

**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	Interrupt.....	14
16	Task .....	14
17	ActionSequence.....	14
17.1	ActionSequence(Action action[]).....	14
17.2	Action peek().....	14
17.3	void pop() .....	15
18	Feedback.....	15
18.1	boolean isError() .....	15
19	Status .....	15
20	StateMonitor.....	15
20.1	public StateMonitor() .....	15
20.2	Status getStatus().....	16
20.3	boolean compare().....	16
20.4	public void run().....	16
21	IdealStateGenerator .....	16
22	ActualStateGenerator .....	16
22.1	void updateState(Feedback f).....	17
23	ModeMonitor .....	17
24	Diagnosis .....	17
25	FaultsandTreatmentsDataStore .....	17
26	TreatmentExecutor ? .....	17

## 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.

<b>abstract class AbstractLowLevel</b>
<b>Methods</b> +abstract boolean executeTask(Task t); +abstract Status getStatus();
<b>Attributes</b> <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.

<b>abstract class AbstractTaskExecutor</b>
<b>Methods</b> +abstract boolean executeTask(Task t);
<b>Attributes</b> <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.

<b>abstract class AbstractActionSequenceSelector</b>
<b>Methods</b> +abstract ActionSequence getActionSequence(Task t);
<b>Attributes</b> <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.

<b>abstract class AbstractActionSequenceDataStore</b>
<b>Methods</b> +abstract ActionSequence[] getActionSequences(Task t);
<b>Attributes</b> 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.

<b>abstract class AbstractHardwareInterface</b>
<b>Methods</b> +abstract boolean executeAction(Action a); +void setInterruptHandler( InterruptHandler ih); +void setActionHandler( ActionHandler ah);
<b>Attributes</b> #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.

<b>public interface ActionHandler</b>
<b>Methods</b> +void receiveAction(Action a);
<b>Attributes</b> <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.

<b>public interface InterruptHandler</b>
<b>Methods</b> +void receiveInterrupt(Interrupt i);
<b>Attributes</b> <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.

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

```
-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.

<b>public class ActionSequenceSelector extends AbstractActionSequenceSelector</b>
<b>Methods</b> +ActionSequenceSelector(); +ActionSequence getActionSequence(Task t); # ActionSequence[] retrieveActionSequences(Task t);
<b>Attributes</b> #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.

<b>public class ActionSequenceDataStore extends AbstractActionSequenceDataStore</b>
<b>Methods</b>
+ActionSequence[] getActionSequences(Task t);
<b>Attributes</b>
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.

<b>public class HardwareInterface extends AbstractHardwareInterface</b>
<b>Methods</b> +HardwareInterface(); +boolean start(); +boolean executeAction(Action a);
<b>Attributes</b> #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.

<b>public class InterruptReader extends Thread</b>
--

<b>Methods</b>
----------------

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

<b>Attributes</b>
-------------------

#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

<b>public class Action</b>
<b>Methods</b> +String toString();
<b>Attributes</b> #int day; #int milliseconds; #int deviceId; #int command; #double args[];

### 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 Interrupt

<code>public class Interrupt</code>
<b>Methods</b>
<b>Attributes</b>

## 16 Task

Up for discussion with ANTS-PSR.

## 17 ActionSequence

An action sequence is a list of actions.

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

### 17.1 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] );  
    }  
}
```

### 17.2 Action peek()

This method returns the next action to be executed.

#### Pseudo Code:

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

### 17.3 void pop()

This method removes the next action.

#### Pseudo Code:

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

## 18 Feedback

<b>public class Feedback</b>
<b>Methods</b> +boolean isError();
<b>Attributes</b> -boolean isError;

### 18.1 boolean isError()

## 19 Status

This is up for discussion with ANTS-PSR.

## 20 StateMonitor

<b>public class StateMonitor extends AbstractStateMonitor implements Runnable</b>
<b>Methods</b> +StateMonitor(); +Status getStatus(); +boolean compare();  +public void run();
<b>Attributes</b> #Diagnosis diagnosis; #Status idealState; #Status actualState; #IdealStateGenerator idealStateGenerator; #ActualStateGenerator actualStateGenerator;

### 20.1 public StateMonitor()

This method initializes a new StateMonitor object.

#### Pseudo Code:

```
public StateMonitor()  
{  
    diagnosis = new Diagnosis();  
    actualState = new Status();  
}
```

```
        actualStateGenerator = new ActualStateGenerator( actualState );  
    }
```

### **20.2 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  
}
```

### **20.3 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 two models are compared by a predetermined set of rules  
}
```

### **20.4 public void run()**

## **21 IdealStateGenerator**

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

<b>public class IdealStateGenerator</b>
<b>Methods</b> #
<b>Attributes</b>

## **22 ActualStateGenerator**

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

<b>public class ActualStateGenerator</b>
<b>Methods</b> +ActualStateGenerator(Status ActualState); #void updateState(Feedback f);
<b>Attributes</b> #Status actualState;

### **22.1 void updateState(Feedback f)**

The actual model is updated by the received feedback.

#### **Pseudo Code:**

```
public void updateState(Feedback f)
{
    update the actual model with the feedback
}
```

### **23 ModeMonitor**

### **24 Diagnosis**

### **25 FaultsandTreatmentsDataStore**

### **26 TreatmentExecutor ?**