Design Rationale

Behavioural Pattern : Strategy
Classes involved: GUIContext, GuiAction, createContractAction, createRequestAction, ViewBidOfferAction, viewContractAction, ViewMessagesToTutorAction and ViewRequestAction.
The classes: createContractAction, createRequestAction, ViewBidOfferAction, viewVOntractAction, ViewMessagesToTutorAction, ViewRequestAction each implement a specific algorithm but all facilitate the bidding and contract creation features and these two features are also linked. So, the Strategy pattern is used here. How this works is that the GUIContext makes a reference to the GuiAction class which consists of methods used by the mentioned classes, so that way the GuiAction class is common to all other classes which have concrete implementations[1]. The disadvantage here is that there are many classes that utilize the GuiAction class and if the shared concrete implementation inside it is changed then we need to make appropriate changes in all the other classes that use it.

Structural Design Pattern : Facade
Classes involved : OnlineMatchingClient, UserFacade, Student and Tutor
Here OnlineMatchingClient is the client which uses the UserFacade class to access the UI initialization methods in Student and Tutor. This is Facade because Student and Tutor classes do more than just showing UI but the client is only concerned with getting the appropriate UI when they login[2]. The facade design is used because it separates the complex functionality of Student and Tutor classes via the UserFacade class from the OnlineMatchingClient. So OnlineMatchingClient only sees the UserFacade class which in turn uses Student and Tutor classes and any changes in Student and Tutor classes will not affect the client[2]. So refactoring and maintaining the code is easier. The disadvantage here is that student and tutor classes share similar methods and to some extent have repeated code.

Design Principle : Dependency Inversion Principle
Classes Involved : BidAction(abstract), CloseBidAction, OpenBidAction and ViewRequestAction
Here ViewRequestAction is the high-level class which uses the abstract BidAction class. The CloseBidAction and OpenBidAction classes act as low-level classes which depend on the abstract BidAction class. This design principle is used because ViewRequestAction shows the details about bids made between students and tutors and these can be open or closed bids. So instead of making ViewRequestAction class depend on CloseBidAction and OpenBidAction classes, the dependency is inverted by introducing the abstract BidAction class[3]. Now we can extend our CloseBidAction and OpenBidAction classes without breaking ViewRequestAction[3]. The disadvantage in our design is that BidAction does not depend too much on either CloseBidAction orOpenBidAction classes, so this makes the design complicated.

Design Principle : Stable Dependencies Principles
In our system, no class depends on all other classes and neither does all classes depend on a single class. Most stable classes are abstract like BidAction class or an interface like GuiAction[4]. This design principle is used because having one stable and all other unstable or one class depending on all the other classes, would mean too much coupling and so maintaining the code base is too difficult[4].

Design Principle : Do Not Repeat Yourself
This design principle holds true in our system because we make use of abstract classes like BidAction and interfaces like GuiAction both of which implement methods that are used by their subclasses. This design is used because we desire less repeated code which means that if a bug needs to be fixed or any change is required, we only make them in the abstract or interface classes only and so we do not need to look for the same code and fix it in multiple places.

References:
[1] Week 6 lecture slides 34 to 44
[2] Week 6 lecture slides 61 to 72
[3] Week 3 lecture slides 38 to 42
[4] Week 4 lecture slides 25 to 30
[5] Learnt in FIT2099