SENG 330 Assignment 3 Design Document

Dana Wiltsie & Keanu Enns

Nov. 16, 2018

Notes to Marker

The application can be built, tested, and run using the commands gradle build, gradle test, and gradle run respectively. Equivalently, the commands can be run through the gradle plugin in Eclipse. The final option for running the application is to right click the class IOTApp.java under package ca.uvic.seng330.assn3 and select “Run as” -> “Java application”.

Two automatic client accounts will be registered to the system upon start up. The user names are “basic” (the basic user account) and “admin” (the admin user account). The password for both accounts is “password”. In addition, the marker may feel free to create other users from the login screen (although not implemented in the admin view for this assignment).

The acceptance tests to be marked are as follows:

- Scenario A: 1-4, 7
- Scenario B: 2,3,6
- Scenario C: 3-5 (note for 4: A known inconvenience exists in the code that deselects a device and defaults to the first device in the list each time it is toggled off, so though it is possible with the other devices, toggling the first camera in the list will be easiest. The toggle button must be clicked 9 times after turning the camera on)
- Scenario D: 1-5 are complete, but choose number 1 for testing
- Scenario E: 1,2
- Scenario F: 1-3.

For the purposes of marking, it is important to know what is not implemented. For example, this application currently has no offline data storage (default devices and users are created upon start up), so changes made will remain if users logout and back in but not if the application is closed. Also, no focus has yet been given to providing the admin with a log of activities (although information is logged to the console), allowing an admin to create users, or assigning devices to users and displaying them in the user’s view accordingly.

The Design

As suggested in class, we decided to go with a “vertical slicing” approach (i.e. imagine we have three rows: View, Controller, Model and we slice them vertically to create columns). This means we started by providing “end-to-end” functionality for each component of the application one by one (or column by column) rather than creating a complex network of views and models and trying to figure out how to connect them via controllers at the end.

We illustrate our design with a class diagram of what we can consider to be the important vertical slices laid down on their side. For simplicity, rather than include all the devices (camera, thermostat, smart plug, and lightbulb), we illustrate the design with only the camera model, controller, and view classes. However, equivalent device counterparts exist and behave in nearly the same way as the camera (i.e. they all extend the abstract class Device and their controllers are associated with the admin controller).

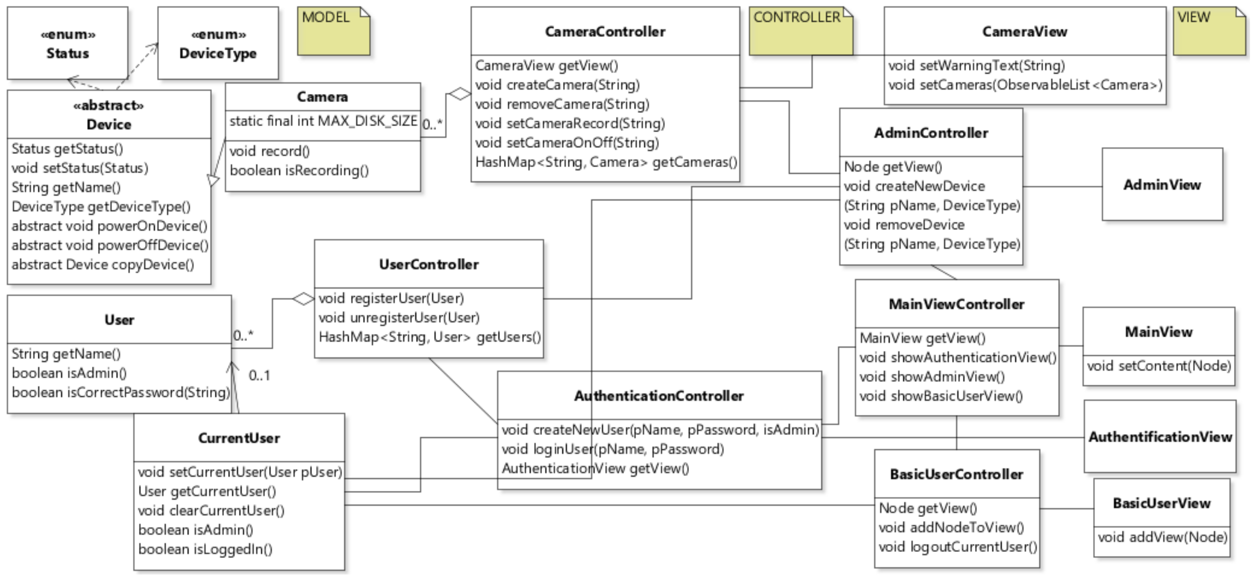

«enum» Status
«enum» DeviceType
MODEL
«abstract» Device
Status getStatus()
void setStatus(Status)
String getName()
DeviceType getDeviceType()
abstract void powerOnDevice()
abstract void powerOffDevice()
abstract Device copyDevice()
Camera
static final int MAX_DISK_SIZE
void record()
boolean isRecording()
0..*
CameraController
CameraView getView()
void createCamera(String)
void removeCamera(String)
void setCameraRecord(String)
void setCameraOnOff(String)
HashMap<String, Camera> getCameras()
CONTROLLER
CameraView
void setWarningText(String)
void setCameras(ObservableList<Camera>)
VIEW
AdminController
Node getView()
void createNewDevice (String pName, DeviceType)
void removeDevice (String pName, DeviceType)
AdminView
UserController
void registerUser(User)
void unregisterUser(User)
HashMap<String, User> getUsers()
User
String getName()
boolean isAdmin()
boolean isCorrectPassword(String)
0..*
0..1
MainViewController
MainView getView()
void showAuthenticationView()
void showAdminView()
void showBasicUserView()
MainView
void setContent(Node)
AuthenticationController
void createNewUser(pName, pPassword, isAdmin)
void loginUser(pName, pPassword)
AuthenticationView getView()
AuthentificationView
CurrentUser
void setCurrentUser(User pUser)
User getCurrentUser()
void clearCurrentUser()
boolean isAdmin()
boolean isLoggedIn()
BasicUserController
Node getView()
void addNodeToView()
void logoutCurrentUser()
BasicUserView
void addView(Node)

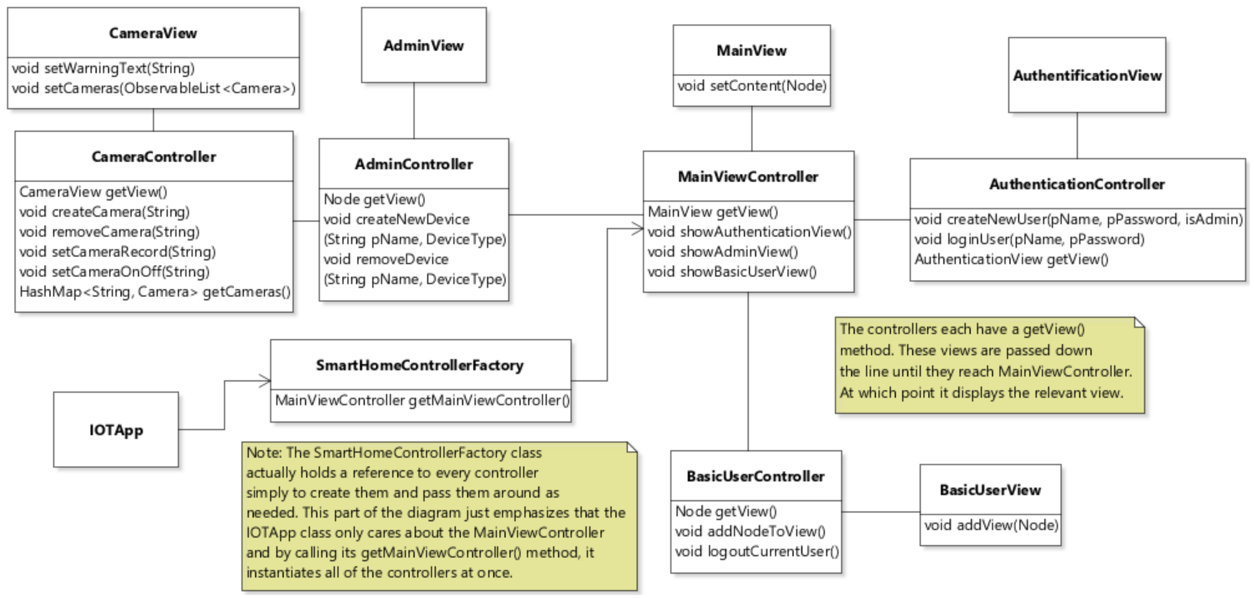
Figure 1. The class diagram of assignment 3 excluding the SmartHomeControllerFactory and IOTApp classes as well as other extraneous details (e.g. exceptions thrown). Models, controllers, and views are laid out from left to right.

Note that the SmartHomeControllerFactory.java class is not included in the diagram in Figure 1 as it holds a reference to every controller as well as the current user class. It essentially consists of a series of "getSomeController()" methods and ensures that each time the application is run, the application uses one of each controller and everything needing a reference to a controller gets the correct reference.

It may be observed that there are two levels of controllers. The user controller and device controllers are more directly retrieving and manipulating data from the models, they store maps between device/user names and instances (where registered users and devices are kept) and return deep copies of these maps for information verification by the other controllers. These are like the lower-level controllers whereas, the other controllers (admin, basic user, authentication, and main view) interact with the views more directly, taking user input and relaying it to the lower-level controllers.

Higher Level Analysis

Now let us turn our attention towards the view section. To get a closer look, we provide another class diagram involving the high-level controllers, the views, the factory class and the main app class.



CameraView
void setWarningText(String)
void setCameras(ObservableList<Camera>)
AdminView
MainView
void setContent(Node)
AuthentificationView
CameraController
CameraView getView()
void createCamera(String)
void removeCamera(String)
void setCameraRecord(String)
void setCameraOnOff(String)
HashMap<String, Camera> getCameras()
AdminController
Node getView()
void createNewDevice (String pName, DeviceType)
void removeDevice (String pName, DeviceType)
MainViewController
MainView getView()
void showAuthenticationView()
void showAdminView()
void showBasicUserView()
AuthenticationController
void createNewUser(pName, pPassword, isAdmin)
void loginUser(pName, pPassword)
AuthenticationView getView()
The controllers each have a getView() method. These views are passed down the line until they reach MainViewController. At which point it displays the relevant view.
SmartHomeControllerFactory
MainViewController getMainViewController()
IOTApp
Note: The SmartHomeControllerFactory class actually holds a reference to every controller simply to create them and pass them around as needed. This part of the diagram just emphasizes that the IOTApp class only cares about the MainViewController and by calling its getMainViewController() method, it instantiates all of the controllers at once.
BasicUserController
Node getView()
void addNodeToView()
void logoutCurrentUser()
BasicUserView
void addView(Node)

Figure 2. A higher-level look at assignment 3. Notice how views are pulled in by the main view controller, from which the IOTApp pulls the current view we want displayed.

As seen in Figure 2, MainViewController.java essentially exists only to display the correct view and give it to the IOTApp class (which gains its reference to it through the factory). This means there is really no user interface belonging to the MainView class (aside from the header and tagline), it acts as a channel for the other views. For example, upon start up the IOTApp class calls showAuthenticationView() in MainViewController which calls the AuthenticationController's getView() method and transfers it to the MainView class using its setContent() method (note that a view is-a Node). Then all the IOTApp class needs to do is call getView() from MainViewController to display the correct view.

Notice the addView() method in BasicUserView.java, this takes a Node as a parameter and is called exclusively by the SmartHomeControllerFactory class. Upon creation, the factory uses its references to the device controllers to give the BasicUserView a reference to each of the device views. In the addView() method BasicUserView adds the references to its children, hence why the basic user view consists of a scrolling pane of device views (with a logout button at the top as well).

Final Notes

We understand it may be beneficial to combine the device classes together employing the strategies of polymorphism, and we intend to do so in the future. However, this implementation was straightforward and guaranteed a manageable system that we could create incrementally (i.e. with our slice method).

It is worth noting that only the most important methods are mentioned in the diagrams (important as in, best demonstrate the design), but there are many methods behind the scenes so to speak, including some that may seem like they are not being used (for example, the createNewUser() method in AdminController). This is simply because we plan to use them in the further development of the project.