```
23    public Rank getRank() { return rank; }
24  }
25
```

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.

7.3 Jukebox: Design a musical jukebox using object-oriented principles.

pg 127

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 CDs? 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, Set<CD> cdCollection,
8                   SongSelector ts) { ... }
9
10   public Song getCurrentSong() { return ts.getCurrentSong(); }
11   public void setUser(User u) { this.user = u; }
12 }
```

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  public 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.

```
public class Playlist {
   private Song song;
   private Queue<Song> queue;
   public Playlist(Song song, Queue<Song> queue) {
      ...
   }
   public Song getNextSToPlay() {
      return queue.peek();
   }
   public void queueUpSong(Song s) {
      queue.add(s);
   }
}
```

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

```
public class CD { /* data for id, artist, songs, etc */ }

public class Song { /* data for id, CD (could be null), title, length, etc */ }

public class User {
   private String name;
   public String getName() { return name; }
   public void setName(String name) {   this.name = name; }
   public long getID() { return ID; }
   public void setID(long iD) { ID = iD; }
   private long ID;
   public User(String name, long iD) { ... }
   public User getUser() { return this; }
   public static User addUser(String name, long iD) { ... }
}
```

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

7.4 Parking Lot: Design a parking lot using object-oriented principles.

pg 127

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 = new ArrayList<ParkingSpot>();
5    protected String licensePlate;
6    protected int spotsNeeded;
7    protected VehicleSize size;
8
9    public int getSpotsNeeded() { return spotsNeeded; }
10   public VehicleSize getSize() { return size; }
11
12   /* Park vehicle in this spot (among others, potentially) */
13   public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
14
15   /* Remove car from spot, and notify spot that it's gone */
16   public void clearSpots() { ... }
17
18   /* Checks if the spot is big enough for the vehicle (and is available). This
19    * compares the SIZE only. It does not check if it has enough spots. */
20   public abstract boolean canFitInSpot(ParkingSpot spot);
21 }
22
23 public class Bus extends Vehicle {
24   public Bus() {
25     spotsNeeded = 5;
26     size = VehicleSize.Large;
27   }
28
29   /* Checks if the spot is a Large. Doesn't check num of spots */
30   public boolean canFitInSpot(ParkingSpot spot) { ... }
31 }
32
33 public class Car extends Vehicle {
34   public Car() {
35     spotsNeeded = 1;
36     size = VehicleSize.Compact;
37   }
38
39   /* Checks if the spot is a Compact or a Large. */
40   public boolean canFitInSpot(ParkingSpot spot) { ... }
41 }
42
43 public class Motorcycle extends Vehicle {
44   public Motorcycle() {
45     spotsNeeded = 1;
46     size = VehicleSize.Motorcycle;
47   }
48
49   public boolean canFitInSpot(ParkingSpot spot) { ... }
50 }
```

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

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 available */
22    public void removeVehicle() { ... }
23 }
```

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

7.5 Online Book Reader: Design the data structures for an online book reader system.

pg 127

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 could 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 getActiveUser() { 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 HashMap<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 HashMap<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 User find(int id) { return users.get(id); }
40    public boolean remove(User u) { return remove(u.getID()); }
41    public boolean remove(int id) {
```

```
42        if (!users.containsKey(id)) {
43           return false;
44        }
45        users.remove(id);
46        return true;
47     }
48  }
49
50  public class Display {
51     private Book activeBook;
52     private User activeUser;
53     private int pageNumber = 0;
54
55     public void displayUser(User user) {
56        activeUser = user;
57        refreshUsername();
58     }
59
60     public void displayBook(Book book) {
61        pageNumber = 0;
62        activeBook = book;
63
64        refreshTitle();
65        refreshDetails();
66        refreshPage();
67     }
68
69     public void turnPageForward() {
70        pageNumber++;
71        refreshPage();
72     }
73
74     public void turnPageBackward() {
75        pageNumber--;
76        refreshPage();
77     }
78
79     public void refreshUsername() { /* updates username display */ }
80     public void refreshTitle() { /* updates title display */ }
81     public void refreshDetails() { /* updates details display */   }
82     public void refreshPage() { /* updated page display */ }
83  }
```

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(int id) { bookId = id; }
12     public String getDetails() { return details; }
13     public void setDetails(String d) { details = d; }
```

```
}

public class User {
   private int userId;
   private String details;
   private int accountType;

   public void renewMembership() {  }

   public User(int id, String details, int accountType) {
      userId = id;
      this.details = details;
      this.accountType = accountType;
   }

   /* Getters and setters */
   public int getID() { return userId; }
   public void setID(int id) { userId = id; }
   public String getDetails() {
      return details;
   }

   public void setDetails(String details) {
      this.details = details;
   }
   public int getAccountType() { return accountType; }
   public void setAccountType(int t) { accountType = t; }
}
```

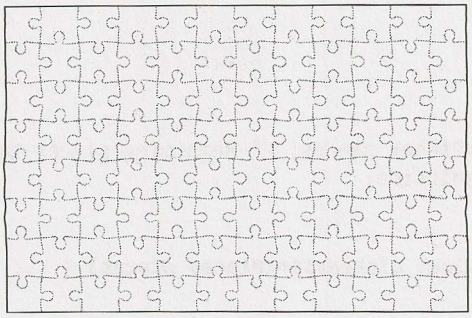
The decision to tear off user management, library, and display into their own classes, when this functionality could have been in the general `OnlineReaderSystem` class, is an interesting one. On a very small system, making this decision could make the system overly complex. However, as the system grows,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

7.6 **Jigsaw:** Implement an NxN 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 edges, returns true if the two edges belong together.

pg 128

SOLUTION

We have a traditional 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)."
- *Relative Position:* "I don't know where this piece is actually located, but I know it is next to this other piece."

For our solution, we will use the absolute position.

We'll need classes to represent `Puzzle`, `Piece`, and `Edge`. Additionally, we'll want enums for the different shapes (`inner`, `outer`, `flat`) and the orientations of the edges (`left`, `top`, `right`, `bottom`).

`Puzzle` will start off with a list of the pieces. When we solve the puzzle, we'll fill in an NxN `solution` matrix of pieces.

`Piece` will have a hash table that maps from an orientation to the appropriate edge. Note that we might rotate the piece at some point, so the hash table could change. The orientation of the edges will be arbitrarily assigned at first.

`Edge` will have just its shape and a pointer back to its parent piece. It will not keep its orientation.

A potential object-oriented design looks like the following:

```
1   public enum Orientation {
2     LEFT, TOP, RIGHT, BOTTOM; // Should stay in this order
3
4     public Orientation getOpposite() {
5       switch (this) {
6         case LEFT: return RIGHT;
7         case RIGHT: return LEFT;
8         case TOP: return BOTTOM;
9         case BOTTOM: return TOP;
10        default: return null;
11      }
12    }
13  }
14
15  public enum Shape {
16    INNER, OUTER, FLAT;
17
18    public Shape getOpposite() {
19      switch (this) {
20        case INNER: return OUTER;
21        case OUTER: return INNER;
22        default: return null;
```