MONASH University

FIT 3077 - Software Engineering: Architecture and Design

**Team Software Architecture Assignment**

# Stage 1 - UML Design Report


## Semester 1, 2015

**Team "CodeIneNoobs":**

Arvin Wiyono (awiy1 - 24282588)

Darren Wong (dwwon6 - 25147064)

# Table of Content

# 1. Introduction

The purpose of this report is to document any UML design diagrams which represent the design decision and the infrastructure of the classes. The diagrams that will be presented include:

- **Conceptual Class Diagram**
  Captures the important concepts and relationships between classes in the Weather Monitor application.

- **Design Class Diagram**
  More in-depth representation of the classes, including the attributes, methods and extra classes utilized to enable the logical functionality of the Visual Basic graphical user interface.

- **Sequence Diagrams**
  Present how objects interact to carry out a functionality of the system to fulfil the given requirements. The Use Cases that are modelled are “Add Location”, “Remove Location” and “Update Location”.

- **State Diagram**
  Describes how the WeatherLocation object changes its state over time in the Weather Monitor application.

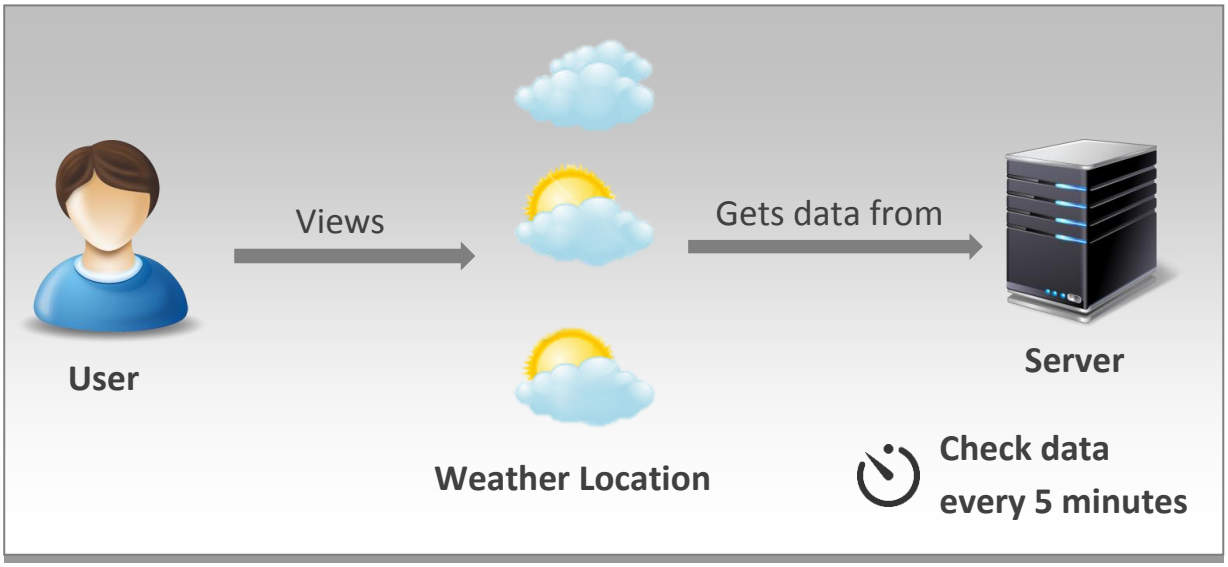
Furthermore, several diagrams will be supported by an explanation regarding its design and weaknesses or limitations if any.

The design decisions were carefully considered so that it can be extended in the later stage of the assignment. It is expected that the system has both flexibility and stability in a balance manner. There have been changes done to the current system compared to the original consistent system based on the repeating feedback from the tutor and lecturer.

*All diagrams will be provided both in (.uxf) and (.pdf) format in the **Documentation Stage 1** folder

## 2. Concept of Application

The concept of the application is to build a Weather Monitor application with a Graphical User Interface (GUI), which allows the user to view the weather information of multiple locations. The system gets the supplied live data from the provided server and then creates separate Monitors for displaying the respective information. The application is required to check the server data every 5 minutes so that user is kept up to date with the latest weather information for their selected locations. This is illustrated as below:



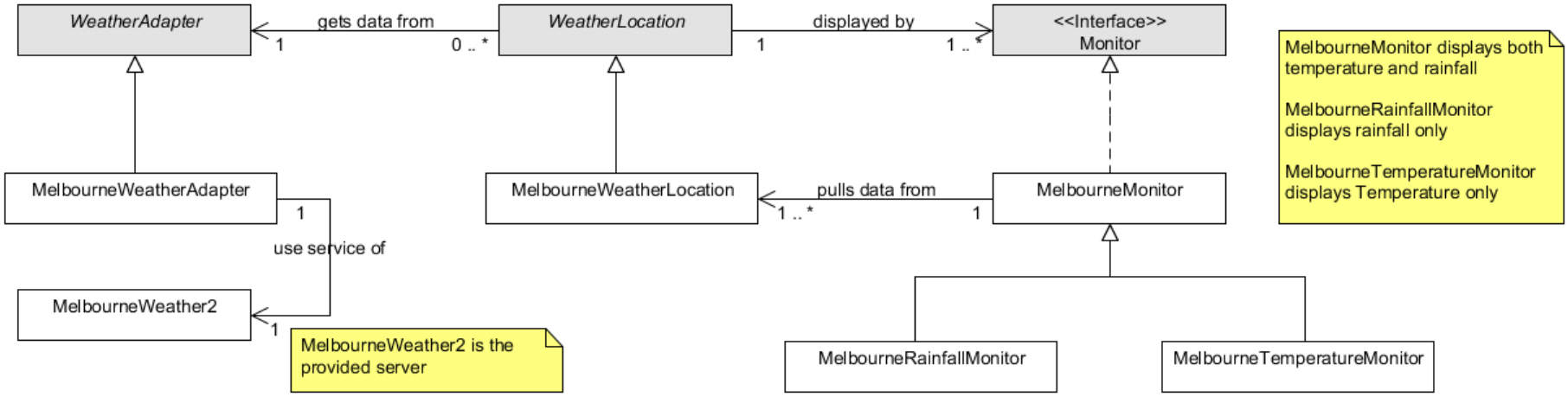

Simple Illustration of Weather Monitor Application

## 3. Stage One Requirements

The requirements for Stage 1 are:

- View the temperature and/or rainfall at selected locations.
- For each location, a separate monitor must be increased.
- The user can choose to view the weather for more than one location simultaneously.
- The User can choose to stop viewing the weather for a location.
- The program should check the web service every 5 minutes and update the monitor(s) as necessary.

## 4. Conceptual Class Diagram




### 4.1. Design Explanation

The above conceptual class diagram represents the architecture of classes in the Weather Monitor application. The design is mainly derived from the ***Observer*** design pattern. This is evident by the associations between WeatherLocation and Monitor classes (including their subclasses). In this case, there is two-way navigability between both classes. This is because a WeatherLocation must know its monitors and MelbouneWeatherLocation must be visible to the MelbourneMonitor class for pulling information.

Furthermore, there is also use of ***Abstract Factory*** pattern between WeatherAdapter and WeatherLocation classes. This is shown by the association, i.e. WeatherLocation (the client) only depends on the services provided by the WeatherAdapter (the supplier).

### 4.2. Strength and Limitation of Design

There are multiple strengths to this design due to its flexibility.

Firstly, in the case we wanted to use another service provider, the WeatherLocation does not need to be recompiled because the WeatherAdapter handles the information that comes from the service provider. Consequently, it becomes the medium of data delivery between a WeatherLocation and the actual server. This further allows multiple service providers to be used in parallel if is needed.

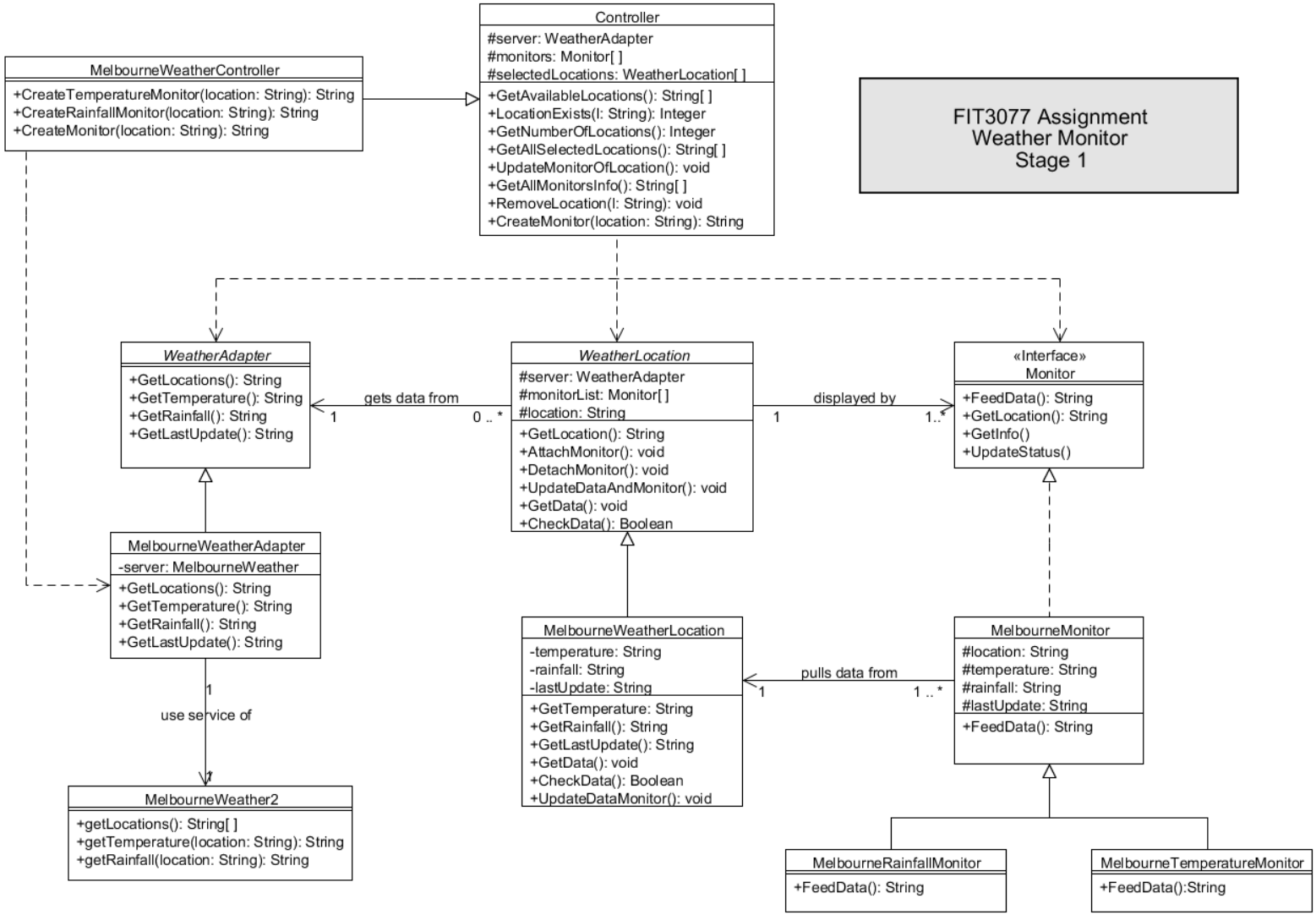
Secondly, we create a **Monitor interface** which serves as a blueprint to the subclasses. This design allows new Monitor subclasses for particular WeatherLocations to be added in the future upgrade. This results in the creation of Monitors with different types of data (numeric/non-numeric) since they are able to have different attributes and methods to pull the data from a particular WeatherLocation.

Similarly, **WeatherLocation abstract class** provides the basic underlying information of a WeatherLocation. It must have/know at least these 3 attributes: **server, monitorList** and the **location** itself. As a result, in the subclasses, we can have different attributes between subclasses. For example, WeatherLocation1 class might have temperature and rainfall, but WeatherLocation2 can possibly have humidity and precipitation. The design absolutely supports extensions to this section of the system.

Lastly, *Observer* design pattern can have two techniques in receiving data from the subject. To begin with, it is important to note that in our design, Monitor class is the Observer and WeatherLocation is the Subject. The first technique is what we currently implement: two-way navigability between WeatherLocation and Monitor. This allows Monitor to **pull** data from the location **with a trade-off that they become tightly coupled**. The second technique asks the WeatherLocation to **push** data to all Monitors without tightly couple the two classes (one-way navigability).

Considering these two options, we would prefer the **pull** technique as it is more efficient in the case as we have a lot of Monitors (worst case scenario) for a particular WeatherLocation at the same time. This frees up the WeatherLocation from the responsibility of pushing the data to an excessive number of Observers compared to the second technique.

## 5. Design Class Diagram

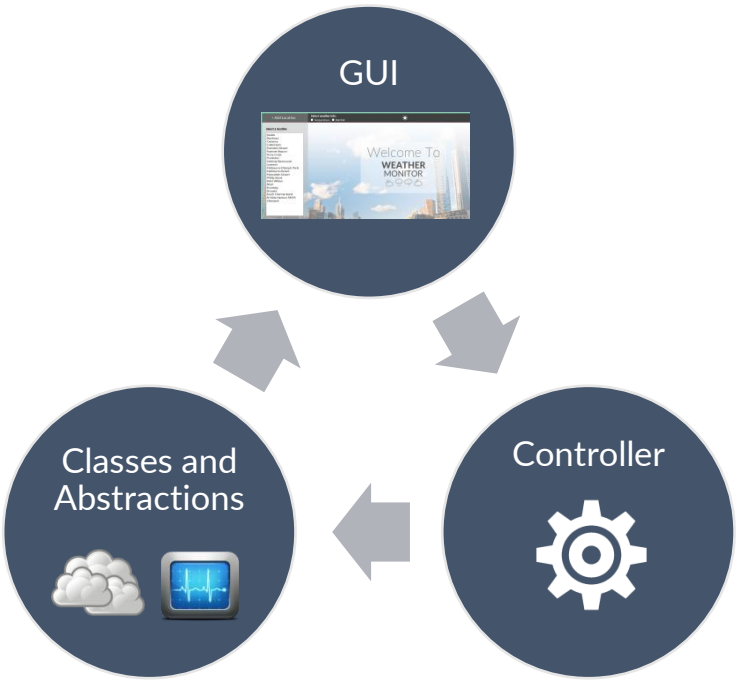


### 5.1. Design Explanation

The design class diagram contains two classes which are not included in the previous conceptual class diagram. We introduce the use of **Controller** class which performs and takes care of any logical thinking behind the GUI. This will result in separation between the GUI and the classes. By doing this way, we can replace the GUI with any kinds of user interface, yet will still yield the same result since both UI and Controller are independent. The organization of the system is based on the *Model-View-Controller* pattern.

Next, it can be seen that WeatherAdater, WeatherLocation and Monitor must be visible to the Controller. This is because it has the responsibility to handle the functionalities of the system. To enhance its cohesion, a subclass namely MelbourneWeatherController is created, which depends on a particular subclass of WeatherAdapter (MelbourneWeatherAdapter).

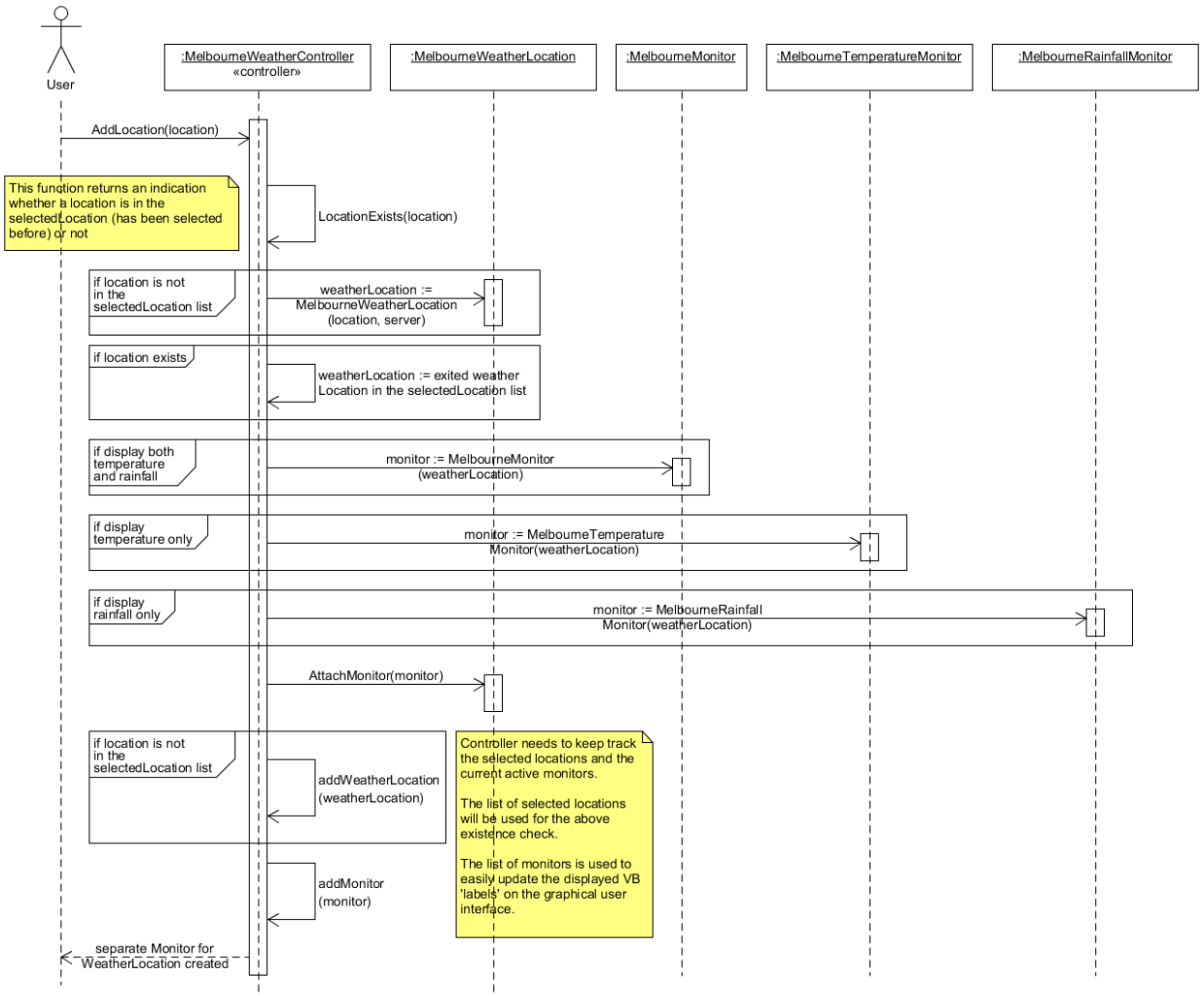
This can be seen as a partial solution as it could not use the service provided by the WeatherAdapter abstract class. However, this is essential for creating 3 different types of required Monitors by calling the three methods: CreateTemperatureMonitor(), CreateRainfallMonitor, CreateMonitor(). The last method simply creates a Monitor object which displays both temperature and rainfall data. Below is an illustration of how the system overview:




Model-View-Controller Pattern in Weather Monitor Application

# 6. Sequence Diagram

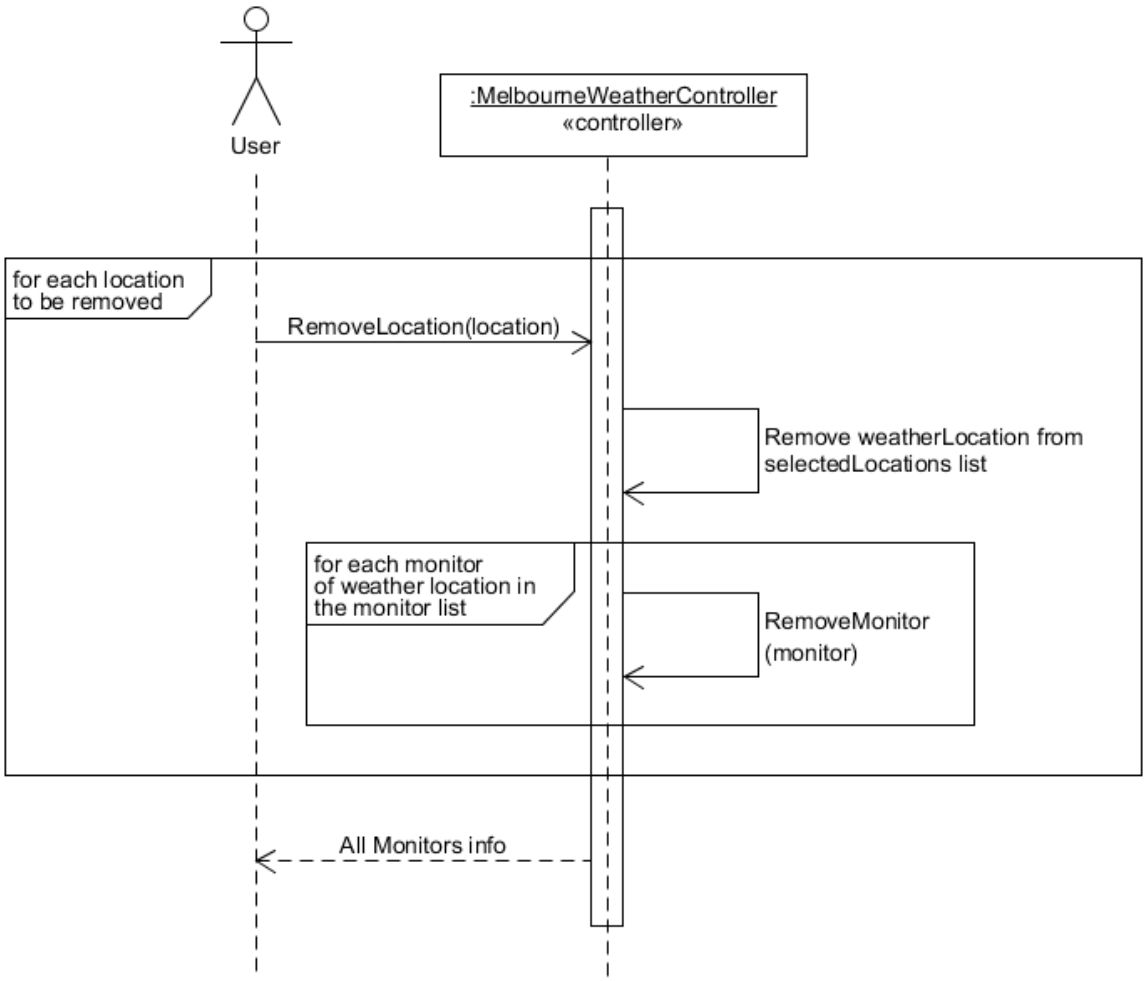
## 6.1. Functionality : Add Location




The *Add Location* functionality is one of the major requirements that the application must have. To begin with, it gives the user 2 options: display temperature or display rainfall. However, if both checkboxes are ticked, both temperature and rainfall will be shown. This leads to the increase of complexity in the controller class since it must know what kind of Monitor object which must be created. Fortunately, it is the responsibility of the Controller subclass rather than the superclass.

As stated in the design class diagram, Controller class depends on both Monitor and WeatherLocation to keep track of previously selected locations. This significantly enhances the speed of creating monitors of an existed location since it doesn't need to access the service provider. The list of Monitors is used for easy update of Visual Basic Label controls in the MainPage.vb.

### 6.2. Functionality : Remove Location

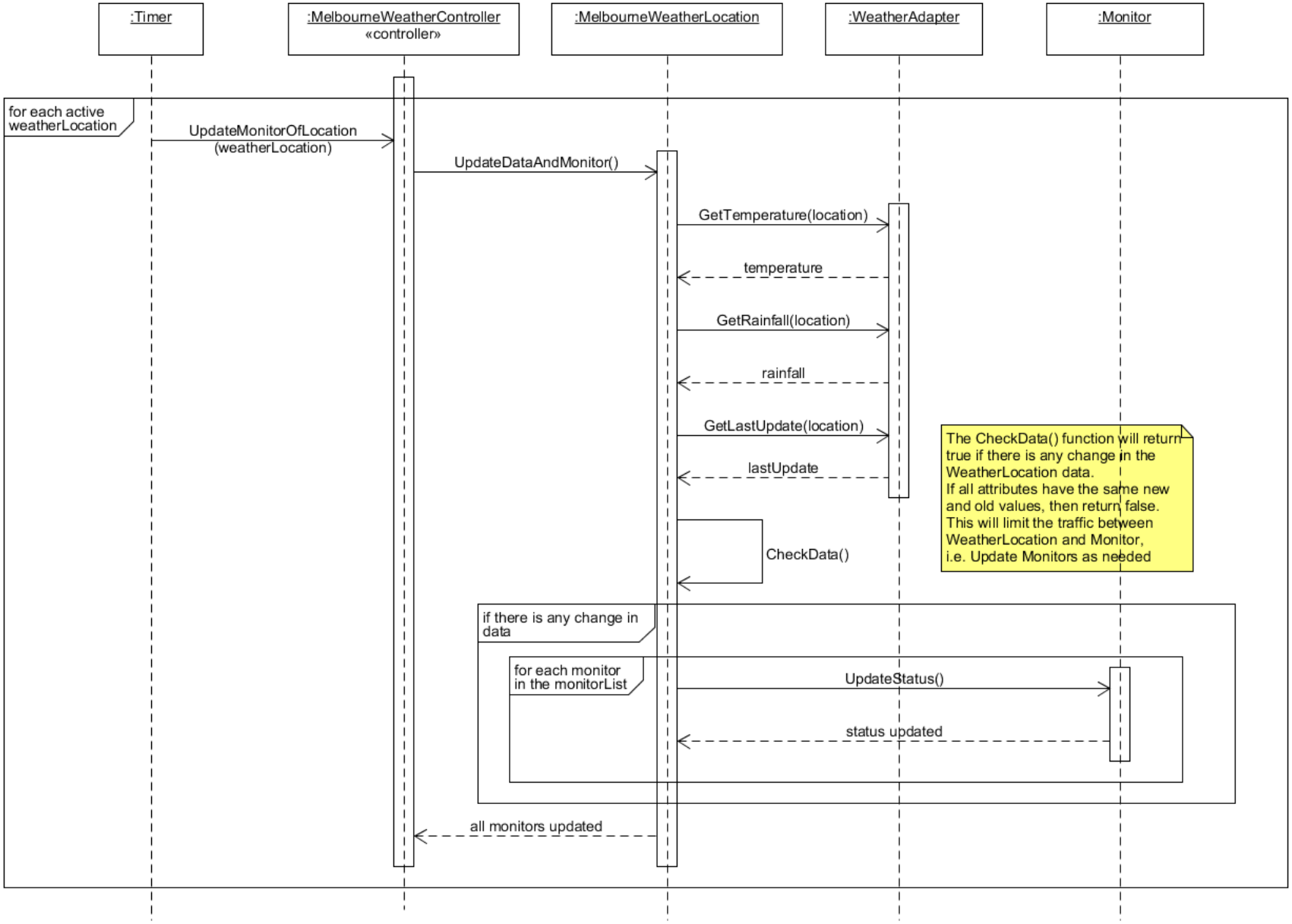



The *Remove Location* sequence diagram is relatively simple. It removes the locations selected by the user in the RemovePage's list box and throws away the responsibility to the Controller class.

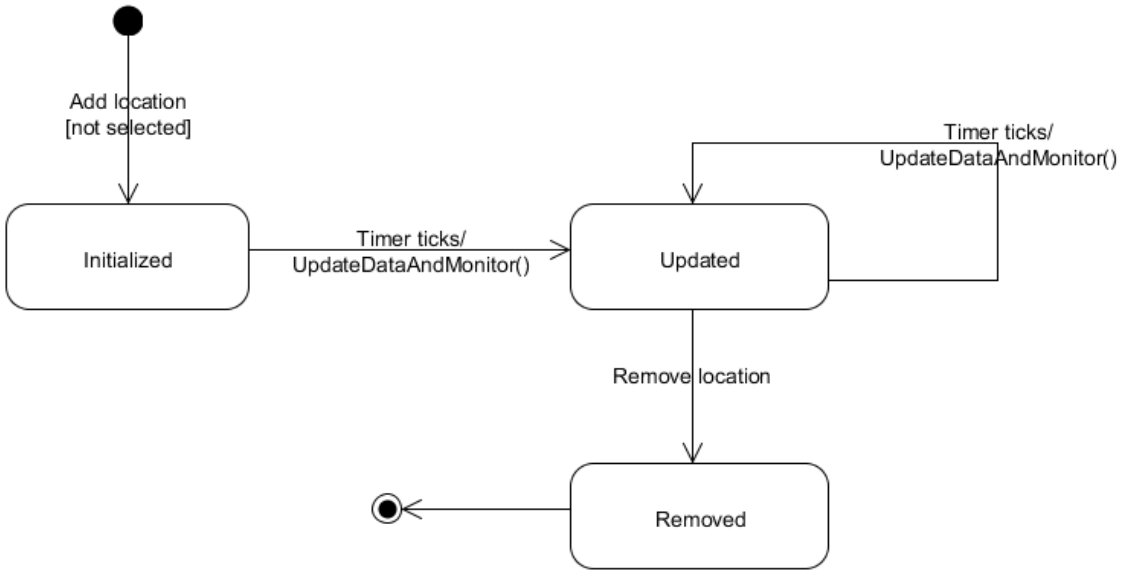
The Controller class then removes each location from the selectedLocations list. As we know that a WeatherLocation object might have more than one monitor, thus it needs to remove the corresponding Monitors in the *monitors* list. At the end, RemoveLocation() function returns all the weather information of the undeleted Monitors so that the Label controls in the MainPage.vb can be re-drawn.

The limitation of this functionality is that it removes all the Monitors of a particular location at once (temperature and rainfall, temperature only, and rainfall only Monitors). A better approach is to list the active Monitors in the RemovePage.vb instead of the locations. However it could potentially increase the complexity of Controller's RemoveLocation() function quite significantly, thus will be harder to implement.

## 6.3. Functionality : Check Data and Update Monitor



## 7. State Diagram: Weather Location




The above diagram describes the states of a WeatherLocation object. There are 3 identified states in the current system:

- **Initialized**
  A WeatherLocation object will be created if it is not yet selected previously by the user. Then it will be directly initialized by getting the most current data from the service provider WeatherAdapter. Thus all attributes/values are initialized automatically during creation. Finally, the Controller will put the object to the selectedLocations list.

- **Updated**
  When the timer ticks, it executes the update routine. This commands all selected WeatherLocations to perform a data transaction again with the WeatherAdapter. Consequently, the attribute values must be updated by the time the update routine is completed. Here, once a WeatherLocation moves to this state, it will always be updated in the future when the timer ticks again.

- **Removed**
  Once the user decides to remove this particular WeatherLocation, the Controller removes it from the selectedLocations list and lastly, the object goes to its final state and is eliminated from the system.