LLD machine coding

💡 https://chat.openai.com/share/a1774573-c457-4096-aa17-eec156c51137

https://lldcoding.com/

Designing a JSON parser

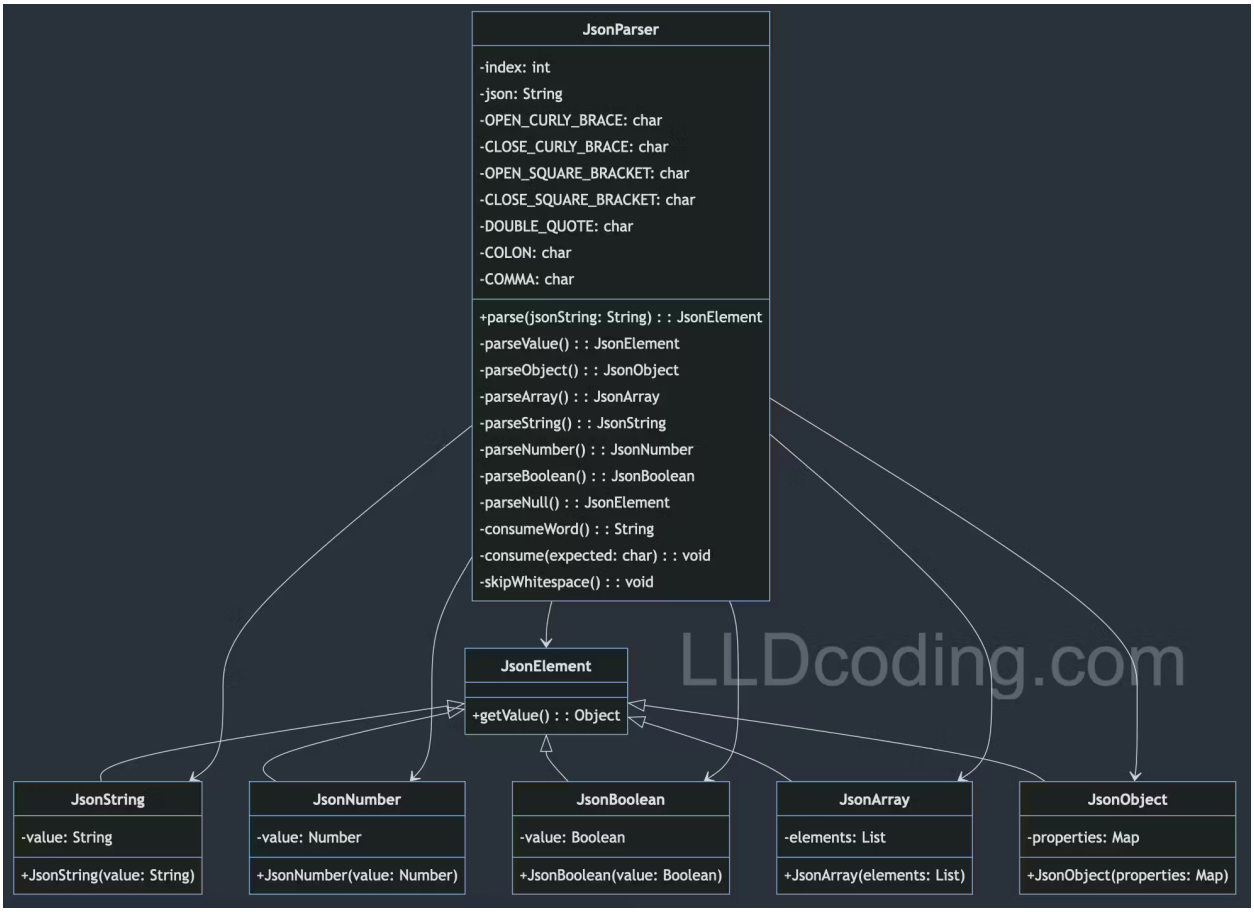
involves several components and considerations. Below is a basic outline of features, design patterns, and a simplified code implementation in Java. Note that JSON parsing is a broad topic, and the implementation can vary based on specific requirements.

Features Required:

1. **Parse JSON:** Read and interpret JSON data.
2. **Handle Different Types:** Support various data types (string, number, boolean, array, object, null).
3. **Error Handling:** Gracefully handle syntax errors and unexpected input.
4. **Nested Structures:** Support nested JSON structures.
5. **Flexible Parsing:** Allow parsing of partial JSON data.
6. **Object Mapping:** Convert JSON data into a usable object structure.
7. **Configurability:** Allow configuration for handling specific JSON structures.

Design Patterns Involved or Used:

1. **Interpreter Pattern:** For parsing JSON syntax.
2. **Composite Pattern:** For handling nested structures.
3. **Factory Method Pattern:** To create different types of JSON elements.
4. **Builder Pattern:** Construct complex JSON objects step by step.
5. **Strategy Pattern:** Different strategies for parsing different types of JSON elements.



Code

COPY

COPY

```
import java.util.*;
import java.util.stream.Collectors;

interface JsonElement {
    Object getValue();
}

class JsonString implements JsonElement {
    private String value;

    public JsonString(String value) {
        this.value = value;
    }

    public Object getValue() {
        return value;
    }
}

class JsonNumber implements JsonElement {
    private Number value;

    public JsonNumber(Number value) {
        this.value = value;
    }

    public Object getValue() {
        return value;
    }
}

class JsonBoolean implements JsonElement {
    private Boolean value;

    public JsonBoolean(Boolean value) {
```

```
        this.value = value;
    }

    public Object getValue() {
        return value;
    }
}

class JsonArray implements JsonElement {
    private List<JsonElement> elements;

    public JsonArray(List<JsonElement> elements) {
        this.elements = elements;
    }

    public Object getValue() {
        return elements.stream().map(JsonElement::getValue).c
ollect(Collectors.toList());
    }
}

class JsonObject implements JsonElement {
    private Map<String, JsonElement> properties;

    public JsonObject(Map<String, JsonElement> properties) {
        this.properties = properties;
    }

    public Object getValue() {
        Map<String, Object> result = new HashMap<>();
        properties.forEach((key, value) -> result.put(key, va
lue.getValue()));
        return result;
    }
}
```

```
public class JsonParser {
    private int index;
    private String json;

    // Constants
    private static final char OPEN_CURLY_BRACE = '{';
    private static final char CLOSE_CURLY_BRACE = '}';
    private static final char OPEN_SQUARE_BRACKET = '[';
    private static final char CLOSE_SQUARE_BRACKET = ']';
    private static final char DOUBLE_QUOTE = '"';
    private static final char COLON = ':';
    private static final char COMMA = ',';

    public JsonElement parse(String jsonString) {
        this.index = 0;
        this.json = jsonString;
        skipWhitespace();
        return parseValue();
    }

    private JsonElement parseValue() {
        char currentChar = json.charAt(index);

        if (currentChar == OPEN_CURLY_BRACE) {
            return parseObject();
        } else if (currentChar == OPEN_SQUARE_BRACKET) {
            return parseArray();
        } else if (currentChar == DOUBLE_QUOTE) {
            return parseString();
        } else if (Character.isDigit(currentChar) || currentC
har == '-') {
            return parseNumber();
        } else if (currentChar == 't' || currentChar == 'f')
{
            return parseBoolean();
        } else if (currentChar == 'n') {
```

```
            return parseNull();
        }

        throw new RuntimeException("Invalid JSON");
    }

    private JsonObject parseObject() {
        Map<String, JsonElement> properties = new HashMap<>
();

        // Consume the opening curly brace
        consume(OPEN_CURLY_BRACE);

        skipWhitespace();
        while (json.charAt(index) != CLOSE_CURLY_BRACE) {
            // Parse property name
            String propertyName = parseString().getValue().to
String();
            skipWhitespace();

            // Consume the colon
            consume(COLON);
            skipWhitespace();

            // Parse property value
            JsonElement propertyValue = parseValue();
            properties.put(propertyName, propertyValue);

            skipWhitespace();

            // Check for a comma, indicating more properties
            if (json.charAt(index) == COMMA) {
                consume(COMMA);
                skipWhitespace();
            }
        }
```

```
        // Consume the closing curly brace
        consume(CLOSE_CURLY_BRACE);

        return new JsonObject(properties);
    }

    private JsonArray parseArray() {
        List<JsonElement> elements = new ArrayList<>();

        // Consume the opening square bracket
        consume(OPEN_SQUARE_BRACKET);

        skipWhitespace();
        while (json.charAt(index) != CLOSE_SQUARE_BRACKET) {
            // Parse array element
            JsonElement element = parseValue();
            elements.add(element);

            skipWhitespace();

            // Check for a comma, indicating more elements
            if (json.charAt(index) == COMMA) {
                consume(COMMA);
                skipWhitespace();
            }
        }

        // Consume the closing square bracket
        consume(CLOSE_SQUARE_BRACKET);

        return new JsonArray(elements);
    }

    private JsonString parseString() {
        // Consume the opening double quote
```

```
        consume(DOUBLE_QUOTE);

        StringBuilder sb = new StringBuilder();
        while (json.charAt(index) != DOUBLE_QUOTE) {
            sb.append(json.charAt(index));
            index++;
        }

        // Consume the closing double quote
        consume(DOUBLE_QUOTE);

        return new JsonString(sb.toString());
    }

    private JsonNumber parseNumber() {
        int startIndex = index;

        // Consume digits and optional decimal point
        while (Character.isDigit(json.charAt(index)) || json.
charAt(index) == '.') {
            index++;
        }

        String numberStr = json.substring(startIndex, index);
        if (numberStr.contains(".")) {
            return new JsonNumber(Double.parseDouble(numberSt
r));
        } else {
            return new JsonNumber(Long.parseLong(numberStr));
        }
    }

    private JsonBoolean parseBoolean() {
        String boolStr = consumeWord();
        if (boolStr.equals("true")) {
            return new JsonBoolean(true);
```

```
        } else if (boolStr.equals("false")) {
            return new JsonBoolean(false);
        }

        throw new RuntimeException("Invalid boolean value");
    }

    private JsonElement parseNull() {
        consumeWord(); // Consume "null"
        return null;
    }

    private String consumeWord() {
        StringBuilder sb = new StringBuilder();
        while (Character.isLetter(json.charAt(index))) {
            sb.append(json.charAt(index));
            index++;
        }
        return sb.toString();
    }

    private void consume(char expected) {
        if (json.charAt(index) == expected) {
            index++;
        } else {
            throw new RuntimeException("Expected: " + expecte
d);
        }
    }

    private void skipWhitespace() {
        while (Character.isWhitespace(json.charAt(index))) {
            index++;
        }
    }
```

```
    public static void main(String[] args) {
        String jsonString = "{ \"name\": \"John\", \"age\": 3
0, \"city\": \"New York\", \"isAdmin\": true, \"scores\": [1
0, 20, 30] }";

        JsonParser jsonParser = new JsonParser();
        JsonElement jsonElement = jsonParser.parse(jsonStrin
g);

        System.out.println(jsonElement.getValue());
    }
}
```

Google Authenticator

we can start by outlining the basic requirements and then proceed to design the classes and methods. Here's a high-level design:

1. **Requirements**:
    - Generate a new secret key for each user.
    - Use the secret key to generate a Time-based One-Time Password (TOTP).
    - Verify the OTP entered by the user.
    - Allow the user to enable/disable two-factor authentication (2FA).
    - Allow the user to reset the 2FA by re-generating a new secret key.
2. **Class Diagram**:
    - `User`: Represents a user with properties like `userId`, `username`, `password`, `secretKey`, `isTwoFactorEnabled`, etc.
    - `TOTPGenerator`: A utility class to generate TOTP based on a given secret key and timestamp.
    - `GoogleAuthenticatorService`: A service class that provides methods for generating a secret key, enabling/disabling 2FA, generating TOTP, and verifying OTP.

- `Database`: Interface for storing and retrieving user information. Can be implemented using any database system (e.g., MySQL, MongoDB).

3. **Sequence of Operations**:
   - User Registration: Generate a secret key for the user and store it in the database.
   - Enable 2FA: Set the `isTwoFactorEnabled` flag to true for the user.
   - OTP Generation: Generate TOTP based on the secret key and timestamp.
   - OTP Verification: Verify the OTP entered by the user with the generated OTP.
4. **Code Implementation** (Java):
   - `User` class:

```
javaCopy code
public class User {
    private String userId;
    private String username;
    private String password;
    private String secretKey;
    private boolean isTwoFactorEnabled;

    // Getters and setters
}
```

   - `TOTPGenerator` class:

```
javaCopy code
public class TOTPGenerator {
    public static String generateTOTP(String secretKey,
long timestamp) {
        // Implementation of TOTP generation algorithm
(e.g., using HMAC-SHA1)
    }
```

```
}
```

- `GoogleAuthenticatorService` class:

```
javaCopy code
public class GoogleAuthenticatorService {
    public String generateSecretKey() {
        // Generate a new secret key
    }

    public boolean enableTwoFactorAuth(User user) {
        // Enable two-factor authentication for the use
r
    }

    public boolean disableTwoFactorAuth(User user) {
        // Disable two-factor authentication for the us
er
    }

    public boolean verifyOTP(User user, String otp) {
        // Verify the OTP entered by the user
    }

    // Other methods for OTP generation, secret key sto
rage, etc.
}
```

- `Database` interface:

```
javaCopy code
public interface Database {
    User getUserById(String userId);
```

```
    void updateUser(User user);

    // Other methods for user management
}
```

This is a basic outline of the design for Google Authenticator. Actual implementation may vary based on specific requirements and technologies used.

Blogging application

we need to model the database schema that will store information about users, blog posts, comments, and any other relevant entities. Here's a basic outline of the database schema using a relational database approach (e.g., MySQL):

1. **Entities**:
    - User: Represents a user of the blogging application.
    - Post: Represents a blog post written by a user.
    - Comment: Represents a comment on a blog post.
2. **Database Schema**:
    - **User Table**:
        - Columns: `id` (primary key), `username`, `email`, `password_hash`, `created_at`, `updated_at`.
    - **Post Table**:
        - Columns: `id` (primary key), `title`, `content`, `author_id` (foreign key referencing `User.id`), `created_at`, `updated_at`.
    - **Comment Table**:
        - Columns: `id` (primary key), `content`, `post_id` (foreign key referencing `Post.id`), `author_id` (foreign key referencing `User.id`), `created_at`, `updated_at`.
3. **Database Relationships**:

- One-to-Many Relationship between User and Post: Each user can have multiple blog posts.
- One-to-Many Relationship between Post and Comment: Each post can have multiple comments.
- Many-to-One Relationship between Comment and User: Each comment is made by a user.
- Many-to-One Relationship between Comment and Post: Each comment is on a post.

4. **Database Tables**:
   - `user` table:

```
sqlCopy code
CREATE TABLE user (
    id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON U
PDATE CURRENT_TIMESTAMP
);
```

   - `post` table:

```
sqlCopy code
CREATE TABLE post (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    author_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON U
```

```
PDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (author_id) REFERENCES user(id)
);
```

- `comment` table:

```
sqlCopy code
CREATE TABLE comment (
    id INT PRIMARY KEY AUTO_INCREMENT,
    content TEXT NOT NULL,
    post_id INT NOT NULL,
    author_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON U
PDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES post(id),
    FOREIGN KEY (author_id) REFERENCES user(id)
);
```

This is a basic schema for a blogging application. Depending on the specific requirements of your application, you may need to add more tables or modify the existing schema.

Designing the low-level design (LLD) for Amazon Prime Video

involves breaking down the system into smaller components and defining how these components interact with each other. Here's a basic outline of the design:

1. **Requirements**:
    - Users should be able to browse and watch videos.
    - Users should be able to create and manage profiles.
    - Admins should be able to upload and manage videos.

- The system should support multiple devices (web, mobile, smart TVs, etc.).
- The system should support video streaming with adaptive bitrate.
- The system should support user authentication and authorization.

2. **Components**:
   - **User Management**: Manages user accounts, profiles, authentication, and authorization.
   - **Video Management**: Manages video metadata, storage, and retrieval.
   - **Streaming Service**: Handles video streaming, including adaptive bitrate streaming.
   - **Device Management**: Supports various devices and platforms.
   - **Database**: Stores user data, video metadata, and other relevant information.
3. **Class Diagram**:
   - `User`: Represents a user with properties like `userId`, `username`, `email`, `password`, etc.
   - `Video`: Represents a video with properties like `videoId`, `title`, `description`, `duration`, `url`, etc.
   - `Profile`: Represents a user profile with properties like `profileId`, `userId`, `name`, `language`, etc.
   - `Device`: Represents a device with properties like `deviceId`, `type`, `name`, `platform`, etc.
   - `AuthenticationService`: Handles user authentication.
   - `AuthorizationService`: Handles user authorization.
   - `VideoService`: Manages video metadata and storage.
   - `StreamingService`: Handles video streaming.
   - `DeviceService`: Manages devices and their capabilities.
4. **Sequence of Operations**:

- User Registration/Login: Authenticate users and create/manage profiles.
- Video Upload/Management: Admins upload/manage videos, and users browse/watch videos.
- Video Streaming: Stream videos based on user device and network conditions.

5. **Code Implementation** (Java):
   - `User` class:

```
javaCopy code
public class User {
    private String userId;
    private String username;
    private String email;
    private String password;
    // Other properties, getters, and setters
}
```

   - `Video` class:

```
javaCopy code
public class Video {
    private String videoId;
    private String title;
    private String description;
    private int duration;
    private String url;
    // Other properties, getters, and setters
}
```

   - `Profile` class:

```
javaCopy code
public class Profile {
    private String profileId;
    private String userId;
    private String name;
    private String language;
    // Other properties, getters, and setters
}
```

- `Device` class:

```
javaCopy code
public class Device {
    private String deviceId;
    private String type;
    private String name;
    private String platform;
    // Other properties, getters, and setters
}
```

- `AuthenticationService`, `AuthorizationService`, `VideoService`, `StreamingService`, `DeviceService`: Implementations for respective services.

Design a Pair Programming Platform like HackerRank CodePair

we need to focus on features that facilitate collaborative coding sessions between two or more developers. Here's a basic outline of the design:

1. **Requirements**:
    - Users should be able to create and join coding sessions.
    - Users should be able to share their code editor in real-time.

- Users should be able to communicate via text/audio/video chat.
- Users should be able to run and debug code together.
- The platform should support multiple programming languages.
- The platform should have code playback feature to review the session.

2. **Components**:
   - **Session Management**: Manages creation, joining, and deletion of coding sessions.
   - **Code Editor**: Provides a shared code editor with syntax highlighting and collaborative editing.
   - **Communication**: Supports text/audio/video chat between participants.
   - **Code Execution**: Allows running and debugging of code within the platform.
   - **Language Support**: Provides support for multiple programming languages.
   - **Code Playback**: Records and replays coding sessions for review.
3. **Class Diagram**:
   - `User`: Represents a user with properties like `userId`, `username`, `email`, etc.
   - `Session`: Represents a coding session with properties like `sessionId`, `participants`, `code`, `chat`, etc.
   - `CodeEditor`: Provides methods for editing and sharing code.
   - `CommunicationService`: Manages communication between participants.
   - `ExecutionService`: Handles code execution and debugging.
   - `PlaybackService`: Manages recording and playback of coding sessions.
4. **Sequence of Operations**:
   - Session Creation: User creates a new coding session.
   - Session Joining: User joins an existing coding session.

- Code Sharing: Participants share and edit code in real-time.
- Communication: Participants communicate via chat and audio/video.
- Code Execution: Participants run and debug code together.
- Code Playback: Participants review the session using code playback feature.

5. **Code Implementation** (Java):
   - `User` class:

```
javaCopy code
public class User {
    private String userId;
    private String username;
    private String email;
    // Other properties, getters, and setters
}
```

   - `Session` class:

```
javaCopy code
public class Session {
    private String sessionId;
    private List<User> participants;
    private String code;
    private String chat;
    // Other properties, getters, and setters
}
```

   - `CodeEditor`, `CommunicationService`, `ExecutionService`, `PlaybackService`: Implementations for respective services.