***Note: Answers may be wrong given the vagueness of some parts of the question paper, so just make suggestions or add in more confident solutions***

1ai) BillingSystem is coupled with the PhoneCall class in which adding PhoneCall-irrelevant methods to the BillingSystem would expose these methods as well to the PhoneCall class.

It is the direct use of the BillingSystem class in charge(), rather than passing in a service which abstracts away from a specific method of charging for calls.

aii)

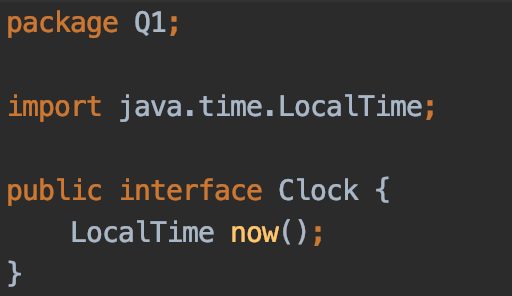
PriceInPence i.e if we want to change currencies? LocalTime reliance?

It is the direct use of the system clock in calculating startTime and endTime, rather than passing in an implementation of a clock for use in calculating these values.

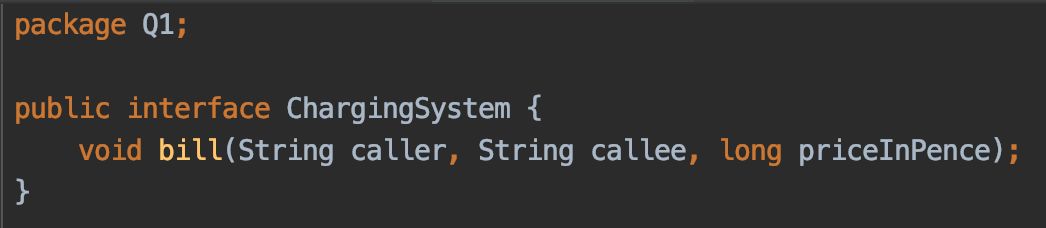
aiii) Coupling increases the fragility of our code, creating dependencies across classes that rely too heavily on each other where a single change of implementation may require modifying coupled classes to adapt to the new solution. For example, if the BillingSystem has to be changed to a SuperNewBillingSystem, this would require modifying the PhoneCall class to use the SuperNewBillingSystem.

b)

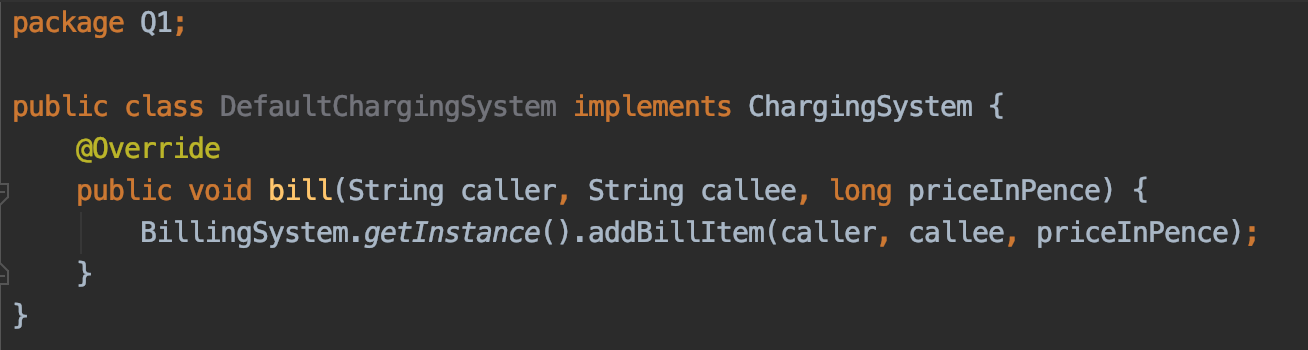
New interface Clock:



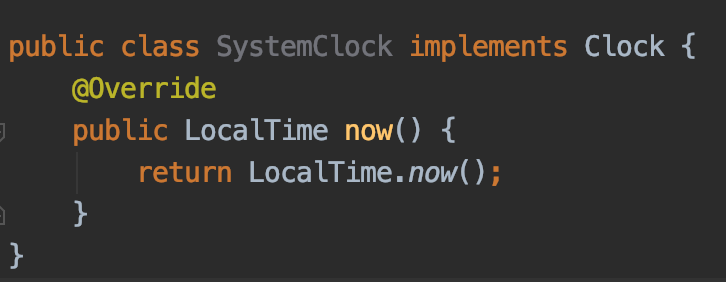
New interface ChargingSystem:



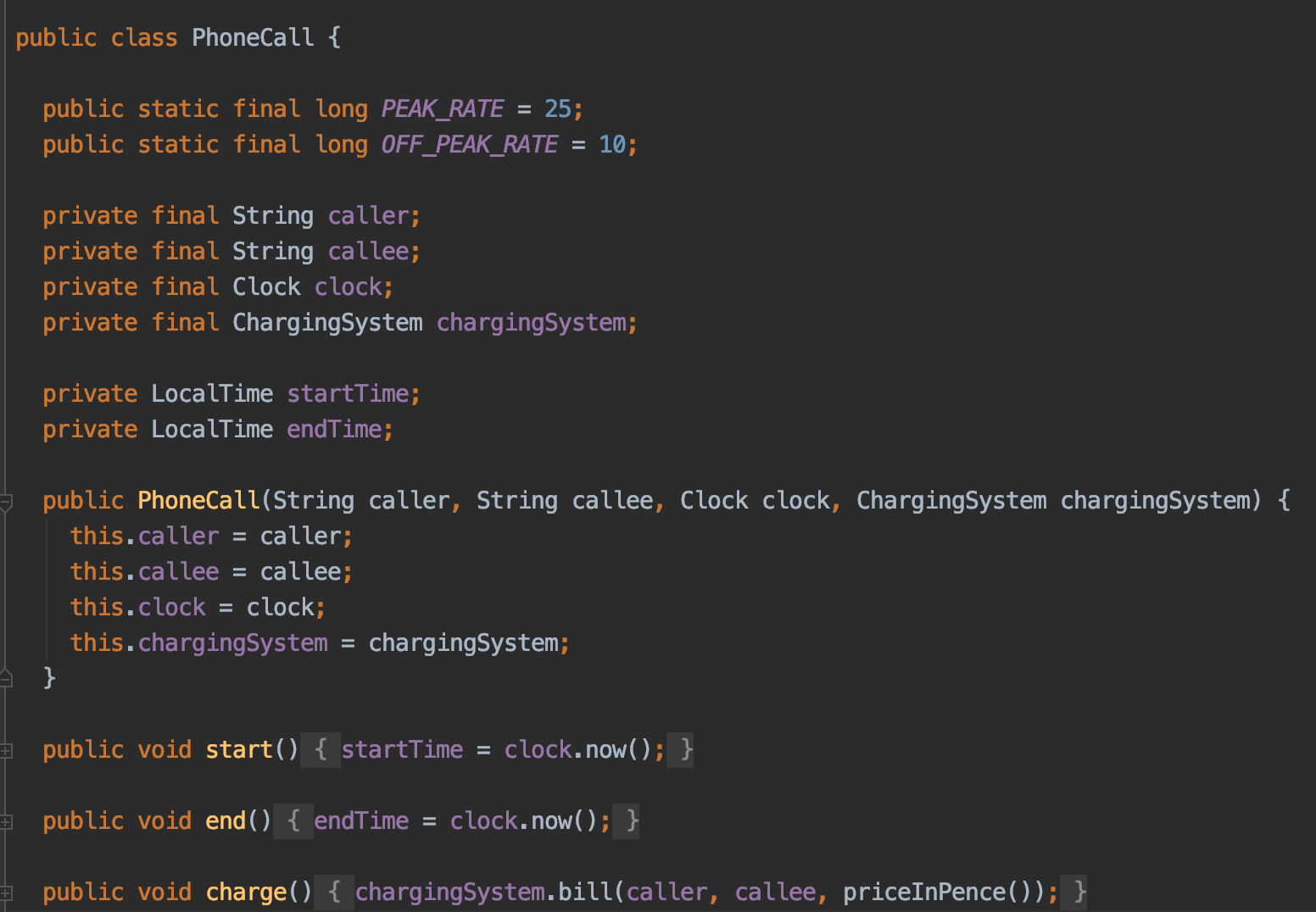
Default implementation DefaultChargingSystem:



Default implementation SystemClock:



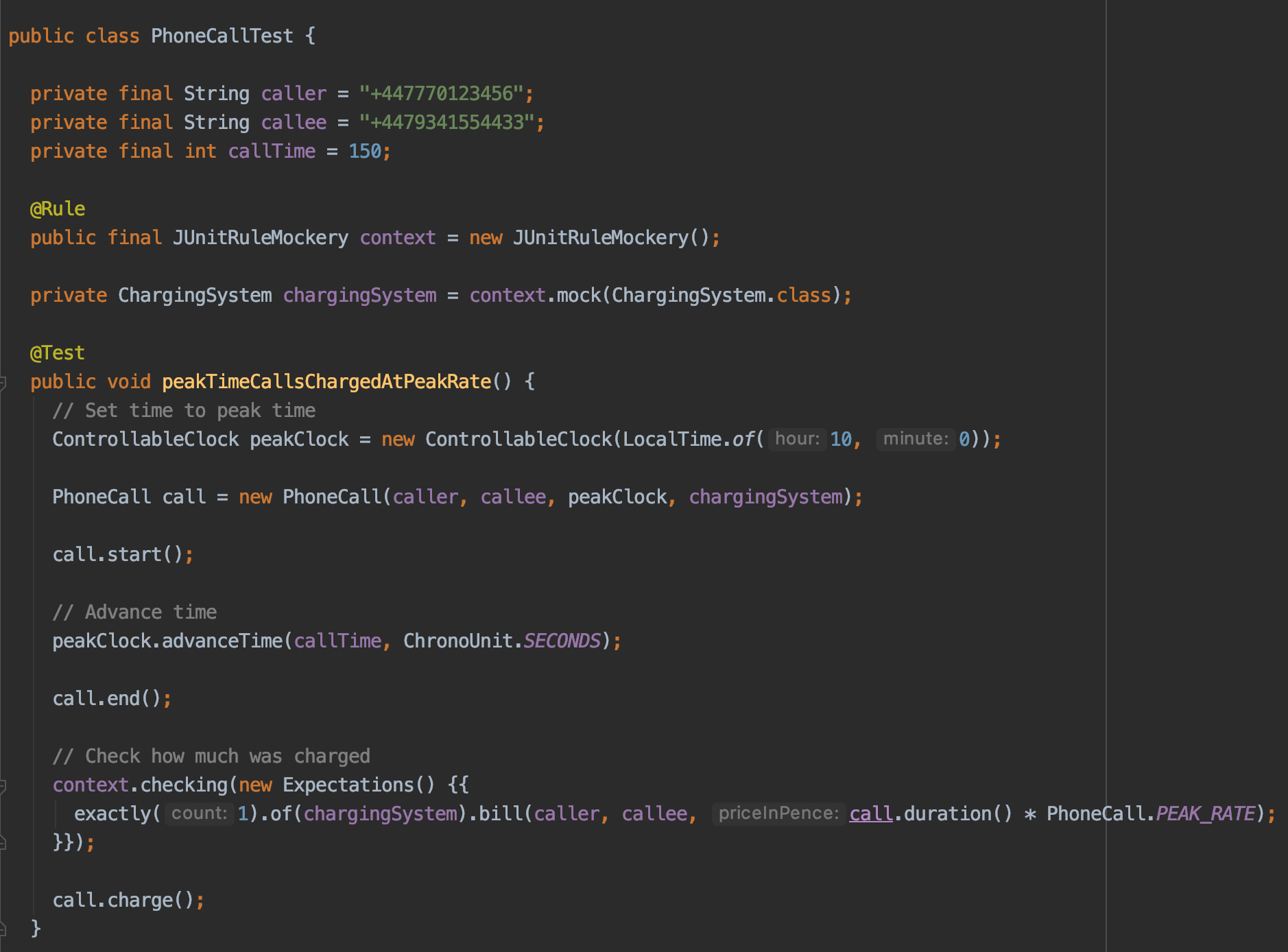
Refactored PhoneCall to take in a Clock (interface) and ChargingSystem (interface):



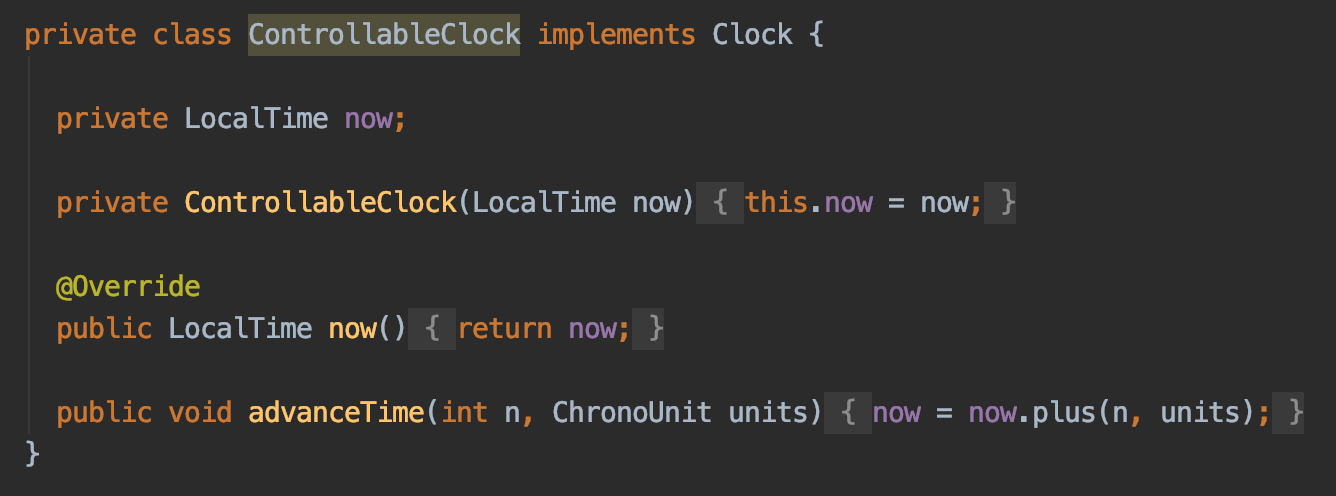
\*\*duration() and the peak/off-peak rates were made public for use in testing, not sure if there is a better solution

Tests:

\*Only the peak-time test is shown, the off-peak one follows a similar structure



\*inner class ControllableClock used for testing:

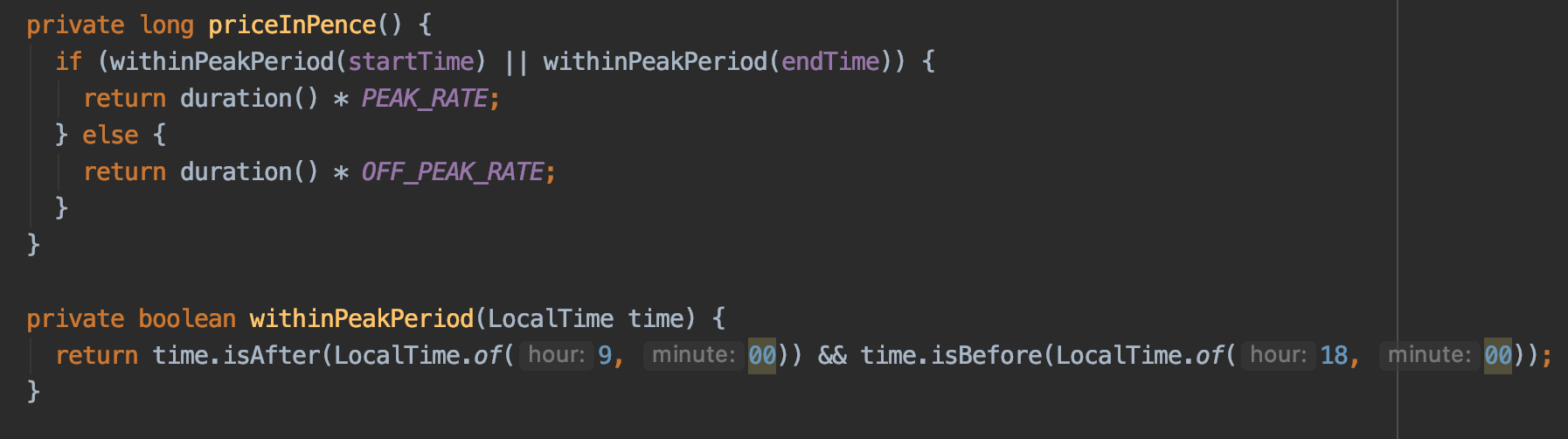


c)

...

d)

Update to PhoneCall:



Tests:

\*Only the test for a call starting right before the start of peak-period is shown

\*The other test follows a similar structure but with the clock time starting at 17:59 (right before peak-period ends)



2ai) There is an abstraction of logic, GUI components and event listeners by separating these into different classes / methods. This reduces the rigidity of the code as all GUI components can be found in one place: the View class, whilst the logic and state storing is managed by the Model.

Also aids mobility, can use the model class somewhere else with little to no change.

aii) The Model can easily be adapted to embrace an observer pattern in which multiple Views can utilise the same Model to display the same data but in different ways. This decoupled approach allows for the views to be swapped out any time for another without having to change the Model.

((Could we mention something about how separatin stuff into MVC means we can unit test the model thoroughly regardless of the implementation of the view, to make sure our logic is correct?))

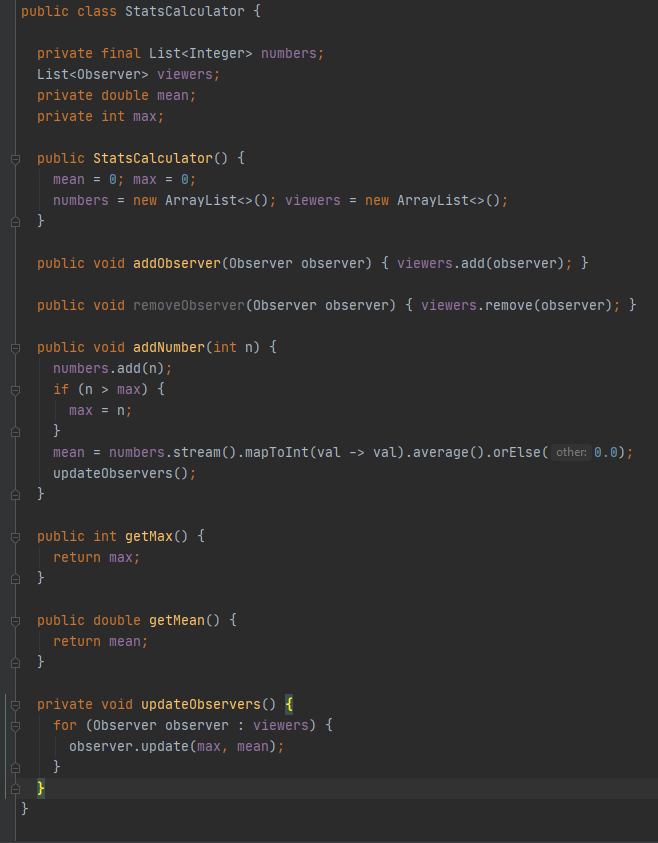
aiii) PAC (Presentation-Abstraction-Control)

b) + c) + d) + e) (quite similar so just smashed all of them at once)

Tests:



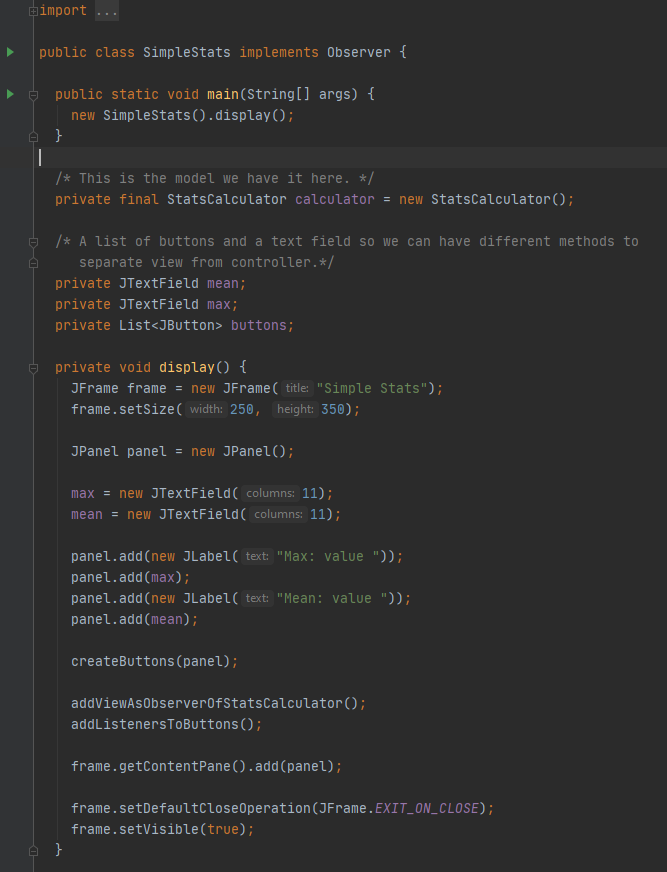
Model:

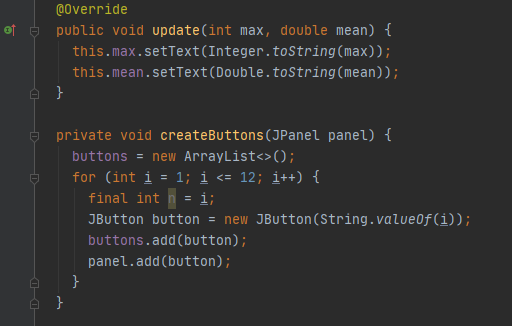


Controller + View (they are in the same class):

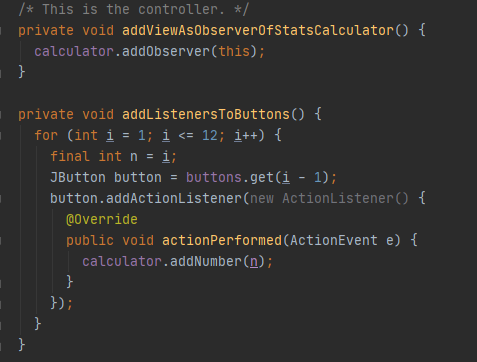
PS: added an observer interface that has a update(int max, double mean);

View part:





And the damn controller:

Junit expect