The beginning steps involved the understanding of the minimum interface elements. We have to identify the various states of the device and understand how do these states interact among each other.

Coming up with a basic state machine diagram was the first step towards the implementation of the final design (?).

The three view states that we identified were **Stopped** state - with value 0 being displayed and with an increment button, **Running** state - with value decrementing on every clock tick; the clock starts after 3 sec wait or if a value of 99 is reached when in the Stopped, and **Beeping** state - when value reaches 0, the device starts beeping and a button press would stop beeps and moves again to the stopped state.

The next step involved recognizing the abstractions for the TimerModel states and representing them on scala code using traits

trait TimerWatchState extends UIHandling with OnTickListener with OnTimeoutListener {

def onEntry(): Unit

def onExit(): Unit

def updateView(): Unit

def getStateName(): Int

def getStateButtonAction(): Int

def onTimeout(): Unit

def onTick(): Unit

}

trait UIHandling {

def onButtonPress(): Unit

def onTimerChanged(value: Int): Unit

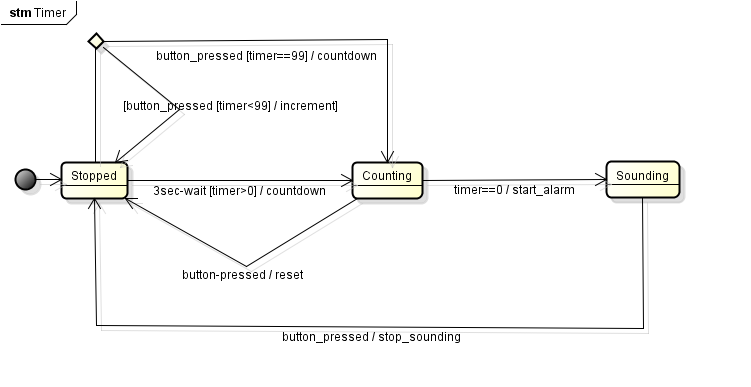
}

The first step to bridging the gap between the view state and the abstractions was the setup actions: start with Stopped state and with no time, value 0, on the view. View has a button that is capable of firing events, like onButtonPress in the UIHandling trait, that make state machine reacts according to its current state. For example, in Stopped state, on each button press, we increment the current amount of time and we start a listener that will start the countdown after 3 seconds. In Running state we decrement the current value by one after each tick of the clock, that notifies UI using a specific listener for it. When the clock reach 0, we move to beeping state to signalize that the timer has runned out. In order to stop beeping, its possible to click on the button. In this way, we go to Stopped state again with zero time.

The most difficult test to write was the UI ones. We haven't any experience with UI testing and any frameworks. Robolectric makes the job easy and this lack of experience make the things interesting to learn. We had to imagine all the case scenario and how the user would interact with the application. Then, we searched for mockito methods that would represent this intent in code.

For mock tests an interesting part caught our attention. Mock tests make really easy to see when a methods has more than one responsibility. The verify assertion shows us when this kind of behaviour occurs.

For models, unit testing helps in guaranteeing that the desired behaviour is acquired with the proposed implementation. This was checked with assertions on values after an chain of actions takes place.



The state diagram is represented in code by state object. The state object receives calls from a Facade, which handles interactions from UI, in the android environment represented by MainActivity, and notifies UI when changes occur in the state itself. The state object also manages the changes between each state - like moving from stopped to running state - and manages the clocks and timeouts that will make the machine operates itself.

We closely followed the stopwatch and clickcounter design implementation and naming conventions, this created clear separations between model, implementations and view.

TimerUiUpdate listener is the UI listener to detect any event occurring at UI.

We initially had 3 states in the state model Stopped, Running and Beeping. Our model was implemented using these three states. However, when it came to implementing the model with editable text field, we decided to use an extra state called Stopped\_Counting that performs the 3 second wait when the text field is filled in. This implementation helped us in performing countdown after 3 second delay in input. Our inital implementation followed very closely to the model.

It is more effective to model first than to code, so we can have an idea about the overall structure that our implementation should follow. Even though the code may not follow the model closely in all cases, modelling first leads to faster implementation and comprehensible design.

Use of VCS helped us to create a separate branch for slight implementation changes that we wanted to incorporate like the addition of editable text field.