Service systems implementation guideline

Version 1.0

Patcharee Thongtra

Telematics Architecture for Play-based Adaptable System (TAPAS) Research Project

Telematics, NTNU

Table of Contents

1. Introduction

This document explains the implementation of service systems; also called as *play*, defined by the TAPAS architecture concepts. This document is supposed to be used by the service system developer.

2. Service system implementation

In this section, we explain the implementation of the service systems step-by-step from the initialization until the service systems are ready for the deployment. After the explanation, an example is also given. We suggest the developer to use an open source Java development framework for the implementation, for example Eclipse.

1. New a Java project for a play.
2. Import a library file, *TapasLib.jar*.
3. Create a folder named *plays* under source file folder of the Java project.
4. Create a *play-description* text file named *role.properties* under the plays folder.
5. Define roles and their manuscripts, which are represented by Java classes, in the play-description text file. Note that the manuscript is based on extended finite state machine. A pair of role and Java class for the manuscript is written with the format *{role_name}={java_class_name}*.
6. New Java classes for the manuscripts. These class names are mentioned in the play-description text file (Step 5).
7. The class in Step 5. needs to extend the abstract class *tapas.Manuscript*.
8. The class in Step 5. needs to implement abstract methods of the abstract class tapas.Manuscript as follows:

    a. constructor()

This is the explicit constructor method. This constructor must call the constructor of the super class tapas.Manuscript.

    b. init()

This is the initial method, which is to implement some tasks if needed before the role starts its behaviors, for example check file status and open file. This method is called when the role is assigned to an actor.

    c. terminate()

This is the termination method, which is to complete some existing tasks and/or to release allocated resources if needed before the role figure is terminated, for example close file or close connections. This method is called when an actor gets a message to stop playing its assigned role.

    d. doAction(int current_state, GenericEvent an_event)

This is action method. The actions depend on current state (current_state), an incoming event (an_event) and current local variables. An event can be a message or an input from the human user. This method is triggered after the role figure gets an event.

e. stateTransition(int current_state, GenericEvent an_event)

This is state transition method. The transitions depend on current state (current_state), an incoming event (an_event) and current local variables. An event can be a message or an input from the human user. This method is triggered after the role figure gets an event.

9. Export the Java project as a Jar file. Now the play, in the Jar file, is ready for the deployment.

2.1 An example of service system implementation

This is a SNMP-based monitoring system example. We name this system by short as *Monitor*. The system is collecting the management information from pre-compiled SNMP agents. The system contains two roles; Main monitoring manager (MMM) and Intermediate monitoring manager (IMM), which they handle part of the *service and capability monitoring functionality*. Here only MMM are explained.

MMM gets monitoring requests from the users. MMM starts deploying and instantiating an IMM in an available node for each request. Then, it distributes a request to the instantiated IMM that are federated managers on behalf of MMM. (IMMs regularly query the management information; specified by monitored variables, from ordinary SNMP agents by using SNMP protocol.) MMM gets the value of the monitored variables from IMMs every interval.

Moreover, MMM can get a request from the users, which the request is to get an available node where MMM can be moved to. MMM can also get a request, which is to move MMM to the available node. After MMM is moved to the new node, MMM automatically connects with existing IMMs. A state diagram illustrating the behaviors of MMM is given below.

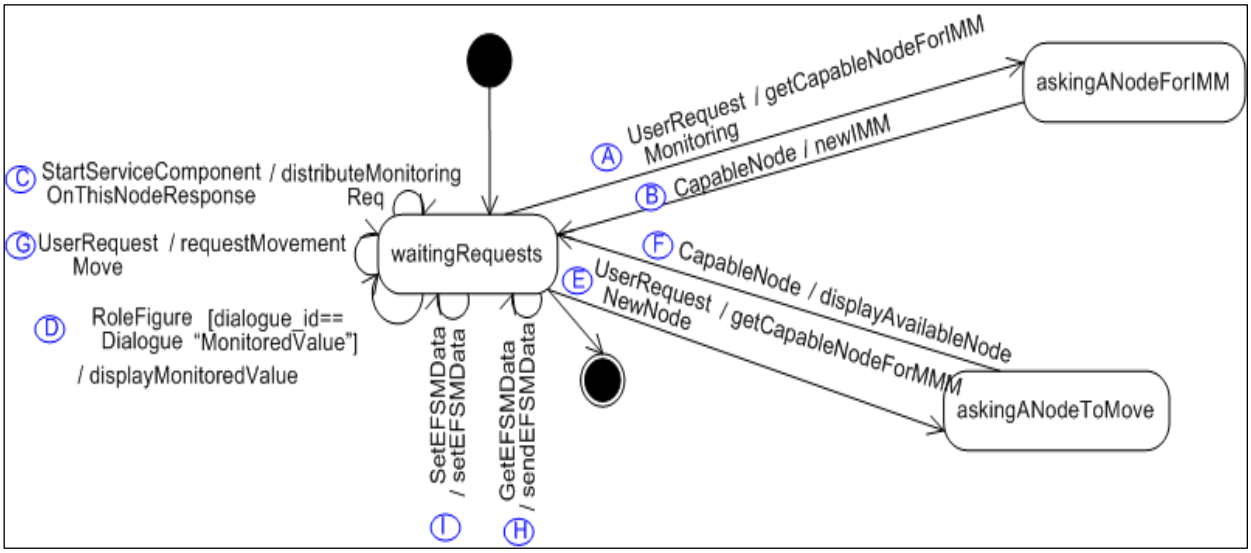


Figure 1: State diagram illustrates the behaviors of MMM.

From the state diagram, MMM has three states; *waitingRequests*, *askingANodeForIMM* and *askingANodeToMove*.

A. When MMM is at the state *waitingRequests* and there is the event *UserRequestMonitoring* happened; the event is generated because the users click a button in the GUI of MMM to request monitoring, the action *getCapableNodeForIMM* will be executed and MMM will move to the state

askingANodeForIMM. The action getCapableNodeForIMM is to get a capable node where IMM will be deployed and instantiated.

B. When MMM is at the state *askingANodeForIMM* and MMM gets the message *CapableNode*; informing about a capable node's IP address, the action *newIMM* will be executed and MMM will move to the state *waitingRequests*. The action newIMM is to send the message *StartServiceComponentOnThisNodeRequest* to Actor Handler on the capable node. Consequently, the Actor Handler creates a new actor or finds an available actor, and assigns the actor the IMM role.

C. When MMM is at the state *waitingRequests* and MMM gets the message *StartServiceComponentOnThisNodeResponse*; informing about the actor id of an actor constituting IMM, the action *distributeMonitoringReq* will be executed. The action distributeMonitoringReq is to distribute the monitoring request to the actor (the instantiated IMM).

D. When MMM is at the state *waitingRequests* and MMM gets the message *RoleFigureDialogue* with the *dialogue_id = MonitoredValue*; informing about the values of the monitored variables, the action *displayMonitoredValue* will be executed. The action displayMonitoredValue is to display those values on the GUI of MMM.

E. When MMM is at the state *waitingRequests* and there is the event *UserRequestNewNode* happened; the event is generated because the users click a button in the GUI of MMM to request for a capable node for MMM, the action *getCapableNodeForMMM* will be executed and MMM will move to the state *askingANodeToMove*. The action getCapableNodeForMMM is to get a capable node where MMM will be deployed and instantiated.

F. When MMM is at the state *askingANodeToMove* and MMM gets the message *CapableNode*; informing about a capable node's IP address, the action *displayAvailableNode* will be executed and MMM will move to the state *waitingRequests*. The action displayAvailableNode is to display the capable node's address on the GUI of MMM.

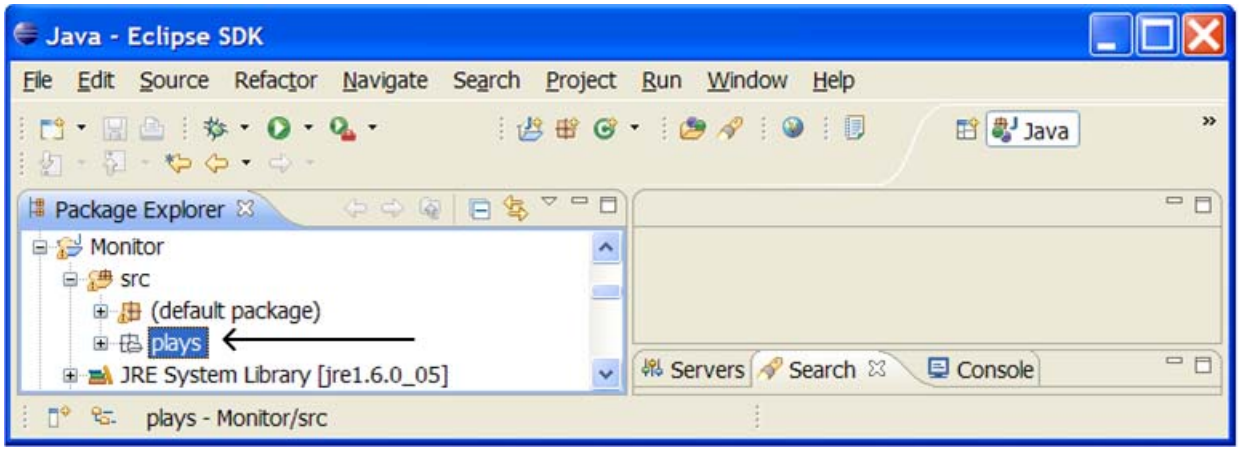
G. When MMM is at the state *waitingRequests* and there is the event *UserRequestMove* happened; the event is generated because the users click a button in the GUI of MMM to move MMM, the action *requestMovement* will be executed. The action requestMovement is to send the message RoleFigureMoveRequest to the Mobility Manager.

H. When MMM is at the state *waitingRequests* and MMM gets the message *GetEFSMData*; getting MMM's EFSM data, the action *sendEFSMData* will be executed. The action sendEFSMData is to send MMM's EFSM data to the Mobility Manager.

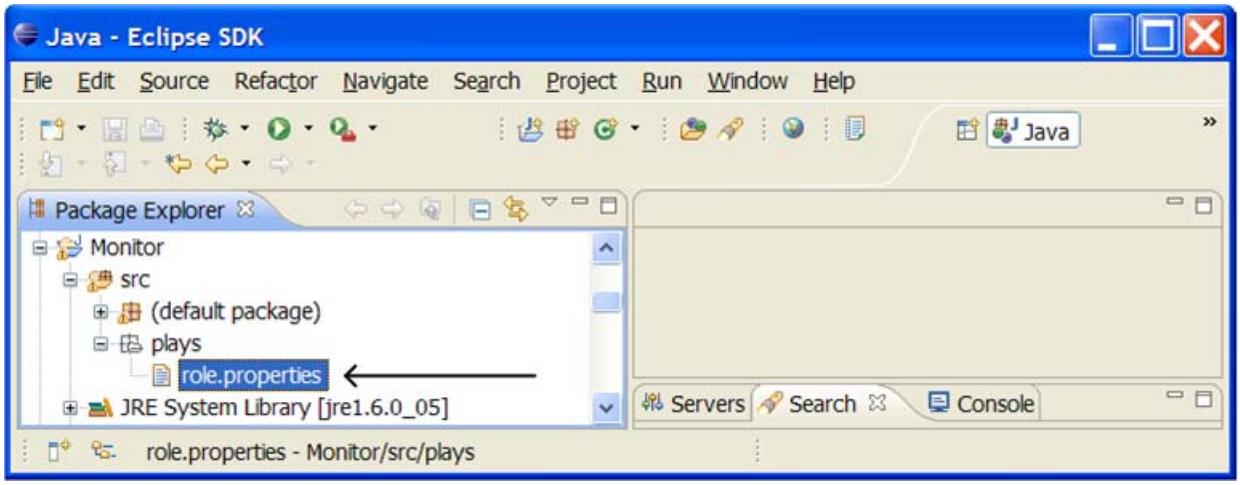
I. When MMM is at the state *waitingRequests* and MMM gets the message *SetEFSMData*; setting MMM's EFSM data, the action *setEFSMData* will be executed. The action setEFSMData is to set EFSM data of MMM itself after it has been moved. Consequently, MMM can connect with existing IMMs.

In the Java project of this system example, there are java classes which they represent the manuscripts of MMM and IMM. Below part of the Java code of the project and the MMM's manuscript, which correlate with Step 3 – 8 explained in Sec. 2, are presented.

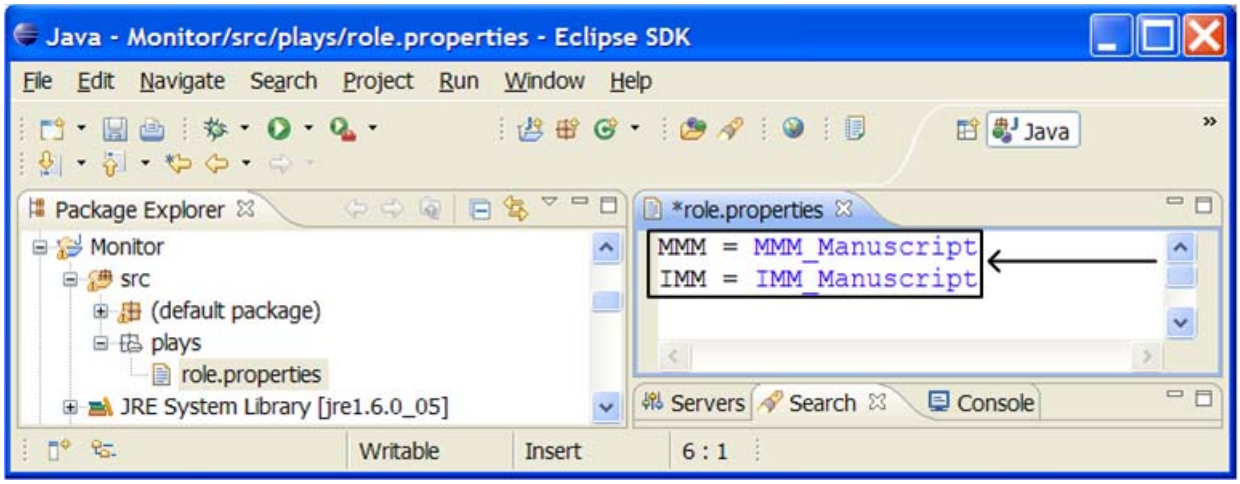
- Create the folder named plays under source file folder of the Java project (Step 3).



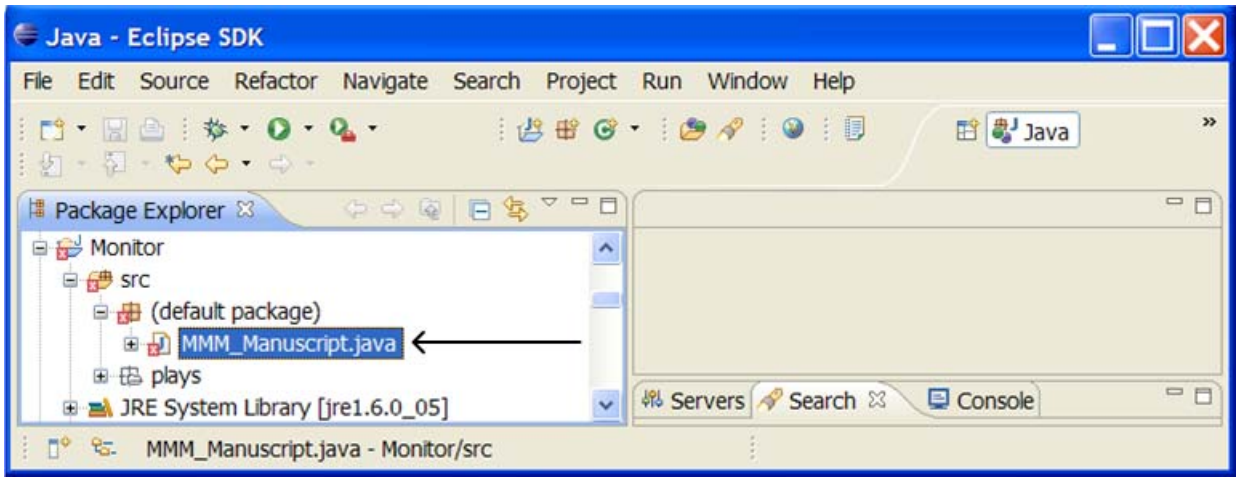
- Create the play-description text file under the plays folder (Step 4).



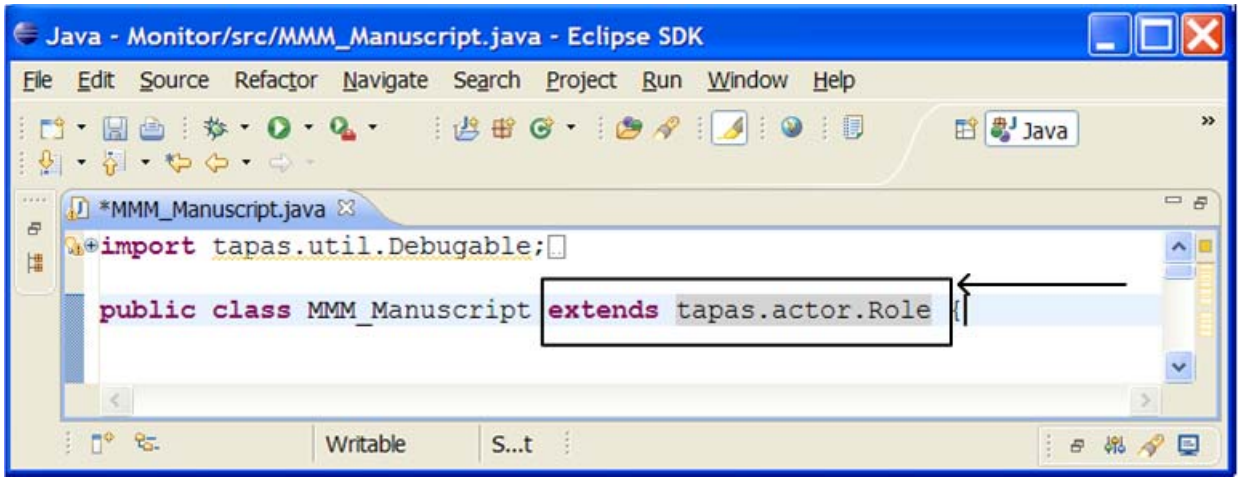
- Define roles and their manuscripts (Step 5). In the screen below, the role MMM and IMM are defined together with their manuscripts that are MMM_Manuscript.java and IMM_Manuscript.java respectively.



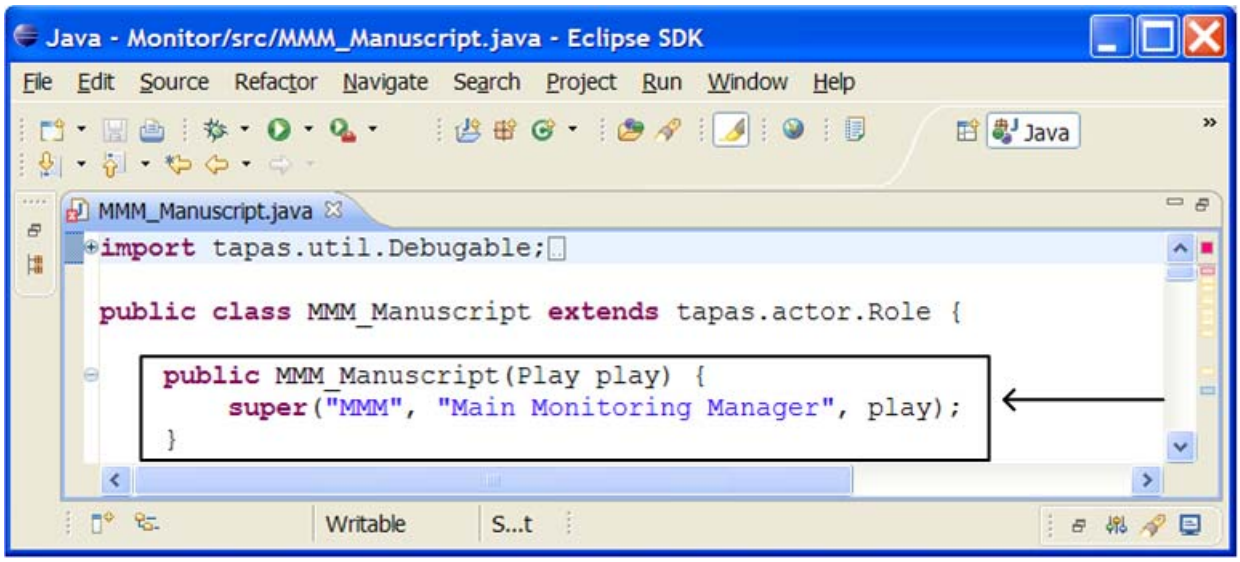
- New the MMM_Manuscript.java (Step 6).



- The MMM_Manuscript class extends the abstract class tapas.Manuscript (Step 7).

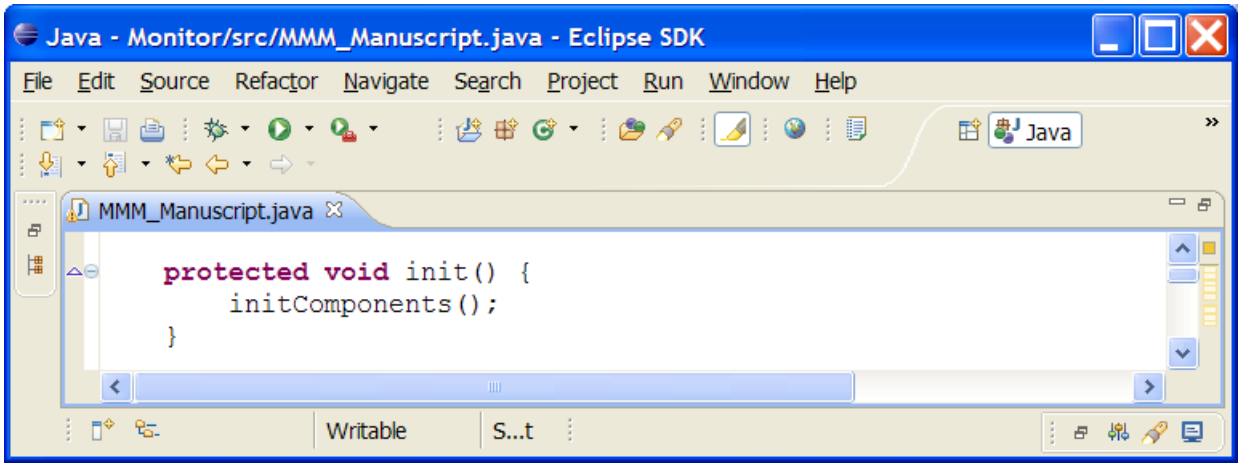


- The MMM_Manuscript class implements abstract methods of the abstract class tapas.Manuscript (Step 8).
    - constructor()



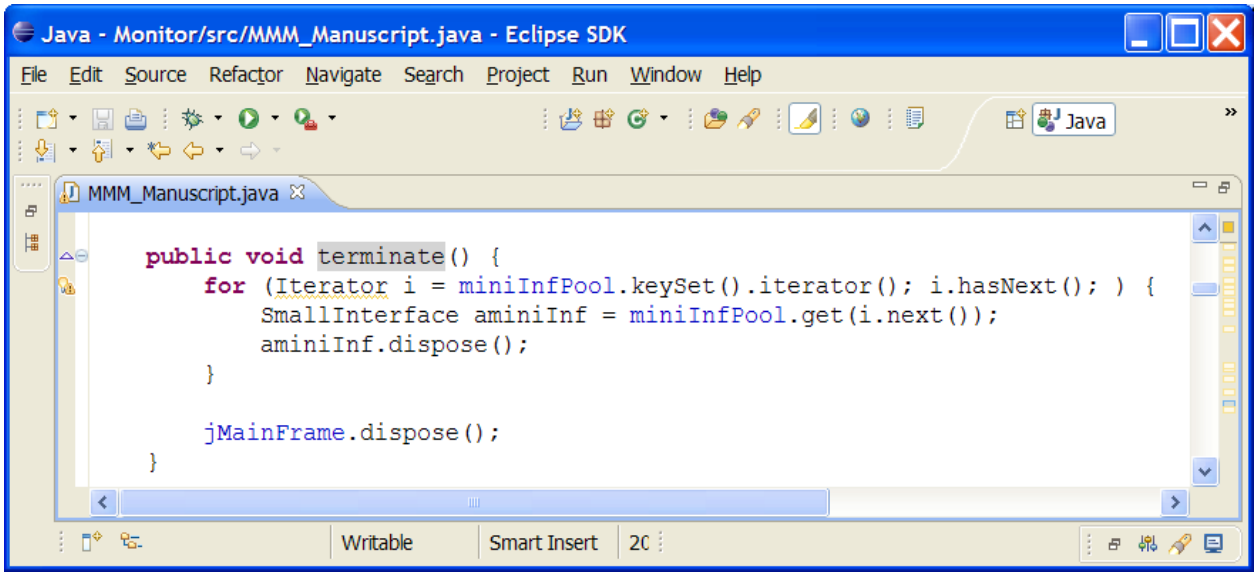
- init()

The initial method of MMM_Manuscript creates the GUI.



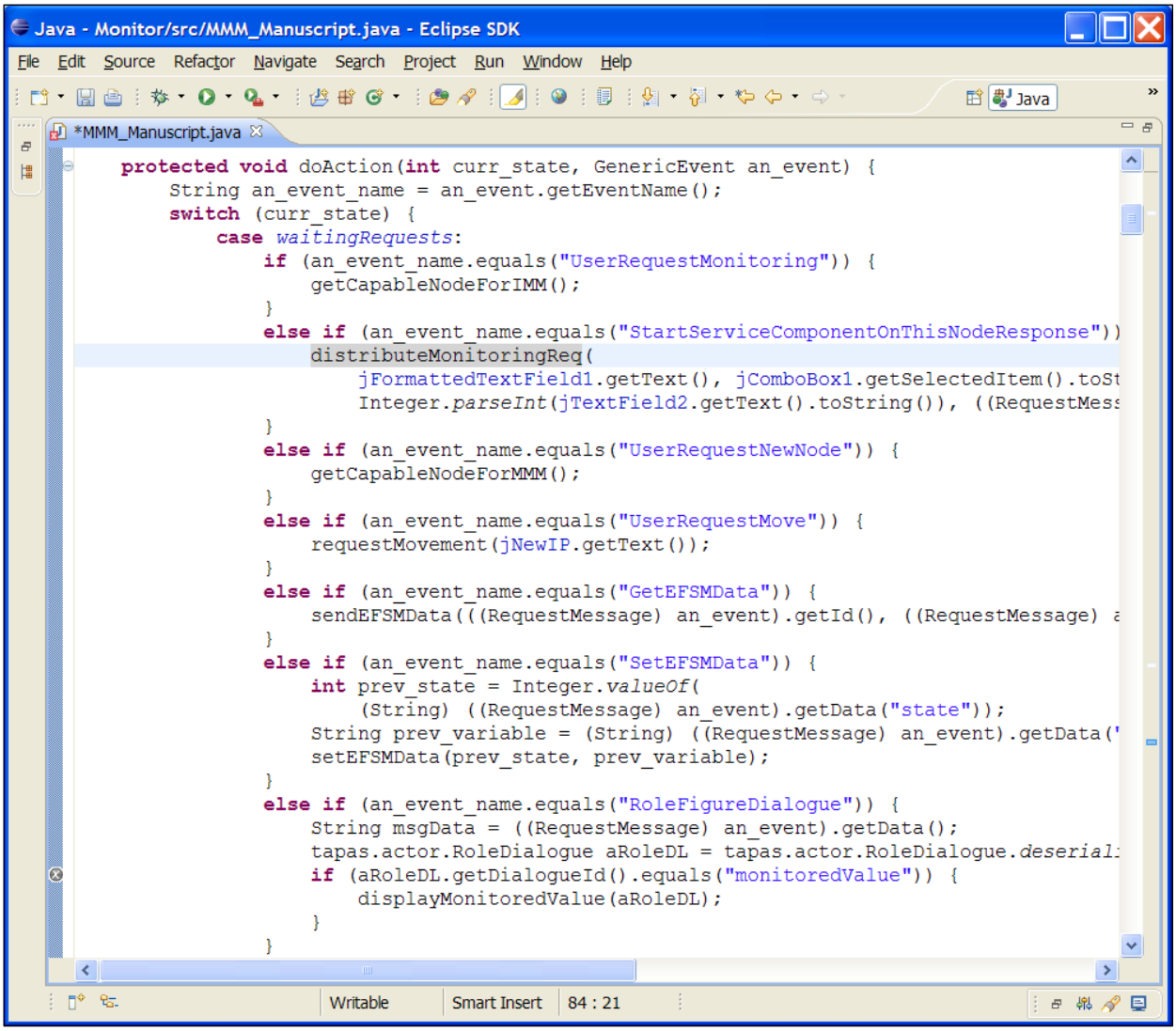
- terminate()

The termination method of MMM_Manuscript destroys the existing GUI.



- doAction(int current_state, GenericEvent an_event)

The action method of MMM_Manuscript is consistency with the state diagram in Fig. 1. For example, when the current state is equal to waitingRequests and the incoming event is equal to StartServiceComponentOnThisNodeResponse, the action distributeMonitoringReq will be executed.



stateTransition(int current_state, GenericEvent an_event)

The state transition method of MMM_Manuscript is consistency with the state diagram in Fig. 1. For example, when the current state is equal to waitingRequests and the incoming event is equal to UserRequestMonitoring, MMM will move to the state AskingANodeForIMM.