# Overview:

* The Strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable
  + Let’s the algorithm vary independently from client to client
  + Conceptually, all of these algorithms do the same things:
    - Just have different implementations
* We can select the behavior of an algorithm dynamically at runtime
  + Can help us to avoid dealing with complex algorithm (specific data structures).
  + We let the client application pass the algorithm to be used as a parameter.
* Allows us to create objects that represent various strategies and a context object whose behavior varies as per its strategy object.
  + The strategy object changes the executing algorithm of the context object.

# Examples:

* In a soccer match, strategies will differ:
  + If Team A is leading Team B by a score of 1-0.
  + Instead of attacking, Team A becomes defensive.
  + Team B goes for an all-out attack to score.
* We can think two dedicated storage devices:
  + When one device becomes “full”, we start storing the data in the second available device.
  + A runtime check is necessary before the storing of data, and based on the situation, we will proceed.
* The Collections.sort() method from the Java API uses the strategy pattern
  + Takes a Comparator parameter
  + Based on the different implementations of Comparator interfaces
    - The Objects are getting sorted in different ways.
* Suppose we have different algorithms for breaking a stream of text into lines.
* Hard wiring all suck algorithms into the classes that require them is not desirable for several reasons.
  + Clients that need line breaking get more complex if they include the line-breaking code.
    - Makes clients bigger and harder to maintain.
  + Different algorithms will be appropriate at different times
    - Do not want to support multiple line breaking algorithms if we do not use them all
  + Difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client.
* We can avoid the above problems by defining classes that encapsulate different line-breaking algorithms (Strategy pattern).

# Principles of the Strategy Pattern:

* Objects have responsibilities.
* Different, specific implementations of these responsibilities are manifested through the use of polymorphism.
* There is a need to manage several different implementations of what is, conceptually, the same algorithm.
* It is a good design practice to separate behaviors that occur in the problem domain from each other (decoupling)
  + Allows me to change the class responsible for one behavior without adversely affecting another.

# Strategy versus State Pattern:

* The strategy and state pattern are very similar.
* Think of the Strategy Pattern as subclasses decide how to implement steps in an algorithm.
* Think of the State Pattern as an alternative to putting lots of conditionals in your context:
  + Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
* One difference between the two is that with the state pattern the Context contains state as an instance variable:
  + There can be multiple tasks whose implementation can be dependent on the state.
  + In the strategy pattern, strategy is passed as argument to the method and the context object does not have any variable to store it.

# Advantages:

* By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
* Applications can switch strategies at run-time (polymorphism).
* Enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
* Simplifies unit testing because each algorithm is in its own class and can be tested through its interface alone.
  + The developer does not need to worry about interactions caused by coupling.
  + Developer is able to test each algorithm independently and not worry about all the combinations possible.

# When to use the strategy pattern?

* When many related classes differ only in their behavior
  + Strategies provide a way to configure a class with one of many behaviors.
* When you need different variants of an algorithm
  + We want our application to be flexible to choose any of the algorithm at runtime for specific task
* When an algorithm uses data that clients should not know about
  + Use the Strategy pattern to avoid exposing complex, algorithm, and specific data structures.
* When a class defines many behaviors, and these appear as multiple conditional statements in its operations.
  + Instead of many conditionals, move related conditional branches into their own Strategy class.

# Implementation:

# Participants:

* **Strategy**:
  + Declares an interface common to all supported algorithms.
  + **Context** uses this interface to call the algorithm defined by a **ConcreteStrategy**.
  + Specifies how the different algorithms are used.
* **ConcreteStrategy**:
  + Implements the algorithm using the **Strategy** interface.
* **Context**:
  + Is configured with a **ConcreteStrategy** object
    - Maintains a reference to a Strategy object or is passed as a parameter to a method.
  + May define an interface lets Strategy access its data
* **Context** uses a specific **ConcreteStrategy**
* **Strategy** and **Context** interact to implement the chosen algorithm:
  + Sometimes **Strategy** must query **Context**
  + **Context** forward requests from its client to **Strategy**.
    - Clients usually create and pass a ConcreteStrategy object to the context.
      * Clients interact with the **Context** exclusively
      * Often a family of **ConcreteStrategy** classes for a client to choose from

# Advantages of implementation:

* A family of algorithms can be defined as a class hierarchy:
  + Can be used interchangeably to alter application behavior without changing its architecture.
  + Inheritance can help factor out common functionality of the algorithms.
* An alternative to subclassing
  + Inheritance offers another way to support a variety of algorithms or behaviors
  + You can subclass a Context class directly to give it different behaviors.
    - Hard-wires the behavior into **Context**.
    - Mixes the algorithm implementation with Context’s.
      * Context is harder to understand, maintain and extend.
    - You cannot vary the algorithm dynamically.
    - You wind up with many related classes whose only difference is the algorithm or behavior they employ.
  + Encapsulating the algorithm in separate Strategy classes
    - Lest you vary the algorithm independently of its context.
      * Easier to switch, understand, and extend.
* Data structures used for implementing the algorithm are completely encapsulating in Strategy classes.
  + The implementation of an algorithm can be changed without affecting the Context class
  + Eliminates conditional statements.
* A choice of implementations
  + Strategies can provide different implementations of the same behavior.
  + Client can choose among strategies with different time and space trade-offs.
* Clients must be aware of different Strategies
  + Has a potential drawback in that a client must understand how Strategies differ before it can select appropriate one.
  + Clients might be exposed to implementation issues.

# Implementation issues:

* The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa
  + Have Context pass data in parameters and Strategy operations
    - Keeps Strategy and Context decoupled
  + Context can pass itself as an argument to Strategy operations
    - Let’s the Strategy call back on the context as required.
* Context class may be simplified if it is meaningful not to have a Strategy object
  + Context checks to see if it has a Strategy object before accessing it
    - If there is one, then Context uses it normally.
    - If there is not a strategy, then Context carries out default behavior.
  + The benefit of this approach is that clients do not have to deal with Strategy objects at all unless they do not like the default behavior.