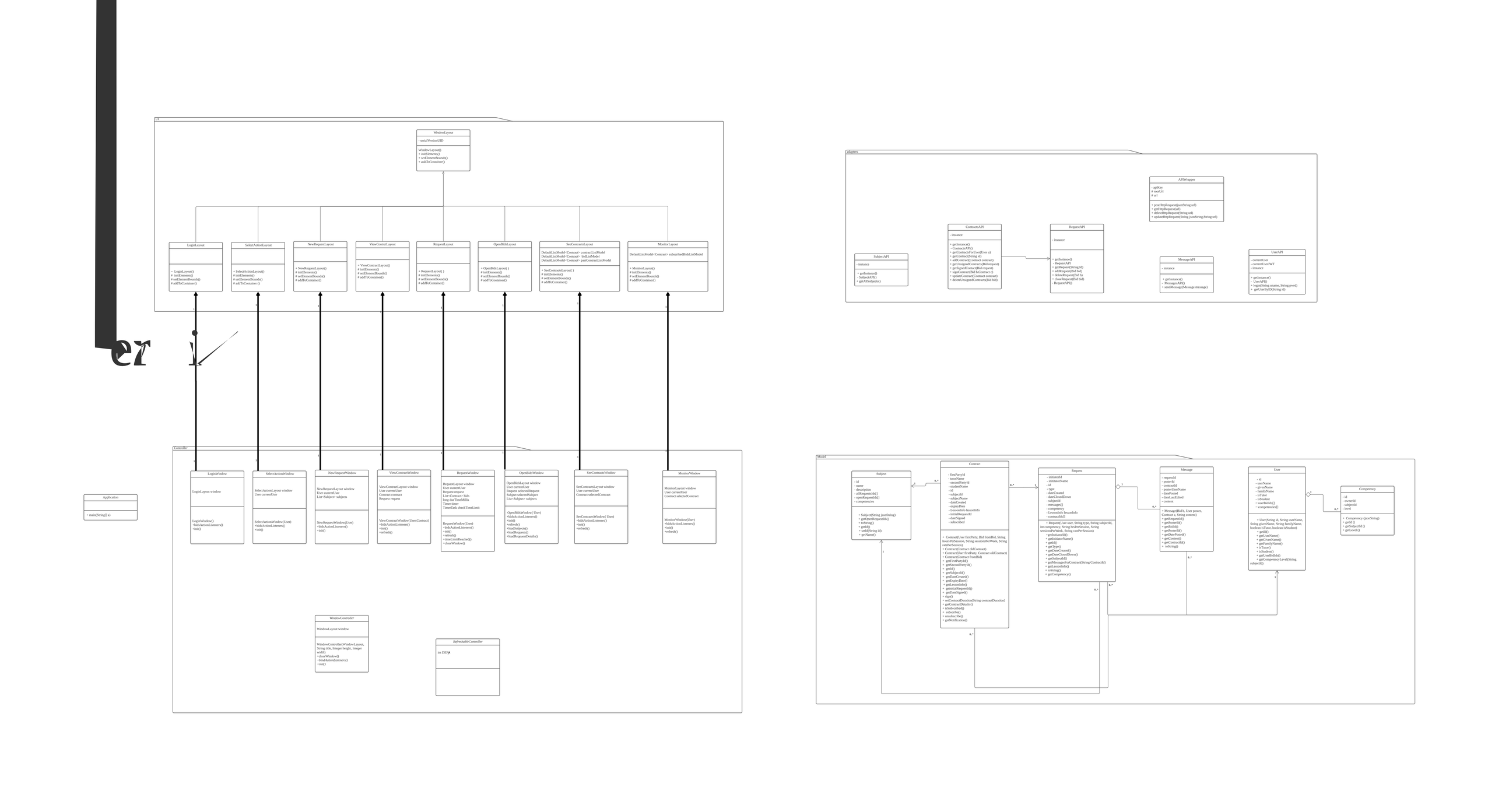
UI
WindowLayout
- serialVersionUID
WindowLayout()
+ initElements()
+ setElementBounds()
+ addToContainer()
LoginLayout
+ LoginLayout()
# initElements()
# setElementBounds()
# addToContainer()
SelectActionLayout
+ SelectActionLayout()
# initElements()
# setElementBounds()
# addToContainer ()
NewRequestLayout
+ NewRequestLayout()
# initElements()
# setElementBounds()
# addToContainer()
ViewContractLayout
+ ViewContractLayout()
# initElements()
# setElementBounds()
# addToContainer()
RequestLayout
+ RequestLayout( )
# initElements()
# setElementBounds()
# addToContainer()
OpenBidsLayout
+ OpenBidsLayout( )
# initElements()
# setElementBounds()
# addToContainer()
SeeContractsLayout
DefaultListModel<Contract> contractListModel
DefaultListModel<Contract> bidListModel
DefaultListModel<Contract> pastContractListModel
+ SeeContractsLayout( )
# initElements()
# setElementBounds()
# addToContainer()
MonitorLayout
DefaultListModel <Contract> subscribedBidsListModel
+ MonitorLayout()
# initElements()
# setElementBounds()
# addToContainer()
Controller
LoginWindow
LoginLayout window
LoginWindow()
+bidsActionListeners()
+init()
SelectActionWindow
SelectActionLayout window
User currentUser
SelectActionWindow(User)
+bidsActionListeners()
+init()
NewRequestWindow
NewRequestLayout window
User currentUser
List<Subject> subjects
NewRequestWindow(User)
+bidsActionListeners()
+init()
ViewContractWindow
ViewContractLayout window
User currentUser
Contract contract
Request request
ViewContractWindow(UserContract)
+bidsActionListeners()
+init()
+refresh()
RequestWindow
RequestLayout window
User currentUser
Request request
List<Contract> bids
long durTimeMillis
Timer timer
TimerTask checkTimeLimit
RequestWindow(User)
+bidsActionListeners()
+init()
+refresh()
+timeLimitReached()
+closeWindow()
OpenBidsWindow
OpenBidsLayout window
User currentUser
Request selectedRequest
Subject selectedSubject
List<Subject> subjects
OpenBidsWindow(User)
+bidsActionListeners()
+init()
+refresh()
+loadSubjects()
+loadRequests()
+loadRequestDetails()
SeeContractsWindow
SeeContractsLayout window
User currentUser
Contract selectedContract
SeeContractsWindow(User)
+bidsActionListeners()
+init()
+refresh()
MonitorWindow
MonitorLayout window
User currentUser
Contract selectedContract
MonitorWindow(User)
+bidsActionListeners()
+init()
+refresh()
Application
+ main(String[ ] a)
WindowController
WindowLayout window
WindowController(WindowLayout, String title, Integer height, Integer width)
+closeWindow()
+bindActionListeners()
+init()
RefreshableController
int DEFA
adapters
SubjectAPI
- instance
+ getInstance()
+ SubjectAPI()
+ getAllSubjects()
ContractsAPI
- instance
+ getInstance()
- ContractsAPI()
+ getContractsForUser(User u)
+ getContract(String id)
+ addContract(Contract contract)
+ getUnsignedContracts(Bid request)
+ getSignedContract(Bid request)
+ signContract(Bid b,Contract c)
+ updateContract(Contract contract)
+ deleteUnsignedContracts(Bid bid)
RequestAPI
- instance
+ getInstance()
- RequestAPI
+ getRequest(String bId)
+ addRequest(Bid bid)
+ deleteRequest(Bid b)
+ closeRequest(Bid bid)
- RequestAPI()
APIWrapper
- apiKey
rootUrl
url
+ postHttpRequest(jsonString,url)
+ getHttpRequest(url)
+ deleteHttpRequest(String url)
+ updateHttpRequest(String jsonString,String url)
MessageAPI
- instance
+ getInstance()
- MessageAPI()
+ sendMessage(Message message)
UserAPI
- currentUser
- currentUserJWT
- instance
+ getInstance()
- UserAPI()
+ login(String uname, String pword)
+ getUserByID(String id)
Model
Subject
- id
- name
- description
- allRequestsIds[]
- openRequestsIds[]
- competencies
+ Subject(String jsonString)
+ getOpenRequestsIds()
+ toString()
+ getId()
+ setId(String id)
+ getName()
Contract
- firstPartyId
- tutorName
- secondPartyId
- studentsName
- id
- subjectId
- subjectName
- dateCreated
- expiryDate
- LessonInfo lessonInfo
- initialRequestId
- dateSigned
- subscribed
+ Contract(User firstParty, Bid fromBid, String hoursPerSession, String sessionsPerWeek, String ratePerSession)
+ Contract(Contract oldContract)
+ Contract(User firstParty, Contract oldContract)
+ Contract(Contract fromBid)
+ getFirstPartyId()
+ getSecondPartyId()
+ getId()
+ getSubjectId()
+ getDateCreated()
+ getExpiryDate()
+ getLessonInfo()
+ getInitialRequestId()
+ getDateSigned()
+ sign()
+ setContractDuration(String contractDuration)
+ getContractDetails ()
+ isSubscribed()
+ subscribe()
+ unsubscribe()
+ getNotification()
Request
- initiatorId
- initiatorName
- id
- type
- dateCreated
- dateClosedDown
- subjectId
- messages[]
- competency
- LessonInfo lessonInfo
- contractIds[]
+ Request(User user, String type, String subjectId, int competency, String hrsPerSession, String sessionsPerWeek, String ratePerSession)
- getInitiatorId()
+ getInitiatorName()
+ getId()
+ getType()
+ getDateCreated()
+ getDateClosedDown()
+ getSubjectId()
+ getMessagesForContract(String ContractId)
+ getLessonInfo()
+ toString()
+ getCompetency()
Message
- requestId
- posterId
- contractId
- posterUserName
- datePosted
- dateLastEdited
- content
+ Message(Bid b, User poster, Contract c, String content)
+ getRequestId()
+ getPosterId()
+ getBidId()
+ getPosterId()
+ getDatePosted()
+ getContent()
+ getContractId()
+ toString()
User
- id
- userName
- givenName
- familyName
- isTutor
- isStudent
~ userBids[]
~ competencies[]
+ User(String id, String userName, String givenName, String familyName, boolean isTutor, boolean isStudent)
+ getId()
+ getUserName()
+ getGivenName()
+ getFamilyName()
+ isTutor()
+ isStudent()
+ getUserBidIds()
+ getCompetencyLevel(String subjectId)
Competency
- id
- ownerId
- subjectId
- level
+ Competency (jsonString)
+ getId ()
+ getSubjectId ()
+ getLevel ()

Design rationale by Andrew And Bassel

1.1 Architectural Patterns

Refactoring

h In the past, there was already some form of **Model-View-Controller architecture** that was implemented However it was observ enforced as the View and Controllers were somewhat behaved together This made the UI classes very large and therefore violate the Single Responsibility Principle .

MVC is now strictly enforced in the system's architecture The UI package contains the views used in the system, h these classes only contain code that deals with the UI such as initializing and defining the properties of text boxes, labels and buttons and do not contain any form of logic code The logic code is separated from the UI code in the Controller Package All classes in the View package inherit from the Window Layout Class while all classes in the Controller package inherit from the Window Controller Class Each Window Layout will have a view of its very own Window Controller class which Each View Class will have a view of its respective Controller Class By doing so, the **Single Responsibility Principle** can be adhered to .

h The classes in the model and adapters packages can together be classified as the Model of the system, they are typically deemed to be the most stable packages of a kind as they model the real world business logic which rarely changes The model was split into 2 packages, namely model and adapters This is because the adapters k package deals with the external API (see 1 2 Design Patterns .).

h The MVC architecture implemented is passive MVC because the model rarely has to interact with the view However in order to fulfill a feature from requirement 2 that requires the system to refresh and update the view every N seconds A **partially-active MVC** solution was employed in which we have the controller class implement a h refreshable interface that forces the controller to prompt the model and update the view at intervals

h The advantages of adopting the MVC architecture is that it narrows down the frame of concentration for the developer We can break the system down into 2 seperate parts and get the best of both sides (UI and k back end look to work on the same system) separately The disadvantage of it is architecture hides in the complexity of h the controller classes The controller classes have to string together the two widely different codes of the model h and the view, which may be time consuming .

1.2 Design Patterns

New Functionality

1 **Observer design pattern**.

Observer was employed in the system to fulfill requirement 1 This can be seen in the MonitorWindowClass, h h which is the controller class that acts as the Publisher in this scenario, it checks for the changes in each Subscription (which is an interface employed by each Contract NOTE: An unsigned contract acts as a bid for the purposes of this system and displays to the user a) the bids that it has subscribed to in a it's ViewLayout which is MonitorLayout The **Open-Closed Principle** is enforced here by making the contract class implement the Subscription interface The bids subscription functionality can be extended easily in the future without having to h change (or modify the Contract class which is) the bid itself .

h When implementing requirement 2 where the system needs to display a notification to the user once a signed contract is reaching its 1 month to expiry In this scenario, the **Observer design pattern** is also employed The SelectActionWindowController that controls the notification panel acts as the publisher and observes for each subscribed contract by the user It then prompts the contract class for a contract expiry notification if the user is subscribed to the contract and displays the notification on the notification panel in the SelectActionLayout (it's View).

2 **Memento design pattern**

In order to fulfil the requirement, the Memento design pattern was employed. This is because when extending a Contract, students do not need to re-enter all the details of the Contract such as rate, sessions, subject and other particulars, all they need to change is the contract duration. In this system, a Contract can be renewed by creating a NEW Contract but using a different constructor which takes in the OLD Contract instance that the student intends to extend the Contract with. The constructor uses this old Contract and creates a new snapshot of the Contract but with the new expiry date and is not signed.

The benefits of doing so is that the external system does not need to know the implementations of creating a new Contract when renewing a Contract yet it still provides a way for the external system to track which contract that the renewed contract originated from. This preserves the encapsulation of the Contract class as the variables in the class can remain private.

Refactoring

3 **Adapter design pattern**.

The most obvious change here is the Adapter design pattern that was extracted from the controller classes in the previous design (refer to the previous video explanation. It was realised that having the API calls nested with the controller classes of the View is not optimal. Therefore, the adapter design pattern was used to abstract the API calls from the controller and view classes so that it can be handled by the classes in the adapters package. Using this design pattern, it was possible to upgrade the current API version from 1 to 2 seamlessly by editing merely 2 lines of code.

This is the biggest advantage of having an adapter, where we can handle upgrades in the API without changing much of the code. The downside to this is that the adapter is a single point of failure. For example, if the root url is entered wrongly, all API connections will fail. Please note that the adapters are still singletons as mentioned in the previous design rationale.

1.3 Package Design Principles

There were no new package design principles that were added into the overall system and they maintain the package design principles as laid out in the previous design rationale.

1 **Common Closure Principle**

By implementing the MVC architecture in a stricter manner, CCP is now more strongly enforced as the reasons behind classes being grouped in their respective packages are now more uniquely defined. For example, the view package classes only deals with the UI and are dependent on the java swing library. The adapters package only deals with the API connections, the controller package classes act as the middlemen between view and model and finally, the model package classes deals with the real world business logic.

Other package design principles that emerge as a result of the MVC implementation are the **Acyclic Dependency Principle and Stable Dependency Principle** as outlined in the previous design rationale.

1.4 Class Design Principles

New Functionality

1 **Open Closed Principle**.

As mentioned earlier, the OCP was enforced when implementing the feature for requirements 1 and 2. This was achieved by making the Contract class implement the subscription interface. In the event of a new functionality to be added to the Subscription feature without having to change the Contract classes, the developer can just update the Subscription interface. This implies that the system welcomes extension of new features but is closed for modification.

Refactoring

2 **Interface Segregation Principle**

A new class design pattern was enforced while performing refactoring. It was noted that if new controller classes were forced to implement an empty refresh() function despite not having to refresh their pages. Therefore the RefreshableInterface was created to separate the view controller classes that need the refresh feature from the ones that do not. This is a form of ISP in action.

The advantage of implementing ISP is that it makes the code neater and reduces the size of the classes as it removes dead and redundant code. However, though not apparent in this assignment, over-use of ISP may result in increased complexity of the code.

3 **Single Responsibility Principle**

As mentioned above, the SRP was also adhered to simply from enforcing a strict MVC architecture. In which each class in the view, controller and model package now has a very distinct purpose and responsibility. The only place where SRP is violated in the system is in the Adapter classes as they have to handle both the API connections and enforcing the rules of accessing the API. However, this violation is negligible.

The advantage of enforcing SRP is that it limits the scope of concentration for the developer working on a class and therefore producing code with fewer bugs and is more easily testable. However, overdoing SRP may result in a very fragmented codebase which encourages coupling and increases code complexity and reduced maintainability.

2.0 Refactoring Techniques

1. Extract Variable

In the refresh() method of the ViewContractWindow class, there used to be a complex sequence of if-else statements to determine the state of each button in the ViewContractWindow (whether to enable them or not). This made each expression hard to implement. The extract variable method was used in this case to replace the if-else statements and the code uses boolean logic now to determine if a button should be enabled or not.

The advantages of doing so is to improve the readability of the code. It has also significantly improves the system performance as nested if-else statements generally eat up a lot of resources due to the repeated checks that are sometimes duplicate.

2. Duplicate Observed Data

In order to strictly enforce the MVC architecture, it was noted that in the previous assignment the responsibilities of the view and controller classes were essentially blended in one class. This means that the code containing the logic and domain data binding the user inputs were in the ViewClasses as well. Therefore, duplicate observed data method is used to separate the responsibility of the controller class out of the ViewClasses, further enforcing the MVC architecture.

The advantages of separating the responsibility of the classes is that SRP can be fulfilled because now each View Class is only responsible for the UI elements while the Controller Classes are responsible for the logic behind each ViewClass. This way we also make room for further extension to the view of the system without having to change the controllers, essentially fulfilling the OCP.

3. Extract Class

The Contract and Request class from the previous system was deemed too large by the team, besides that there were similar variables that exist in both the Contract and Request class. Therefore the extract class.method was used to extract the LessonInfo class out from the large Contract and Request Class.

This will improve code maintainability and further enforce the idea of SRP in the system. This also prevents shotgun surgery as previously any edits in the Contract class will require edits in the Request class as well.