1) Obtaining the Sum of Squares of all the Values using existing sumTree method

DECORATOR: It seems to me that this is the best choice when we are trying to *modify/add behavior* of an Object without changing the Object's internal code. Similar to the book’s example of LowerCaseReader decorating IOStream, taking as input IOStream’s input, we could have a ValueSquared decorating Node whose only added functionality would be taking Node’s GetValue() and returning the square of this value. It sort of depends on the sumTree() method, but because the ValueSquared decorator would be a subclass of the Node, we could call sumTree(decoratedRootNode) and get the proper result.

PROXY: It seems to me that this is the best choice to access a Node's .getValue() method; we could create a Proxy with a reference to the Node, and the Proxy’s getValue() method would invoke the Node's getValue() method and returns the square of it. However, generally, Proxies should not be subclasses, but rather use the same interface as the object they are proxying. Although a proxy could probably work to solve this problem, it is not the best option because Decorators are specifically made to modify the behavior of an inherited Object without changing the Object itself.

FACTORY METHOD: I don't think this design pattern would really help us much because we do not want to create objects; it is useful when we can't anticipate the type of objects we want to create or want subclasses to specify the type of created object, but this is more-or-less irrelevant for the task at hand because we are only working with an object of one type.

ITERATOR:

We are told that we want to be using the sumTree(Node rootNode) method we are already given, so we can assume that it knows how to iterate through the data structure based on the rootNode and its List<Node> of children; the Iterator would not help with obtaining the sum of squares of getValue() without modifying sumTree(Node rootNode).

2) E-commerce Website Model

STATE: For this, I would use primarily a STATE based design. I believe for this implementation, we would want a reference to the context as a whole and give the Order the opportunity to change its state (from NEW ORDER to PAID ORDER to SHIPPED ORDER), which means the STATE is a better design pattern than the STRATEGY for this model. We want a USER to be able to perform different TASKS based on the ORDER, and the best way to ensure that this remains consistent is by allowing the Order to have multiple states, which make it very easy to check if we can do specific tasks.

STRATEGY: According to the book, Strategy objects only handle specific tasks for each strategy, whereas STATE will provide the implementation of everything within the context. We seem to be concerned with the interaction of a USER and an ORDER more than the specific tasks of an Order, so I think using order states is more appropriate than a Strategy-based implementation.

COMMAND: It doesn’t seem to me like this is the best design because it does not seem like we need to parameterize different requests or queue different requests, but rather ensure complete control given one Order request. We don’t have multiple objects with different functions like turning a TV on, changing the channel, turning up the volume, etc. which we want to combine into one complete control system, which would be the optimal use for a COMMAND based design.

PROXY: I don't see how we could use a PROXY design to improve this implementation without requiring oodles of if statements to determine the Order’s acceptable behavior based on the Proxy.

3) Sequential Access (Subsequent Integer Functions)

TEMPLATE METHOD:

STRATEGY:

COMPOSITE: Using the COMPOSITE would imply that we want to use components that all come together to form a single larger object (like a menu and its subitems), but this does not seem to be the case in this example at all. We are not concerned with this in our implementation of a Java Range object.

FAÇADE: A FAÇADE’S purpose would be to provide a unified system to control a bunch of smaller components of a subsystem. From the example in the book, a façade can be used to simplify user interaction across numerous components. We only want to have one Range component and we will allow the user to create this based on their desired functions. The façade strategy does not aid in this.