```python
import numpy as np
import functools
import lib
import week6

class StochasticGradientDescent(lib.GradientDescent):
    def __init__(self):
        self._iteration = 0
        self._max_iter = 1000
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._f_star = 0
        self._debug = False
        self._beta = 0
        self._function_generator = None
        self._dimension = None
        self._algorithm = None
        self._function = None
        self._sum = None
        self._x_value = None
        self._old_x_value = None
        self._step_coeff = None
        self._converged_value = None
        self._grad_value = None
        self._m = None
        self._v = None
        self._adam_grad = None
        self._beta = None
        self._beta2 = None
        self._step_size = None
        self._z = None

    def adam_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complet step
        self._grad_value = self._gradient(self._x_value)
        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
        # grad_value * grad_value gives element-wise product of np array
        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
        self._old_x_value = self._x_value
        self._iteration += 1
        m_hat = self._m / (1-(self._beta ** self._iteration))
        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
        v_hat_aug = v_hat**(0.5) + self._epsilon
        self._adam_grad = m_hat / v_hat_aug
        self._x_value = self._x_value - self._step_size * self._adam_grad
        return True

    def polyak_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complet step
        self._iteration += 1
        numerator = self._function(self._x_value) - self._f_star
        self._grad_value = self._gradient(self._x_value)
        denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
        if denominator == 0.0:
            # do nothing this step (hope for non-zero on next mini-batch)
            return False
        self._old_x_value = self._x_value
        step = numerator/denominator
        self._x_value = self._x_value - step * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        return True  # completed step

    def constant_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complete step
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._x_value = self._x_value - self._step_size * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        return True  # completed step

    def rmsprop_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._x_value = self._x_value - self._alpha_n * self._grad_value
        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
        self._alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
        self._step_coeff = self._alpha_n
        return True


    def heavy_ball_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._z = self._beta * self._z + self._step_size * self._grad_value
        self._x_value = self._x_value - self._z
```

```
101:         return True
102:
103:     # pass a function which generates the function to be evaluated,
104:     # e.g. with different minibatches at each iteration
105:     def function_generator(self, fg):
106:         self._function_generator = fg
107:         return self
108:
109:     def alg(self, a):
110:         if a == "constant":
111:             self.step = self.constant_step
112:         elif a == "polyak":
113:             self.step = self.polyak_step
114:         elif a == "rmsprop":
115:             self.step = self.rmsprop_step
116:             if self._step_size is None:
117:                 raise Exception("Need step_size to initialize rmsprop")
118:             if self._x_value is None:
119:                 raise Exception("Need start/x_value to initialize rmsprop")
120:             self._sum = np.zeros(self._x_value.shape)
121:             self._alpha_n = np.zeros(self._x_value.shape)
122:             self._alpha_n.fill(self._step_size)
123:         elif a == "adam":
124:             self.step = self.adam_step
125:             if self._x_value is None:
126:                 raise Exception("Need start/x_value to initialize rmsprop")
127:             self._m = np.zeros(self._x_value.shape, dtype=np.float64)
128:             self._v = np.zeros(self._x_value.shape, dtype=np.float64)
129:         elif a == "heavy_ball":
130:             self.step = self.heavy_ball_step
131:             self._z = 0
132:         else:
133:             raise Exception(f"Alg {a} NYI")
134:         self.function_name = a
135:         return self
136:
137:     def f_star(self, f_st):
138:         self._f_star = f_st
139:         return self
140:
```