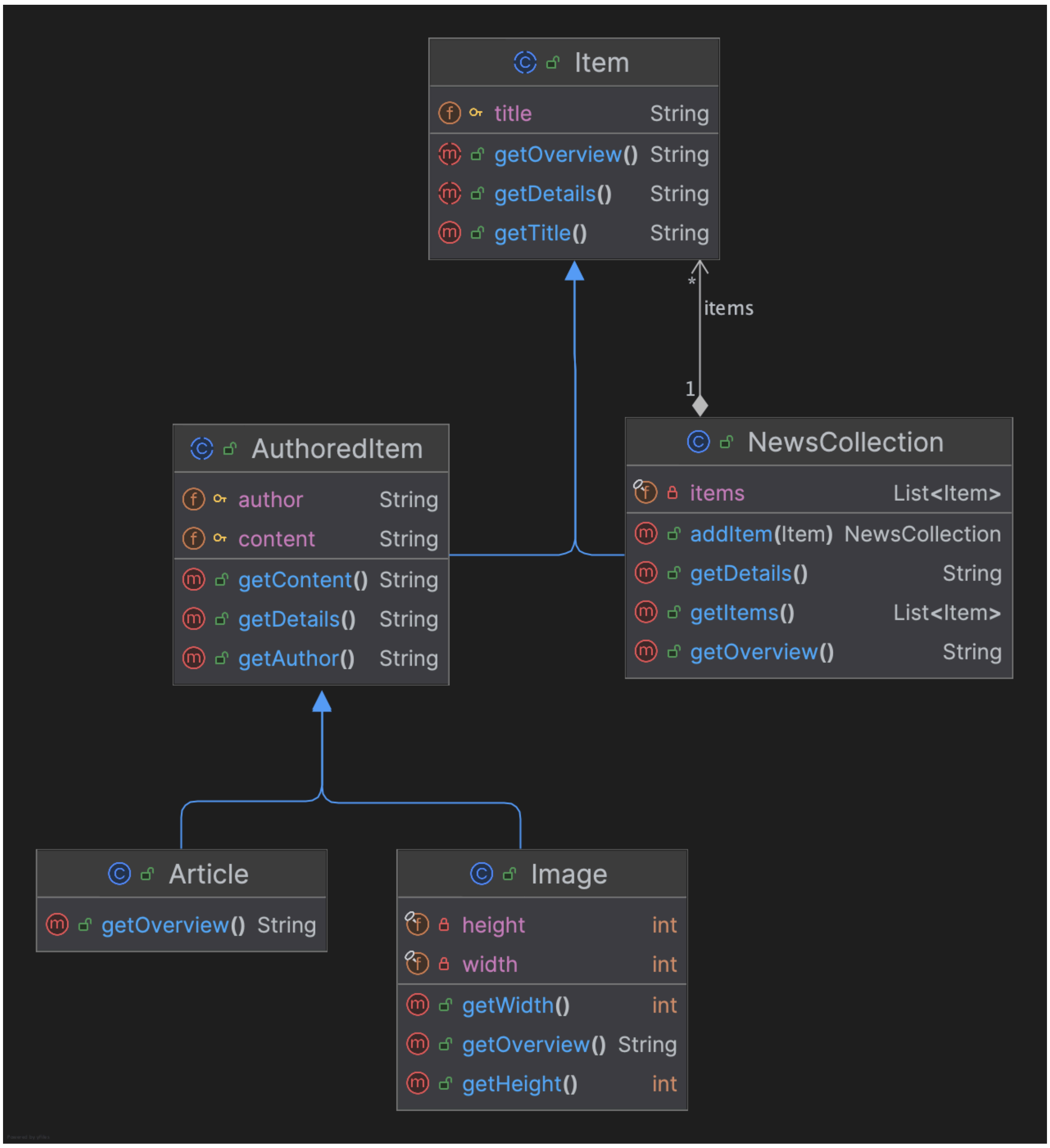
Task 1

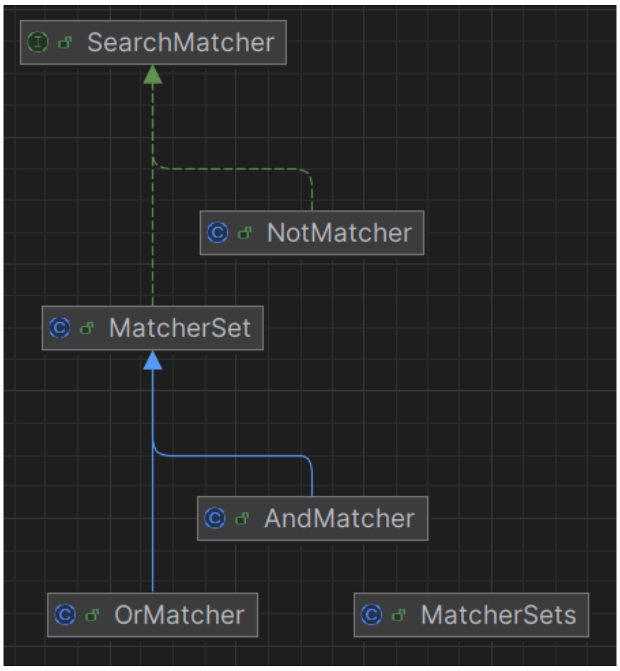
a)

Below you can see the class diagram of our solution of 1 b).
The *Component* is represented by **Item**. The *Atomics* **Article** and **Image** inherit from the abstract class **AuthoredItem**. **NewsCollection** is the *Composite*.



Task 2

(2- a)



Composite Pattern

The Composite pattern allows for the composition of objects into tree structures to represent part-whole hierarchies. In this case, MatcherSet can contain multiple SearchMatcher objects, including other MatcherSet objects, allowing for complex search criteria to be built.

Include:

- Component interface
  - `SearchMatcher`. Defines the common interface for all objects in the composition.
- Composite abstract class
  - `MatcherSet`. Acts as a container for SearchMatcher objects and can contain other MatcherSet objects
- Concrete Composites
  - `AndMatcher`. Provide specific behavior by extending abstract composite class: returns true only if ALL matchers in the set match the entry
  - `OrMatcher`. Provide specific behavior by extending abstract composite class: returns true if ANY matcher in the set matches the entry
- Leaf
  - `NotMatcher`. Represents an individual matcher that does not contain other matchers.

Difference: In the classic Composite pattern, a program usually has methods to add and remove child components freely. However, in the `MatcherSet` class the design is a bit more restrictive. Instead of allowing us to remove matchers (children), it uses a protected list to manage them. This means new matchers can be added, but once they're in, they can't be taken out. This approach keeps things simpler and more controlled, but it also means less flexibility compared to the traditional Composite pattern.

Factory Method Pattern

The Factory Method pattern usually has a method that decides which specific class to instantiate. In the `MatcherSets` class, the `build` method creates either an `AndMatcher` or an `OrMatcher` based on the `MatcherType` provided. This way, the client doesn't need to know the details of how each matcher is created; we just call the `build` method and get the right type of matcher.

Include:

- Creator
    - `MatcherSets`. Declares the factory method that returns an object of type MatcherSet.
- Abstract Product
    - `MatcherSet`. Defines the interface of objects the factory method creates.
- Concrete Products
    - `AndMatcher`. Implement the MatcherSet interface and are created by the factory method.
    - `OrMatcher`. Implement the MatcherSet interface and are created by the factory method.

Difference:
For the most part, the current implementation of Composite pattern works without differences from the classic one. Often with the Composite pattern instances are created, but since this is not required, the build method is called directly.

(2 - b)

For importers to create file import logic for different formats, the Importer class provides a flexible foundation.

The Importer subclasses (such as format importers) do not wrap or improve other Importer instances; instead, they implement their own logic.
There is no dynamic composition of behavior; instead, each importer subclass (such as BibTeXImporter or RISImporter) defines distinct behavior.

The only exception to this is the CustomImporter class: This class wraps the Importer class and allows to customize the behaviour of the Importer class and its subclasses.
The importer class is designed for extensibility, where each subclass is a standalone implementation for specific formats.

The implementation of the importer class varies from the original design pattern of the decorator pattern. It is rather a mixture of the decorator pattern and the composite pattern. But for the solution of this task we will focus on which classes represent which classes in the decorator pattern.

<u>Component:</u> **Importer**
In this implementation the *Component* is not an abstract class or an interface, but an actual class.

<u>Concrete Component:</u> **Importer & subclasses** (except CustomImporter)
Importer itself and its subclasses (except CustomImporter) are concrete implementations of the *Component*.

<u>Decorator:</u> **CustomImporter**
CustomImporter wraps the *Component* Importer and subsequently all its subclasses.

<u>Concrete Decorator:</u> **CustomImporter**
CustomImporter is not only a *Decorator*, but a *Concrete Decorator*.

What's special about this implementation is that the Importer is the *Component* as well as a *Concrete Component*. Similarly the CustomImporter is the *Decorator* and a *Concrete Decorator*.

A cleaner implementation of the Decorator pattern would create an abstract class or an interface as the *Component*, as well as an interface or abstract class as the *Decorator*.