```
      }
    }
  }

  public class Puzzle {
    private LinkedList<Piece> pieces; /* Remaining pieces to put away. */
    private Piece[][] solution;
    private int size;

    public Puzzle(int size, LinkedList<Piece> pieces) { ... }


    /* Put piece into the solution, turn it appropriately, and remove from list. */
    private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
                                   int column, Orientation orientation) {
      Piece piece = edge.getParentPiece();
      piece.setEdgeAsOrientation(edge, orientation);
      pieces.remove(piece);
      solution[row][column] = piece;
    }

    /* Find the matching piece in piecesToSearch and insert it at row, column. */
    private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int col);

    /* Solve puzzle. */
    public boolean solve() { ... }
  }

  public class Piece {
    private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();

    public Piece(Edge[] edgeList) { ... }

    /* Rotate edges by "numberRotations". */
    public void rotateEdgesBy(int numberRotations) { ... }

    public boolean isCorner() { ... }
    public boolean isBorder() { ... }
  }

  public class Edge {
    private Shape shape;
    private Piece parentPiece;
    public Edge(Shape shape) { ... }
    public boolean fitsWith(Edge edge) { ... }
  }
```

Algorithm to Solve the Puzzle

Just as a kid might in solving a puzzle, we'll start with grouping the pieces into corner pieces, border pieces, and inside pieces.

Once we've done that, we'll pick an arbitrary corner piece and put it in the top left corner. We will then walk through the puzzle in order, filling in piece by piece. At each location, we search through the correct group of pieces to find the matching piece. When we insert the piece into the puzzle, we need to rotate the piece to fit correctly.

The code below outlines this algorithm.

```
/* Find the matching piece within piecesToSearch and insert it at row, column. */
boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int column) {
  if (row == 0 && column == 0) { // On top left corner, just put in a piece
    Piece p = piecesToSearch.remove();
    orientTopLeftCorner(p);
    solution[0][0] = p;
  } else {
    /* Get the right edge and list to match. */
    Piece pieceToMatch = column == 0 ? solution[row - 1][0] :
                                       solution[row][column - 1];
    Orientation orientationToMatch = column == 0 ? Orientation.BOTTOM :
                                                   Orientation.RIGHT;
    Edge edgeToMatch = pieceToMatch.getEdgeWithOrientation(orientationToMatch);

    /* Get matching edge. */
    Edge edge = getMatchingEdge(edgeToMatch, piecesToSearch);
    if (edge == null) return false; // Can't solve

    /* Insert piece and edge. */
    Orientation orientation = orientationToMatch.getOpposite();
    setEdgeInSolution(piecesToSearch, edge, row, column, orientation);
  }
  return true;
}

boolean solve() {
  /* Group pieces. */
  LinkedList<Piece> cornerPieces = new LinkedList<Piece>();
  LinkedList<Piece> borderPieces = new LinkedList<Piece>();
  LinkedList<Piece> insidePieces = new LinkedList<Piece>();
  groupPieces(cornerPieces, borderPieces, insidePieces);

  /* Walk through puzzle, finding the piece that joins the previous one. */
  solution = new Piece[size][size];
  for (int row = 0; row < size; row++) {
    for (int column = 0; column < size; column++) {
      LinkedList<Piece> piecesToSearch = getPieceListToSearch(cornerPieces,
        borderPieces, insidePieces, row, column);
      if (!fitNextEdge(piecesToSearch, row, column)) {
        return false;
      }
    }
  }

  return true;
}
```

The full code for this solution can be found in the downloadable code attachment.

7.7 **Chat Server:** 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?

pg 128

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 split 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 `UserManager` 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.

```
/* UserManager serves as a central place for core user actions. */
public class UserManager {
  private static UserManager instance;
  /* maps from a user id to a user */
  private HashMap<Integer, User> usersById;

  /* maps from an account name to a user */
  private HashMap<String, User> usersByAccountName;

  /* maps from the user id to an online user */
  private HashMap<Integer, User> onlineUsers;

  public static UserManager getInstance() {
    if (instance == null) instance = new UserManager();
    return instance;
  }

  public void addUser(User fromUser, String toAccountName) { ... }
  public void approveAddRequest(AddRequest req) { ... }
  public void rejectAddRequest(AddRequest req) { ... }
  public void userSignedOn(String accountName) { ... }
  public void userSignedOff(String accountName) { ... }
}
```

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.approveAddRequest` 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 `UserManager` 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.

```
public class User {
  private int id;
  private UserStatus status = null;

  /* maps from the other participant's user id to the chat */
  private HashMap<Integer, PrivateChat> privateChats;

  /* list of group chats */
  private ArrayList<GroupChat> groupChats;

  /* maps from the other person's user id to the add request */
  private HashMap<Integer, AddRequest> receivedAddRequests;

  /* maps from the other person's user id to the add request */
  private HashMap<Integer, AddRequest> sentAddRequests;

  /* maps from the user id to user object */
  private HashMap<Integer, User> contacts;

  private String accountName;
  private String fullName;

  public User(int id, String accountName, String fullName) { ... }
  public boolean sendMessageToUser(User to, String content){ ... }
  public boolean sendMessageToGroupChat(int id, String cnt){...}
  public void setStatus(UserStatus status) { ... }
  public UserStatus getStatus() { ... }
  public boolean addContact(User user) { ... }
  public void receivedAddRequest(AddRequest req) { ...}
  public void sentAddRequest(AddRequest req) { ... }
  public void removeAddRequest(AddRequest req) { ... }
  public void requestAddUser(String accountName) { ... }
  public void addConversation(PrivateChat conversation) { ... }
  public void addConversation(GroupChat conversation) { ... }
  public int getId() { ... }
  public String getAccountName() { ... }
  public String getFullName() { ... }
}
```

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.

```
public abstract class Conversation {
  protected ArrayList<User> participants;
  protected int id;
  protected ArrayList<Message> messages;

  public ArrayList<Message> getMessages() { ... }
  public boolean addMessage(Message m) { ... }
  public int getId() { ... }
}

public class GroupChat extends Conversation {
  public void removeParticipant(User user) { ... }
  public void addParticipant(User user) { ... }
}

public class PrivateChat extends Conversation {
  public PrivateChat(User user1, User user2) { ...
  public User getOtherParticipant(User primary) { ... }
}

public class Message {
  private String content;
  private Date date;
  public Message(String content, Date date) { ... }
  public String getContent() { ... }
  public Date getDate() { ... }
}
```

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

```
public class AddRequest {
  private User fromUser;
  private User toUser;
  private Date date;
  RequestStatus status;

  public AddRequest(User from, User to, Date date) { ... }
  public RequestStatus getStatus() { ... }
  public User getFromUser() { ... }
  public User getToUser() { ... }
  public Date getDate() { ... }
}

public class UserStatus {
   private String message;
   private UserStatusType type;
   public UserStatus(UserStatusType type, String message) { ... }
   public UserStatusType getStatusType() { ... }
   public String getMessage() { ... }
}

public enum UserStatusType {
  Offline, Away, Idle, Available, Busy
}

public enum RequestStatus {
  Unread, Read, Accepted, Rejected
}
```

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.

Q1: 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?

7.8 **Othello:** 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.

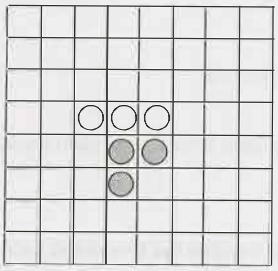
pg 128

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 the 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.

However, making Game a singleton means 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.

```java
public enum Direction {
  left, right, up, down
}

public enum Color {
  White, Black
}

public class Game {
  private Player[] players;
  private static Game instance;
  private Board board;
  private final int ROWS = 10;
  private final int COLUMNS = 10;

  private Game() {
    board = new Board(ROWS, COLUMNS);
    players = new Player[2];
    players[0] = new Player(Color.Black);
    players[1] = new Player(Color.White);
  }

  public static Game getInstance() {
    if (instance == null) instance = new Game();
    return instance;
  }

  public Board getBoard() {
    return board;
  }
}
```

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

```
public class Board {
  private int blackCount = 0;
  private int whiteCount = 0;
  private Piece[][] board;

  public Board(int rows, int columns) {
    board = new Piece[rows][columns];
  }

  public void initialize() {
    /* initialize center black and white pieces */
  }

  /* Attempt to place a piece of color color at (row, column). Return true if we
   * were successful. */
  public boolean placeColor(int row, int column, Color color) {
    ...
  }

  /* Flips pieces starting at (row, column) and proceeding in direction d. */
  private int flipSection(int row, int column, Color color, Direction d) { ... }

  public int getScoreForColor(Color c) {
    if (c == Color.Black) return blackCount;
    else return whiteCount;
  }

  /* Update board with additional newPieces pieces of color newColor. Decrease
   * score of opposite color. */
  public void updateScore(Color newColor, int newPieces) { ... }
}
```

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.

```
public class Piece {
  private Color color;
  public Piece(Color c) { color = c; }

  public void flip() {
    if (color == Color.Black) color = Color.White;
    else color = Color.Black;
  }

  public Color getColor() { return color; }
}
```

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. Player.getScore() will call out to the Game object to retrieve this value.

```
public class Player {
  private Color color;
  public Player(Color c) { color = c; }

  public int getScore() { ...  }
  
  public boolean playPiece(int r, int c) {
    return Game.getInstance().getBoard().placeColor(r, c, color);
  }

  public Color getColor() { return color; }
}
```

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 to 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.

7.9 **Circular Array:** Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. If possible, the class should use a generic type (also called a template), and should support iteration via the standard `for (Obj o : circularArray)` notation.

pg 128

SOLUTION

This problem really has two parts to it. First, we need to implement the `CircularArray` class. Second, we need to support iteration. We will address these parts separately.

Implementing the CircularArray class

One way to implement the `CircularArray` class is to actually shift the elements each time we call `rotate(int shiftRight)`. Doing this is, of course, not very efficient.

Instead, we can just create a member variable head which points to what should be *conceptually* viewed as the start of the circular array. Rather than shifting around the elements in the array, we just increment head by `shiftRight`.

The code below implements this approach.

```
public class CircularArray<T> {
  private T[] items;
  private int head = 0;

  public CircularArray(int size) {
    items = (T[]) new Object[size];
  }

  private int convert(int index) {
    if (index < 0) {
      index += items.length;
    }
    return (head + index) % items.length;
  }

  public void rotate(int shiftRight) {
    head = convert(shiftRight);
  }

  public T get(int i) {
    if (i < 0 || i >= items.length) {
```