Critical Analysis of OOP Principles-

1) Classes should be open for extension and closed for modification

In object-oriented programming, the open–closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.
This principle is fairly implemented in the code. The classes, Person and Service which are generalised are extended by Student and Admin to perform their specific tasks. Also, defining private variables prevents modification to some extent.

2) Depend on abstractions, do not depend on concrete classes

In '*simple*' terms, this means that when we rely upon a concrete instance of an object - we're building a dependency in to our code (albeit without that intention), this then limits the ability to re-use it.
In our code, we have failed to follow this principle. The classes we have created are concrete and have many constrains (like the parameters required to declare the constructor) which prevent us from using it in some other situation. An interface based format might have helped with this.

3) Program to an interface, not implementation

Coding to interfaces is a technique to write classes based on an interface; interface that defines what the behavior of the object should be. It involves creating an interface first, defining its methods and then creating the actual class with the implementation.
This principle again could have been solved, and ideally should have been by creating interfaces of the classes that we were trying to implement. This would have solved two problems, It would have increased the flexibility of the code as well as implemented a test driven development. This would have further in turn helped our group as we coded as well.

4) Favour composition over inheritance

Favoring Composition over Inheritance is a principle in object-oriented programming (OOP). Classes should achieve polymorphic behaviour and code reuse by their composition rather than inheritance from a base or parent class. To get the higher design flexibility, the design principle says that composition should be favored over inheritance.
Undoubtedly, this concept grants higher flexibility but in our particular case, it feels that favouring composition instead of inheritance would be an overkill. The inheritance principle fairly satisfies what we want to implement with minimal addition to code. If we were to expand with multiple teachers and multiple courses, composition would definitely come in handy.

Observations of code with respect to Creational Design Pattern:

Our code, as we have been expecting, fail to implement the creational design pattern. However, after learning about this particular pattern, it has become evident as to how we could have used this design pattern to better our code in both sense and efficiency.

Creational Design Pattern mainly deal with class instantiation and object instantiation. We in our own program, use these concepts which could have been improved.

Abstract Factory method and Factory method pattern – It states to define just the interface/abstract class and let the subclasses decide which class to instantiate. In our particular code, we could have let the Person parent class to have been an instance class and let student and teacher each implement them, thus inceasing what could have been done.

Singleton - Singleton Pattern says that just"define a class that has only one instance and provides a global point of access to it".
In our particular code, Test file is common to all classes. What we should have done was to create a global instance of this class which would in turn allow all objects to call it instead of initializing it multiple times.

Builder – This pattern suggests to construct a complex object from simple objects using a step-by-step approach. In our program, StudentService and AdminService classes can be considered as complex classes. These are nested classes which already extend Person class and Service class. This layered manner of object creation could have been divided in a better by use of properly defined classes interfaces.

Further, looking back, we surely could have kept in mind several concepts of Structural Design patterns and Behavioural Design patterns to better our code.