



Dipartimento di Elettronica, Informazione e Bioingegneria

Politecnico di Milano

Prof. Matteo Camilli, Elisabetta Di Nitto, and Matteo Rossi

20133 Milano (Italia)

Piazza Leonardo da Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Software Engineering 2 – Written Exam 2 (WE2)

February 5th, 2024

Last Name, First Name

Id number (Matricola)

Number of paper sheets you are submitting as part of the exam

Notes

- A. You must write your name and student ID (matricola) on each piece of paper you hand in.
- B. You may use a pencil.
- C. Incomprehensible handwriting is equivalent to not providing an answer.
- D. The use of any electronic apparatus (computer, cell phone, camera, etc.) is strictly forbidden, except for an ebook reader.
- E. The exam is composed of 2 parts, one focusing on requirements, and one focusing on design. Read carefully all points in the text.
- F. Total available time for WE2: 1h and 30 mins**

System Description: ContainerOrchestrator (CO)

Consider a pre-existing container management system (CMS) managing *software containers* (e.g., Docker containers). CMS can be available in multiple instances, one per *computing node*. Each CMS instance provides operations to create, execute, stop, and destroy containers on the corresponding computing node. Moreover, it is able to return a list of the containers running on the node at a given time. Containers are created from *images*, which are available on a catalogue known by each CMS instance. CMS instances are able to download and upload these images from the catalogue.

Each computing node has a *monitoring agent* that gathers information regarding the status of the node (whether it is functioning or not, whether a CMS instance is running on it, etc.) and that can, in case, reboot the whole node restarting also the CMS.

We are tasked to develop a new system, **ContainerOrchestrator (CO)**, that interfaces users with the CMS instances for the purpose of controlling the creation and management of containers on multiple nodes. Users of CO can write scripts that define plans of operations to be executed (e.g., start container x on node Y , stop container k on node J , migrate container w from node Y to node L if the CPU utilization on Y is over threshold h , create a new container from image I). CO can read and execute these scripts by sending the corresponding commands to the CMS instances. In addition, while a script is being executed, CO can exploit the monitoring agents to collect the status of each node – for instance, to assess the CPU utilization of a node or to check that a node is alive. Based on the information provided by monitoring agents, CO can perform the actions defined in the currently active script. For example, as soon as it sees that the CPU utilization of node Y is over the threshold h , CO can start the migration procedure by requesting the CMS instance of node L to run container w and then by requesting the CMS instance of node Y to stop w . Additionally, CO can ask the monitoring agent to reboot a CMS instance if needed.

Part 1 Requirements (7 points)

RASD_Q1 (3 points)

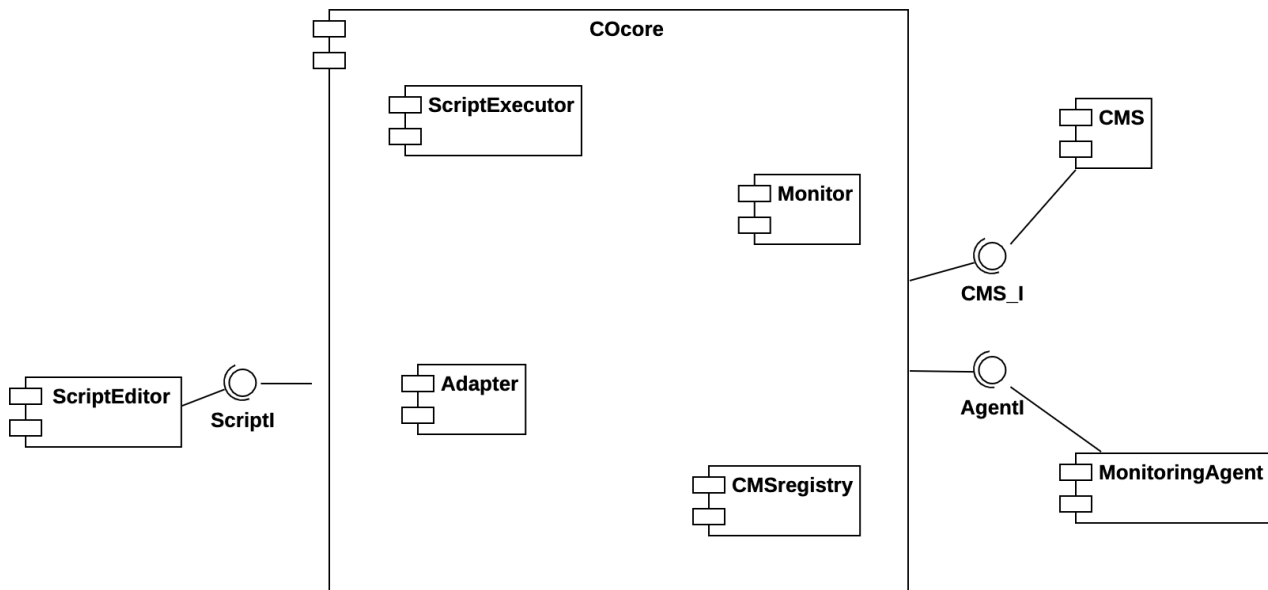
Use a UML Use Case diagram to identify the actors relevant for the CO system and the corresponding use cases. Provide a description of the actors and of the purposes of each use case.

RASD_Q2 (4 points)

Use a UML Class Diagram to describe the domain of the CO system.

Part 2 Design (7 points)

Consider the following UML Component Diagram.



The diagram depicts various components related to the CO system. Component *ScriptEditor* allows users to create scripts and send them for execution on the *COcore*. Component *COcore* is, as the name suggests, the core of the CO system. *COcore* has four subcomponents:

- *ScriptExecutor* is in charge of executing scripts created by users. In doing so, it sends the appropriate commands to the CMS instances and acquires monitoring information.
- *Adapter* is the component in charge of rebooting nodes, stopping and restarting CMS instances, requesting to CMS instances the restart of containers.
- *Monitor* checks the current situation of the nodes and provides this information to the components that need it.
- *CMSRegistry* is the component in charge of keeping track of the endpoints of each running CMS instance.

Components *CMS* and *MonitoringAgent* represent the elements mentioned in the system description above. As stated above, they can be available in multiple instances.

DD_Q1 (2 points)

Complete the UML Component Diagram with the necessary connections and interfaces between the depicted components.

DD_Q2 (3 points)

For each interface identified in the UML Component Diagram (i.e., *ScriptI*, *CMS_I* and *AgentI* and those defined in DD_Q1) define the operations they provide (you can use a UML Class Diagram, if useful, to define suitable types for the operation parameters) and provide a short description for each of them.

DD_Q3 (2 points)

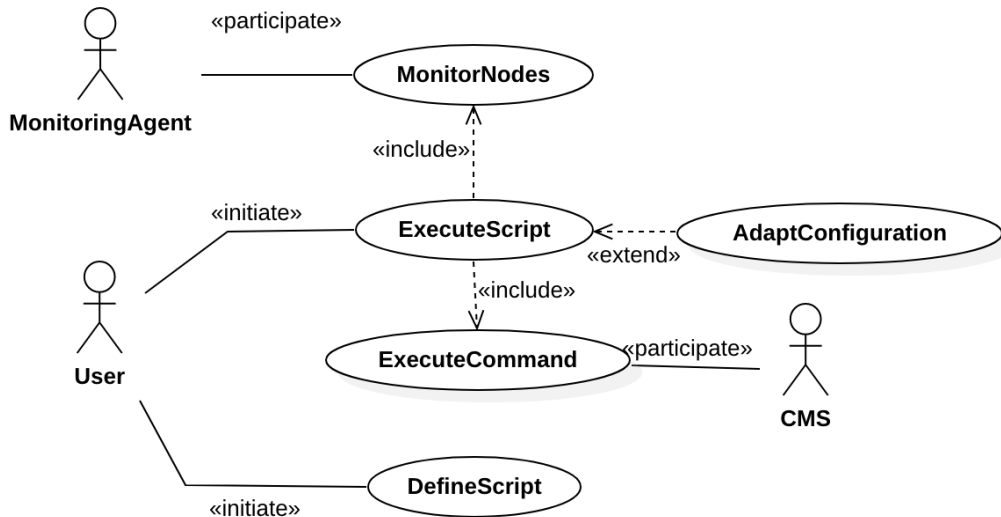
Define a UML Sequence Diagram that shows the interaction between the components of the architecture when the following scenario occurs: a new command requesting the creation of a container is read from the script and is executed. Then, a node fails and the system restarts it.

Solutions

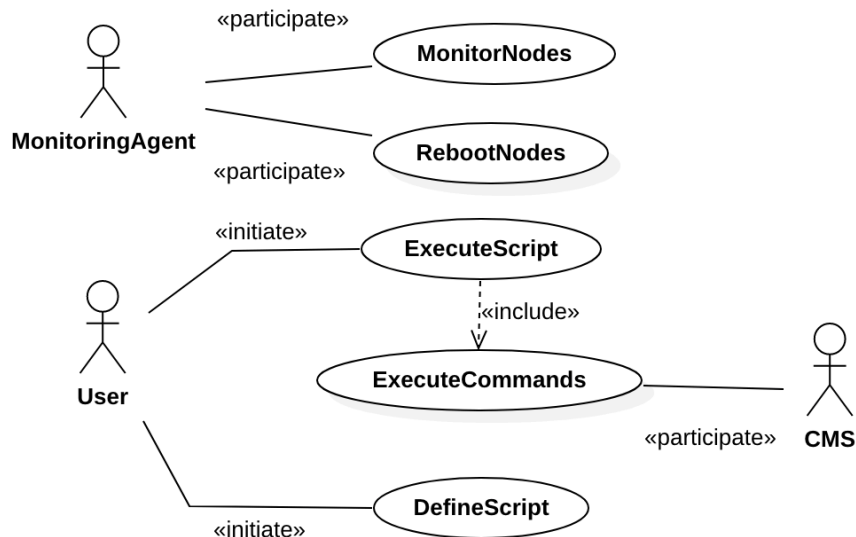
RASD_Q1 (3 points)

The following UC diagrams show different approaches to the monitoring of CMSs and their respective nodes.

In the first approach, shown in the UC diagram below, monitoring is carried out during the execution of the scripts, in a “synchronous” manner (i.e., as part of the script execution).



In the second approach, instead, monitoring occurs in an “asynchronous” manner; that is, it is performed independent of the scripts; when anomalies are detected, corrective reboot actions are taken.



Actors:

- *MonitoringAgent*: it collects the status of each node and provides CO with this information. Based on this input, CO can perform the actions defined in the current script (e.g., migration procedures).
- *User*: a User actor writes scripts that define plans of operations (in the form of scripts) and then asks the system to execute such plans.
- *CMS*: a CMS instance provides operations to create, execute, stop, and destroy containers on the corresponding computing node.

Use cases for the first solution:

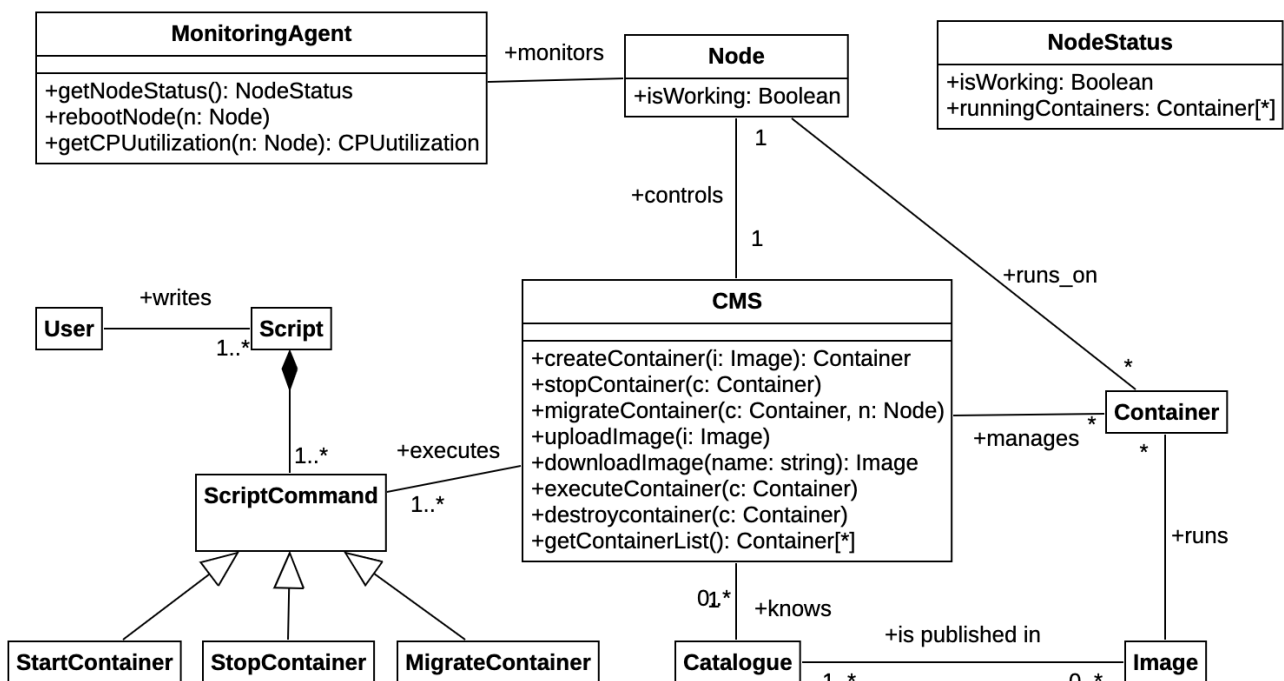
- *DefineScript*: initiated by a user, who interacts with CO to define the plan of operations to be executed.
- *ExecuteScript*: initiated by a user, who asks the system to execute the plan of operations previously defined. The execution of the plan includes the monitoring of the status of the nodes; this information is used to decide about corrective actions, if they are necessary.
- *ExecuteCommands*: is included in *ExecuteScripts*. Its main purpose is to send the commands defined in the script to the CMS instances for execution.
- *AdaptConfiguration*: this use case is an extension of the *ExecuteScript* UC, in that it includes actions that, based on the information gathered from the *MonitoringAgents*, modify the configuration of the nodes (i.e., what containers are run on them) when necessary.
- *MonitorNodes*: this use case describes the collection of data regarding the operational status and performance metrics of nodes, which occurs while a script is executing.

Use cases for the second solution:

- *DefineScript*: same as the first solution.
- *ExecuteScript*: initiated by a user, who asks the system to execute the plan of operations previously defined.
- *ExecuteCommands*: same as the first solution.
- *MonitorNodes*: this use case describes systematic collection of data regarding the operational status and performance metrics of nodes, which occurs independently from the execution of scripts.
- *RebootNode*: this use case is triggered by the CO that asks the monitoring agent to reboot a CMS instance if needed based on the collected operational status of the corresponding node.

Note that, based on the request, the use case diagram must be defined from the perspective of the CO system. In this sense, those activities that involve the other actors but not CO are not part of this diagram.

RASD_Q2 (4 points)



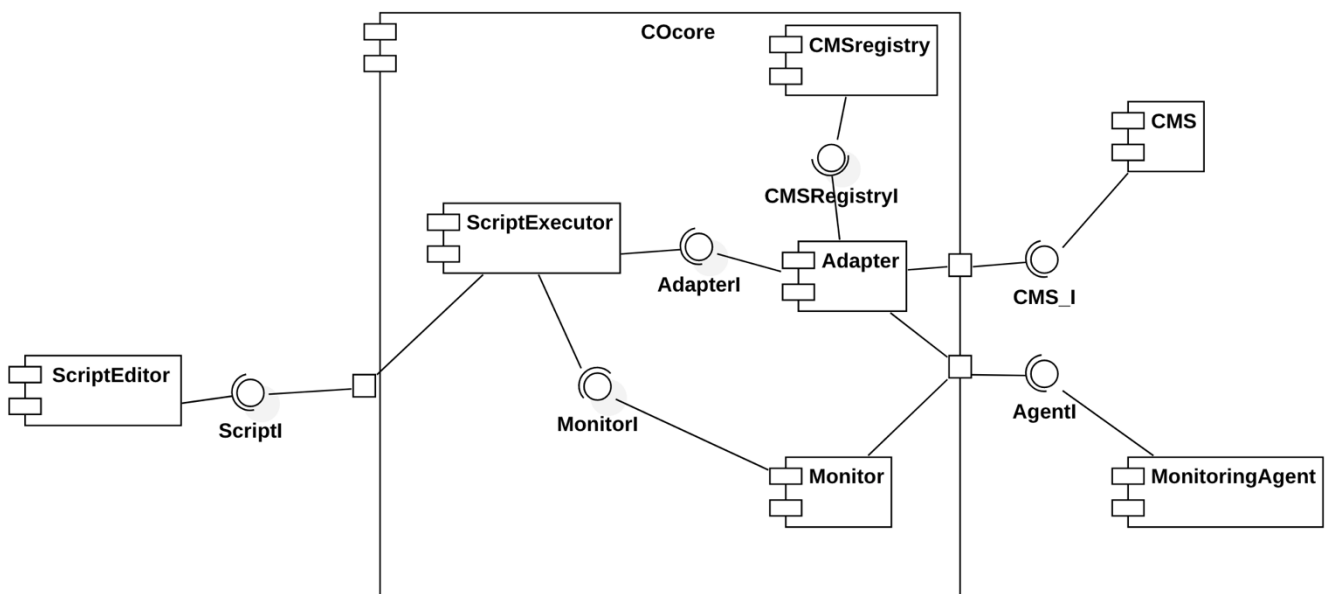
DD_Q1 (2 points)

The key points of the architecture are the following:

- According to the text, *ScriptExecutor* sends appropriate commands to CMS instances and acquires monitoring information. To limit the number of *COcore* components using the external interfaces, the interaction with CMSs occurs always through the *Adapter* component and the interaction with monitoring agents to acquire monitoring information occurs always through *Monitor*.
- The *Monitor* component knows about *MonitoringAgent* endpoints. A different solution could consist in relying on the *CMSregistry* to this purpose. In this case, *Monitor* would use the interface offered by *CMSregistry* to retrieve the reference to the *MonitoringAgents*.

As above, two solutions are proposed, which differ in the way they handle the interaction between the *ScriptExecutor* and the *Monitor* component (i.e., whether the monitoring occurs “synchronously” or “asynchronously” with respect to the execution of the scripts).

Solution 1:

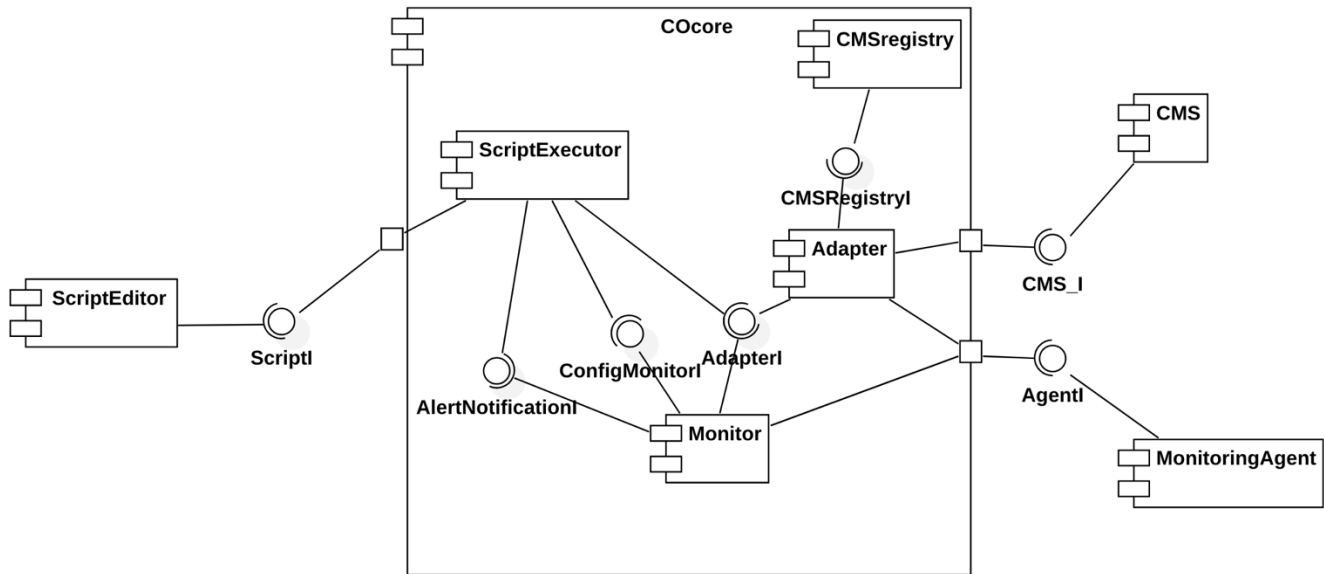


ScriptExecutor offers the *ScriptI* interface exposed by *COcore*. Through this interface, *ScriptExecutor* receives the script to be executed. While executing a script, *ScriptExecutor* interacts with *Adapter*, through the *AdapterI* interface, to which it passes the commands that are to be executed, either by some *MonitoringAgent*, or by a CMS. To interact with these last ones, *Adapter* uses their interfaces *CMS_I* and *AgentI*. Moreover, it relies on the interface offered by *CMSregistry* (*CMSRegistryI*) to acquire information about the endpoints of the installed CMSs/Agents. For simplicity, we assume that *CMSregistry* is initialized at the system startup.

If the currently active script includes conditional instructions that are executed if a certain condition to be monitored becomes true (e.g., CPU utilization about a certain threshold), then the *ScriptExecutor* will have to periodically contact *Monitor* to acquire the needed information and, possibly, execute the corresponding action. To this end, *Monitor* offers to *ScriptExecutor* the interface *MonitorI*, through which *ScriptExecutor* can retrieve monitoring information. Upon request, *Monitor* calls the *MonitoringAgents* through the *AgentI* interface and provides a response to *ScriptExecutor*.

As a result, in this solution, the *Monitor* component is a passive element that is used by the *ScriptExecutor* as an intermediary with the *MonitoringAgents*.

Solution 2:



In this case, the role of the *Monitor* component becomes more prominent and active. In fact, we want to avoid the need for *ScriptExecutor* to continuously poll *Monitor* to gather the information it is interested in. Moreover, we want to keep *Monitor* autonomous and able to control the status of CMSs independently of the commands received by *ScriptExecutor*, so to restart nodes/CMSs as needed by relying directly on the services offered by the *Adapter* and the *MonitoringAgents*.

To achieve this behavior, *Monitor* will be active by itself and will continuously poll *MonitoringAgents* and send requests to them or to the *Adapter* to restart/stop/... nodes or containers. Moreover, *Monitor* offers the *ConfigMonitorI* interface to allow the *ScriptExecutor* to configure specific monitoring activities. For instance, through this interface, the *ScriptExecutor* is able to set an alert in case the CPU utilization overcomes its threshold, passing to the *Monitor* also the endpoint of its provided interface *AlertNotificationI*, through which the *Monitor* can send notifications to the *ScriptExecutor*.

DD_Q2 (3 points)

CMS and *MonitoringAgent* are external actors, whose interfaces have been identified in the UML Class Diagram of question RASD_2.

Interface *CMS_I* offers the following operations (from the UML class diagram):

```
Container createContainer(Image i)
void stopContainer(Container c)
void migrateContainer(Container c, Node n)
void executeContainer(Container c)
void destroyContainer(Container c)
Container [] getContainerList()
```

Notice that the operations concerning the uploading and downloading of images are not used by *COcore*, so they are not part of this specific interface.

Interface *AgentI* offers the following operations (from UML class diagram):

```
NodeStatus getNodeStatus(Node n)
void rebootNode(Node n)
CPUUtilization getCPUUtilization(Node n)
```

Interface *ScriptI* offers the operation:

```
void executeScript (Script s)
```

that allows the script editor to execute a script.

Interface *AdapterI* offers the following operation:

```
void executeCommand (Command c).
```

Command is a type including the following fields: *commandName*, *parameterList*, *recipientGroup*. *commandName* can correspond to an enumeration including all possible commands accepted by either *CMS* or *MonitoringAgents*. *parameterList* will be a list of key/value pairs representing the name and corresponding value of each of the parameters needed to run the specific command. Finally, we can make the hypothesis that *recipientGroup* can take as value a specific node, or *Any* in case the command can be executed on any *CMS* or *MonitoringAgent* chosen by the *Adapter*, or *Broadcast* in case the command is sent to all *CMS* or *MonitoringAgents* controlled by the CO.

Interface *CMSRegistryI* offers the following operations:

```
EndPoint getCMSEndPoint (Node n)
```

```
EndPoint getAgentEndPoint (Node n)
```

```
void addNode (Node n)
```

```
void addCMS (CMS_EndPoint c, Node n)
```

```
void addAgent (Agent_EndPoint a, Node n)
```

Given a node controlled by CO, *getCMSEndPoint* and *getAgentEndPoint* return the endpoints of the CMS running on it or of the *MonitoringAgent* monitoring it, respectively. The *add* operations instead are executed when a new *Node*, *CMS* or *MonitoringAgent* is inserted in the system as a result of the execution of a script command.

Interface *MonitorI* (interface defined in solution 1)

```
NodeStatus getNodeStatus (Node n)
```

```
void rebootNode (Node n)
```

Interface *ConfigMonitorI* (interface defined in solution 2)

```
void setMonitoringCondition (Condition c, AlertNotificationI handler)
```

Condition is a type including the following elements:

- *Metric*: the metric to be monitored (e.g., CPU utilization or memory utilization)
- *Threshold*: a threshold value
- *Operator*: this can be *lower*, *lowerEqual*, *greater*, *greaterEqual*

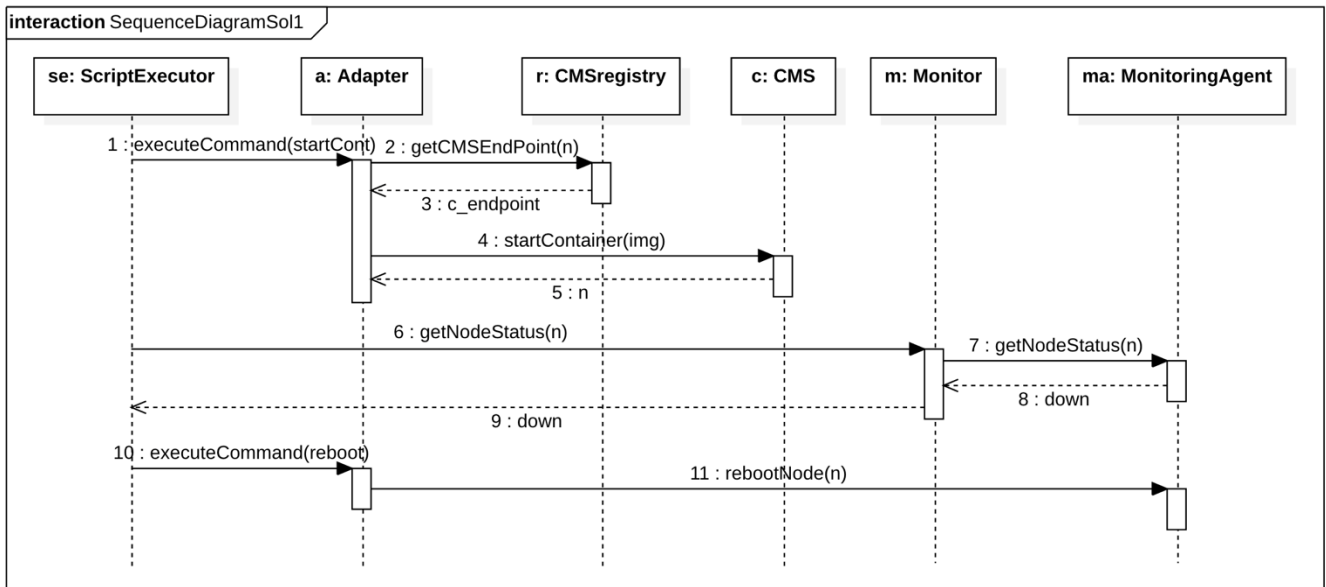
Interface *AlertNotificationI* (interface defined in solution 2) allows the *ScriptExecutor* to be informed by the *Monitor* about relevant changes in some monitored metrics. It offers the following operation:

```
void alert (Condition c)
```

where *c* is the condition that, according to the collected monitoring information, does not hold any more. This operation results in the execution of a thread internally to the *ScriptExecutor* that may trigger the execution of an interaction with the *Adapter*.

DD_Q3 (2 points)

The following sequence diagram is associated with the architectural solution 1.



For brevity, the diagram shows only the request from *se* to *m* that returns the information that the node is down. During the execution of the system, *se* will periodically poll *m*, which, in turn, will poll all agents to know about the status of nodes.

The following sequence diagram is associated with the architectural solution 2. As mentioned above, in this case *Monitor* is more autonomous and takes care of the so-called repair process independently from the other parts of CO.

