1. In this case, a lot of code does not respect the Open-Closed principle. It would be better if the class ControlPanel inherits from other classes, due to the separation of concerns. And in the ControlPanel class, it will use the parameters in terms of light, such as lightOnCmd, lightOffCmd and does not contain some basic functionalities to be inherited for an electronic. If we want to add some new electronics, we have to rewrite the code, resulting in the violation of Open-Closed principle. ControlPanel method follows the Open-Closed principle, because it has four parameters: LightOnCommand, lightOnCmd, LightOffCommand, lightOffCmd, and separating the concerns help us modify the code easily instead of rewriting.
2. In this case, the code respects the Liskov Substitution. We say S is a subtype of T if we can replace object1(S) with object2(T). From the code, we know IndicatorLight is the subclass and Light is the parent class; they have different type, but they have the same functionality. The subclass does not perform some other functions, both of them have light on and light off and have same main functionality. However, rectangle and square won't respect the Liskov Substitution. It is true square is a type of rectangle, but a square's width and height are equal; changing a rectangle's width does not affect its height or changing a rectangle's height does not affect its width. If we derive a square from a rectangle, its width and height are always equal. Neither type is properly a subtype of the other, so that this case does not respect the Liskov Substitution.
3. The interface Device contains some basic functions: On(), Off(), and IndicatorLight and SecurityCamera would use these functions as well. However, startRecording() and stopRecording() are only suitable for SecurityCamera instead of IndicatorLight, this design does not respect Interface Segregation, because the principle is that Cilents should not be forced to depend on interfaces that they do not use.
   Positive consequences with Segregation Principle: it can increase the readability and maintainbility of our code.

   Negative consequences with Segregation Principle: some needless methods/classes may be confused.

   Positive consequences without Segregation Principle: there won't be too many interfaces, because developers may create a separate interface for each of non-overlapping subset.

   Negative consequences without Segregation Principle: it is possible that big interfaces are not cohesive.
   In the future, since IndicatorLight is forced to depend on an interface that contains some not suitable things, so that each class that implements such an interface will throw an error which is not relevant to the class.
4. a) For classes, a dependency means a class uses another class. If class A is mentioned in class B, then we say A is depends on B. For an application, the dependency is a component or part of the application functionally without which is partially or completely impossible, it could be the database, a library. In my opinion, most of dependencies point down, because we describe that high level points low level as point down and high level classes usually call the low classes. Low classes contain some basic functionality, higher classes will use

these functionalities, so they usually call the lower classes, then it’s point down. It is possible to have a mix of different dependencies in some special cases.

b)

Original code:

```java
public class ISU{
        public void cs352(){
            System.out.println(“I am studying cs352”);
        }
        Public void cs362(){
            System.out.println(“I am study cs362”);
        }
}
```

After dependency inversion:

```java
Public interface course {
        Void study();
}
Public class cs352 implements course {
        @Override
        Public void study(){
            System.out.println(“I am studying cs352”);
        }
}
Public class cs362 implements course{
        @Override
        Public void study(){
            System.out.println(“I am study cs 362”);
        }
}
Public class ISU{
        Public void study(course c){
            c.study();
        }
}
```

Add an interface, and cs352 and cs362 implement it. In ISU class, use course(interface) as parameter and to call study().

c) In the example I provided in problem b, the benefit is obvious. For a new course, we only need to create a new class and pass the parameter for a new course, and we do not need to change the code in lower classes. To sum up, it can reduce the coupling between classes and improve the stability of the system, and it is also able to make the code readability and maintainability (decrease the risk if stakeholders change the requirements).