
    }
  }
  return -1;
}
```

It will take 10 days in the worst case to get a result with this approach.

Optimal Approach (7 days)

We can actually optimize this slightly more, to return a result in just seven days. This is of course the minimum number of days possible.

Notice what each test strip really means. It's a binary indicator for poisoned or unpoisoned. Is it possible to map 1000 keys to 10 binary values such that each key is mapped to a unique configuration of values? Yes, of course. This is what a binary number is.

We can take each bottle number and look at its binary representation. If there's a 1 in the `i`th digit, then we will add a drop of this bottle's contents to test strip `i`. Observe that 2^{10} is 1024, so 10 test strips will be enough to handle up to 1024 bottles.

We wait seven days, and then read the results. If test strip `i` is positive, then set bit `i` of the result value. Reading all the test strips will give us the ID of the poisoned bottle.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2    runTests(bottles, strips);
3    ArrayList<Integer> positive = getPositiveOnDay(strips, 7);
4    return setBits(positive);
5  }
6
7  /* Add bottle contents to test strips */
8  void runTests(ArrayList<Bottle> bottles, ArrayList<TestStrip> testStrips) {
9    for (Bottle bottle : bottles) {
10     int id = bottle.getId();
11     int bitIndex = 0;
12     while (id > 0) {
13       if ((id & 1) == 1) {
14         testStrips.get(bitIndex).addDropOnDay(0, bottle);
15       }
16       bitIndex++;
17       id >>= 1;
18     }
19   }
20 }
21
22 /* Get test strips that are positive on a particular day. */
23 ArrayList<Integer> getPositiveOnDay(ArrayList<TestStrip> testStrips, int day) {
24   ArrayList<Integer> positive = new ArrayList<Integer>();
25   for (TestStrip testStrip : testStrips) {
26     int id = testStrip.getId();
27     if (testStrip.isPositiveOnDay(day)) {
28       positive.add(id);
29     }
30   }
31   return positive;
32 }
33
34 /* Create number by setting bits with indices specified in positive. */
35 int setBits(ArrayList<Integer> positive) {
36   int id = 0;
37   for (Integer bitIndex : positive) {
38     id |= 1 << bitIndex;
39   }
40   return id;
41 }
```

This approach will work as long as $2^T >= B$, where T is the number of test strips and B is the number of bottles.

7

Solutions to Object-Oriented Design

7.1 Deck of Cards: Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

pg 127

SOLUTION

First, we need to recognize that a "generic" deck of cards can mean many things. Generic could mean a standard deck of cards that can play a poker-like game, or it could even stretch to Uno or Baseball cards. It is important to ask your interviewer what she means by generic.

Let's assume that your interviewer clarifies that the deck is a standard 52-card set, like you might see used in a blackjack or poker game. If so, the design might look like this:

```
1  public enum Suit {
2    Club (0), Diamond (1), Heart (2), Spade (3);
3    private int value;
4    private Suit(int v) { value = v; }
5    public int getValue() { return value; }
6    public static Suit getSuitFromValue(int value) { ... }
7  }
8
9  public class Deck <T extends Card> {
10   private ArrayList<T> cards; // all cards, dealt or not
11   private int dealtIndex = 0; // marks first undealt card
12
13   public void setDeckOfCards(ArrayList<T> deckOfCards) { ... }
14
15   public void shuffle() { ... }
16   public int remainingCards() {
17      return cards.size() - dealtIndex;
18   }
19   public T[] dealHand(int number) { ... }
20   public T dealCard() { ... }
21 }
22
23 public abstract class Card {
24   private boolean available = true;
25
26   /* number or face that's on card - a number 2 through 10, or 11 for Jack, 12 for
27    * Queen, 13 for King, or 1 for Ace */
28   protected int faceValue;
29   protected Suit suit;
```

```
30
31    public Card(int c, Suit s) {
32       faceValue = c;
33       suit = s;
34    }
35
36    public abstract int value();
37    public Suit suit() { return suit; }
38
39    /* Checks if the card is available to be given out to someone */
40    public boolean isAvailable() { return available; }
41    public void markUnavailable() { available = false; }
42    public void markAvailable() { available = true; }
43 }
44
45 public class Hand <T extends Card> {
46    protected ArrayList<T> cards = new ArrayList<T>();
47
48    public int score() {
49       int score = 0;
50       for (T card : cards) {
51          score += card.value();
52       }
53       return score;
54    }
55
56    public void addCard(T card) {
57       cards.add(card);
58    }
59 }
```

In the above code, we have implemented `Deck` with generics but restricted the type of `T` to `Card`. We have also implemented `Card` as an abstract class, since methods like `value()` don't make much sense without a specific game attached to them. (You could make a compelling argument that they should be implemented anyway, by defaulting to standard poker rules.)

Now, let's say we're building a blackjack game, so we need to know the value of the cards. Face cards are 10 and an ace is 11 (most of the time, but that's the job of the Hand class, not the following class).

```
1  public class BlackJackHand extends Hand<BlackJackCard> {
2     /* There are multiple possible scores for a blackjack hand, since aces have
3      * multiple values. Return the highest possible score that's under 21, or the
4      * lowest score that's over. */
5     public int score() {
6        ArrayList<Integer> scores = possibleScores();
7        int maxUnder = Integer.MIN_VALUE;
8        int minOver = Integer.MAX_VALUE;
9        for (int score : scores) {
10          if (score > 21 && score < minOver) {
11             minOver = score;
12          } else if (score <= 21 && score > maxUnder) {
13             maxUnder = score;
14          }
15       }
16       return maxUnder == Integer.MIN_VALUE ? minOver : maxUnder;
17    }
18
```

```
19    /* return a list of all possible scores this hand could have (evaluating each
20     * ace as both 1 and 11 */
21    private ArrayList<Integer> possibleScores() { ... }
22
23    public boolean busted() { return score() > 21; }
24    public boolean is21() { return score() == 21; }
25    public boolean isBlackJack() { ... }
26 }
27
28 public class BlackJackCard extends Card {
29    public BlackJackCard(int c, Suit s) { super(c, s); }
30    public int value() {
31       if (isAce()) return 1;
32       else if (faceValue >= 11 && faceValue <= 13) return 10;
33       else return faceValue;
34    }
35
36    public int minValue() {
37       if (isAce()) return 1;
38       else return value();
39    }
40
41    public int maxValue() {
42       if (isAce()) return 11;
43       else return value();
44    }
45
46    public boolean isAce() {
47       return faceValue == 1;
48    }
49
50    public boolean isFaceCard() {
51       return faceValue >= 11 && faceValue <= 13;
52    }
53 }
```

This is just one way of handling aces. We could, alternatively, create a class of type `Ace` that extends `BlackJackCard`.

An executable, fully automated version of blackjack is provided in the downloadable code attachment.

7.2 **Call Center:** Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.

pg 127

SOLUTION

All three ranks of employees have different work to be done, so those specific functions are profile specific. We should keep these things within their respective class.

There are a few things which are common to them, like address, name, job title, and age. These things can be kept in one class and can be extended or inherited by others.

Finally, there should be one `CallHandler` class which would route the calls to the correct person.

Note that on any object-oriented design question, there are many ways to design the objects. Discuss the trade-offs of different solutions with your interviewer. You should usually design for long-term code flexibility and maintenance.

We'll go through each of the classes below in detail.

`CallHandler` represents the body of the program, and all calls are funneled first through it.

```
1  public class CallHandler {
2    /* 3 levels of employees: respondents, managers, directors. */
3    private final int LEVELS = 3;
4
5    /* Initialize 10 respondents, 4 managers, and 2 directors. */
6    private final int NUM_RESPONDENTS = 10;
7    private final int NUM_MANAGERS = 4;
8    private final int NUM_DIRECTORS = 2;
9
10   /* List of employees, by level.
11    * employeeLevels[0] = respondents
12    * employeeLevels[1] = managers
13    * employeeLevels[2] = directors
14    */
15   List<List<Employee>> employeeLevels;
16
17   /* queues for each call's rank */
18   List<List<Call>> callQueues;
19
20   public CallHandler() { ... }
21
22   /* Gets the first available employee who can handle this call.*/
23   public Employee getHandlerForCall(Call call) { ... }
24
25   /* Routes the call to an available employee, or saves in a queue if no employee
26    * is available. */
27   public void dispatchCall(Caller caller) {
28     Call call = new Call(caller);
29     dispatchCall(call);
30   }
31
32   /* Routes the call to an available employee, or saves in a queue if no employee
33    * is available. */
34   public void dispatchCall(Call call) {
35     /* Try to route the call to an employee with minimal rank. */
36     Employee emp = getHandlerForCall(call);
37     if (emp != null) {
38       emp.receiveCall(call);
39       call.setHandler(emp);
40     } else {
41       /* Place the call into corresponding call queue according to its rank. */
42       call.reply("Please wait for free employee to reply");
43       callQueues.get(call.getRank().getValue()).add(call);
44     }
45   }
```

```
46
47    / *An employee got free. Look for a waiting call that employee can serve. Return
48     * true if we assigned a call, false otherwise. */
49    public boolean assignCall(Employee emp) { ... }
50 }
```

`Call` represents a call from a user. A call has a minimum rank and is assigned to the first employee who can handle it.

```
1  public class Call {
2     / *Minimal rank of employee who can handle this call. */
3     private Rank rank;
4
5     / *Person who is calling. */
6     private Caller caller;
7
8     / *Employee who is handling call. */
9     private Employee handler;
10
11    public Call(Caller c) {
12       rank = Rank.Responder;
13       caller = c;
14    }
15
16    / *Set employee who is handling call. */
17    public void setHandler(Employee e) { handler = e; }
18
19    public void reply(String message) { ... }
20    public Rank getRank() { return rank; }
21    public void setRank(Rank r) { rank = r; }
22    public Rank incrementRank() { ... }
23    public void disconnect() { ... }
24 }
```

`Employee` is a super class for the `Director`, `Manager`, and `Respondent` classes. It is implemented as an abstract class since there should be no reason to instantiate an `Employee` type directly.

```
1  abstract class Employee {
2     private Call currentCall = null;
3     protected Rank rank;
4
5     public Employee(CallHandler handler) { ... }
6
7     / *Start the conversation */
8     public void receiveCall(Call call) { ... }
9
10    / *the issue is resolved, finish the call */
11    public void callCompleted() { ... }
12
13    / *The issue has not been resolved. Escalate the call, and assign a new call to
14     * the employee. */
15    public void escalateAndReassign() { ... }
16
17    / *Assign a new call to an employee, if the employee is free. */
18    public boolean assignNewCall() { ... }
19
20    / *Returns whether or not the employee is free. */
21    public boolean isFree() { return currentCall == null; }
22
```