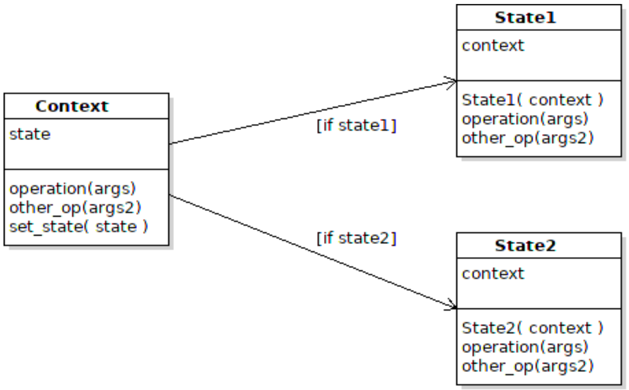
Lab 7: Stopwatch with States, GUI using Event Loop

There are two parts to this lab:
1. Apply the State Pattern to the Stopwatch you did in a previous lab.
2. Create a UI for the Stopwatch that updates itself every 100 milliseconds.

Part 1. State Pattern

The behavior of some objects depends strongly on what "state" the object is in. Parsers (programs that try to interpret files according to some grammar) are an example of this.

The object whose behavior depends on state is called the *context*. It can greatly simplify programming if you create one class for each *state* of the context, and have the context delegate that behavior to the state object.



For example, in `Context` the code for `operation` would look like:

```
def operation(self, *args):
    # delegate the operation to the current state
    # self.state is the current state (could be state1 or state2)
    return self.state.operation(*args)
```

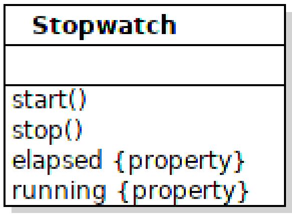
The State Pattern is very similar to the Strategy Pattern. The *states* are *strategies*. A distinctive feature of the State Pattern is the context delegates entire methods to states, and the states usually have a way to change the current state of the context (a set_state or `enter_state` method).

Reference

Wikipedia State Pattern https://en.wikipedia.org/wiki/State_pattern

In the Stopwatch:

what are the states of a stopwatch?
what behavior depends on the state?



The states are: `running` and `stopped`

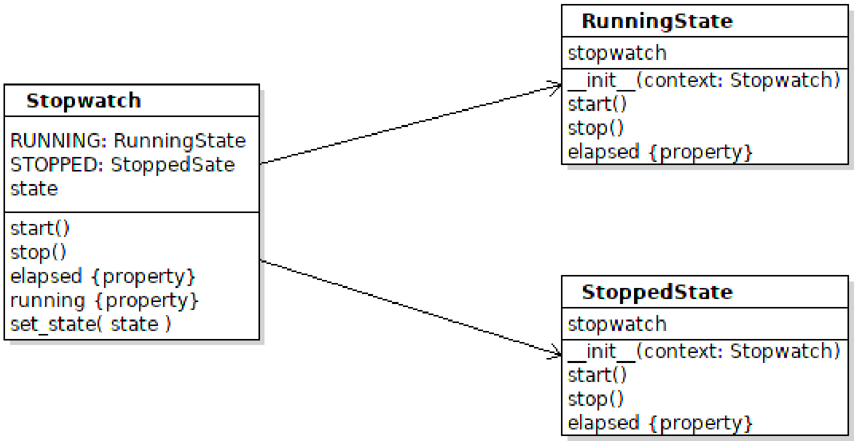
Most behavior depends on the current state. You can see this in the code; almost every method contains:

```
if self.running:
    ...
else:
    ...
```

That means the *behavior depends on state* (running or stopped).

You will apply the State Pattern to the Stopwatch. When you are finished there won't be any "if" statements in the code at all!

To do this, you need to write a RunningState and StoppedState class, and create one instance of those objects in the Stopwatch.



1.1 Define RunningState and StoppedState classes. Each of these classes contains the methods of the Stopwatch that depend on state. The code should handle only the case where the stopwatch is running (RunningState) or stopped (StoppedState).

```
class RunningState:
    """Behavior of the stopwatch when it is in the running state.

    The only attribute is a reference to the Stopwatch (context),
    which the state uses to get attributes of the Stopwatch.
    Do not define any other attributes here.
    """
    def __init__(self, stopwatch: Stopwatch):
        # save a reference to the context
        self.stopwatch = stopwatch

    def start(self):
        # in the running state, what to do when start is invoked?
        ???

    def stop(self):
        # in the running state, what to do when stop is invoked?
        #TODO write the code
        # finally, change Stopwatch state to "stopped"
        self.stopwatch.set_state(self.stopwatch.STOPPED)

    @property
    def elapsed(self):
        # what is the elapsed time when stopwatch is in the running state?
        return ???
```

Write the code for StoppedState. It has the same methods as RunningState.

1.2 Add States to Stopwatch.

In the Stopwatch you need to make 3 changes:

1. create an attribute for each state (RunningState, StoppedState)
2. add a **state** attribute that always refers to the current state
3. modify all the state-dependent methods to delegate their behavior to the current state

```
class Stopwatch:
    def __init__(self):
        # a constant for each state of the Stopwatch
        self.RUNNING = RunningState(self)
        self.STOPPED = StoppedState(self)
        # set the initial state of the stopwatch
        self.set_state(self.STOPPED)
        # your original code (may be different from this)...
        self._elapsed = 0.0

    def set_state(self, state):
        # change the state of the stopwatch
        self.state = state
```

1.3 Modify Stopwatch methods to *delegate* to the current state.

```
class Stopwatch:

    def start(self):
        self.state.start()

    #TODO delegate stop and elapsed to the state.

    @property
    def running(self):
        # The stopwatch can perform this itself.
        # It is just 1 line of code using self.state
        return ???   # no self._running (boolean)
```

1.4 Test it!

When you are done:

- there should not be any "if" statements in any of the classes
- you don't need a self._running attribute (boolean).

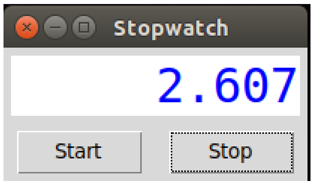
Part 2. Graphical UI for Stopwatch

Write a graphical UI for the Stopwatch that updates every 50-100 milliseconds. The two things you will learn from this part are:

how to schedule a future event (method call) using **`after`**

use *dependency injection* to *inject* a Stopwatch reference into the StopwatchUI. You did this in the UnitConverterUI, too.

Your Graphical UI should look something like this:



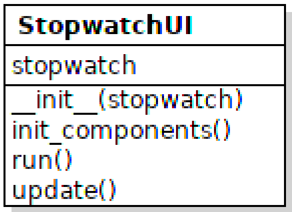
Create a class named StopwatchUI for the user interface.

The StopwatchUI should have these methods:

`init_components()` - create and layout the components, as usual.

`run()` - call update and then start the mainloop

`update()` - update the time displayed on the UI and then schedule a new call to update.



`tkinter.Tk` has an after method that schedules a function (or method) call at a later time:

```
after(milliseconds, func, *args)
```

after a delay of milliseconds, call the function func with the given args.

In the StopwatchUI:

```
def update(self):
    get the elapsed time from stopwatch and show it on the display (a
    Label). Use a Control Variable

    call self.after() to schedule another call to update in 50
    milliseconds.
```

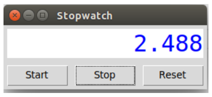
Part 3: Add "reset" Behavior to the Stopwatch & UI

Add a "reset" behavior to the Stopwatch and modify the behavior of "start".

`reset()` - if the Stopwatch is running, reset does nothing. If the stopwatch is stopped, then reset the elapsed time to 0.
`start()` - if the stopwatch is running, then start does nothing. If the stopwatch is stopped, then (re)start the stopwatch, but do not reset the elapsed time. That is, add to any existing elapsed time.

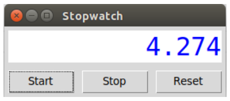
In the GUI, add a "`Reset`" button.

Stopped:



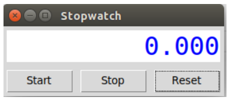
2.488

Press Start (no reset):



4.274

Stop & Reset:

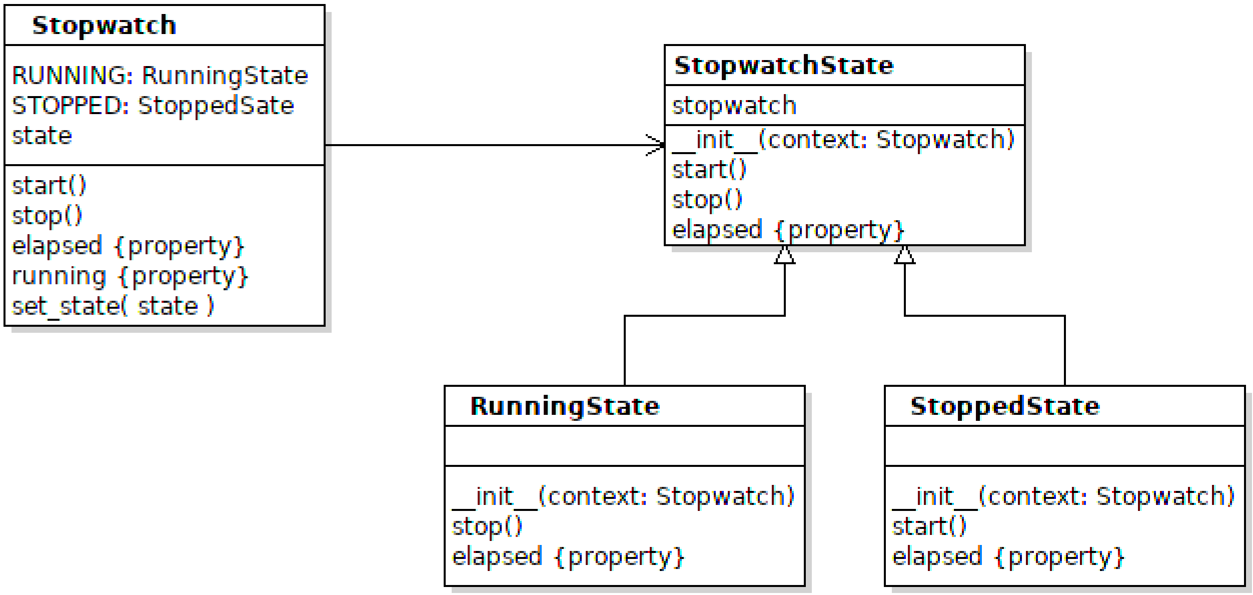


0.000

States with Superclass (Optional)

The State Pattern is typically implemented with states that extend some superclass (StopwatchState). The methods in the state superclass do nothing or provide some default behavior. This can simplify writing the concrete states, since you only need to implement the methods that do something specific. Other methods just inherit from the superclass. For example:

```
class StopwatchState:
    """Superclass for stopwatch states."""

    def __init__(self, stopwatch: Stopwatch):
        # save a reference to the context
        self.stopwatch = stopwatch

    def start(self):
        pass

    def stop(self):
        pass
    @property
    def elapsed(self):
        return 0

class RunningState(StopwatchState):
    """Behavior of the stopwatch when it is in the running state"""

    def __init__(self, stopwatch: Stopwatch):
        super().__init__(stopwatch)

   # inherit start() from superclass - does nothing

   def stop(self):
       # override stop() for this state.
```


Stopwatch
RUNNING: RunningState
STOPPED: StoppedSate
state
start()
stop()
elapsed {property}
running {property}
set_state(state)
StopwatchState
stopwatch
__init__(context: Stopwatch)
start()
stop()
elapsed {property}
RunningState
__init__(context: Stopwatch)
stop()
elapsed {property}
StoppedState
__init__(context: Stopwatch)
start()
elapsed {property}