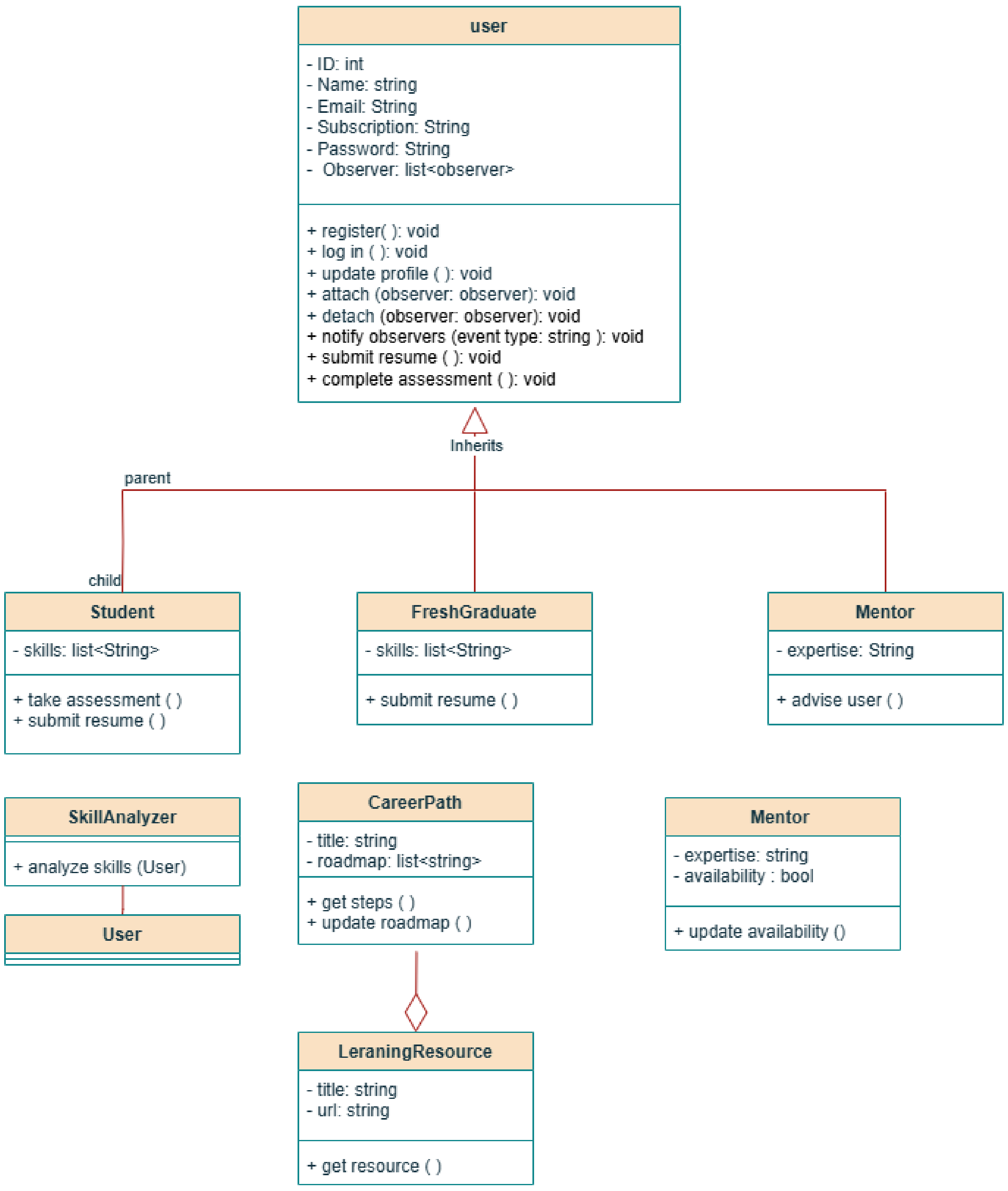
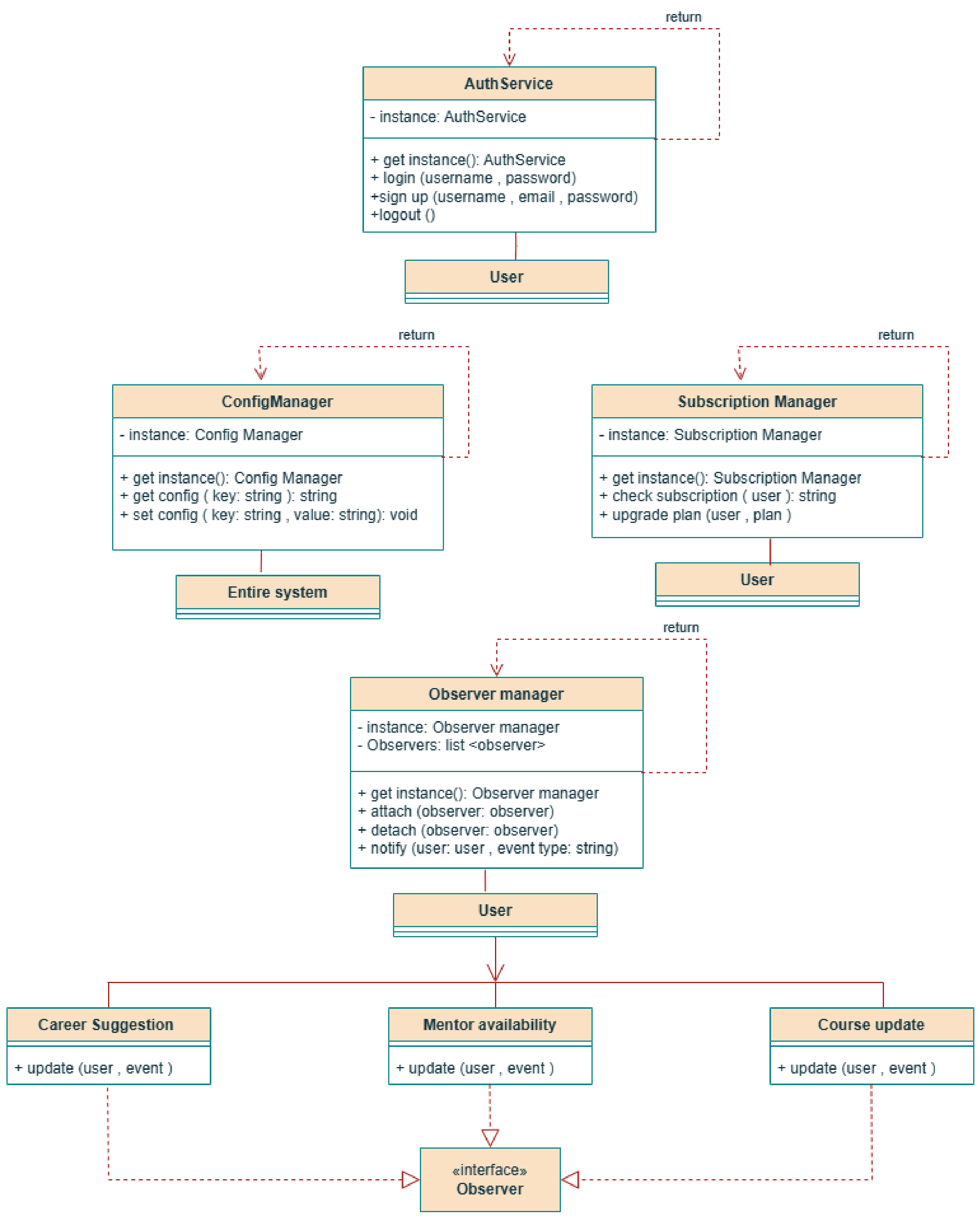
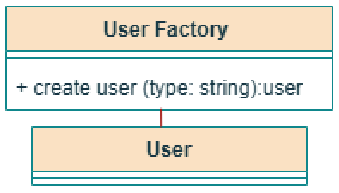
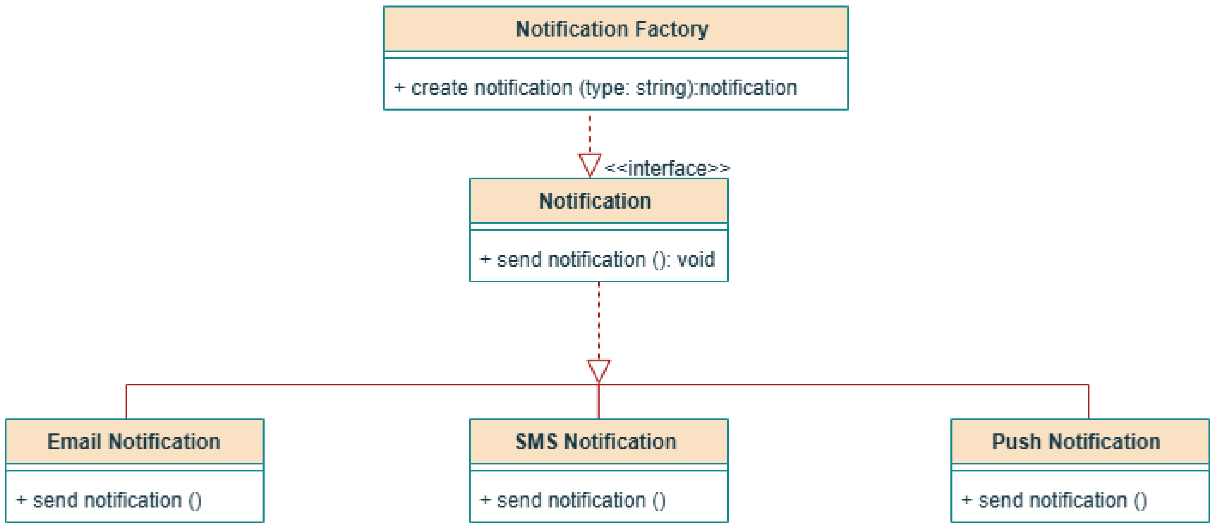
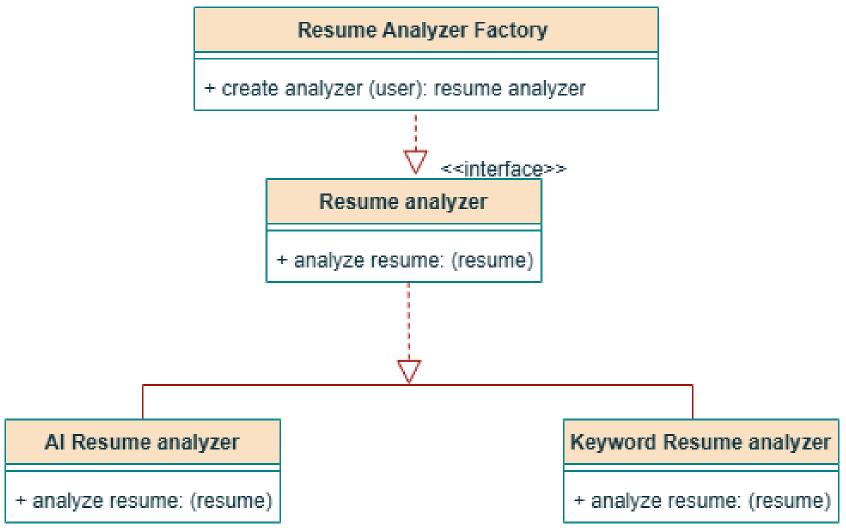
1/ class diagram with applied design patterns:




return
AuthService
- instance: AuthService
+ get instance(): AuthService
+ login (username , password)
+sign up (username , email , password)
+logout ()
User
return
ConfigManager
- instance: Config Manager
+ get instance(): Config Manager
+ get config (key: string): string
+ set config (key: string , value: string): void
Entire system
return
Subscription Manager
- instance: Subscription Manager
+ get instance(): Subscription Manager
+ check subscription (user): string
+ upgrade plan (user , plan)
User
return
Observer manager
- instance: Observer manager
- Observers: list <observer>
+ get instance(): Observer manager
+ attach (observer: observer)
+ detach (observer: observer)
+ notify (user: user , event type: string)
User
Career Suggestion
+ update (user , event)
Mentor availability
+ update (user , event)
Course update
+ update (user , event)
«interface»
Observer


User Factory
+ create user (type: string):user
User


Notification Factory
+ create notification (type: string):notification
<<interface>>
Notification
+ send notification (): void
Email Notification
+ send notification ()
SMS Notification
+ send notification ()
Push Notification
+ send notification ()


Resume Analyzer Factory
+ create analyzer (user): resume analyzer
<<interface>>
Resume analyzer
+ analyze resume: (resume)
AI Resume analyzer
+ analyze resume: (resume)
Keyword Resume analyzer
+ analyze resume: (resume)

2/ Explanation of responsibilities per class:

== Singleton Classes

1. AuthService (Singleton)
    - Manages user authentication (login/logout).
    - Ensures there is a single instance handling authentication across the system.
2. SubscriptionManager (Singleton)
    - Manages user subscriptions (free vs. premium).
    - Provides methods to check the user's plan and upgrade it.
3. ConfigManager (Singleton)
    - Stores system-wide configurations.
    - Ensures configuration values can be accessed globally.
4. ObserverManager (Singleton)
    - Manages the list of observers.
    - Controls attachment, detachment, and notification of observers.

ObserverManager is a Singleton because:

- It centralizes observer management.
- It prevents redundant observer lists inside each `User`.
- It ensures global consistency and improves memory efficiency

== Observer Pattern Classes

5. Observer (Interface)
    - Acts as the base for all observers.
    - Defines the `update(user, event)` method, which all observers implement.
6. CareerSuggestion (Observer)
    - Monitors when users submit resumes or complete assessments.

  - Provides career suggestions based on the extracted skills.

7. MentorAvailability (Observer)
   - Tracks when a user updates their profile (to match with available mentors).
   - Notifies users if a mentor becomes available.

8. CourseUpdate (Observer)
   - Notifies users when a new relevant course is added based on their skills.

Explanation:

- `User` notifies observers when certain events happen (e.g., `submitResume()`, `completeAssessment()`).
- The `User` class does not own the observers permanently, but it depends on them at runtime.

== User and Subclasses

9. User (Observable)
   - Represents a system user.
   - Can be observed by CareerSuggestion, MentorAvailability, and CourseUpdate.
   - Provides core methods like `register()`, `login()`, and `updateProfile()`.

10. Student (Subclass of User)

- Specifically represents students.
- Can take skill assessments and submit resumes.

11. FreshGraduate (Subclass of User)

- Represents users with work experience.
- Focuses on submitting resumes.

12. CareerAdvisor (Subclass of User)

- Provides career advice to students.

== Factory Pattern Classes

13. UserFactory (Factory)

- Creates instances of different types of users.

14. NotificationFactory (Factory)

- Creates different types of notifications (Email, SMS, Push).

15. Notification (Base Class)

- Base class for all notifications.
- Defines the `sendNotification()` method.

16. EmailNotification / SMSNotification / PushNotification (Concrete Notification Subclasses)

- Implement `sendNotification()` differently based on the medium.

17. ResumeAnalyzerFactory (Factory)

- Decides which resume analyzer to use (AI or Keyword-based).

18. ResumeAnalyzer (Interface)

- Defines the `analyze(user)` method, implemented by concrete analyzers.

19. AIResumeAnalyzer / KeywordResumeAnalyzer (Concrete Resume Analyzers)

- AIResumeAnalyzer: Used for premium users.
- KeywordResumeAnalyzer: Used for free users.

== Career & Learning System

20. SkillAnalyzer

- Analyzes the skills of a user.
- Helps generate career suggestions.

21. CareerPath

- Represents a career track with steps.
- Links to learning resources.

22. LearningResource

- Stores educational materials.
- Can be accessed through career paths.

3/ Relationships Between Classes

1. Observer Pattern Relationships

- **User (Observable) is observed by:**
  - **CareerSuggestion → Notifies users of career paths based on resume analysis.**
  - **MentorAvailability → Notifies users when a mentor is available.**
  - **CourseUpdate → Notifies users of new courses.**
- **ObserverManager (Singleton) manages observers:**
  - **Stores observers in a list.**
  - **Ensures `attach()`, `detach()`, and `notifyObservers(eventType)` work.**
- **When a User performs an action (e.g., submits a resume), `notifyObservers(eventType)` is called, triggering all relevant observers.**

OOP relations:

== ObserverManager (Singleton) → Itself

- **Arrow Type: Curved arrow pointing to the class**
- **OOP Relationship Name: "Singleton Pattern (Static Self-Reference)"**
- **Explanation:**
  - **The `ObserverManager` follows the Singleton pattern, meaning it has a static reference to itself via `getInstance()`.**
  - **This ensures that only one instance of `ObserverManager` exists in the system.**

== `User → ObserverManager`

- **Arrow Type: Solid line (association)**
- **OOP Relationship Name: Association**

Explanation:
User interacts with `ObserverManager` to attach/detach observers and notify them when events happen.
This is a direct association, meaning `User` uses (but does not own) the `ObserverManager` instance.
It is accessed through a static `getInstance()` method, since `ObserverManager` is a Singleton.

== `User → Observer (Interface)`

- **Arrow Type: Solid line (association)**
- **OOP Relationship Name: Association**

Explanation:
The `User` class maintains a list of observers (`list<Observer>`) and notifies them upon events.
This is part of the Observer pattern, where the subject (User) holds references to observers via their interface.
The link is maintained dynamically (attach/detach), not hard-coded.

== `ObserverManager → Observer (Interface)`

- **Arrow Type: Solid line (association or aggregation)**
- **OOP Relationship Name: Aggregation (or Association)**

Explanation:
`ObserverManager` maintains a list of observers (`list<Observer>`) to manage global observer handling.
Because the observers are passed in and not owned by `ObserverManager`, this can be seen as aggregation.
Still, in UML, if no lifecycle dependency is modeled, it is shown as a plain association.

== `User → Concrete Observers (CareerSuggestion, MentorAvailability, CourseUpdate)`

- **Arrow Type: Solid line (association — in context of the pattern)**
- **OOP Relationship Name: Observer Pattern Association**

Explanation:
The `User` notifies these concrete observers (if attached).
Although the reference is to `Observer` interface, the actual implementation is one of the concrete classes.
In UML, this is typically represented by associating `User` with the `Observer` interface — not every implementation.

== `CareerSuggestion, MentorAvailability, CourseUpdate → Observer`

- **Arrow Type: Dashed line with white triangle (realization)**
- **OOP Relationship Name: Realization**

Explanation:
Each of these concrete observer classes implements the `Observer` interface.
This is a standard interface-implementation relationship.
In UML, this is shown as a dashed line with a white triangle pointing to the interface.

---

2. User and Subclass Relationships

- **User has subscriptions managed by:**
  - **SubscriptionManager (Singleton)**
- **Student and FreshGraduate interact with:**
  - **ResumeAnalyzer (AI or Keyword)**
  - **CareerSuggestion (Observer)**
- **CareerAdvisor provides career advice.**

- **User has authentication controlled by:**
    - **AuthService (Singleton)**

OOP relations:

== User → Student, FreshGraduate, and CareerAdvisor (User is the parent class of them)

- **Arrow Type: Solid line with a hollow triangle**
- **OOP Relationship Name: "Inheritance (Generalization)"**
- **Explanation:**
    - `Student`, `FreshGraduate`, and `CareerAdvisor` **inherit from the** `User` **class.**
    - **This represents a generalization relationship where** `User` **is the parent class (superclass), and the others are subclasses.**
    - **This allows polymorphism: a** `Student` **or** `FreshGraduate` **can be treated as a generic** `User` **when necessary.**

== User → AuthService

- **Arrow Type: Solid line (association)**
- **OOP Relationship Name: "Association"**
- **Explanation:**
    - **The** `User` **interacts with** `AuthService` **during login, registration, and authentication.**
    - **This is a direct association, meaning** `User` **relies on** `AuthService` **to verify credentials.**

---

3. Factory Pattern Relationships

- **UserFactory creates different types of users.**
- **NotificationFactory generates different types of notifications.**
    - **Calls** `createNotification(type)`**.**
    - **Returns either EmailNotification, SMSNotification, or PushNotification.**
- **ResumeAnalyzerFactory determines whether:**
    - **AIResumeAnalyzer is used (for premium users).**

    - **KeywordResumeAnalyzer is used (for free users).**

== ResumeAnalyzerFactory → ResumeAnalyzer (Interface)

- **Arrow Type: Dashed line with a hollow triangle**
- **OOP Relationship Name: "Realization (Implementation)"**
- Explanation:
    - The `ResumeAnalyzerFactory` creates objects that implement the `ResumeAnalyzer` interface.
    - This follows the Factory Method Pattern.

== ResumeAnalyzer (Interface) → Concrete Analyzers (AIResumeAnalyzer, KeywordResumeAnalyzer)

- **Arrow Type: Dashed line with a hollow triangle**
- **OOP Relationship Name: "Realization (Implementation)"**
- Explanation:
    - `AIResumeAnalyzer` (for premium users) and `KeywordResumeAnalyzer` (for free users) implement the `ResumeAnalyzer` interface.
    - The system chooses the correct analyzer using the `ResumeAnalyzerFactory`.

🔹 4. Resume Analysis & Career Growth

- **ResumeAnalyzer (AI or Keyword) is called when:**
    - **A user submits a resume.**
    - **Analyzes the skills and generates a report.**
- **SkillAnalyzer extracts key skills from the user’s profile.**
    - **Feeds data into CareerSuggestion.**
- **CareerPath is linked to LearningResource.**
    - **Users can access learning materials to improve their skills.**

Summary table:

From Class	To Class	Relationship Type	Relation Name (OOP)
`ObserverManager`	`Observer (Interface)`	Association	`Manages`
`User`	`ObserverManager`	Association	`Uses`
`User`	`Observer (Interface)`	Association (Observable-Observer)	`Notifies`
`Observer`	`CareerSuggestion, MentorAvailability, CourseUpdate`	Inheritance	`Implements`
`User`	`Student, FreshGraduate, CareerAdvisor`	Inheritance	`Extends`
`AuthService (Singleton)`	`User`	Association	`Authenticates`
`SubscriptionManager (Singleton)`	`User`	Association	`Tracks Subscription`
`SkillAnalyzer`	`User`	Association	`Analyzes Skills`
`ResumeAnalyzerFactory`	`ResumeAnalyzer`	Factory Method	`Creates`
`UserFactory`	`User`	Factory Method	`Creates`
`NotificationFactory`	`Notification`	Factory Method	`Creates`
`CareerPath`	`LearningResource`	Aggregation	`Has`