First: Each Applied Design Pattern with Explanation

1. Singleton Pattern Explanation

- ❖ **In general:** The Singleton Pattern ensures that a class has only one instance throughout the lifecycle of the application, and provides a global access point to that instance.
- ❖ **In the SkillSync project**: The Singleton Pattern is used in the AuthService class. We implemented AuthService as a Singleton because user authentication (registering new users, logging in existing users, and verifying passwords) must be consistent and centralized across the backend.

Instead of creating multiple AuthService instances for each login/register operation, a single shared AuthService instance is used.
This ensures that:

- No duplicated login logic exists
- No inconsistent authentication behavior happens between requests
- Only one "source of truth" controls all authentication processes

---

2. Factory Pattern Explanation

- ❖ **In general:** The Factory Pattern is a creational design pattern that provides a method to create objects without exposing the creation logic to the client.
  Instead of calling `new` directly, you call a factory method, which chooses and returns the correct object.
- ❖ **In the SkillSync project**: the Factory Pattern is implemented in the ResumeAnalyzerFactory class.
  This factory dynamically chooses which type of resume analyzer to use based on the user's subscription plan:
- If the user has a **"premium"** subscription → Use `AIResumeAnalyzer`
- If the user has a **"free"** subscription → Use `KeywordResumeAnalyzer`

This decision is **completely hidden from the controller** — the controller simply asks the factory for an analyzer, without worrying about the internal decision logic.

---

3. Observer Pattern Explanation

- ❖ **In general:** The Observer Pattern defines a one-to-many relationship between objects:
  When one object (the Subject) changes state, all its dependent objects (Observers) are automatically notified and updated.
- ❖ **In the SkillSync project**: the Observer Pattern is implemented in the NotificationService system.
- NotificationService is the Subject: it manages a list of observers.

- Observers are notification types: `EmailNotifier`, `SMSNotifier`, and `PushNotifier`.

Whenever an important event happens (e.g., resume analyzed successfully), the `NotificationService` notifies all attached observers, which then send emails, SMS messages, or push notifications automatically.

The controllers (like `resumeController.js`) don't have to directly manage notifications — they only trigger a notify event once!

Second: How and Where It Is Used

1. Singleton Pattern — AuthService

❖ How It Is Used:

- The **AuthService** class is designed to always return the **same instance** whenever it is imported and used in the project.

- The Singleton instance handles all authentication-related logic like:

  - **Registering users** (hashing passwords and saving to DB)

  - **Logging users in** (validating passwords and generating JWT tokens)

❖ Where It Is Used:

- 📁 `services/AuthService.js` → Singleton logic

- 📁 `controllers/authController.js` → Imports AuthService to call `.register()` and `.login()`

- The controller doesn't create multiple instances — it simply **uses the shared AuthService instance**.

❖ Example in `authController.js:`

```
const authService = require('../services/AuthService');

exports.login = async (req, res) => {
  const { email, password } = req.body;
```

---

2. Factory Pattern — ResumeAnalyzerFactory

❖ How It Is Used:

- The **ResumeAnalyzerFactory** dynamically chooses which analyzer to use based on the user's subscription type.
- It hides the complexity from the controller.
- If a user is premium → Factory returns an `AIResumeAnalyzer`.
- If a user is free → Factory returns a `KeywordResumeAnalyzer`.

❖ Where It Is Used:

- 📁 `factories/ResumeAnalyzerFactory.js` → Factory logic
- 📁 `controllers/resumeController.js` → Calls the factory to get an analyzer before analyzing the uploaded resume.

❖ Example in `resumeController.js`:

```
const ResumeAnalyzerFactory =
require('../factories/ResumeAnalyzerFactory');

const analyzer =
ResumeAnalyzerFactory.getAnalyzer(req.user.subscription);

    const result = analyzer.analyze(resumeText);
```

---

3. Observer Pattern — NotificationService

❖ **How It Is Used:**

- **NotificationService** acts as the subject that manages and triggers notification observers.

- Observers like `EmailNotifier`, `SMSNotifier`, and `PushNotifier` attach themselves to the NotificationService.

- When a business event occurs (e.g., resume analysis completed), the controller **calls `notify()` once**, and all attached observers are triggered automatically.

❖ **Where It Is Used:**

- 📁 `observers/NotificationService.js` → Subject logic

- 📁 `observers/EmailNotifier.js`, `SMSNotifier.js`, `PushNotifier.js` → Observer implementations

- 📁 `controllers/resumeController.js` → Attaches observers and calls `.notify()` after analysis.

❖ **Example in `resumeController.js`:**

```
const notifier = new NotificationService();

    notifier.attach(new EmailNotifier());

    notifier.attach(new SMSNotifier());

    notifier.attach(new PushNotifier());

    notifier.notify(` Resume analyzed. Score: ${result.score}`);
```

Third: Benefits & Trade-offs of Each Pattern

❖ **Singleton Pattern — AuthService:**

Aspects	Details
Benefit	Ensures that only one instance of `AuthService` exists throughout the app. This centralizes authentication logic (login, register) and avoids inconsistent behavior across different parts of the backend. It also improves memory efficiency because no repeated instantiation happens.
Trade-off	Singleton patterns can introduce hidden dependencies across the project. Overusing them can make unit testing harder because the Singleton "hides" internal state unless mocked carefully. Must be cautious when using global singletons in very large apps.

❖ Factory Pattern — ResumeAnalyzerFactory

Aspects	Details
Benefit	Allows clean creation of analyzers (`KeywordResumeAnalyzer` or `AIResumeAnalyzer`) without exposing the creation logic to the controller. This makes the system flexible: if new types of analyzers are added in the future (e.g., `AdvancedAIResumeAnalyzer`), no change is needed in the controller code — only the Factory needs to be updated.
Trade-off	Slightly adds an extra layer of abstraction. If the factory becomes very complex (many conditions), it could become a "God factory," which is harder to maintain unless organized properly.

❖ Observer Pattern — NotificationService

Aspects	Details
Benefit	Achieves high decoupling between the notification logic and the core application. Whenever a notification-worthy event occurs (like analyzing a resume), the application can notify multiple observers (email, SMS, push) without the controller needing to know who or how notifications are sent. New notification types can be added dynamically without modifying existing code.
Trade-off	Observers can cause performance problems if too many notifications are triggered in a chain without careful handling (e.g., dozens of observers triggered for a single event). Also, debugging observer interactions can be harder if not properly logged.

- **Summary Table for Quick Revision:**

Pattern	**Benefit**	**Trade-off**
Singleton	Centralized shared instance, saves memory	Harder unit testing if overused
Factory	Cleaner object creation, scalable analyzer types	Factory class can grow complex
Observer	Decouples notifications from main logic	Can impact performance if overused

---

Forth: Code Samples with Justifications

1. Singleton Pattern — AuthService

- **services/AuthService.js**

```
class AuthService {

  static instance;

  constructor() {

    if (AuthService.instance) return AuthService.instance;

    AuthService.instance = this;

  }

  async register({ name, email, password, subscription, role }) {

    const existing = await User.findOne({ email });

    if (existing) throw new Error('User already exists');

    const hashed = await bcrypt.hash(password, 10);

    const user = new User({ name, email, password: hashed,
subscription, role });

    await user.save();
    return user;

  }

  async login({ email, password }) {

    const user = await User.findOne({ email });

    if (!user) throw new Error('User not found');

    const valid = await bcrypt.compare(password, user.password);

    if (!valid) throw new Error('Invalid password');

    const token = jwt.sign({ id: user._id, role: user.role },
process.env.JWT_SECRET, { expiresIn: '2h' });

    return { user, token };

  }

}

module.exports = new AuthService();  // Singleton instance
```

🔵 Justification:

- The `AuthService` class controls both login and register operations.
- Only one instance (`new AuthService()`) is created and exported.
- Controllers (`authController.js`) import and use the same instance everywhere.
- This matches Singleton pattern principles: one shared global access point, controlled instantiation.

---

2. Factory Pattern — ResumeAnalyzerFactory

- ❖ **factories/ResumeAnalyzerFactory.js**

```
const KeywordResumeAnalyzer =
require('../services/analyzers/KeywordResumeAnalyzer');
```

```
const AIResumeAnalyzer =
require('../services/analyzers/AIResumeAnalyzer');

class ResumeAnalyzerFactory {

  static getAnalyzer(subscriptionType) {

    if (subscriptionType === 'premium') {

      return new AIResumeAnalyzer();

    } else {

      return new KeywordResumeAnalyzer();

    }

  }

}

module.exports = ResumeAnalyzerFactory;
```

🔵 **Justification:**

- The controller doesn't decide which analyzer class to create.
- Instead, it asks the Factory and the factory decides based on subscription.
- This shows the Factory Pattern principle: centralized object creation logic, hiding complexity from the user (controller).

---

3. Observer Pattern — NotificationService

❖ observers/NotificationService.js

```
class NotificationService {

    constructor() {

      this.observers = [];

    }

    attach(observer) {

      this.observers.push(observer);
    }   detach(observer) {

      this.observers = this.observers.filter(obs => obs !==
observer);

    }

    notify(message) {

      this.observers.forEach(observer =>
observer.update(message));

    }

  }

  module.exports = NotificationService;
```

❖ **Example usage in resumeController.js**

```
 const notifier = new NotificationService();
    notifier.attach(new EmailNotifier());
    notifier.attach(new SMSNotifier());
    notifier.attach(new PushNotifier());
    notifier.notify(`Resume analyzed. Score: ${result.score}`);
```

🔵 Justification:

- The controller attaches multiple notifiers (observers) dynamically.
- The controller only calls `notify()` once — and all observers react automatically.
- This perfectly matches the Observer Pattern principle: one-to-many dependency, automatic update propagation.