Peer Review

Do the design and implementation follow design principles?
There seems to be an attempt to use the Event Bus-pattern in TimerManager, which is good for reducing coupling. The placement of the bus, however, breaks the MVC pattern. As the model should not care for what happens outside of it (i.e. in the controller and view) the bus should be placed outside of the model package.

The use of the observer pattern in viewController is more appropriately placed, to loosen coupling and also to increase modularity, since an arbitrary number of instances can subscribe as listeners, which is a step in the right direction for following OCP.

Since there are some circular dependencies, e.g between Course and StudySession, the Single responsibility principle is not followed.
Due to the lack of JavaDoc and comment, maintainability is somewhat hindered.

Is the code documented?
The Requirement and Analysis document was nicely structured and gave a good understanding of the project. The user stories were well written and the design model quite extensive. Looking at the User Interface section, they gave a good description of how to navigate the app and the UI itself was intuitive and well made.

Code comments, however, were lacking. Only a few classes were commented which made it harder to get insight into the codebase. Even if the authors themselves might find the comments redundant or that there wasn't enough time, the few classes that were commented made it a lot easier to comprehend the responsibilities and features of the class which in turn makes the code easier to maintain. Perhaps some Javadoc?

Are proper names used?
The name for the class Moment is hard to understand. Because it doesn't give a clear picture of what it really is. There are also instances where the internal variable names inside methods are not properly used. For example toto1 in courseMainPage-class addToDo-method.
Other than that it was proper names.

Is the design modular? Are there any unnecessary dependencies?
The design of the application tries to apply som modular techniques but unfortunately, they are very tightly coupled with each other. Since the views have controller-like functionality you don't really get the positive modular functionality the mvc-pattern usually gives. If you want to change the UI you have to redo all the controller-like methods since they are put in the same classes.

The course-class has an image-variable that might fit better in the presentation layer.

There is a circular dependency in Course and StudySession that is referencing each other. Try and solve which of the two is the aggregate root.

Some views are using setters that exist in entities. To avoid views directly changing entities, use a method in a controller to change the model and maybe implement interfaces that hide that functionality from outside the model.

Some entities instantiate new objects of other classes, this is very tight coupling and should be handled somewhere else and put in as an argument when constructing the aggregate. Otherwise, you have to change the entities that instantiate when you change the other class.

Does the code use proper abstractions?
The package “event” has a lot of classes of different types of “Timers” but is it really necessary to have 7 different types of timers which also have some weird abstraction? And it’s often not a good idea to have abstract classes in the model. The views had good abstractions, for example, CourseManager, and PanelItemManager.

Is the code well tested?
We think the tests for StudySession meet the criteria for good junit tests. But there should be more tests for the other classes in the model. Maybe use @Before and @After in the tests if you want to?

Are there any security problems, are there any performance issues?
While using final as a parameter is a good way to prevent mutability by preventing the references from being changed, perhaps the authors should also look into other ways of achieving immutability e.g by declaring some variables as final.

Ultimately, the authors themselves know best what level of immutability they need and require for their application and how they wish to achieve this.

Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?
There are some actions going on in the model that may be more suitable in the view. For instance the Image in Course.

There are many methods and classes which do not seem to be used or implemented yet. Without almost any documentation and no code, it is hard to understand the wanted functionality of the classes and methods. The overall understanding of the code is also hard to grasp with all this extra code. Maybe this code could be placed in a developer branch and not master.

Can the design or code be improved? Are there better solutions?
Instead of abstracting the Timer in the model, why not let the views/controllers handle it differently for a different result.
Create controller-classes so that the views can easily be updated without having to recreate controller-like methods.
Create a factory to instantiate new objects instead of having new Object() in the entities
CourseManager is a controller and shouldn’t be an abstract class?