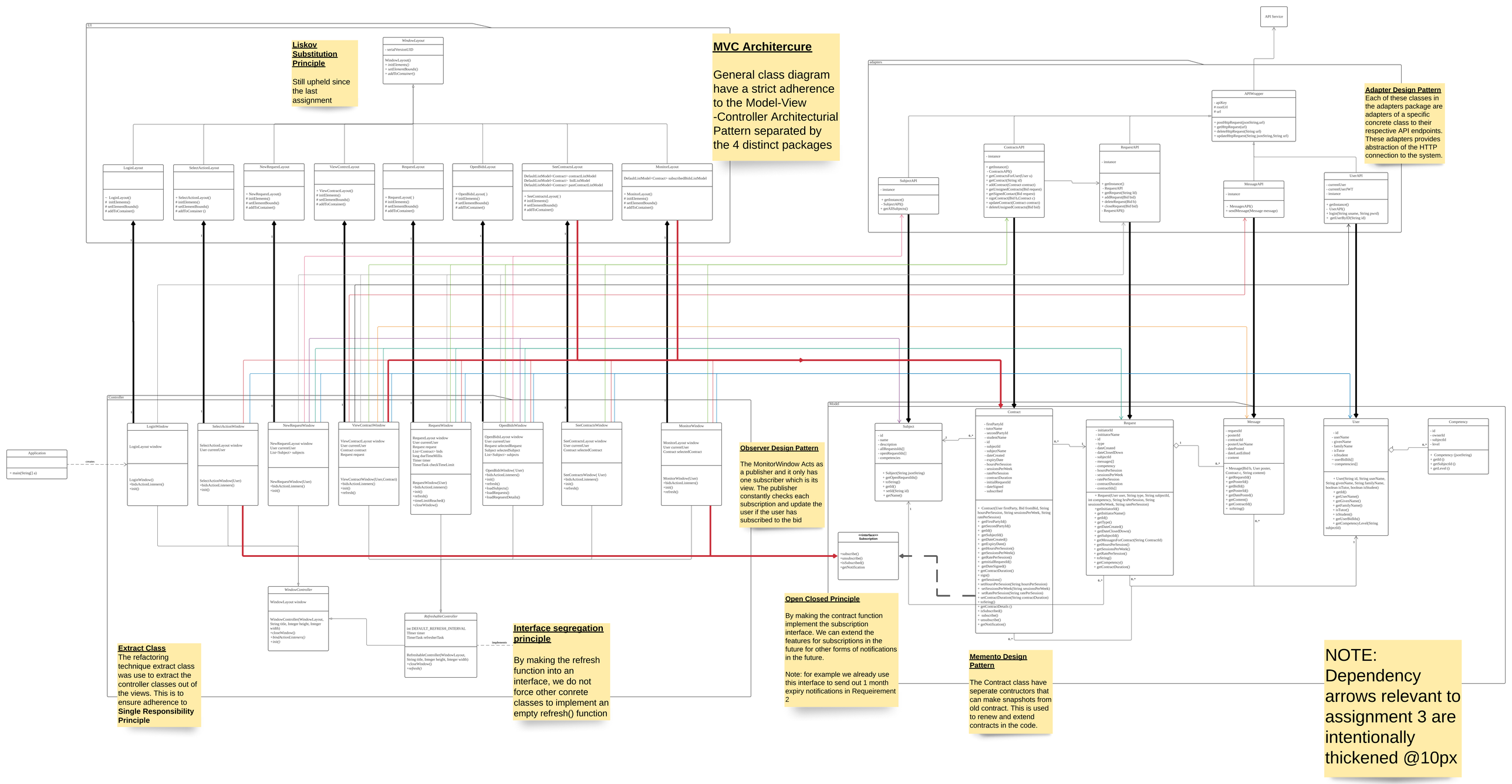
UI
WindowLayout
- serialVersionUID
WindowLayout()
+ initElements()
+ setElementBounds()
+ addToContainer()
LoginLayout
SelectActionLayout
NewRequestLayout
ViewContrctLayout
RequestLayout
OpenBidsLayout
SeeContractsLayout
MonitorLayout
Liskov Substitution Principle
Still upheld since the last assignment
MVC Architecure
General class diagram have a strict adherence to the Model-View -Controller Architecturial Pattern separated by the 4 distinct packages
API Service
adapters
APIWrapper
- apiKey
rootUrl
url
+ postHttpRequest(jsonString,url)
+ getHttpRequest(url)
+ deleteHttpRequest(String url)
+ updateHttpRequest(String jsonString, String url)
Adapter Design Pattern
Each of these classes in the adapters package are adapters of a specific concrete class to their respective API endpoints. These adapters provides abstraction of the HTTP connection to the system.
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 Id)
+ addRequest(Bid bid)
+ deleteRequest(Bid b)
+ closeRequest(Bid bid)
- RequestAPI()
MessageAPI
- instance
- MessagesAPI()
+ sendMessage(Message message)
UserAPI
- currentUser
- currentUserJWT
- instance
+ getInstance()
- UserAPI()
+ login(String uname, String pwrd)
+ getUserByID(String id)
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(User,Contract)
+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()
Observer Design Pattern
The MonitorWindow Acts as a publisher and it only has one subscriber which is its view. The publisher constantly checks each subscription and update the user if the user has subscribed to the bid
Application
+ main(String[ ] a)
creates
WindowController
WindowLayout window
WindowController(WindowLayout, String title, Integer height, Integer width)
+closeWindow()
+bindActionListeners()
+init()
RefreshableController
int DEFAULT_REFRESH_INTERVAL
Timer timer
TimerTask refresherTask
RefreshableController(WindowLayout, String title, Integer height, Integer width)
+closeWindow()
+refresh()
implements
Interface segregation principle
By making the refresh function into an interface, we do not force other conrete classes to implement an empty refresh() function
Extract Class
The refactoring technique extract class was use to extract the controller classes out of the views. This is to ensure adherence to Single Responsibility Principle
Model
Subject
- id
- name
- description
- allRequestsIds[ ]
- openRequestsIds[ ]
- competencies
+ Subject(String jsonString)
+ getOpenRequestIds()
+ toString()
+ getId()
+ setId(String id)
+ getName()
Contract
- firstPartyId
- tutorName
- secondPartyId
- studentName
- id
- subjectId
- subjectName
- dateCreated
- expiryDate
- hoursPerSession
- sessionsPerWeek
- ratePerSession
- contractDuration
- initialRequestId
- dateSigned
- subscribed
+ Contract(User firstParty, Bid fromBid, String hoursPerSession, String sessionsPerWeek, String ratePerSession)
+ getFirstPartyId()
+ getSecondPartyId()
+ getId()
+ getSubjectId()
+ getDateCreated()
+ getExpiryDate()
+ getHoursPerSession()
+ getSessionsPerWeek()
+ getRatePerSession()
+ getInitialRequestId()
+ getDateSigned()
+ getContractDuration()
+ sign()
+ getSessions()
+ setHoursPerSession(String hoursPerSession)
+ setSessionsPerWeek(String sessionsPerWeek)
+ setRatePerSession(String ratePerSession)
+ setContractDuration(String contractDuration)
+ toString()
+ getContractDetails ()
+ isSubscribed()
+ subscribe()
+ unsubscribe()
+ getNotification()
<<interface>> Subscription
+subscribe()
+unsubscribe()
+isSubscribed()
+getNotification
Open Closed Principle
By making the contract function implement the subscription interface. We can extend the features for subscriptions in the future for other forms of notifications in the future.
Note: for example we already use this interface to send out 1 month expiry notifications in Requeirement 2
Request
- initiatorId
- initiatorName
- id
- type
- dateCreated
- dateClosedDown
- subjectId
- messages[]
- competency
- hoursPerSession
- sessionsPerWeek
- ratePerSession
- contractDuration
- 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)
+ getHoursPerSession()
+ getSessionsPerWeek()
+ getRatePerSession()
+ toString()
+ getCompetency()
+ getContractDuration()
Memento Design Pattern
The Contract class have seperate contructors that can make snapshots from old contract. This is used to renew and extend contracts in the code.
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
- userBidIds[ ]
- 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 ()
NOTE: Dependency arrows relevant to assignment 3 are intentionally thickened @10px

Design rationale by AndrewAndBassel

1.1 Architectural Patterns

The architectural pattern is a pattern that emerges from the system as a whole. It is the most obvious pattern of all and can be instantly identified from a glance of the class diagram.

Refactoring

In the past, there was already some form of **Model-View-Controller architecture** that was implemented. However it was loosely enforced as the View and Controllers were somewhat blended together. Logic code was embedded in the classes that initialized the UI elements and its properties. This made the UI classes very large and therefore violate the Single Responsibility Principle.

MVC is now strictly enforced in the system's architecture. This can be seen from the categorization of classes in different packages. The UI package contains the views used in the system, these classes only contain code that deal with the UI such as initializing and defining the properties of textboxes, 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, these classes deal with the logic behind the user inputs and acts as the middle men between the model and views. All classes in the View package inherit from the WindowLayout Class while all classes in the Controller package inherit from the WindowController Class. Each WindowLayout will have its very own WindowController class. (i.e. Each View Class will have its own respective Controller Class). By doing so, the **Single Responsibility Principle** can be adhered to.

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 all and 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 package deals with the external API, this separation provides some form of abstraction to the model as mentioned below (see 1.2 Design Patterns)

The MVC architecture implemented is passive MVC because it is noted that models rarely have to interact with the views. 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 refreshable interface that forces the controller to prompt the model and update the view at intervals.

The advantages of adopting the MVC architecture is that it narrows down the frame of concentration for the developer. A developer may be specialised in either UI design or back-end algorithms but rarely in both aspects. Therefore, by employing the MVC architecture, we can break the system down into 2 seperate parts and get the best of talents from both sides (UI and backend) to work on the same system separately. The disadvantage of this architecture lies in the complexity of the controller classes. The controller classes have to string together the two wildly different codes of the model and the views, which may be time consuming.

1.2 Design Patterns

These patterns were found in different segments of the code when implementing the new functionality and during refactoring.

New Functionality

In the process of extending the functionality of the system, several design principles and design patterns were upheld in the system.

1. **Observer design pattern**

Observer was employed in the system to fulfill requirement 1. This can be seen in the MonitorWindow Class, 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 all the bids that it has subscribed to via it's VIew 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 change (or modify) the Contract class which is the bid itself.

A very good example of the OCP in action is 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 SelectActionWindow controller that controls the notification panel acts as the publisher and looks 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 fulfill requirement 3, 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. Therefore the system does not need the implementation of creating a new Contract. 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 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. The disadvantage of memento is that the program may take up alot of memory space if multiple Contract instances are created, making the application Therefore there should be a limit to the number of renewed contracts.

In a less direct manner, a modified memento design pattern was also employed when implementing the buy-out bid functionality in the previous system. When buying out a request, the tutor essentially agrees to all terms stated in the request and all terms are essentially copied one-for-one to the contract. Therefore, a Contact that is bought out is essentially a snapshot of the Request that it was before. The Contract class has a constructor that takes in a Request and generates an identical snapshot of the Request but in the form of a Contract.

<u>Refactoring</u>

In this part of the project, several other design patterns and design principles have emerged as a result of 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 Views 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 v1 to v2 seamlessly by editing merely 2 lines of code.

This is the biggest advantage of having an adapter, 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 generally follow the package design principles as laid out in the previous design rationale.

1. **Common Close Principle**

By implementing the MVC architecture in a stricter manner, CCP is now more strongly enforced as the reasons behind classes being put in their respective packages are now more uniquely defined. For example, the view package classes only deal 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 middle-men between views and models 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 view-controller classes were forced to implement an empty refresh() function despite not having to refresh their pages. Therefore the Refreshable interface 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**

Also 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 code which encourages coupling and increases code complexity and reduced maintainability.

2.0 Refactoring Techniques

Several refactoring techniques have been applied in this assignment

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 also slightly 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. **Extract Class**

In order to strictly enforce the MVC architecture, it was noted that in the previous assignment the responsibilities of the views and controllers classes were essentially blended in one class. This meant that the code containing the logic behind the user inputs were in the View Classes as well. Therefore we used the extract class method to separate the responsibility of the controller class out of the View Classes, 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 View Class. This way we also make room for further extension to the views of the system without having to change the controllers, essentially fulfilling the OCP.