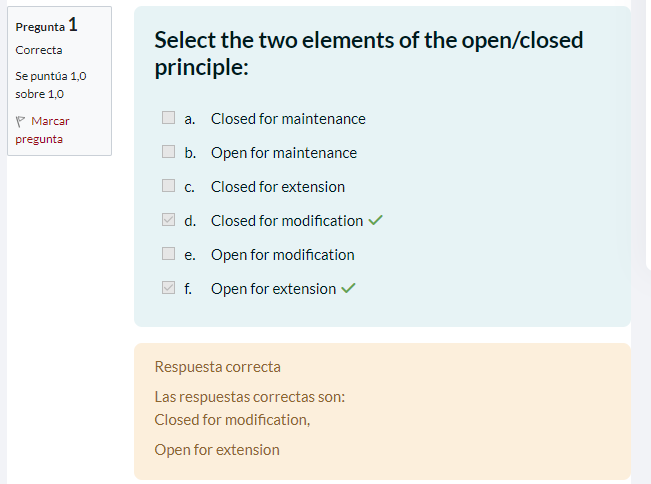
***Partial correction:***

***Software Modeling***

****

**Justification:**

I chose: Closed for modification and Open for extension.

Because the SOLID (Open/Closed) principle establishes that the code must be open for extension, that is, we must write it so that we can add new functionality, without changing the existing code that is closed for modification.



Interfaz de usuario gráfica, Texto, Aplicación

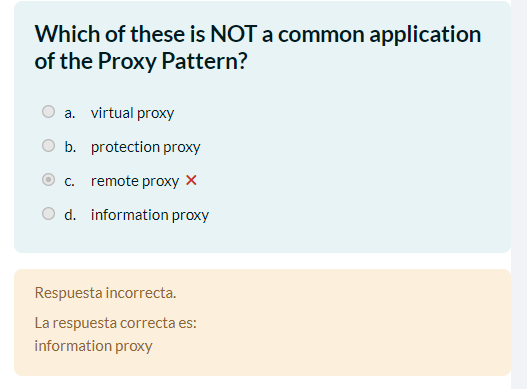
Descripción generada automáticamente

**Justification:**

I chose: Legacy rejected.

Because of the Interface Segregation Principle, as it suggests that clients should not be forced to rely on interfaces they do not use, i.e., that an interface should have only the methods that are relevant to the implementation class. And that's why you can avoid code odors related to unnecessary dependencies and bloated interfaces.





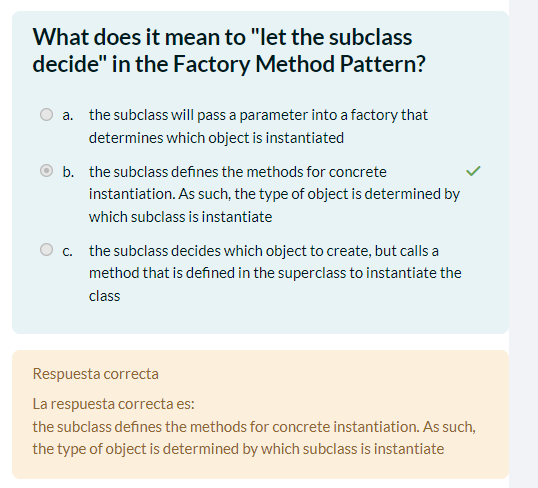
**Justification:**

The Proxy pattern is used to control access to an object or to provide a substitute for an expensive or remote object. Common types of proxy include Virtual Proxy, which creates expensive objects only when necessary; the Protection Proxy, which controls access to an object; and Remote Proxy, which represents objects in a different address space, such as remote objects on a network or web services. However, information proxy is not a common use of the Proxy pattern.

**Error:**

Error: select the remote proxy, since it was the only one I thought I had not heard of, and consider that the information proxy referred to the chace memory.

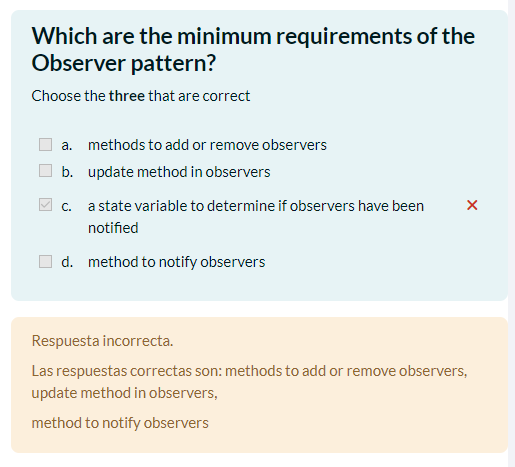




**Justification:**

Select option b, because in the Factory Method pattern, subclasses define the methods for the specific creation. That is, each specific subclass implements its own factory method to create specific objects.





**Justification:**

Methods for adding or removing observers: The Observer pattern implies that an object the subject, maintains a list of observers interested in its changes. Therefore, it is necessary to have methods to add and remove observers from this list.

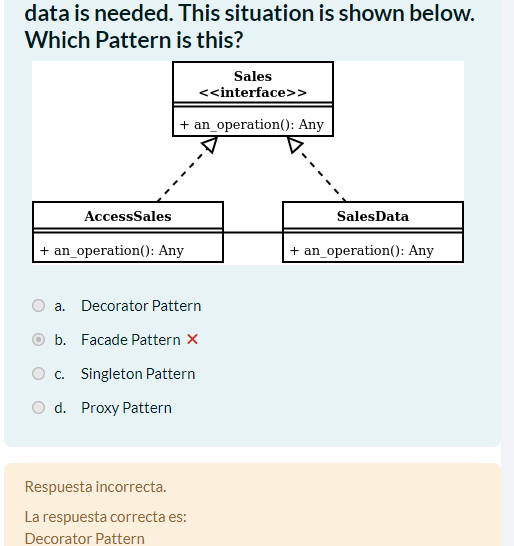
Update method in observers: Observers must have an update method that is called when the subject changes its state. This method allows observers to react to relevant changes.

Method for notifying observers: When the subject changes its status, it must notify all registered observers. Therefore, a method is needed to notify observers about changes.

**Error:**

I don't know how I could have been wrong because a state variable to determine if the observers have been notified is not something necessary to implement it.





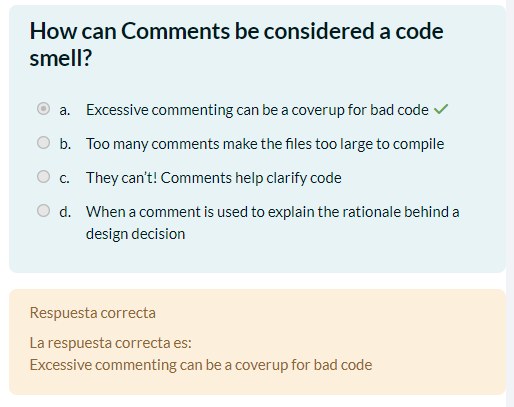
**Justification:**

The Decorator Pattern is best suited for this case because it allows adding additional functionality to an existing class without modifying its original structure. In other words, you can “decorate” an object with extra features without altering its code base. In this case, the object to decorate is the SalesData class. The pattern allows you to extend the capabilities of SalesData using decorators such as AccessSales, which provides curated sales data based on user credentials.

**Error:**

I realized that I did not understand the question well, or well I could not translate it correctly, since I took it as if two different calls were made to the interface depending on what we wanted to do.





**Justification:**

I select Excessive Commenting, because if a code has too many comments, it is a red flag, as it could be a sign that the code itself is not clear enough. And the programmer, realizing that his code was not readable or self-explanatory, decided to go crazy and explain the code through comments.



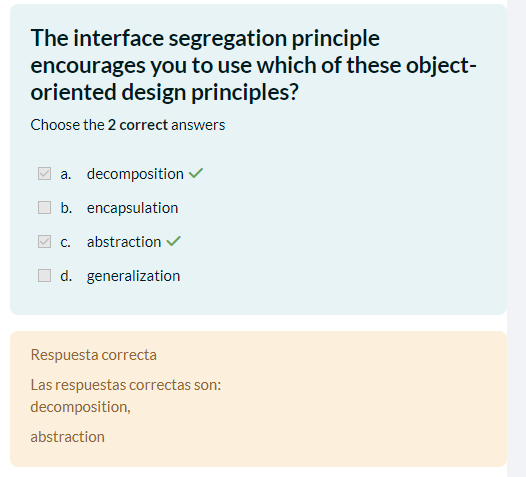
Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

**Justification:**

The Observer Pattern is the most suitable for this case, since it allows establishing a one-to-many relationship between objects. In this scenario, we have one class that represents a mailbox and another that is the owner of the mailbox. The owner wants to know when new mail arrives in the mailbox. The Observer Pattern allows the owner (observer) to register to receive notifications when the mailbox (subject) changes its state. The mailbox will automatically notify the owner when there are changes.





**Justification:**

The Interface Segregation Principle suggests that we should divide large interfaces into smaller, specific parts (decomposition) and create abstract interfaces that are relevant to each client (abstraction). This improves cohesion and prevents classes from implementing unnecessary methods.

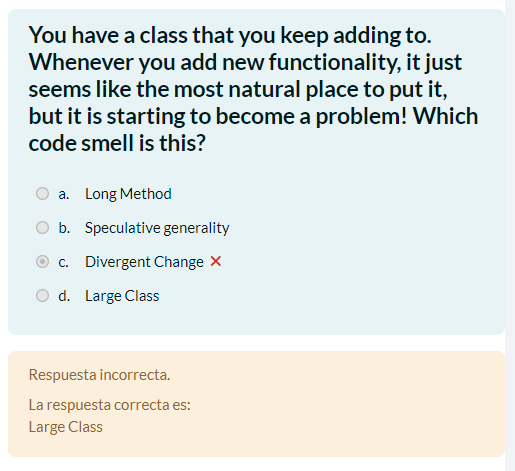


Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Justification:**

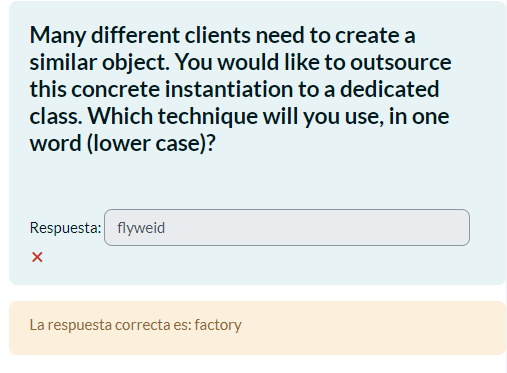
a, is the correct option and that by having multiple potential handlers, but only one will be in charge of the request, it is generally the case of using the “Chain of Responsibility” design pattern since this is used when you have a chain of objects that can handle a request, and each object in the chain decides whether it can handle the request or should pass it to the next object in the chain.



**Justification:**

The correct answer is the Large Class odor code. This smell refers to a class that has too many responsibilities and is doing too much. Being that ideally, a class should have only one responsibility (Single Responsibility Principle). When a class becomes large and spans many lines of code, it may be an indicator that it is taking on too many tasks.





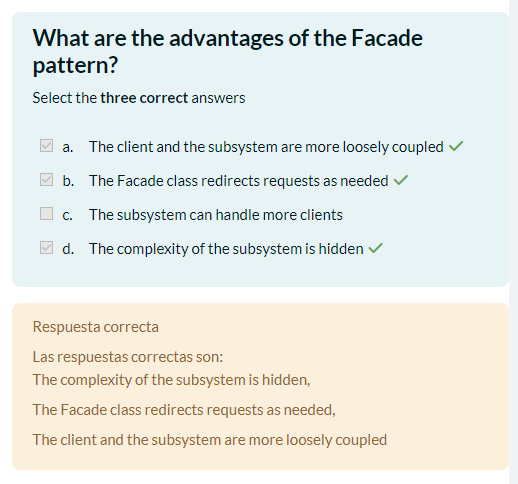
**Justification:**

The Factory pattern is suitable because it centralizes the creation of similar objects in a single class, simplifying client code and facilitating future changes to the creation logic without modifying the client code. Additionally, it improves code reusability and maintainability.

**Error:**

Again I couldn't read the question correctly, I thought it referred to reusing common parts.





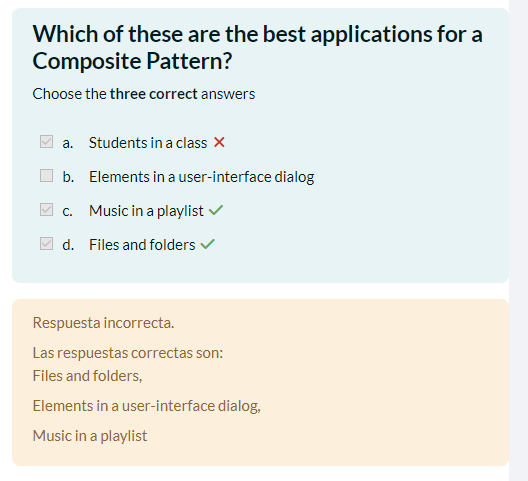
**Justification:**

The client and the subsystem are more loosely coupled: The Facade pattern reduces the direct dependency between the client and the subsystem, making maintenance easier.

The complexity of the subsystem is hidden: Hides the complexity of the subsystem, presenting a simple interface to the client.

The Facade class redirects requests as needed: Redirects client requests to the correct subsystem components, simplifying interaction.





**Justification:**

Elements in a user-interface dialog: The Composite pattern allows user interface elements to be treated uniformly, whether simple or composite, facilitating the management and manipulation of hierarchical structures.

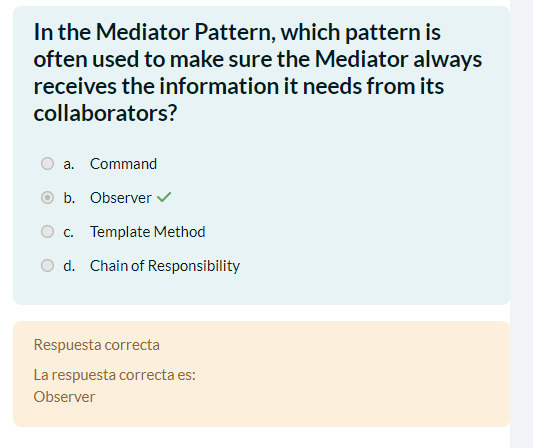
Music in a playlist: The Composite pattern is ideal for playlists, as it allows you to manage both individual songs and groups of songs (playlists) in the same way, simplifying organization and playback.

Files and folders: The Composite pattern is perfect for file systems, as it allows files and folders (which can contain other files and folders) to be treated uniformly, facilitating management operations.

**Error:**

I thought that because the music list was one of the correct ones, so would the student list, and I didn't want to risk it because of the interface because it was difficult for me to visualize it.





**Justification:**

It is a Mediator because the Observer pattern is often used to ensure that the Mediator always receives the necessary information from his collaborators. The Observer pattern allows objects (collaborators) to automatically notify the Mediator of any state changes, ensuring that the Mediator is always aware of relevant updates without requiring constant querying. This facilitates centralized communication and coordination without direct coupling between collaborators.



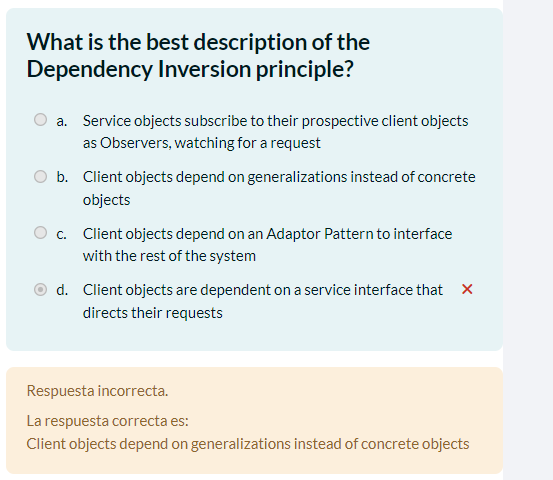
Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Justification:**

a was the correct option because it violates the Principle of Least Knowledge or Law of Demeter, which states that an object should interact only with its immediate friends and not with internal objects of other objects.





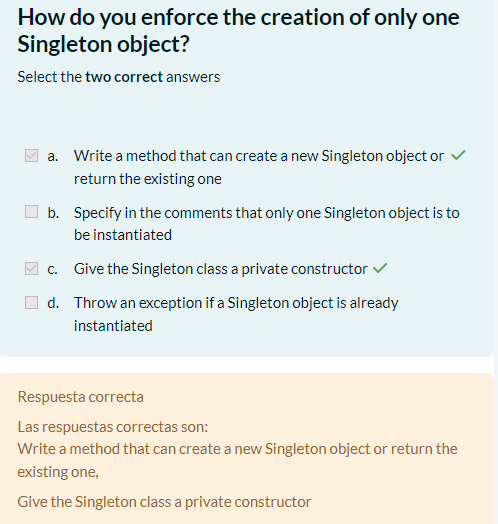
**Justification:**

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules, but rather both should depend on abstractions (for example, interfaces or abstract classes). This principle also implies that abstractions should not depend on details, but rather that details should depend on abstractions.

**Error:**

In this one I didn't quite understand what the answers mean xd



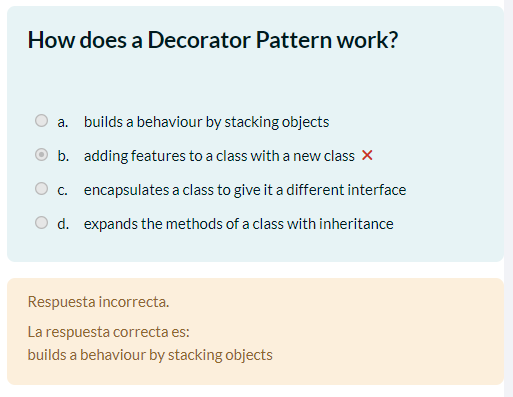


**Justification:**

Write a method that can create a new Singleton object or return the existing one: A static method that handles the creation of the singleton instance.

Give the Singleton class a private constructor: By making the constructor private, you avoid creating instances outside the class.





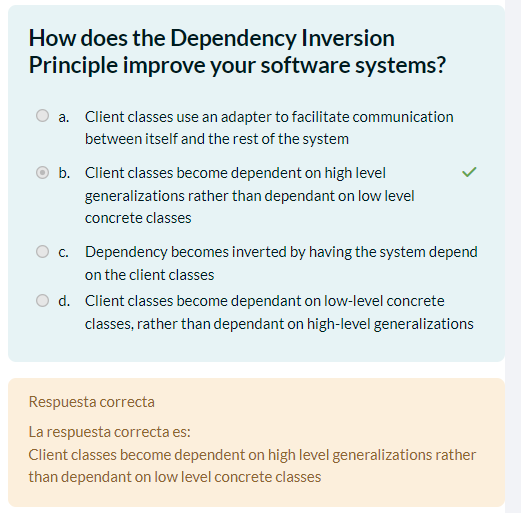
**Justification:**

El patrón Decorador funciona al apilar objetos que tienen funcionalidades adicionales sobre una clase base. Cada objeto decorador envuelve a otro objeto, proporcionando una funcionalidad adicional sin modificar la interfaz de la clase base. Esto permite la composición flexible de comportamientos, ya que los objetos pueden agregarse o eliminarse dinámicamente para modificar el comportamiento de la clase base.

**Error:**

I didn't have the concept very clear, and my English failed to interpret well.

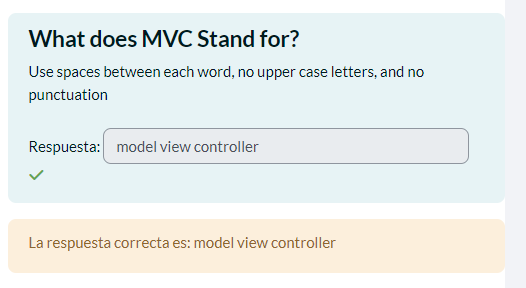




**Justification:**

The Dependency Inversion Principle (DIP) improves software systems by promoting a more flexible and decoupled design. By making client classes depend on high-level generalizations (abstractions), rather than directly depending on low-level concrete classes, the system becomes more modular and easier to maintain. This allows specific implementations to be easily exchanged without modifying the client code, making it easier to adapt the system to future changes and reuse the code.





**Justification:**

without words xd



Texto

Descripción generada automáticamente con confianza media

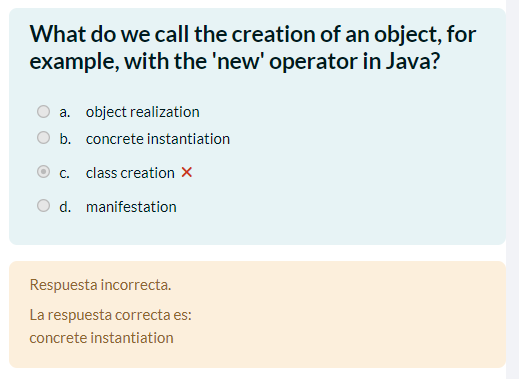
**Justification:**

The Mediator pattern is suitable for coordinating communication between different components of a complex system, as in this case, where you have different sensors and components of a machine represented by separate classes. The Mediator acts as a centralized intermediary that facilitates the interaction between these components, avoiding direct coupling between them.

**Error:**

The question confused me a little, but I considered that you could use command to make requests for classifying the pieces





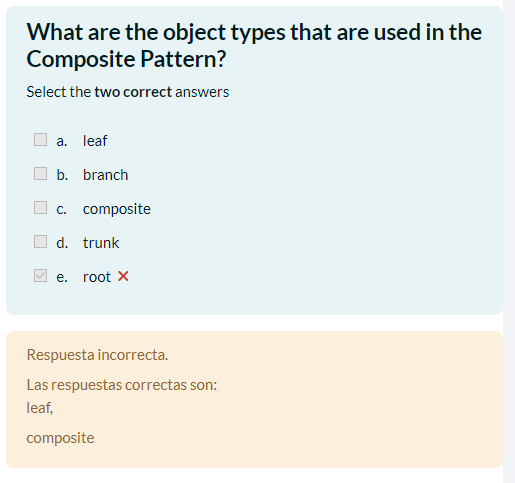
**Justification:**

The correct term for creating an object, for example, with the 'new' operator in Java, is "concrete instantiation". This refers to the process of creating a specific (concrete) instance of a class at runtime.

**Error:**

In this case, I have no justification, I screwed up with such an easy question, I don't know if I didn't think it through with the English terms xd





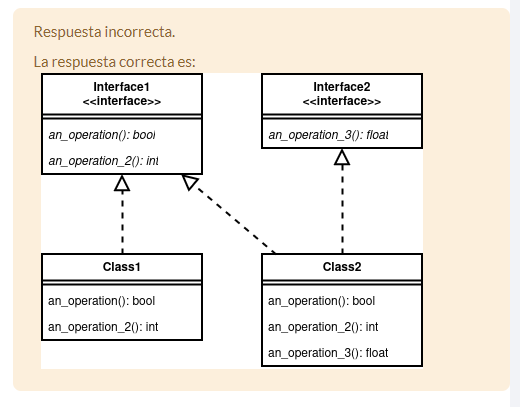
**Justification:**

In the Composite pattern, two types of objects are used: leaves and composites. Leaves represent the individual elements in the hierarchical structure, while compounds represent nodes that may contain other objects, including leaf objects and other compounds.

**Error:**

In this case I did not remember the term "leaf", and decided to go with the tree term that I remembered "root".





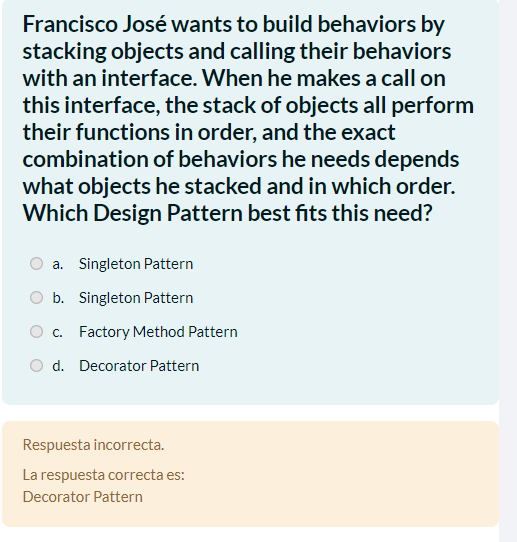
**Justification:**

The correct answer is option b because it shows the segregation of the interface into two separate interfaces, Interface1 and Interface2, each with specific methods that serve the needs of the classes that implement them. This approach follows the Interface Segregation Principle (ISP), which states that clients should not depend on interfaces they do not use, reducing overhead and improving code cohesion and maintainability.

**Error:**

I chose option d, which forces classes to implement methods they do not need, contravening the ISP. Which creates unnecessary dependencies and can lead to increased complexity and the implementation of empty or non-functional methods, reducing efficiency and design clarity.





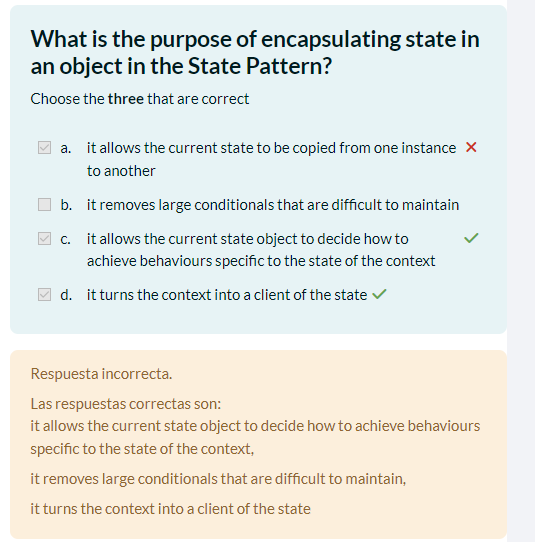
**Justification:**

The Decorator pattern best fits Franz Joseph's need to construct behaviors by stacking objects and calling their behaviors using an interface. In this pattern, multiple decorator objects can be stacked, each adding additional functionality to the chain of objects. When a method is called on the interface, all objects in the chain execute their functions in order, and the exact combination of behaviors depends on the stacked objects and their order.

**Error:**

I cannot understand this question from the translation and decided not to answer.





**Justification:**

b. Remove large conditionals that are difficult to maintain: Eliminates long and complicated conditionals, improving code maintainability.

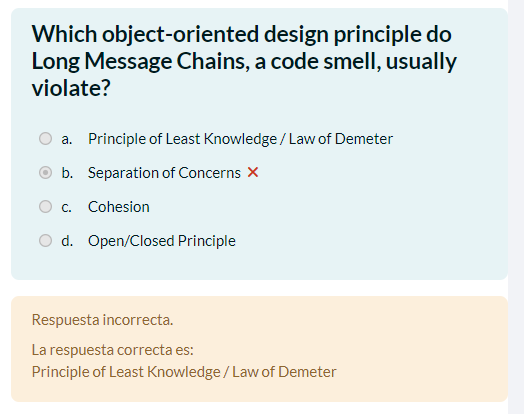
c. Allow the current state object to decide how to achieve behaviors specific to the state of the context: Allow the current state object to determine how to achieve specific behaviors depending on the state of the context, facilitating flexible design.

d. Turn the context into a client of the state: The context becomes a client of the state, which separates responsibilities and allows the context to focus on its core functionality without worrying about state implementation details.

**Error:**

I didn't understand option b very well, and despite my doubts I chose option a (bad option)





**Justification:**

The Long Message Chains violate the Principle of Least Knowledge or Law of Demeter, which states that an object should interact only with its immediate friends and not with internal objects of other objects. Long message chains imply excessive dependency between objects, indicating that one object knows too much about the internal structure of other objects. This can result in overcoupling and loss of encapsulation, making system maintenance and evolution difficult.

**Error:**

I thought about separation of interests since this divides the system into several modules, but in this case, little to do with it xd



Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

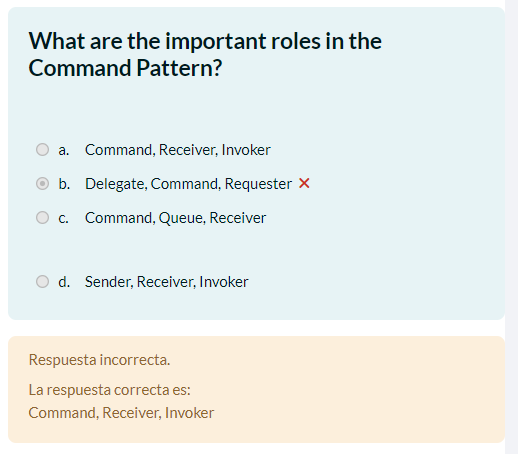
**Justification:**

The Command pattern is not typically used to send commands to external services or third-party libraries, as it is designed to manage internal application operations, such as supporting undo/redo, building macros, and handling the user interface. Other patterns, such as the Adapter pattern or the Facade pattern, are more appropriate for interacting with external services.

**Error:**

Because I didn't see it very clearly implemented for creating a user interface, I decided to put that option.





**Justification:**

In the Command pattern, the key roles are:

Command: Represents a request as an object, encapsulating both the action and its parameters.

Receiver: It is responsible for carrying out the action requested by the command. It can be any object that has the logic necessary to perform the operation.

Invoker: It is responsible for executing commands. Maintains a reference to the command and executes it when necessary. It can be an object that controls the script or a user interface that interacts with the user to execute the commands.

**Error:**

I wasn't clear what the roles were.





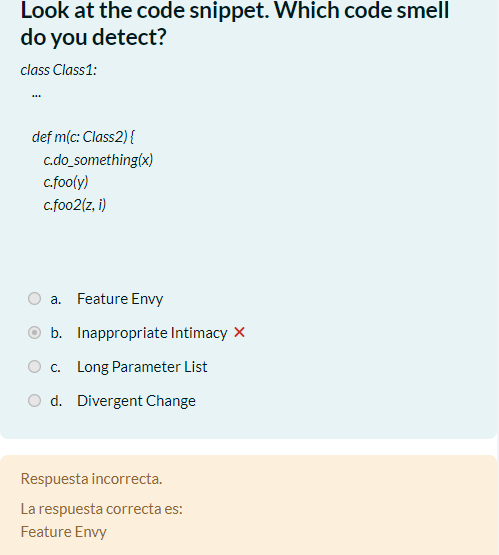
**Justification:**

To connect an application to a third-party library but wanting to keep the coupling loose because it is believed that the library could change in the future, the Adapter pattern is necessary, since this allows us to adapt the interface of the third-party library to an interface consistent that your object can call transparently. This way, the application object can communicate with the library indirectly through a consistent interface, and if the library changes, only the adapter will need to be adjusted instead of the entire application.

**Error:**

I didn't understand this point very well because of the translation, but I thought that by using a facade it could be avoided that if the code changed later it would not affect the operation.





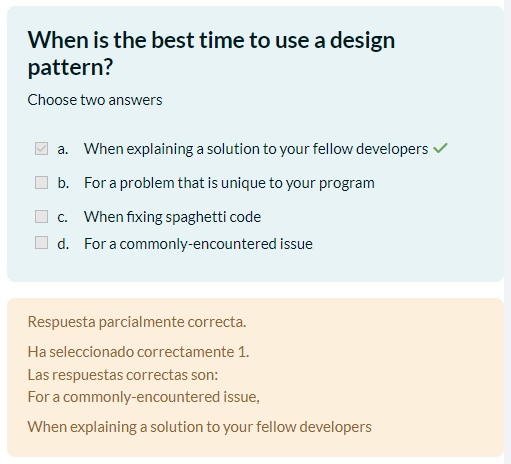
**Justification:**

Feature Envy code smell refers to a situation where a method appears to be more interested in the attributes or methods of another class than its own. In the code, the method m of the class Class1 overuses the methods (do\_something, foo, foo2) and attributes of the class Class2, suggesting that m should be located in Class` instead of Class1. Which indicates a lack of cohesion in the code design and may be an indicator of a better distribution of responsibilities between classes.

**Error:**

I thought it might be Inappropriate Intimacy, due to so many calls to Class2.

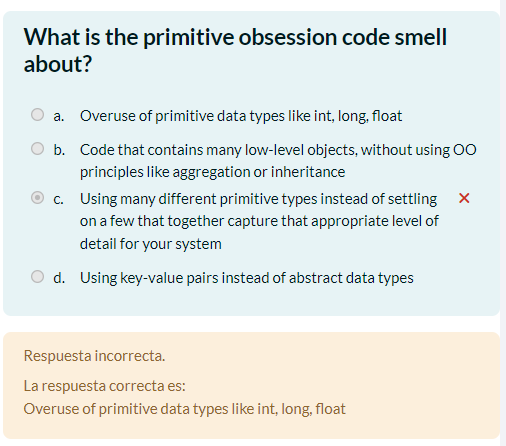




**Justification:**

Design patterns are tested and documented solutions to common problems in software design. Therefore, it is best to use a design pattern when we are faced with a problem that is commonly encountered in software development.





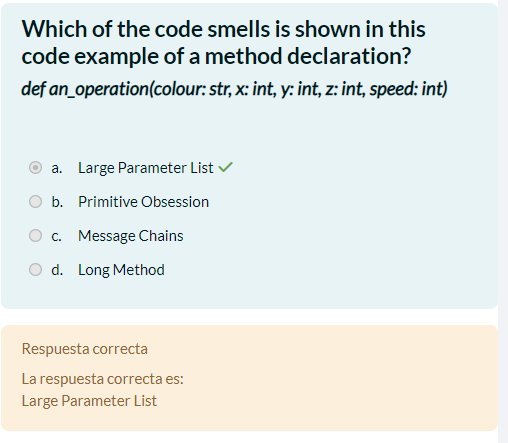
**Justification:**

Primitive obsession code smell refers to the overuse of primitive data types such as int, long, float, etc., instead of encapsulating related logic in classes or objects with specific semantics. This can lead to code that is difficult to understand, maintain, and extend, since related logic is scattered throughout the code and is not encapsulated in classes that represent domain concepts more clearly and expressively.

**Error:**

I was not clear about the concept of primitive obsession.

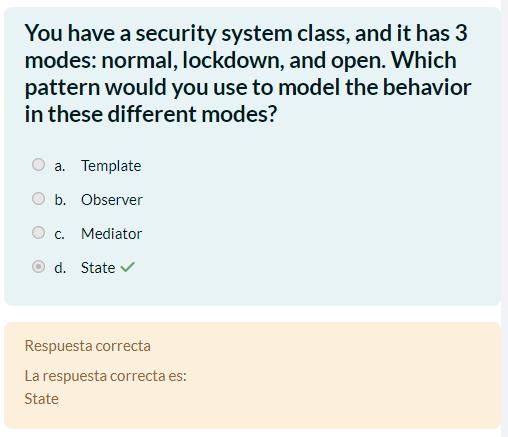




**Justification:**

The Large Parameter List code smell refers to methods that have a long list of parameters. In the example provided, the an\_operation method has five parameters (color, x, y, z, speed), which can make it difficult to understand and maintain. Having a long list of parameters can indicate a lack of cohesion in the method design and can make the code less readable and more error-prone.

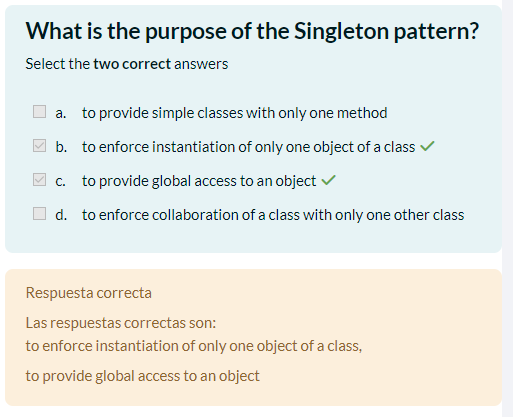




**Justification:**

The State pattern is suitable for modeling behavior in different states of an object. In this case, the security system has three different modes: normal, lock and open. Each of these modes represents a different state that the security object can be in. Using the State pattern, separate classes can be defined for each state (normal, blocking, open), where each class implements the specific behavior associated with that state.



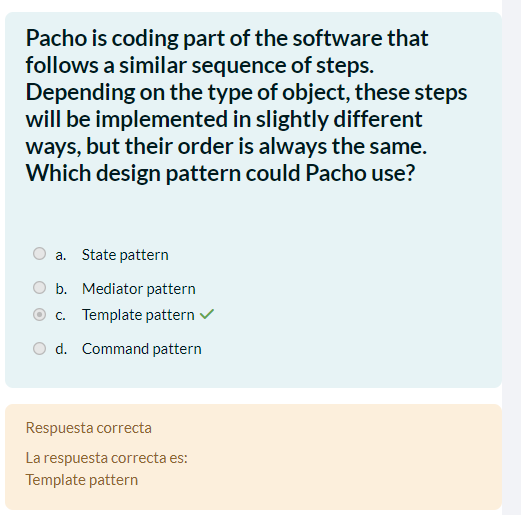


**Justification:**

to enforce instantiation of only one object of a class: The Singleton pattern is used to ensure that a class has only one instance and provides a global access point to that instance.

to provide global access to an object: In addition to ensuring that there is only one instance of the class, the Singleton also provides a mechanism to globally access that instance. This allows different application components to access the same instance of the Singleton at any time and from anywhere in the code.

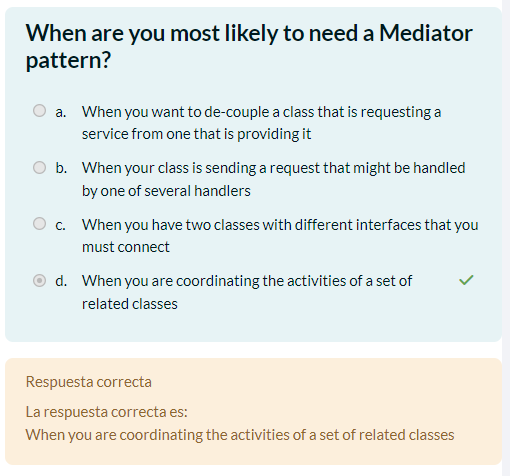




**Justification:**

The Template pattern is ideal for situations where a similar sequence of steps is followed, but implementation details may vary depending on the type of object. This allows Pacho to define a common structure for steps in a base class and allow subclasses to implement specific details as needed.

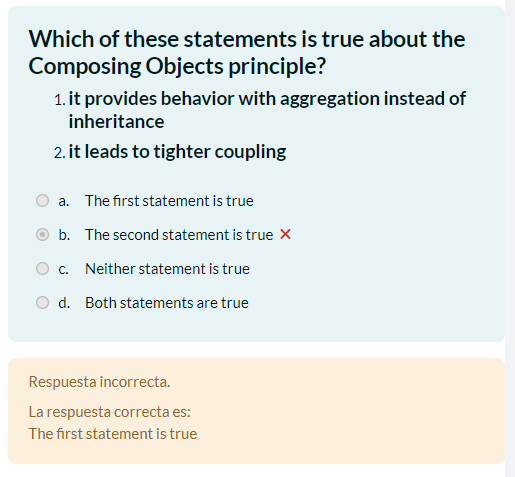




**Justification:**

The correct answer is When you are coordinating the activities of a set of related classes, because the Mediator pattern is used to manage and centralize communication between multiple related classes, avoiding direct dependencies and facilitating the coordination of their activities. This reduces complexity and improves the modularity of the system.





**Justification:**

The correct answer is because the principle of Composing Objects refers to the composition of objects to provide behavior through aggregation rather than inheritance. This promotes code reuse and flexibility by allowing objects to be combined in different ways without the restrictions of a rigid inheritance hierarchy. On the other hand, compounding generally leads to lower coupling, not higher coupling, which makes the second statement incorrect.

**Error:**

I didn't understand this point well, and I just put the answer for the sake of putting it.