Analysis of the application

1. Encapsulate what varies:
   Encapsulating what varies is an approach for dealing with details that change regularly. When code is constantly updated to accommodate new features or requirements, it tends to become tangled. We reduce the amount of code that will be affected by a change in requirements by separating the sections that are prone to change. In our application, we implemented simulation using simulate method with choices as parameters. These choices also have sub-choices for different scenarios in the one choice. So, if a requirement arises to change the code related to one sub-choice, it will not affect the working of other scenarios.
   For example, if we want to modify the implementation of an object sliding from an inclined surface, we need to just modify one sub-choice and it will not affect other cases of sliding. Encapsulation also helps if need arises to introduce one more sub-choice in simulation of sliding. In this way, implementing becomes a lot easier.
2. Favor Composition over Inheritance
   Instead of inheriting from a base or parent class, classes should achieve polymorphic behavior and code reuse through composition. Since Java does not allow multiple inheritance, composition is favored over inheritance as it makes it easier to have these classes as private members. Composition helps in changing behavior of code easily. Instead of inheriting from a base or parent class, it allows the development of complicated kinds by combining objects (components) of different types. It also increases the readability of code.
   We have used an instance of “Object” class as a parameter for constructor of sliding and rolling objects.
3. Program to an interface not implementation
   Programming to an interface begins with the creation of an interface, followed by the definition of its methods, and finally the creation of the actual class with the implementation. This can be used to change the behavior of a program at run-time. If the code is based on implementation, as soon as code changes the code stops working. This doesn’t lock the runtime object into the code. In our project, “rollBehavior” and “slideBehavior” interfaces could have been created. All objects with rolling behavior will implement rollBehavior interface.

These also helps in readability of code and the objects just need to employ relevant methods from interface.

4. Classes should be open for extension and closed modification
   The classes can extend its behavior without modifying the code for class. It helps in maintaining the correctness of code as it removes the possibility of writing the code again to extend the scope of class. The design of our classes should be able to add new functionality as new requirements are generated and a once tested class should not be modified. In our project, we can implement different cases of rolling by using more generally an object as input rather than specific rolling object. This way it helps in extending the definition to more general bodies if and when the functionality is implemented.

Strategy Design Pattern

A class's behavior or algorithm can be altered at runtime using the Strategy pattern. This design pattern is classified as a behavior pattern. This design pattern is really helpful when there are a lot of similar classes that only differ in the way they execute some behavior. We generate objects that represent numerous strategies and a context object whose behavior varies depending on its strategy object in the Strategy pattern. Instead of executing the work on its own, the context delegates it to a linked strategy object. The context isn't in charge of choosing the best algorithm for the job. Instead, the user informs the context of the desired strategy. The context object's running algorithm is changed by the strategy object. The context class must have a field for storing a reference to one of the strategies. Instead of executing the work on its own, the context delegates it to a linked strategy object. The context isn't in charge of choosing the best algorithm for the job. Instead, the user informs the context of the desired strategy. The Open/Closed principle is used in this pattern. We don't need to change the context [closed for changes], but we can pick and choose any implementation [open for additions].

1. Create a Strategy interface

This interface is implemented by the concrete classes. In our project, a slide interface can be created which will be implemented by concrete classes.

2. Creating concrete classes that implement the Strategy interface. These classes will override the methods in interface. Different cases of sliding of objects have different concrete classes that simulate the motion of objects. This way introduction of different scenarios is much easier.
3. Creating a context class.
   The context class adds a field for storing a reference to a strategy object. This class should only interact with the concrete strategy classes through Strategy interface.
4. Use the context for behavior changes when it changes the strategy
   The context will decide the strategy to implement the function. So, if the user inputs sliding motion on an inclined surface, the context will use relevant strategy to simulate.

This strategy can be also used to simulate different scenarios such as rolling, collision, spring, etc. This will be helpful when extending the scope of these simulations for future versions.