

Table of Contents

1	AbstractLowLevel	3
1.1	abstract boolean executeTask(Task t)	3
1.2	abstract Status getStatus()	3
2	AbstractTaskExecutor	3
2.1	abstract boolean executeTask(Task t)	3
3	AbstractActionSequenceSelector	3
3.1	abstract ActionSequence getActionSequence(Task t).....	4
4	AbstractActionSequenceDataStore	4
4.1	abstract ActionSequence[] getActionSequence(Task t).....	4
5	AbstractHardwareInterface	4
5.1	abstract boolean executeAction(Action a)	4
5.2	void setInterruptHandler(InterruptHandler ih).....	4
5.3	void setActionHandler(ActionHandler ah)	4
6	ActionHandler	5
6.1	receiveAction(Action a)	5
7	InterruptHandler	5
7.1	receiveInterrupt(Interrupt i).....	5
8	LowLevel	5
8.1	LowLevel().....	6
8.2	executeTask(Task t)	6
8.3	Status getStatus().....	6
9	TaskExecutor.....	6
9.1	TaskExecutor()	7
9.2	boolean executeTask(Task t).....	7
9.3	void run().....	8
9.4	void suspend()	8
9.5	void resume().....	9
9.6	stopThread()	9
9.7	boolean isRunning()	9
9.8	void setHardwareInterface(AbstractHardwareInterface hi)	9
10	ActionSequenceSelector.....	9
10.1	ActionSequenceSelector().....	10
10.2	ActionSequence getActionSequence(Task t).....	10
10.3	ActionSequence[] retrieveActionSequences(Task t)	10
11	ActionSequenceDataStore.....	10
11.1	ActionSequence[] getActionSequence(Task t)	10
12	HardwareInterface	11
12.1	public HardwareInterface()	11
12.2	void start()	11
12.3	boolean executeAction(Action a).....	11
13	InterruptReader.....	12
13.1	InterruptReader(BufferedReader b, InterruptHandler ih)	12
13.2	void run().....	12
13.3	void stopThread()	13

13.4	boolean isRunning()	13
14	Action	13
14.1	public String toString()	13
15	Task	14
16	ActionSequence	14
16.1	ActionSequence(Action action[])	14
16.2	Action peek()	14
16.3	void pop()	14
17	Feedback	15
17.1	boolean isError()	15
18	Status	15
19	StateMonitor	15
19.1	public StateMonitor()	16
19.2	Status getStatus()	16
19.3	boolean compare()	16
19.4	public void run()	16
19.5	Action getTreatment(Feedback f)	16
19.6	void actionSent(Action a)	17
20	IdealStateGenerator	17
20.1	IdealStateGenerator(Status idealState)	17
20.2	receiveAction(Action a)	17
21	ActualStateGenerator	18
21.1	ActualStateGenerator(Status actualState)	18
21.2	void updateState()	18
21.3	queryHardware()	18
22	Diagnosis	18
22.1	Diagnosis()	19
22.2	ActionSequence getTreatment(Error e)	19
23	FaultsandTreatmentsDataStore	19
23.1	FaultsandTreatmentsDataStore()	19
23.2	ActionSequence getTreatment(Error e)	20

1 AbstractLowLevel

This class defines how the low level system will operate. Any low level system will need to be able to execute a task and return status information about the overall system.

abstract class AbstractLowLevel
Methods +abstract boolean executeTask(Task t); +abstract Status getStatus();
Attributes <i>None</i>

1.1 abstract boolean executeTask(Task t)

This method takes a task given to it by the high level and translates it into actions that are sent to the hardware interface.

1.2 abstract Status getStatus()

This method sends status information back to high level.

2 AbstractTaskExecutor

The TaskExecutor is responsible for translating a given task into actions that are then sent to the hardware interface.

abstract class AbstractTaskExecutor
Methods +abstract boolean executeTask(Task t);
Attributes <i>None</i>

2.1 abstract boolean executeTask(Task t)

This method takes a give task and breaks it up into actions sequences which are then sent to the hardware interface.

3 AbstractActionSequenceSelector

The ActionSequenceSelector selects action sequences for a given task. This class provides a point where ActionSequences can be intelligently selected.

abstract class AbstractActionSequenceSelector
Methods +abstract ActionSequence getActionSequence(Task t);
Attributes <i>None</i>

3.1 *abstract ActionSequence* **getActionSequence(Task t)**

This method returns an action sequence that accomplishes the given task.

4 **AbstractActionSequenceDataStore**

The ActionSequenceDataStore interfaces with the action sequence data store.

abstract class AbstractActionSequenceDataStore
Methods +abstract ActionSequence[] getActionSequences(Task t);
Attributes None

4.1 *abstract ActionSequence[]* **getActionSequence(Task t)**

This method queries the data store for every action sequence that is related to the given task.

5 **AbstractHardwareInterface**

The HardwareInterface provides interface for communication between the low level system and the simulated hardware.

abstract class AbstractHardwareInterface
Methods +abstract boolean executeAction(Action a); +void setInterruptHandler(InterruptHandler ih); +void setActionHandler(ActionHandler ah);
Attributes #InterruptHandler interruptHandler; //place to send interrupt messages #ActionHandler actionHandler; //place to send actions

5.1 *abstract boolean* **executeAction(Action a)**

This method executes a given action by sending it to the simulation.

5.2 *void* **setInterruptHandler(InterruptHandler ih)**

This method sets the InterruptHandler for the HardwareInterface.

Pseudo Code:

```
public void setInterruptHandler(InterruptHandler ih)
{
    interruptHandler = ih;
}
```

5.3 *void* **setActionHandler(ActionHandler ah)**

This method sets the ActionHandler for the HardwareInterface.

Pseudo Code:

```
public void setActionHandler(ActionHandler ah)
```

```
{
    actionHandler = ah;
}
```

6 ActionHandler

This interface allows a class to obtain action messages that are received by the hardware interface.

public interface ActionHandler
Methods +void receiveAction(Action a);
Attributes <i>None</i>

6.1 receiveAction(Action a)

This method is called when an action is sent down to the hardware interface. The action that was sent is stored in a.

7 InterruptHandler

This interface allows a class to obtain interrupt messages that received by the hardware interface.

public interface InterruptHandler
Methods +void receiveInterrupt(Interrupt i);
Attributes <i>None</i>

7.1 receiveInterrupt(Interrupt i)

This method is called when an interrupt is received by the hardware interface. The interrupt that was received is stored in i.

8 LowLevel

The low level provides the high level with an interface to execute tasks and get status information about the overall system. This low level implementation has three main components, a TaskExecutor, StateMonitor, and a HardwareInterface. The TaskExecutor is responsible for executing the tasks sent to the low level. The StateMonitor is responsible for monitoring the state of the spacecraft and reporting status information about the overall system.

public class Lowlevel extends AbstractLowLevel
Methods +LowLevel(); +boolean executeTask(Task t); +Status getStatus();
Attributes

```
-TaskExecutor taskExecutor; //responsible for translating a task into actions and issuing
    //the actions to the hardware
-StateMonitor stateMonitor; //responsible for monitoring the status of the hardware
    system //and detecting any errors that occur
-HardwareInterface hardwareInterface; //provides communication between the low level
    //and the simulation
```

8.1 LowLevel()

This method initializes a new low level object.

Pseudo Code:

```
public LowLevel()
{
    hardwareInterface = new HardwareInterface();

    taskExecutor = new TaskExecutor();
    taskExecutor.setHardwareInterface( hardwareInterface );

    stateMonitor = new StateMonitor();
}
```

8.2 executeTask(Task t)

This method takes a task given to it by the high level and translates it into actions that are sent to the hardware interface.

Pseudo Code:

```
public boolean executeTask(Task t)
{
    return taskExecutor.executeTask( t );
}
```

8.3 Status getStatus()

This method sends status information back to high level.

Pseudo Code:

```
public Status getStatus()
{
    return stateMonitor.getStatus();
}
```

9 TaskExecutor

The TaskExecutor is responsible for translating a given task into actions that are then sent to the hardware interface.

```
public class TaskExecutor extends AbstractTaskExecutor implements Runnable
```

Methods

```
+TaskExecutor();
+boolean executeTask(Task t);
+void run
```

```
+void suspend();  
+void resume();  
+stopThread();  
+boolean isRunning();  
+void setHardwareInterface(AbstractHardwareInterface hi);
```

Attributes

```
#ActionSequenceSelector actionSequenceSelector; //selects a sequence of actions that  
//will complete the task  
#boolean isRunning; //allows the task executor to be stopped  
#boolean isSuspended; //allows the TaskExecutor to be suspended and resumed  
#boolean isTask; //determines if a task is currently being executed  
#Task task; //task to be executed  
#AbstractHardwareInterface hardwareInterface; //interface to hardware  
#boolean taskStatus; //determines whether task was successful or not
```

9.1 TaskExecutor()

This method is responsible for initializing a new TaskExecutor object.

Pseudo Code:

```
public TaskExecutor()  
{  
    actionSequenceSelector = new ActionSequenceSelector();  
    isRunning = true;  
    isSuspended = false;  
    isTask = false;  
    isFeedback = false;  
    taskStatus = false;  
    run();  
}
```

9.2 boolean executeTask(Task t)

This method takes a give task and breaks it up into actions sequences which are then sent to the hardware interface.

Pseudo Code:

```
public boolean executeTask(Task t)  
{  
    //wait for the current task to finish executing  
    while( isTask );  
  
    //set t to be the new task to execute  
    isTask = true;  
    this.task = t;  
    //notify the TaskExecutor that there is a new task to be executed  
    .....  
    //wait for the task to compete  
    .....  
    return taskStatus;  
}
```

9.3 *void run()*

This method takes the next task to be executed, breaks it up into an action sequence which are then sent to the hardware interface.

Pseudo Code:

```
public void run()
{
    ActionSequence as;
    Action action;

    while(isRunning())
    {
        //wait for a task to come
        while( !isTask )
        {
            if( isSuspended )
                wait be be resumed
        }

        taskStatus = true;

        as = actionSequenceSelector.getActionSequence( t );

        while( as.hasMoreActions() )
        {
            while( isSuspended );

            if( !isRunning )
                break;

            action = as.peek();

            if( !hardwareInterface.executeAction( action ) )
            {
                taskStatus = false;
                break;
            }

        }

        notify that task has stopped
        isTask = false;
    }
}
```

9.4 *void suspend()*

This method suspends the exeuction of the TaskExecutor.

Pseudo Code:

```
public void suspend()
{
    isSuspended = true;
}
```

9.5 *void resume()*

This method resumes the execution of the TaskExecutor.

Pseudo Code:

```
public void resume()  
{  
    isSuspended = false;  
}
```

9.6 *stopThread()*

This method stops the TaskExecutor.

Pseudo Code:

```
public void stopThread()  
{  
    isRunning = false;  
    isTask = false;  
}
```

9.7 *boolean isRunning()*

This method determines if the TaskExecutor is running.

Pseudo Code:

```
public boolean isRunning()  
{  
    return isRunning;  
}
```

9.8 *void setHardwareInterface(AbstractHardwareInterface hi)*

This method sets the hardware interface which the TaskExecutor will send actions to.

Pseudo Code:

```
public void setHardwareInterface(AbstractHardwareInterface hi)  
{  
    hardwareInterface = hi;  
}
```

10 ActionSequenceSelector

The ActionSequenceSelector selects action sequences for a given task. This class provides a point where ActionSequences can be intelligently selected.

public class ActionSequenceSelector extends AbstractActionSequenceSelector
Methods
+ActionSequenceSelector();
+ActionSequence getActionSequence(Task t);
ActionSequence[] retrieveActionSequences(Task t);
Attributes
#ActionSequenceDataStore actionSequenceDataStore; //interface to actual data store

10.1 ActionSequenceSelector()

This method initializes a new ActionSequenceSelector object.

Pseudo Code:

```
public ActionSequenceSelector()  
{  
    actionSequenceDataStore = new ActionSequenceDataStore();  
}
```

10.2 ActionSequence getActionSequence(Task t)

This method returns an action sequence that accomplishes the given task. If more than one action sequence exist for the given the task, the most suitable action sequence is selected.

Pseudo Code:

```
public ActionSequence getActionSequence(Task t)  
{  
    ActionSequence[] sequences;  
    ActionSequence as;  
    sequences = retrieveActionSequences( t );  
  
    intelligent selection happens here and a single action sequence  
    is returned  
  
    return as;  
}
```

10.3 ActionSequence[] retrieveActionSequences(Task t)

This method queries the ActionSequenceDataStore for all action sequence that will fullfill the given task

Pseudo Code:

```
protected ActionSequence[] retrieveActionSequences(Task t)  
{  
    query data store for all action sequences for given task  
}
```

11 ActionSequenceDataStore

The ActionSequenceDataStore interfaces with the action sequence data store.

public class ActionSequenceDataStore extends AbstractActionSequenceDataStore
Methods +ActionSequence[] getActionSequences(Task t);
Attributes None

11.1 ActionSequence[] getActionSequence(Task t)

This method queries the data store for every action sequence that is related to the given task.

Pseudo Code:

```
public ActionSequence[] getActionSequence(Task t)
{
    query data store for list of action sequences for the give task
}
```

12 HardwareInterface

The HardwareInterface provides interface for communication between the low level system and the simulated hardware.

public class HardwareInterface extends AbstractHardwareInterface
Methods +HardwareInterface(); +boolean start(); +boolean executeAction(Action a);
Attributes #InterruptReader interruptReader; //reads interrupts sent on stderr #BufferedReader in; //reads feedback msgs sent on stdin #BufferedWriter out; //sends commands to the simulation

12.1 public HardwareInterface()

This method initializes a new HardwareInterface object. A connection is made to the simulation and the appropriate read threads are created.

Pseudo Code:

```
public HardwareInterface()
{
    initialize connection to simulation
}
```

12.2 void start()

This method begins reading info from the server.

Pseudo Code:

```
public void start()
{
    interruptReader = new InterruptReader(stderr, //from simulation
                                         interruptHandler);

    interruptReader.start();
}
```

12.3 boolean executeAction(Action a)

This method executes a given action by sending it to the simulation. If the action is successful, true is returned. If the action fails the state monitor is asked to try to fix the situation.

Pseudo Code:

```
public boolean executeAction(Action a)
```

```
{
    Send command to simulation
    Get feedback for command

    if feedback is good
        inform TaskExecutor that action was successful
    Otherwise
        inform StateMonitor that a correction is needed
        if correction worked
            inform TaskExecutor that action was successful
        Otherwise
            inform TaskExecutor that action failed
}
```

13 InterruptReader

The InterruptReader is responsible for reading interrupts sent from the simulation which are sent via standard error stream.

public class InterruptReader extends Thread
--

Methods

+InterruptReader(BufferedReader b, InterruptHandler ih); +void run(); +void stopThread(); +boolean isRunning();
--

Attributes

#BufferedReader read; //hardware interface stderr #boolean isRunning; //flag to control stopping the InterruptReader #InterruptHandler interruptHandler;
--

13.1 InterruptReader(BufferedReader b, InterruptHandler ih)

This method initializes a new InterruptReader object.

Pseudo Code:

```
public InterruptReader(BufferedReader b, InterruptHandler ih)
{
    this.read = b;
    interruptHandler = ih;
}
```

13.2 void run()

This method reads interrupts set by the simulation.

Pseudo Code:

```
public void run()
{
    while( isRunning() )
    {
        read interrupts
        interruptHandler.receiveInterrupt( i );
    }
}
```

}

13.3 void stopThread()

This method stops a the thread and closes the stream that interrupts are being sent on.

Pseudo Code:

```
public void stopThread()
{
    isRunning = false;
    read.close();
}
```

13.4 boolean isRunning()

This method determines whether or not the InterruptReader is running. True is returned if the InterruptReader is running, otherwise false.

Pseudo Code:

```
public boolean isRunning()
{
    return isRunning;
}
```

14 Action

Action is command that will be sent to the simulation.

public class Action
Methods +String toString();
Attributes #int day; //day the action is to be executed #int milliseconds; //the time in milliseconds for when the action is to be executed #int deviceId; //the device the command is for #int command; //the command to issue to the device #double args[]; //any arguments that are a part of the command

14.1 public String toString()

This method converts an action into string form.

Pseudo Code:

```
public String toString()
{
    String actionString = new String("(" + day + " "
        + milliseconds
        + " ) ( "
        + deviceId
        + " "
        + command );

    for(int x = 0; x < args.length(); x++)
    {
```

```
        actionString += args[x];  
    }  
    actionString += " )";  
    return actionString;  
}
```

15 Task

Up for discussion with ANTS-PSR.

16 ActionSequence

An action sequence is a list of actions.

public class ActionSequence
Methods +ActionSequence(Action action[]); +Action peek(); +pop();
Attributes #Stack actionStack; //stores actions

16.1 ActionSequence(Action action[])

This method creates a new ActionSequence object. The object is passed an array of actions.

Pseudo Code:

```
public ActionSequence(Action action[])  
{  
    actionStack = new Stack();  
    for(int x = (action.length() - 1); x >= 0; x--)  
    {  
        actionStack.push( (Object) action[x] );  
    }  
}
```

16.2 Action peek()

This method returns the next action to be executed.

Pseudo Code:

```
public Action peek()  
{  
    return (Action) actionStack.peek();  
}
```

16.3 void pop()

This method removes the next action.

Pseudo Code:

```
public void pop()  
{  
    actionStack.pop();  
}
```

17 Feedback

public class Feedback
Methods +boolean isError();
Attributes -boolean isError;

17.1 boolean isError()

This method determines whether or not the feedback message is an error.

Pseudo Code:

```
public boolean isError()  
{  
    return true if the message is error else false  
}
```

18 Status

This is up for discussion with ANTS-PSR.

19 StateMonitor

The state monitor is responsible for keeping track of the spacecraft's state. It is the job of the state monitor to detect errors in the spacecraft's operations. If an error is detected the state monitor tries to rectify the situation by applying a quick fix. If the quick fix fails then the high level is notified about the situation.

public class StateMonitor extends AbstractStateMonitor implements Runnable
Methods +StateMonitor(); +Status getStatus(); +boolean compare(); + void run(); +Action getTreatment(Feedback f); +void actionSent(Action a);
Attributes #Diagnosis diagnosis; #Status idealState; #Status actualState; #IdealStateGenerator idealStateGenerator; #ActualStateGenerator actualStateGenerator; #int Mode; //the mode the StateMonitor is in

19.1 public StateMonitor()

This method initializes a new StateMonitor object.

Pseudo Code:

```
public StateMonitor()  
{  
    initialize the diagnosis engine  
    create the actual and ideal state models  
    create the actual and ideal state generators  
}
```

19.2 Status getStatus()

This function returns status information to the high level.

Pseudo Code:

```
public Status getStatus()  
{  
    create a new status object  
    populate status object  
    return the new status object  
}
```

19.3 boolean compare()

This function compares the expected state with the current state. If the two state objects are logically equivalent true is returned. If the two objects are not equivalent false is returned.

Pseudo Code:

```
public boolean compare()  
{  
    the Actual and ideal state models are compared by a\  
    predetermined set of rules to determine if the spacecraft is in\  
    the state it should be  
}
```

19.4 public void run()

This method is responsible for comparing the actual state and the ideal state periodically.

Pseudo Code:

```
public void run()  
{  
    inform ActualStateGenerator to update its information  
    periodically compare ideal model and actual model  
}
```

19.5 Action getTreatment(Feedback f)

This method provides a treatment action to fix an action that failed. Information about the failed action is stored in f.

Pseudo Code:

```
public Action getTreatment(Feedback f)
{
    if the state monitor's mode is already in treatment mode
        a null action is set back

    The state monitor's mode is set to treatment

    the diagnosis engine is consulted and a treatment for the failed\
    action is obtained
    the treatment action is then returned
}
```

19.6 void actionSent(Action a)

This method informs the StateMonitor that an action has been sent to the hardware. This allows the StateMonitor to know when to compare the actual state with the ideal state.

Pseudo Code:

```
public void actionSent(Action a)
{
    inform the actualStateGenerator to update its state info
}
```

20 IdealStateGenerator

The IdealStateGenerator monitors actions sent to the hardware interface and modifies the expected state accordingly.

public class IdealStateGenerator
Methods +IdealStateGenerator(Status idealState); +receiveAction(Action a);
Attributes #Status idealState;

20.1 IdealStateGenerator(Status idealState)

This method initializes a new IdealStateGenerator.

Pseudo Code:

```
public IdealStateGenerator(Status idealState)
{
    a new IdealStateGenerator is initialized
    a reference to the state model that the IdealStateGenerator will\
    update is given
}
```

20.2 receiveAction(Action a)

This method is called when a new action has been sent to the hardware. The ideal state model is updated based on the action received.

Pseudo Code:

```
public void receiveAction(Action a)
{
    update the ideal state model
    inform the State monitor that a new action has been sent
}
```

21 ActualStateGenerator

The ActualStateGenerator monitors the hardware interface and updates the ActualState accordingly.

public class ActualStateGenerator
Methods +ActualStateGenerator(Status actualState); +void updateState(); #void queryHardware();
Attributes #Status actualState;

21.1 ActualStateGenerator(Status actualState)

This method initializes a new ActualStateGenerator object.

Pseudo Code:

```
public ActualStateGenerator(Status actualState)
{
    a new ActualStateGenerator is initialized
    the ActualStateGenerator receives a reference to the state model\
    it is responsible for updating
}
```

21.2 void updateState()

This method informs the ActualStateGenerator to update its state.

21.3 queryHardware()

This method queries the spacecrafts hardware in order to determine the current state of the space craft.

Pseudo Code:

```
public void queryHardware()
{
    The hardware is queried for state information
    the actualState is updated based on the feedback received from\
    the query
}
```

22 Diagnosis

This class is responsible for diagnosing a given error and coming up with a treatment for the error.

public class Diagnosis
Methods +Diagnosis(); +ActionSequence getTreatMent(Error e);
Attributes #FaultsandTreatmentsDataStore fandTStore;

22.1 Diagnosis()

This method initializes a new diagnosis object.

Pseudo Code:

```
public Diagnosis()  
{  
    initialize connect ot the actual data store  
}
```

22.2 ActionSequence getTreatMent(Error e)

This method queries the data store for a treatment for the give error.

Pseudo Code:

```
public ActionSequence getTreatMent(Error e)  
{  
    query the fandTStore for a treatment for the given error  
}
```

23 FaultsandTreatmentsDataStore

This class provides and inteface to the Faults and Treatment Data Store. This allows the faults and treatments to stored in a variety of ways.

public class FaultsandTreatmentsDataStore
Methods +FaultsandTreatmentsDataStore(); +ActionSequence getTreatment(Error e);
Attributes <i>None</i>

23.1 FaultsandTreatmentsDataStore()

This method initializes a new FaultsandTreatmentsDataStore object by connecting to the actual data store.

Pseudo Code:

```
public class FaultsandTreatmentsDataStore  
{  
    initialize connection to the actual data store  
}
```

23.2 ActionSequence *getTreatment(Error e)*

This method queries the data store to find a treatment for the given error.

Pseudo Code:

```
public ActionSequence getTreatment(Error e)
{
    query data store for treatment for given error
}
```