Object-Oriented Design

Concepts *and Algorithms: Solutions*

8.1 *Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.*

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
27               * for Jack, 12 for 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
38              public Suit suit() { return suit; }
39
40              /* Checks if the card is available to be given out to someone */
41              public boolean isAvailableQ { return available; }
42              public void markUnavailable() { available = false; }
43
44              public void markAvailable() { available = true; }
45      }
46
47      public class Hand <T extends Card> {
48              protected ArrayList<T> cards = new ArrayList<T>();
49
50              public int score() {
51                      int score = 0;
52                      for (T card : cards) {
53                              score += card.value();
54                      }
55              return score;
56              }
57
58              public void addCard(T card) {
59                      cards.add(card);
60              }
61      }
```

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,
3                * since aces have multiple values. Return the highest possible
4                * score that's under 21, or the 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
20               * (evaluating each 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 BlackDackCard(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 (IsAceQ) 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.

8.2 *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.*

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 is implemented as a singleton class. It represents the body of the program, and all calls are funneled first through it.

```
1   public class CallHandler {
2       private static CallHandler instance;
3
4       /* 3 levels of employees: respondents, managers, directors. */
5       private final int LEVELS = 3;
6
7   /* Initialize 10 respondents, 4 managers, and 2 directors. */
8   private final int NUM_RESPONDENTS = 10;
9   private final int NUM_MANAGERS = 4;
10  private final int NUM_DIRECTORS = 2;
11
12  /* List of employees, by level.
13   *employeeLevels[0] = respondents
14   *employeeLevels[1] = managers
15   *employeeLevels[2] = directors
16   */
17  List<List<Employee>> employeeLevels;
18
19  */ queues for each call's rank */
20  List<List<Call>> callQueues;
21
22  protected CallHandler() { ... }
23
24  /* Get instance of singleton class. */
25  public static CallHandler getInstance() {
26      if (instance == null) instance = new CallHandler();
27      return instance;
28  }
```

```
29
30              /* Gets the first available employee who can handle this call. */
31              public Employee getHandlerForCall(Call call) { ... }
32
33              /* Routes the call to an available employee, or saves in a queue
34               * if no employee available. */
35              public void dispatchCall(Caller caller) {
36                      Call call = new Call(caller);
37                      dispatchCall(call);
38              }
39
40              /* Routes the call to an available employee, or saves in a queue
41               * if no employee available. */
42              public void dispatchCall(Call call) {
43                      /* Try to route the call to an employee with minimal rank. */
44                      Employee emp = getHandlerForCall(call);
45                      if (emp != null) {
46                              emp.receiveCall(call);
47                              call . setHandler (emp) ;
48                      } else {
49                              /* Place the call into corresponding call queue according
50                               * to its rank. */
51                              call.reply ("Please wait for free employee to reply");
52                              callQueues.get (call.getRank().getValue()).add(call);
53                      }
54              }
55
56              /* An employee got free. Look for a waiting call that emp. can
57               * serve. Return true if we assigned a call, false otherwise. */
58              public boolean assignCall(Employee emp) { ... }
59      }
```

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() { }
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
14               * assign a new call to the employee. */
15              public void escalateAndReassign() { ... }
16              }
17
18              Assign a new call to an employee, if the employee is free. */
```

```
19              public boolean assignNewCall() { ... }
20
21              /* Returns whether or not the employee is free. */
22              public boolean isFree() { return currentCall == null; }
23
24              public Rank getRank() { return rank; }
25      }
26
```

The Respondent, Director, and Manager classes are now just simple extensions of the Employee class.

```
1               class Director extends Employee {
2               public Director() {
3                       rank = Rank.Director;
4               }
5       }
6
7       class Manager extends Employee {
8               public Manager() {
9                       rank = Rank.Manager;
10              }
11      }
12
13      class Respondent extends Employee {
14              public Respondent() {
15                      rank = Rank.Responder;
16              }
17      }
```

This is just one way of designing this problem. Note that there are many other ways that are equally good.

This may seem like an awful lot of code to write in an interview, and it is. We've been much more thorough here than you would need. In a real interview, you would likely be much lighter on some of the details until you have time to fill them in.

8.3 *Design a musical jukebox using object-oriented principles.*

SOLUTION

In any object-oriented design question, you first want to start off with asking your interviewer some questions to clarify design constraints. Is this jukebox playing CD? Records? MP3s? Is it a simulation on a computer, or is it supposed to represent a physical jukebox? Does it take money, or is it free? And if it takes money, which currency? And does it deliver change?

Unfortunately, we don't have an interviewer here that we can have this dialogue with. Instead, we'll make some assumptions. We'll assume that the jukebox is a computer simulation that closely mirrors physical jukeboxes, and we'll assume that it's free.

Now that we have that out of the way, we'll outline the basic system components.

- Jukebox
- CD
- Song
- Artist
- Playlist
- Display (displays details on the screen)

Now, let’s break this down further and think about the possible actions.

- Playlist creation (includes add, delete, and shuffle)
- CD selector
- Song selector
- Queuing up a song
- Get next song from playlist

A user also can be introduced:

- Adding
- Deleting
- Credit information

Each of the main system components translates roughly to an object, and each action translates to a method. Let's walk through one potential design.

The Jukebox class represents the body of the problem. Many of the interactions between the components of the system, or between the system and the user, are channeled through here.

```
1       public class Jukebox {
2               private CDPlayer cdPlayer;
```

```
3               private User user;
4               private Set<CD> cdCollection;
5               private SongSelector ts;
6
7               public Jukebox(CDPlayer cdPlayer, User user,
8                                       Set<CD> cdCollection, SongSelector  ts) {
9                       …
10              }
11
12              public Song getCurrentSong() {
13                      return ts.getCurrentSong();
14              }
15
16              public void setUser (User u) {
17                      this. user = u;
18              }
19      }
```

Like a real CD player, the CDPlayer class supports storing just one CD at a time. The CDs that are not in play are stored in the jukebox.

```
1       class CDPlayer {
2               private Playlist p;
3               private CD c;
4
5               /* Constructors. */
6               public CDPlayer(CD c, Playlist p) { ... }
7               public CDPlayer(Playlist p) { this.p = p; }
8               public CDPlayer(CD c) { this.c = c; }
9
10              /* Play song */
11              public void playSong(Song s) { ... }
12
13              /* Getters and setters */
14              public Playlist getPlaylist() { return p; }
15              public void setPlaylist(Playlist p) { this.p = p; }
16
17              public CD getCD() { return c; }
18              public void setCD(CD c) { this.c = c; }
19      }
```

The Playlist manages the current and next songs to play. It is essentially a wrapper class for a queue and offers some additional methods for convenience.

```
1       public class Playlist {
2               private Song song;
3               private Queue<Song> queue;
4               public Playlist(Song song, Queue<Song> queue) {
5                       …
6               }
7               public Song getNextSToPlayQ {
8                       return queue. peek();
9               }
10              public void queueUpSong(Song s) {
11                      queue. add(s);
12              }
13      }
```

The classes for CD, Song, and User are all fairly straightforward. They consist mainly of member variables and getters and setters.

```
1       public class CD {
2               /* data for id, artist, songs, etc */
3       }
4
5       public class Song {
6               /* data for id, CD (could be null), title, length, etc */
7       }
8
9       public class User {
10              private String name;
11              public String getName() { return name; }
12              public void setName(String name) { this.name }
13              public long getlD() { return ID; }
14              public void setID(long ID) { ID = iD; }
15              private long ID;
16              public User(String name, long iD) { … }
17              public User getUser() { return this; }
18              public static User addUser(String name, long iD) { ... }
19      }
```

This is by no means the only "correct" implementation. The interviewer's responses to initial question, as well as other constraints, will shape the design of the jukebox classes.

8.4 *Design a parking lot using object-oriented principles.*

SOLUTION

The wording of this question is vague, just as it would be in an actual interview. This requires you to have a conversation with your interviewer about what types of vehicles it can support, whether the parking lot has multiple levels, and so on.

For our purposes right now, we'll make the following assumptions. We made these specific assumptions to add a bit of complexity to the problem without adding too much. If you made different assumptions, that's totally fine.

- The parking lot has multiple levels. Each level has multiple rows of spots.
- The parking lot can park motorcycles, cars, and buses.
- The parking lot has motorcycle spots, compact spots, and large spots.
- A motorcycle can park in any spot.
- A car can park in either a single compact spot or a single large spot.
- A bus can park in five large spots that are consecutive and within the same row. It cannot park in small spots.

In the below implementation, we have created an abstract class Vehicle, from which Car, Bus, and Motorcycle inherit. To handle the different parking spot sizes, we have just one class ParkingSpot which has a member variable indicating the size.

```
1       public enum VehicleSize { Motorcycle, Compact, Large }
2
3       public abstract class Vehicle {
4               protected ArrayList<ParkingSpot> parkingSpots =
5                       new ArrayList<ParkingSpot>();
6               protected String licensePlate;
7               protected int spotsNeeded;
8               protected VehicleSize size;
9
10              public int getSpotsNeeded() { return spotsNeeded; }
11              public VehicleSize getSize() { return size; }
12
13              /* Park vehicle in this spot (among others, potentially) */
14              public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
15
16              /* Remove car from spot, and notify spot that it's gone */
```

```
17              public void clearSpots() { ... }
18
19              /* Checks if the spot is big enough for the vehicle (and is
20               * available). This compares the SIZE only. It does not check if it
21               * has enough spots. */
22              public abstract boolean canFitInSpot(ParkingSpot spot);
23      }
24
25      public class Bus extends Vehicle {
26              public Bus() {
27                      spotsNeeded = 5;
28                      size = VehicleSize.Large;
29              }
30
31              /* Checks if the spot is a Large. Doesn't check num of spots */
32              public boolean canFitInSpot(ParkingSpot spot) { ... }
33      }
34
35      public class Car extends Vehicle {
36              public Car() {
37                      spotsNeeded = 1;
38                      size = VehicleSize.Compact;
39              }
40
41              /* Checks if the spot is a Compact or a Large. */
42              public boolean canFitlnSpot(ParkingSpot spot) { ... }
43      }
44
45      public class Motorcycle extends Vehicle {
46              public Motorcycle() {
47                      spotsNeeded = 1;
48                      size = VehicleSize.Motorcycle;
49              }
50
51              public boolean canFitInSpot(ParkingSpot spot) { … }
52      }
```

The ParkingLot class is essentially a wrapper class for an array of Levels. By implementing it this way, we are able to separate out logic that deals with actually finding free spots and parking cars out from the broader actions of the ParkingLot. If we

didn’t do it this way, we would need to hold parking spots in some sort of double array (or hash table which maps from a level number to the list of spots). It's cleaner to just separate ParkingLot from Level.

```
1       public class ParkingLot {
2               private Level[ ] levels;
3               private final int NUM_LEVELS = 5;
4
5               public ParkingLot() { ... }
6
7               /* Park the vehicle in a spot (or multiple spots).
8                * Return false if failed. */
9               public boolean parkVehicle(Vehicle vehicle) { ... }
10      }
11
12      /* Represents a level in a parking garage */
13      public class Level {
14              private int floor;
15              private ParkingSpot[ ] spots;
16              private int availableSpots = 0; // number of free spots
17              private static final int SPOTS_PER_ROW = 10;
18
19              public Level(int flr, int numberSpots) { ... }
20
21              public int availableSpots() { return availableSpots; }
22
23              /* Find a place to park this vehicle. Return false if failed. */
24              public boolean parkVehicle(Vehicle vehicle) { ... }
25
26              /* Park a vehicle starting at the spot spotNumber, and
27               * continuing until vehicle.spotsNeeded. */
28              private boolean parkStartingAtSpot(int num, Vehicle v) { ... }
29
30              /* Find a spot to park this vehicle. Return index of spot, or -1
31               * on failure. */
32              private int findAvailableSpots(Vehicle vehicle) { ... }
33
34              /* When a car was removed from the spot, increment
35              * availableSpots */
36              public void spotFreed() { availableSpots++; }
```

```
37      }
```

The parkingSpot is implemented by having just a variable which represents the size of the spot. We could have implemented this by having classes for LargeSpot, CompactSpot, and MotorcycleSpot which inherit from ParkingSpot, but this is probably overkill. The spots probably do not have different behaviors, other than their sizes.

```
1       Public class ParkingSpot {
2               private Vehicle vehicle;
3               private VehicleSize spotSize;
4               private int row;
5               private int spotNumber;
6               private Level level;
7
8               public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}
9
10              public boolean isAvailable() { return vehicle == null; }
11
12              /* Check if the spot is big enough and is available */
13              public boolean canFitVehicle(Vehicle vehicle) { ... }
14
15              /* Park vehicle in this spot. */
16              public boolean park(Vehicle v) { ... }
17
18              public int getRow() { return row; }
19              public int getSpotNumber() { return spotNumber; }
20
21              /* Remove vehicle from spot, and notify level that a new spot is
22               * available */
23              public void removeVehicle() { ... }
24      }
```

A full implementation of this code, including executable test code, is provided in the downloadable code attachment.

8.5 *Design the data structures for an online book reader system.*

SOLUTION

Since the problem doesn't describe much about the functionality, let's assume we want to design a basic online reading system which provides the following functionality:

- User membership creation and extension.
- Searching the database of books.
- Reading a book.
- Only one active user at a time
- Only one active book by this user.

To implement these operations we may require many other functions, like get, set, update, and so on. The objects required would likely include User, Book, and Library.

The class OnlineReaderSystem represents the body of our program. We would implement the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty. Instead, we've chosen to tear off these components into Library, UserManager, and Display classes.

```
1       public class OnlineReaderSystem {
2               private Library library;
3               private UserManager userManager;
4               private Display display;
5
6               private Book activeBook;
7               private User activeUser;
8
9               public OnlineReaderSystem() {
10                      userManager = new UserManager();
11                      library = new Library();
12                      display = new Display();
13              }
14
15              public Library getLibrary() { return library; }
16              public UserManager getUserManager() { return userManager; }
17              public Display getDisplay() { return display; }
18
19              public Book getActiveBook() { return activeBook; }
20              public void setActiveBook(Book book) {
21                      activeBook = book;
22                      display. displayBook(book);
23              }
```

```
24
25              public User getActiveUserQ { return activeUser; }
26              public void setActiveUser(User user) {
27                      activeUser = user;
28                      display. displayUser( user);
29              }
30      }
```

We then implement separate classes to handle the user manager, the library, and the display components.

```
1       public class Library {
2               private Hashtable<Integer, Book> books;
3
4               public Book addBook(int id, String details) {
5                       if (books. containsKey(id)) {
6                               return null;
7                       }
8                       Book book = new Book(id, details);
9                       books. put(id, book);
10                      return book;
11              }
12
13              public boolean remove(Book b) { return remove(b.getID()); }
14              public boolean remove(int id) {
15                      if (!books. containsKey(id)) {
16                              return false;
17                      }
18                      books. remove (id);
19                      return true;
20              }
21
22              public Book find (int id) {
23                      return books. get (id);
24              }
25      }
26
27      public class UserManager {
28              private Hashtable< Integer, User> users;
29
30              public User addUser(int id, String details, int accountType) {
```

```
31                      if (users. containsKey(id)) {
32                              return null;
33                      }
34                      User user = new User(id, details, accountType);
35                      users.put(id, user);
36                      return user;
37              }
38
39              public boolean remove(User u) {
40                      return remove(u.getlD());
41              }
42
43              public boolean remove(int id) {
44                      if (! users. containsKey(id)) {
45                              return false;
46                      }
47                      users. remove(ld);
48                      return true;
49              }
50
51              public User find(int id) {
52                      return users.get(id);
53              }
54      }
55
56      public class Display {
57              private Book activeBook;
58              private User activeUser;
59              private int pageNumber = 0;
60
61              public void displayUser(User user) {
62                      activeUser = user;
63                      refreshUsername();
64              }
65
66              public void displayBook(Book book) {
67                      pageNumber =0;
68                      activeBook = book;
69
70                      refreshTitle();
```

```
71                      refreshDetails();
72                      refreshPage();
73              }
74
75              public void turnPageForward() {
76                      pageNumber++;
77                      refreshPage();
78              }
79
80              public void turnPageBackward() {
81                      pageNumber- -;
82                      refreshPage();
83              }
84
85              public void refreshUsername() { /* updates username display */ }
86              public void refreshTitle() { /* updates title display */ }
87              public void refreshDetails() { /* updates details display */ }
88              public void refreshPage() { /* updated page display */ }
89      }
```

The classes for User and Book simply hold data and provide little true functionality.

```
1       public class Book {
2               private int bookId;
3               private String details;
4
5               public Book(int id, String det) {
6                       bookId = id;
7                       details = det;
8               }
9
10              public int getID() { return bookId; }
11              public void setID(lnt id) { bookId = id; }
12              public String getDetails() { return details; }
13              public void setDetails(String d) { details = d; }
14      }
15
16      public class User {
17              private int userId;
18              private String details;
19              private int accountType;
```

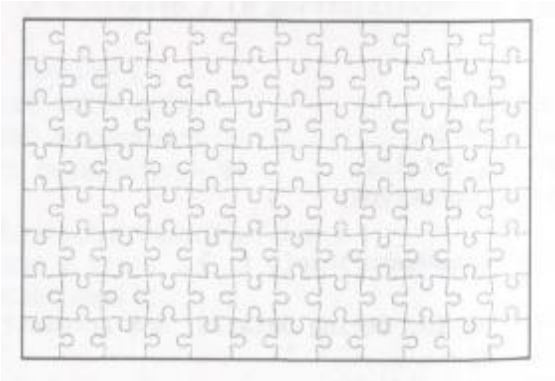
```
20
21              public void renewMembership() { }
22
23              public User(int id, String details, int accountType) {
24                      userId = id;
25                      this.details = details;
26                      this.accountType = accountType;
27              }
28
29              /* Getters and setters */
30              public int getID() { return userId; }

31              public void setID(int id) { userId = id; }
32              public String getDetails() {
33                      return details;
34              }
35
36              public void setDetails(String details) {
37                      this.details = details;
38              }
39              public int getAccountType() { return accountType; }
40              public void setAccountType(int t) { accountType = t; }
41      }
```

The decision to tear off user management, library, and display into their own classes, when this functionality could have been in the general OnlineReaderSystem class, is an interesting one. On a very small system, making this decision could make the system overly complex. However, as the system grows, and more and more functionality gets added to OnlineReaderSystem, breaking off such components prevents this main class from getting overwhelmingly lengthy.

8.6 *Implement a jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a fitsWith method which, when passed two puzzle pieces, returns true if the two pieces belong together.*

SOLUTION

We will assume that we have a traditional, simple jigsaw puzzle. The puzzle is grid-like, with rows and columns. Each piece is located in a single row and column and has four edges. Each edge comes in one of three types: inner, outer, and flat. A corner piece, for example, will have two flat edges and two other edges, which could be inner or outer.



As we solve the jigsaw puzzle (manually or algorithmically), we'll need to store the position of each piece. We could think about the position as absolute or relative:

- Absolute Position: "This piece is located at position (12, 23)." Absolute position would belong to the Piece class itself and would include an orientation as well.
- Relative Position: “I don’t know where this piece is actually located, but I know that it is next to this other piece.” The relative position would belong to the Edge class.

For our solution, we will use only the relative position, by adjoining edges to neighboring edges.

A potential object-oriented design looks like the following:

```
1       class Edge {
2               enum Type { inner, outer, flat }
3               Piece parent;
4               Type type;
5               int index; // Index into Piece.edges
6               Edge attached_to; // Relative position
7
8               /* See Algorithm section. Returns true if the two pieces
9                * should be attached to each other. */
10              boolean fitswith(Edge edge) { ... };
11      }
```

```
12
13      class Piece {
14              Edge[ ] edges;
15              boolean isCorner() { ... }
16      }
17
18      class Puzzle {
19              Piece[ ] pieces; /* Remaining pieces left to put away. */
20              Piece[ ][ ] solution;
21
22              /* See algorithm section. */
23              Edge[ ] inners, outers, flats;
24              Piece[ ] corners;
25
26              /* See Algorithm section. */
27              void sort() { ... }
28              void solve() { ... }
29      }
```

Algorithm to Solve the Puzzle

We will sketch this algorithm using a mix of pseudocode and real code.

Just as a kid might in solving a puzzle, we'll start with the easiest pieces first: the corners and edges. We can easily search through all the pieces to find just the edges. While we're at it though, it probably makes sense to group all the pieces by their edge types.

```
1               void sort() {
2                       for each Piece p in pieces { ;
3                               if (P has two flat edges) then add p to corners
4                       for each edge in p.edges {
5                               if edge is inner then add to inners
6                               if edge is outer then add to outers
7                       }
8               }
9       }
```

We now have a quicker way to zero in on potential matches for any given edge. We then go through the puzzle, line by line, to match pieces.

The solve method, implemented below, operates by picking an arbitrary start with. It then finds an open edge on the corner and tries to match it to an open piece. When it finds a match, it does the following:

1. Attaches the edge.
2. Removes the edge from the list of open edges.
3. Finds the next open edge.

The next open edge is defined to be the one directly opposite the current edge, if it is available. If it is not available, then the next edge can be any other edge. This will cause the puzzle to be solved in a spiral-like fashion, from the outside to the inside.

The spiral comes from the fact that the algorithm always moves in a straight line, whenever possible. When we reach the end of the first edge, the algorithm moves to the only available edge on that corner piece—a 90-degree turn. It continues to take 90-degree turns at the end of each side until the entire outer edge of the puzzle is completed. When that last edge piece is in place, that piece only has one exposed edge remaining, which is again a 90-degree turn. The algorithm repeats itself for subsequent rings around the puzzle, until finally all the pieces are in place.

This algorithm is implemented below with Java-like pseudocode.

```
1       public void solve() {
2               /* Pick any corner to start with */
3               Edge currentEdge = getExposedEdge(corner[0]);
4
5               /* Loop will iterate in a spiral like fashion until the puzzle
6               * is full. */
7               while (currentEdge != null) {
8                       /* Match with opposite edges. Inners with outers, etc. */
9                       Edge[ ] opposites = currentEdge.type == inner ?
10                                              outers : inners;
11                      for each Edge fittingEdge in opposites {
12                              if (currentEdge.fitsWith(fittingEdge)) {
13                                      attachEdges(currentEdge, fittingEdge); //attach edge
14                                      removeFromlist(currentEdge);
15                                      removeFromList(fittingEdge);
16
17                                      /* get next edge */
18                                      currentEdge = nextExposedEdge(fittingEdge);
```

```
19                                      break; // Break out of inner loop. Continue in outer.
20                              }
21                      }
22              }
23      }
24
25      public void removeFromList(Edge edge) {
26              (edge.type == flat) return;
27              Edge[ ] array = currentEdge.type == inner ? inners : outers;
28              array.remove(edge);
29      }
30
31      /* Return the opposite edge if possible. Else, return any exposed
32       * edge. */
33      public Edge nextExposedEdge(Edge edge) {
34              int next_index = (edge.index + 2) % 4; // Opposite edge
35              Edge next_edge = edge.parent.edges[next_index];
36              if isExposed(next_edge) {
37                      return next_edge;
38              }
39              return getExposedEdge(edge.parent);
40      }
41
42      public Edge attachEdges(Edge e1, Edge e2) {
43              el.attached_to = e2;
44              e2.attached_to = e1;
45      }
46
47      public Edge isExposed(Edge e1) {
48              return edge.type != flat && edge.attached_to == null;
49      }
50
51      public Edge getExposedEdge(Piece p) {
52              for each Edge edge in p.edges {
53                      if (isExposed(edge)) {
54                              return edge;
55                      }
56              }
57              return null;
58      }
```

For simplicity, we're represented inners and outers as an Edge array. This is actually not a great choice, since we need to add and removed elements from it frequently. If we were writing a real code, we would probably want to implement these variables as linked lists.

Writing the full code for this problem in an interview would be far, far too much work. More likely, you would be asked to just sketch out the code.

8.7 *Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?*

SOLUTION

Designing a chat server is a huge project, and it is certainly far beyond the scope of what could be completed in an interview. After all, teams of many people spend months or years creating a chat server. Part of your job, as a candidate, is to focus on an aspect of the problem that is reasonably broad, but focused enough that you could accomplish it during an interview. It need not match real life exactly, but it should be a fair representation of an actual implementation.

For our purposes, we'll focus on the core user management and conversation aspects: adding a user, creating a conversation, updating one's status, and so on. In the interest of time and space, we will not go into the networking aspects of the problem, or how the data actually gets pushed out to the clients.

We will assume that "friending" is mutual; I am only your contact if you are mine. Our chat system will support both group chat and one-on-one (private) chats. We will not worry about voice chat, video chat, or file transfer.

What specific actions does it need to support?

This is also something to discuss with your interviewer, but here are some ideas:

- Signing online and offline.
- Add requests (sending, accepting, and rejecting).
- Updating a status message.

- Creating private and group chats.
- Adding new messages to private and group chats.

This is just a partial list. If you have more time, you can add more actions.

What can we learn about these requirements?

We must have a concept of users, add request status, online status, and messages.

What are the core components of the system?

The system would likely consist of a database, a set of clients, and a set of servers. We won't include these parts in our object-oriented design, but we can discuss the overall view of the system.

The database will be used for more permanent storage, such as the user list or chat archives. A SQL database is a good bet, or, if we need more scalability, we could potentially use BigTable or a similar system.

For communication between the client and servers, using XML will work well. Although it’s not the most compressed format (and you should point this out to your interviewer), it’s nice because it's easy for both computers and humans to read. Using XML will make your debugging efforts easier—and that matters a lot.

The server will consist of a set of machines. Data will be divided up across machines, requiring us to potentially hop from machine to machine. When possible, we will try to replicate some data across machines to minimize the lookups. One major design constraint here is to prevent having a single point of failure. For instance, if one machine controlled all the user sign-ins, then we'd cut off millions of users potentially if a single machine lost network connectivity.

What are the key objects and methods?

The key objects of the system will be a concept of users, conversations, and status messages. We've implemented a User-Management class. If we were looking more at the networking aspects of the problem, or a different component, we might have instead dived into those objects.

```
1       /* UserManager serves as a central place for core user actions. */
1       public class UserManager {
2               private static UserManager instance;
3               /* maps from a user id to a user */
```

```
4               private HashMap<Integer, User> usersById;
5
6               /* maps from an account name to a user */
7               private HashMap<String, User> usersByAccountName;
8
9               /* maps from the user id to an online user */
10              private HashMap<Integer, User> onlineUsers;
11
12              public static UserManager getInstance() {
13                      if (instance == null) instance = new UserManager();
14                      return instance;
15              }
16
17              public void addUser(User fromUser, String toAccountName) { ... }
18              public void approveAddRequest(AddRequest req) { ... }
19              public void rejectAddRequest(AddRequest req) { ... }
20              public void userSignedOn(String accountName) { ... }
21              public void userSignedOff(String accountName) { ... }
22      }
```

The method receivedAddRequest, in the User class, notifies User B that User A has requested to add him. User B approves or rejects the request (via UserManager. approvedAddRequest or rejectAddRequest), and the UserManager takes care of adding the users to each other's contact lists.

The method sentAddRequest in the User class is called by User-Manager to add an AddRequest to User A's list of requests. So the flow is:

1. User A clicks "add user" on the client, and it gets sent to the server.
2. User A calls requestAddUser(User B).
3. This method calls UserManager. addUser.
4. UserManager calls both User A.sentAddRequest and User B.receivedAddRequest.

Again, this is just *one* way of designing these interactions. It is not the only way or even the only "good" way.

```
1       public class User {
2               private int id;
3               private UserStatus status null;
4
```

```
5           /* maps from the other participant's user id to the chat */
6           private HashMap<Integer, PrlvateChat> privateChats;
7
8           /* maps from the group chat id to the group chat */
9           private ArrayList<GroupChat> groupChats;
10
11          /* maps from the other person's user id to the add request */
12          private HashMap<Integer, AddRequest> receivedAddRequests;
13
14          /* maps from the other person's user id to the add request */
15          private HashMap<Integer, AddRequest> sentAddRequests;
16
17          /* maps from the user id to the add request */
18          private HashMap<Integer, User> contacts;
19
20          private String accountName;
21          private String fullName;
22
23          public User(int id, String accountName, String fullName) { … }
24          public boolean sendMessageToUser(User to, String content){ … }
25          public boolean sendMessageToGroupChat(int id, String cnt){ … }
26          public void setStatus(UserStatus status) { ... }
27          public UserStatus getStatus() { ... }
28          public boolean addContact(User user) { ... }
29          public void receivedAddRequest(AddRequest req) { … }
30          public void sentAddRequest (AddRequest req) { ... }
31          public void removeAddRequest(AddRequest req) { ... }
32          public void requestAddUser(String accountName) { … }
33          public void addConversation(PrivateChat conversation) { … }
34          public void addConversation(GroupChat conversation) { … }
35          public int getId() { ... }
36          public String getAccountName() { ... }
37          public String getFullName() { ... }
38      }
```

The Conversation class is implemented as an abstract class, since all Conversations must be either a GroupChat or a PrivateChat, and since these two classes each have their own functionality.

```
1       public abstract class Conversation {
```

```
2               protected ArrayList<User> participants;
3               protected int id;
4               protected ArrayList<Message> messages;
5
6               public ArrayList<Message> getMessages() { … }
7               public boolean addMessage(Message m) { … }
8               public int getId() { ... }
9       }
10
11      public class GroupChat extends Conversation {
12              public void removeParticipant(User user) { ... }
13              public void addParticipant(User user) { ... }
14      }
15
16      public class PrivateChat extends Conversation {
17              public PrivateChat(User user1, User user2) { ...
18              public User getOtherParticipant(User primary) { … }
19      }
20
21      public class Message {
22              private String content;
23              private Date date;
24              public Message(String content, Date date) { ... }
25              public String getContent() { ... }
26              public Date getDate() { … }
27      }
```

AddRequest and UserStatus are simple classes with little functionality. Their main purpose is to group data that other classes will act upon.

```
1       public class AddRequest {
2               private User fromUser;
3               private user toUser;
4               private Date date;
5               Requeststatus status;
6
7               public AddRequest(User from, User to, Date date) { … }
8               public RequestStatus getStatus() { … }
9               public User getFromUser() { ... }
10              public User getToUser() { ... }
11              public Date getDate() { ... }
```

```
12      }
13
14      public class UserStatus {
15              private String message;
16              private UserStatusType type;
17              public UserStatus(UserStatusType type, String message) f
18              public UserStatusType getStatusTypeQ { ... }
19              public String getMessageQ { ... }
20      }
21
22      public enum UserStatusType {
23              Offline, Away, Idle, Available, Busy
24      }
25
26      public enum RequestStatus {
27              Unread, Read, Accepted, Rejected
28      }
```

The downloadable code attachment provides a more detailed look at these methods, including implementations for the methods shown above.

What problems would be the hardest to solve (or the most interesting)?

The following questions may be interesting to discuss with your interviewer further.

Q 1: How do we know if someone is online—I mean, really, really know?

While we would like users to tell us when they sign off, we can't know for sure. A user's connection might have died, for example. To make sure that we know when a user has signed off, we might try regularly pinging the client to make sure it's still there.

Q2: How do we deal with conflicting information?

We have some information stored in the computer's memory and some in the database. What happens if they get out of sync? Which one is "right"?

Q3: How do we make our server scale?

While we designed out chat server without worrying—too much—about scalability, in real life this would be a concern. We'd need to split our data across many servers, which would increase our concern about out-of-sync data.

Q4: How we do prevent denial of service attacks?
Clients can push data to us—what if they try to DOS (denial of service) us? How do we prevent that?

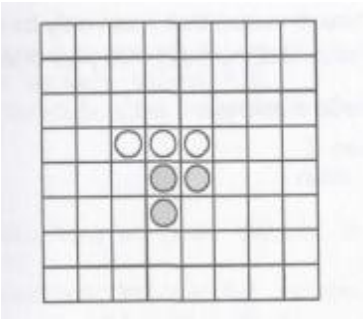
8.8 *Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or* both *the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.*

SOLUTION

Let’s start with an example. Suppose we have the following moves in an Othello game:

1. Initialize the board with two black and two white pieces in the center. The black pieces are placed at the upper left hand and lower right hand corners.
2. Play a black piece at (row 6, column 4). This flips the piece at (row 5, column 4) from white to black.
3. Play a white piece at (row 4, column 3). This flips the piece at (row 4, column 4) from black to white.

This sequence of moves leads to the board below.



The core objects in Othello are probably the game, the board, the pieces (black or white), and the players. How do we represent these with elegant object-oriented design?

Should BlackPiece and WhitePiece be classes?

At first, we might think we want to have a BlackPiece class and a WhitePiece class, which inherit from an abstract Piece. However, this is probably not a great idea. Each piece may flip back and forth between colors frequently, so continuously destroying and creating what is really the same object is probably not wise. It may be better to just have a Piece class, with a flag in it representing the current color.

Do we need separate Board and Game classes?

Strictly speaking, it may not be necessary to have both a Game object and a Board object. Keeping the objects separate allows us to have a logical separation between the board (which contains just logic involving placing pieces) and the game (which involves times, game flow, etc.). However, the drawback is that we are adding extra layers to our program. A function may call out to a method in Game, only to have it immediately call Board. We have made the choice below to keep Game and Board separate, but you should discuss this with your interviewer.

Who keeps score?

We know we should probably have some sort of score keeping for the number of black and white pieces. But who should maintain this information? One could make a strong argument for either Game or Board maintaining this information, and possibly even for Piece (in static methods). We have implemented this with Board holding this information, since it can be logically grouped with the board. It is updated by Piece or Board calling colorChanged and colorAdded methods within Board.

Should Game be a Singleton class?

Implementing Game as a singleton class has the advantage of making it easy for anyone to call a method within Game, without having to pass around references to the Game object.

Making Game a singleton though means that it can only be instantiated once. Can we make this assumption? You should discuss this with your interviewer.

One possible design for Othello is below.

```
1       public enum Direction {
2               left, right, up, down
```

```
3       }
4
5       public enum Color {
6               White, Black
7       }
8
9       public class Game {
10              private Player[ ] players;
11              private static Game instance;
12              private Board board;
13              private final int ROWS = 10;
14              private final int COLUMNS = 10;
15
16      private Game() {
17              board = new Board(ROWS, COLUMNS);
18              players = new Player[2];
19              players[0] = new Player(Color.Black);
20              players[1] = new Player(Color.White);
21      }
22
23      public static Game getInstance() {
24                      if (instance == null) instance = new Game();
25                      return instance;
26              }
27
28              public Board getBoard() {
29                      return board;
30              }
31      }
```

The Board class manages the actual pieces themselves. It does not handle much of the game play, leaving that up to the Game class.

```
1       public class Board {
2               private int blackCount = 0;
3               private int whiteCount = 0;
4               private Piece[ ][ ] board;
5
6       public Board(int rows, int columns) {
7               board = new Piece[rows][columns];
```

```
8       }
9
10      public void initialize() {
11              /* initialize center black and white pieces */
12      }
13
14      /* Attempt to place a piece of color color at (row, column).
15       * Return true if we were successful. */
16      public boolean placeColor(int row, int column, Color color) {
17          …
18      }
19
20      /* flips pieces starting at (row, column) and proceeding in
21       * direction d. */
22      private int flipSection(int row, int column, Color color,
23                                      Direction d) { ... }
24
25      public int getScoreForColor(Color c) {
26              if (c == Color.Black) return blackCount;
27              else return whiteCount;
28      }
29
30      /* Update board with additional newPieces pieces of color
31       *newColor. Decrease score of opposite color. */
32      public void updateScore(Color newColor, int newPieces) { ... }
33      }
```

As described earlier, we implement the black and white pieces with the Piece class, which has a simple Color variable representing whether it is a black or white piece.

```
1       public class Piece {
2               private Color color;
3               public Piece(Color c) { color = c; }
4
5               public void flip() {
6                       if (color == Color.Black) color = Color.White;
7                       else color = Color.Black;
8               }
9
10              public Color getColor() { return color; }
```

```
11      }
```

The Player holds only a very limited amount of information. It does not even hold its own score, but it does have a method one can call to get the score PIayer.getScore() will call out to the GameManager to retrieve this value.

```
12      public class Player {
13              private Color color;
14              public Player(Color c) { color = c; }
15
16              public int getScore() { ... }
17
18              public boolean playPiece(int r, int c) {
19                      return Game.getlnstance() .getBoard() .placeColor(r, c, color);
20              }
21
22              public Color getColor() { return color; }
23      }
```

A fully functioning (automated) version of this code can be found in the downloadable code attachment.

Remember that in many problems, what you did is less important than *why* you did it. Your interviewer probably doesn't care much whether you chose implement Game as a singleton or not, but she probably does care that you took the time to think about it and discuss the trade-offs.

8.9 Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

SOLUTION

Many candidates may see this problem and instantly panic. A file system seems so low level!

However, there's no need to panic. If we think through the components of a file system, we can tackle this problem just like any other object-oriented design question.

A file system, in its most simplistic version, consists of Files and Directories. Each Directory contains a set of Files and Directories. Since Files and Directories share so

many characteristics, we've implemented them such that they inherit from the same class, Entry.

```
1       public abstract class Entry {
2           protected Directory parent;
3           protected long created;
4           protected long lastUpdated;
5           protected long lastAccessed;
6           protected String name;
7
8       public Entry(String n, Directory p) {
9           name = n;
10          parent = p;
11          created = System.currentTimeMillis();
12          lastUpdated = System.currentTimeMillis();
13          lastAccessed = System.currentTimeMillis();
14      }
15
16      public boolean delete() {
17          if (parent == null) return false;
18          return parent.deleteEntry(this);
19      }
20
21      public abstract int size();
22
23      public String getFullPath() {
24          if (parent == null) return name;
25          else return parent.getFullPath() + “/" + name;
26      }
27
28      /* Getters and setters. */
29      public long getCreationTime() { return created; }
30      public long getLastUpdatedTime() { return lastUpdated; }
31      public long getLastAccessedTime() { return lastAccessed;
32      public void changeName(String n) { name = n; }
33      public String getName() { return name; }
34  }
35
36  public class File extends Entry {
37      private String content;
```

```
38          private int size;
39
40          public File(String n, Directory p, int sz) {
41                  super(n, p);
42                  size = sz;
43          }
44
45          public int size() {return size; }
46          public String getContents() { return content; }
47          public void setContents(String c) { content = c; }
48  }
49
50  public class Directory extends Entry {
51          protected ArrayList<Entry> contents;
52
53          public Directory(String n, Directory p) {
54                  super(n, p);
55                  contents = new ArrayList<Entry>();
56          }
57
58          public int size() {
59                  int size = 0;
69                  for (Entry e : contents) {
61                          size += e.size();
62                  }
63                  return size;
64          }
65
66          public int numberOfFiles() {
67                  int count = 0;
68                  for (Entry e : contents) {
69                          if (e instanceof Directory) {
70                                  count++; // Directory counts as a file
71                                  Directory d = (Directory) e;
72                                  count += d.numberOfFiles();
73                          } else if (e instanceof File) {
74                                  count++;
75                          }
76                  }
77                  return count;
```

```
78              }
79
80              public boolean deleteEntry(Entry entry) {
81                      return contents.remove(entry);
82              }
83
84              public void addEntry(Entry entry) {
85                      contents.add(entry);
86              }
87
88              protected ArrayList<Entry> getContents() { return content; }
89      }
```

Alternatively, we could have implemented Directory such that it contains separate lists for files and subdirectories. This makes the numberOfFiles() method a bit cleaner, since it doesn't need to use the instanceof operator, but it does prohibit us from cleanly sorting files and directories by dates or names.

8.10 *Design and implement a hash table which uses chaining (linked lists) to handle collisions.*

SOLUTION

Suppose we are implementing a hash table that looks like Hash<K, V>. That is, the hash table maps from objects of type K to objects of type V.

At we might think our data structure would look something like this:

```
1       public class Hash<K, V> {
2               LinkedList<V>[ ] items;
3               public void put(K key, V value) { ... }
4               public V get(K key) { . . . }
5       }
```

Note that items is an array of linked lists, where items [i] is a linked list of all objects with keys that map to index i (that is, all the objects that collided at i).

This would seem to work until we think more deeply about collisions.

Suppose we have a very simple hash function that uses the string length.

```
1       public int hashCodeOfKey(K key) {
2               return key.toString() .length() % items.length;
3       }
```

The keys jim and bob will map to the same index in the array, even though they are different keys. We need to search through the linked list to find the actual object that corresponds to these keys. But how would we do that? All we've stored in the linked list is the value, not the original key.

This is why we need to store both the value and the original key.

One way to do that is to create another object called Cell which pairs keys and values. With this implementation, our linked list is of type Cell.

The code below uses this implementation.

```
1       public class Hash<K, V> {
2               private final int MAX_SIZE = 10;
3               LinkedList<Cell<K, V>>[ ] items;
4
5               public Hash() {
6                       items = (LinkedList<Cell<K, V>>[ ]) new LinkedList[MAX_SIZE];
7               }
8
9               /* Really, really stupid hash. */
10              public int hashCodeOfKey(K key) {
11                      return key.toString().length() % items.length;
12              }
13
14              public void put(K key, V value) {
15                      int x = hashCodeOfKey(key);
16                      if (items[x] == null) {
17                              items[x] = new LinkedList<Cell<K, V»();
18                      }
19
20                      LinkedList<Cell<K, V>> collided = items[x];
21
22                      /* Look for items with same key and replace if found */
23                      for (Cell<K, V> c : collided) {
24                              if (c.equivalent(key)) {
25                                      collided.remove( c ) ;
```

```
26                                      break;
27                              }
28                      }
29
30                      Cell<K, V> cell = new Cell<K, V>(key, value);
31                      collided.add(cell);
32              }
33
34              public V get(K key) {
35                      int x = hashCodedOfKey(key);
36                      if (items[x] == null) {
37                              return null;
38                      }
39                      LinkedList<Cell<K, V>> collided = items [x];
40                      for (Cell<K, V> c : collided) {
41                              if (c.equivalent(key)) {
42                                      return c.getValue();
43                              }
44                      }
45
46                      return null;
47              }
48      }
```

The Cell class pairs the data value and its key. This will allow us to search through the linked list (created by "colliding," but different, keys) and find the object with the exact key value.

```
1       public class Cell<K, V> {
2               private K key;
3               private V value;
4               public Cell(K k, V v) {
5                       key = k;
6                       value = v;
7               }
8
9               public boolean equivalent(Cell<K, V> c) {
10                      return equivalence.getKey());
11              }
12
```

```
13              public boolean equivalent(K k) {
14                      return key.equals(k);
15              }
16
17              public K getKey() { return key; }
18              public V getValue() { return value; }
19      }
```

Another common implementation for a hash table is to use a binary search tree as the underlying data structure. Retrieving an element will no longer be 0(1) (although, technically it's not 0(1) if there are many collisions), but it prevents us from creating an unnecessarily large array to hold items.