# EventsSequenceParser

To start things off, every XML file should have the following structure:

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<EventsSequence>  
</EventsSequence>

Then, every action you want to add to the sequence should be represented with XML tags. The tags you can use can be found in the **Engine.Events.EventType enum**.

**EventsSequenceParser** contains the following methods:

* public static EventsSequence ParseEventsSequence(XmlNode)
* public static EventsSequence ParseEventsSequence(TextAsset)

The method that takes a **TextAsset** as an argument calls the one that takes a **XmlNode**. They both return an **EventsSequence** based on XML content.

Any exception is caught and logged. If a fatal exception occurs, an empty **EventsSequence** is returned. Unknown event types declared in XML raise an **ArgumentException**.

# XmlGameProgressionParser

This parser handles the registration and edition of variables. There are 3 types of variables:

* **ControlSwitch**: represents a Boolean variable.
* **ControlVariable**: represents a Int32 variable.
* **ControlTimer**: represents a Timer variable.

Any exception is caught and logged. If a fatal exception occurs, an empty object of the desired type is returned.

## Parse ControlSwitch

A **ControlSwitch** statement looks like this:

<ControlSwitch Key=”switch1” Value=”true” />  
<ControlSwitch Key=”switch2” Source=”switch1” />

In this example, the first statement registers a variable **switch1** with the value **true**. The second statement registers a variable **switch2** with the value of the **switch1** variable. Please note the **value** is copied, not the **reference**.

If you try to use a **Source** that is not registered, an exception will be thrown.  
If you try to store a non-boolean value, an exception will be thrown.

## Parse ControlVariable

A **ControlVariable** statement looks like this:

<ControlVariable Key=”var1”>

<Operation>Set</Operation>

<Operand Type=”Constant”>10</Operand>

</ControlVariable>  
<ControlVariable Key=”var1”>

<Operation>Add</Operation>

<Operand Type=”Random”>1,5</Operand>

</ControlVariable>

<ControlVariable Key=”var2”>

<Operation>Set</Operation>

<Operand Type=”Variable”>var1</Operand>

</ControlVariable>

In this example, a variable **var1** is registered and has its value set to the constant value of 10. Then, a random integer between 1 and 5 (inclusive) is added to **var1**. Finally, a variable **var2** is registered and has its value set to the value of **var1**. Please note the **value** is copied, not the **reference**.

The **Operation** node can take the following value: Set(=), Add(+), Sub(-), Mul(x), Div(/), Mod(%).

The **Operand** node can take the following values: **Constant**, **Variable**, **Random**.   
In case of **Random**, the bounds must be defined by two Int32 constants, separated by a comma.  
In case of **Variable**, the provided key must have been registered beforehand and must be of type **Int32**. If it has not been registered, an exception will be thrown.

If you try to store a non-Int32 value, an exception will be thrown.

## Parse ControlTimer

A **ControlTimer** statement looks like this:

<ControlTimer Key=”timer1”>

<Action>Start</Action>

<Duration>10</Duration>

</ControlTimer>

<ControlTimer Key=”timer1”>

<Action>Stop</Action>

</ControlTimer>

In this example, a timer **timer1** is registered and started with a **MaximumDuration** property of 10 seconds. Then, we stop **timer1** from running.

The **Action** node can take **Start** or **Stop** as value. If no duration is defined for a **Start** action, an exception will be thrown. Any **Duration** set for a **Stop** action is ignored.

An exception is thrown if you try to give a non-Int32 value to **Duration**.

An exception is thrown if you try to stop a timer that was not registered beforehand.

# XmlFlowControlParser

This parser handles the conditional branches. There are 3 types of conditions:

* **SwitchCondition**: handle conditions involving **Booleans**.
* **VariableCondition**: handle conditions involving **Int32**.
* **TimerCondition**: handle conditions involving **Timer** objects.

All conditions (inheriting from **ConditionalBranch**) can be obtained through the following method:

* public static ConditionalBranch ParseConditionalBranch(XmlNode)

The provided XmlNode must have the following attributes:

* **Type**: the type of condition involved.
* **FirstMember**: depending on the type of condition, this can be a primitive value or the key to a variable.
* **SecondMember**: depending on the type of condition, this can be a primitive value of the key to a variable.

Sequences to execute when the condition is met or not met can be defined inside the ConditionalBranch node using the **true** and **false** tags. One sequence is mandatory, but you can also a sequence sequence.

Any exception is caught and logged. If a fatal exception occurs, a **null** object is returned.

## Parsing a SwitchCondition

A **SwitchCondition** looks like:

<ConditionalBranch Type=”SwitchCondition” FirstMember=”switch1” SecondMember=”false” Condition=”Equal”>

<true></true>

<false></false>  
</ConditionalBranch >

In this example, a **SwitchCondition** is defined. It tries to evaluate **switch1 == false**. The **Condition** attribute can take **Equal (==)** and **NotEqual (!=)** as value.

**FirstMember** and **SecondMember** can be either be variables or Boolean constants. Please note that an error will be raised in case you define a Boolean constant for both members.

**Any Boolean variable used must be registered first through the ControlSwitch event command.**

## Parsing a VariableCondition

A **VariableCondition** looks like:

<ConditionalBranch Type=”VariableCondition” FirstMember=”var1” SecondMember=”var2” Condition=”Equal”>

<true></true>

<false></false>  
</ConditionalBranch >

In this example, a **VariableCondition** is defined. It tries to evaluate **var1 == var2**. The **Condition** attribute can take the following values:

* Equal (==)
* GreaterThan (>)
* SmallerThan (<)
* Different (!=)
* EqualOrGreaterThan (>=)
* EqualOrSmallerThan (<=)

**FirstMember** and **SecondMember** can be either be variables or Int32 constants. Please note that an error will be raised in case you define a Int32 constant for both members.

**Any Int32 variable used must be registered first through the ControlVariable event command.**

## Parsing a TimerCondition

A **TimerCondition** looks like:

<ConditionalBranch Type=”TimerCondition” FirstMember=”timer1” SecondMember=”20” Condition=”Before”>

<true></true>

<false></false>  
</ConditionalBranch >

In this example, a **TimerCondition** is defined. It tries to evaluate **timer1.CurrentTime < 20**. The **Condition** attribute can take **Before** or **After** as value.

When comparing a timer to a fixed value, the condition checks whether the **CurrentTime** property of the timer is smaller or greater than the fixed value.

When comparing two timers, the condition evaluates the **CurrentTime** property of both timers and compare the values based on the **Condition** attribute.

**FirstMember** and **SecondMember** can be either be variables or Int32 constants. Please note that an error will be raised in case you define a Int32 variable for both members.

**Any Timer used must be registered first through the ControlTimer event command.**

# XmlMessageParser

This parser handles the commands to display dialog boxes on screen. There are 3 types of dialog boxes:

* **DisplayDialog**: display a simple text message, with a locutor.
* **DisplayChoiceList**: display a message with a list of choices the player can pick from.
* **DisplayInputNumber**: display a box where a player can select several digits to input a value or numerical code.

Any exception is caught and logged. If a fatal exception occurs, an empty object is returned.

## Parse DisplayDialog

A **DisplayDialog** looks like this:

<DisplayDialog>

<DialogBoxStyle>Classic</DialogBoxStyle>

<DialogBoxPosition>Bottom</DialogBoxPosition>

<Locutor>Hero</Locutor>

<Message>The message you want to display</Message>

<FaceGraphics>Face1</FaceGraphics>  
</DisplayDialog >

In this example, a dialog box is displayed on screen. It uses the **Classic** style (from **Classic** or **Transparent**) and displays the box at the bottom of the screen (from **Bottom, Center** or **Top**).

A **Locutor** is also defined but is optional. Then the message is provided. Finally, a key can be provided to display graphics for the face. The graphics is displayed only if the provided key has been registered. If not, it will simply be ignored.

## Parse DisplayChoiceList

A **DisplayChoiceList** looks like this:

<DisplayChoice>

<Message>The message you want to display</Message>

<Choices>

<Choice Id=”0”>My first choice</Choice>

<Choice Id=”1”>My second choice</Choice>

</Choices>  
</DisplayChoice>

In this example, a dialog box with a message is displayed, along several choices. Each choice is defined by an **Id** (string) and a text. If no choice is provided, only the message will be displayed.

## Parse DisplayInputNumber

A **DisplayInputNumber** looks like this:

<InputNumber DigitsCount=”3” />

In this example, a dialog box with 3 digits is displayed. The player a freely select 3 different digits from 0 to 9.

An exception is thrown if a non-Int32 value is provided.

# XmlTimingParser

This parser handles timing operations through event commands.

There is currently only one command: **Wait**. It can be used to wait for a fixed number of seconds (float).

Any exception is caught and logged. If a fatal exception occurs, an empty object is returned.

## Parse Wait

A **Wait** statement looks like this:

<Wait Duration=”3.4” />

In this example, the engine will simply wait 3.4 seconds until the next event command from the sequence. Please note that a simple dot (.) should be used to write floating point numbers.

An exception is thrown if the provided duration cannot be parsed as a float.