Note: all the scenes from folder «Sample» add to Build Settings before testing.

Events run the world

Events run the behavior of objects. The object reacting to an event may not always be aware of the object that generates the event. The matter becomes more complicated when events and objects grow in number. Plugin **SimpleEvent** allows you to make the events common to the entire system. Events remind us an organized newsletter:

1. Anyone who wants can subscribe to the newsletter.
2. When a publisher prepares news, he transmits this information to a distributor.
3. The distributor provides news to all subscribers.
4. Those who do not wish to receive news any more can always unsubscribe.

So, we have three existing roles now:

- Subscriber – someone who wants to know about the events,
- Publisher – someone who generates the events,
- Distributor – a person who informs all concerned about the occurrence of the event.

Let's make a few clarifications:

1. The event is at the head of all.
2. The number of subscribers and publishers is not limited.
3. Nothing prevents a subscriber to be the publisher, and the publisher to be a subscriber.
4. Subscribers and a publisher do not know anything about each other.
5. A distributor only knows what events are interested for this or that subscriber, and only sends the information.

Objective

For each chapter its own scene with examples of the use of the plugin is created. Every opportunity will be considered on the example of the tests. Our goal is to acquaint you with all the features of the plugin **SimpleEvent**.

For the test, there are three modes of operation:

1) **Record** is a registration of the subscriptions to various events

2) **Play** is an initialization of the selected event

3) **Stop** is a restart to the initial state

Record

In each scene there are several objects (planets), which we'll sign for the events. Clicking on the planet allows you to add an event subscriber for it. As an indicator, a satellite of the planet is created. When you hover over the satellite in the information window, you will see detailed information about the event. Clicking the satellite will help you to unsubscribe the planet from this event and will destroy the satellite.

The system includes three events: **Fireworks, Shield** and **Explosion**. Switching between the events is made from the special menu. When you subscribe, the currently selected event is used.

The number of subscriptions for a single planet is artificially limited to 12 in the system, in order not to overload the scene with satellites to the planet. However, this does not mean that the number of subscriptions for a single object has to be small.

Play

When the desired number of subscriptions is added, you can initiate the current event from the menu and then watch as each subscription is processed for this event. Subscriptions made for other events must remain inactive. When in use, delay to the launch of the next event is added to the system, for clarity.

Stop

This mode is used to reset the scene to its original state and delete all subscriptions. In this case, the system generates another event to be subscribed by all the planets when creating, and as a handler the procedure that removes all subscriptions is used. Total in the scene there are 4 events, three for the tests and one to restart the test. Now let's see how it is implemented in the plugin.

Preparation

To keep things as simple as possible, we'll distinguish the events by names. Let's create a class that will keep a list of events names as constants.

```
public class EventName
{

#region Event Names

       public const string E_CHANGE = "change";
       public const string E_FIREWORKS = "fireworks";
       public const string E_SHIELD = "shield";
       public const string E_EXPLOSION = "war";

#endregion
}
```

Each event must have a unique name. Subscription and distribution will be carried out according to this name.

To exchange the data, a static class "Store" is used. All public values of the basic variables are stored in it. As we shall see later, one would pass them as parameters together with the events, but in order not to complicate the model, let's use a common storage.

Simple Events

First of all, let's try just to sign up for the event. To do this, you can download the scene Sample/SimpleEvent.

Creating a Distributor

Distributor is created automatically when you first call to the plugin and requires no additional configuration.

Creating a Subscriber

Subscription to the test events like **Fireworks, Shield** and **Explosion** we shall add to the objects interactively. To select the name of interactive event, let's add static function to the class **Planet:**

```
Planet.getEventName()
```

Subscriptions reset is done with the help of the event **Change of status**.

All the code we are interested in is in the class **Sample/Script/Planet**

1. To begin with, let's use the desired namespace:

```
using KingDOM.Event;
```

now plugin **SimpleEvent** is available to us.

2. The next step is to create a method that will handle the event.
   For the event **Change of status** (E_CHANGE) let's create method:

```
void hnChangeStatus(SimpleEvent evnt)
    {
       for (int i = 0; i<maxIndicator; i++ )
       {
              if (indicators[i] is Sputnik)
                 (indicators[i] as Sputnik).destroyIndicator();
       }
    }
```

The method has only one argument of type **SimpleEvent**, containing all the necessary information about the event. When the event **Change of status** occurs, this handler removes all the satellites, and the subscriptions from the planet are removed with them as well.

Handlers for the interactive events are created in a similar way:

```
virtual public void hnFireworks(SimpleEvent evnt)
{
       CreateFirework();
}
virtual public void hnShield(SimpleEvent evnt)
{
       CreateShield();
}
virtual public void hnExplosion(SimpleEvent evnt)
{
       CreateExplosion();
}
```

3. We subscribe to the event **Change of status** when creating an object:

```
void Awake ()
{
       Sender.AddEvent(EventName.E_CHANGE, hnChangeStatus);
}
```

Here we ask the **Sender** to add a subscription for the event (`EventName.E_CHANGE`), and when it is onset, to provide the information about the event into the processing method (hnChangeStatus).

Now, even if the planet is hidden in the scene, it will respond to the event. If you want to subscribe to the function only when the object is active, you should subscribe to the method **OnEnable.**

For the interactive events, the addition of the event will be made only when you click on a planet in the **Record** Mode:

```
public virtual void OnMouseDown()
{
	if (Store.Regime == Store.TestRegime.Record)
	{
		Sender.AddEvent(getEventName(), getEventHandler());
		CreateEventReaction();
	}
}
```

where the functions **getEventName()** and **getEventHandler()** return the text name of the event and the handler for the current event. After you create a subscription, you should perform a visual display of the processing reaction.

Note: It is allowed to do several similar subscription for the same event.

4. If you want to destroy the object (planet), don't forget to unsubscribe from the event **Change of Status**:

```
void OnDestroy()
{
	Sender.RemoveEvent(EventName.E_CHANGE, hnChangeStatus);
}
```

In this case, we do the reverse operation: we ask the **Sender** to unsubscribe the event (EventName.E_CHANGE) and when it's onset, to stop transmitting the information about the event into the processing method (hnChangeStatus).

When removing, from the subscriptions queue the first existing subscription with an appropriate name and subscription handler is removed.

If you want the subscription to be inactive when you hide an object, you can unsubscribe from the event in the method **OnDisable.**

When the satellite in the method «Sputnik.OnDisable()» is destroyed, we should unsubscribe from the event.

That is all! Congratulations! We signed up for the event.

Creating a Publisher

This operation isn't difficult as well. All that is required is to inform the distributor in the right place that the event is occurred. In this case, events calling is performed in the class CenterController, which is responsible for the processing action of the user in the menu.

For the interactive event notification of the event occurs in the method **OnPlaying**, which will be called when the button "Play" is activated.

```
virtual protected void OnPlaying()
{
	Sender.SendEvent(Planet.getEventName(), this);
}
```

This command sends the **Sender** the name of the event (the name of the current event from the familiar function Planet.getEventName ()) and a reference to the object that called the event (this).

In future, we will override this method for other tests, so it is defined as a virtual one.

For the event **Change of Status**, the event sending is performed when the button "Stop" is pressed:

```
public void OnStop()
{
	Sender.SendEvent(EventName.E_CHANGE, this);
	if (info != null) info.text = startText;
	timeRemaining = 4.0f;
	OnRecord();
}
```

Here, the **Sender** sends an event notification (EventName.E_CHANGE) and a reference to the object that called the event (this).

Now, let's run the scene and do the test.

Sending the parameter

The main benefit of the news is an opportunity to convey useful information.

To distinguish the parameters, we need unique names for these parameters. Let's create another file with the constants of parameter names:

```
public class EventParm
{
#region Parameters

	public const string V_POWER = "power";

#endregion
}
```

The tests will be carried out in a scene Sample/ParmEvent. In this test, the menu has got a slider to change the value of the parameter "Power", which we will submit to the events at the time of its creation. When processing an event, we will change the effect of the event according to the received parameter.

Raising an event with the parameter

To pass the information with some event, it is necessary to add a parameter (the name and value) when you call the event in the class CenterController2:

```
protected override void OnPlaying()
{
	Sender.SendEvent(Planet.getEventName(), this, EventParm.V_POWER, power);
}
```

As you can see, the information is transmitted by the addition of KeyValuePair. For the key, as well as with the events, we use the parameter name. As for the value, the object of any type can be transferred. The number of parameters to be passed at the same time is not limited, the only condition is that everyone should be referred to as a KeyValuePair . The procedure for passing parameters does not matter.

Note: If you pass multiple parameters with the same key name, then in the end with the event will be given only one of these parameters, and some information will be lost.

Processing of the received parameters

All the basic information about the event resulting in the handler method can be extracted from the object type SimpleEvent (see tables below.)

Property	Type	Description
eventName	`string`	Event name
target	`object`	Event source
args	`Dictionary<string, object>`	List of the parameters passed with the event
stop	`bool`	Interruption of the event distribution. If this flag is set to"true" in the event handler, all the handlers, following in the line for the current one, will not receive the notice of the occurrence of the event.

To use it easily**, SimpleEvent** has got the following methods:

Method	Description
`bool` ExistParm<T>(`string` key)	This allows you to check the presence of the parameter with the help of the key and the type of value that is attached to the event.
T GetParm<T>(`string` key)	This allows you to get the parameter value with the help of the key from the parameter list, attached to the event.

Note: In this system there are two similar methods without an explicit type. In this case, the responsibility for checking the type of the value falls on the shoulders of the programmer.

Now let's try to get this value in the processing of interactive event in the methods of the relevant events**.**

```
override public void hnFireworks(SimpleEvent evnt)
{
	float power = 1;
	if (evnt != null && evnt.ExistParm<float>(EventParm.V_POWER))
		power = evnt.GetParm<float>(EventParm.V_POWER);
	CreateFirework(power);
}
```

As you can see, first we've checked that the parameter exists in the list of transmitted pairs. Then we extract the value, and finally, we call the handler with the new power value. For the events **Explosion** (E_EXPLOSION) and **Shield** (E_SHIELD) handlers are changed similarly.

Call priority

It happens that someone of the subscribers wants to receive the news before others. Let's suppose that the minister wants to be the first reader of the newspaper, before it appears in circulation.

The tests will be carried out in a scene **Sample/PriorityEvent**. In this test, the menu has got a slider to change the value of the parameter "priority". Unlike parameter passing, priority will be set when putting the event handler in the queue of the subscriptions.

In order to determine the sequence of execution of processing methods, you can only specify subscription priority with the integer.

The following should be considered:
1. The higher the value, the earlier the execution queue.

2. Subscriptions with the same priority are executed in the order of registration.
3. If the priority is not specified, then by default it is considered that the priority is 0.

For example:

```
Sender.AddEvent(EventName.MY_EVENT, hnListenEvent);

Sender.AddEvent(EventName.MY_EVENT, 3, hnListenEvent2);

Sender.AddEvent(EventName.MY_EVENT, 5, hnListenEvent3);

Sender.AddEvent(EventName.MY_EVENT, -3,hnListenEvent4);
```

Now let us see in what sequence the handlers will be called. First, a call without specifying a priority takes the priority of 0 by default. Then we have a priority subscription and last we obtain the following order:

```
hnListenEvent3 (priority of 5),

hnListenEvent2 (priority of 3),

hnListenEvent  (priority of 0 by default),

hnListenEvent4 (priority of -3)
```

Subscription registration with priority

In order to specify the priority when subscribing, just specify the setting priority when calling after the name of the event.

Thus, all that is required is to rewrite the method that handles a click on the planet.

```
public override void OnMouseDown()
{
	if (Store.Regime == Store.TestRegime.Record)
	{
		Sender.AddEvent(getEventName(), Store.Priority, getEventHandler());
		CreateEventReaction();
	}
}
```

Unsubscribe priority

To unsubscribe from the event with a priority, you must specify in a delete command on which priority the subscription was done. As you remember, we unsubscribe destroying the object **Sputnik** from a method **OnDestroy ():**

```
void OnDestroy()
{
	Sender.RemoveEvent(EventName, EventPriority, EventHandler);
}
```

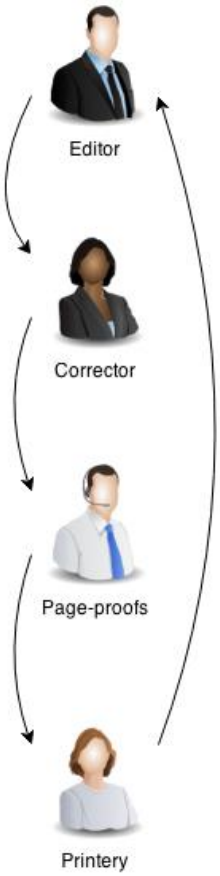
The event name, priority, and handler are passed into the method in the same way as we subscribed to the event. All these parameters are filled in when creating the satellite and contain all the information about the subscription for which the indicator is created. The first match is deleted from the subscription on all three of these parameters in the existing list of subscriptions.

Sending by hierarchy

Let's imagine the editorial office. There are many departments that prepare the information each in its own direction. Each boss is only responsible for its direction and he or she monitors only the department entrusted to him. Before the reader gets the news, initially it has to pass several difficult stages on its way from the publisher. Moreover, in different departments different people are interested in news preparation.

Journalists or writers prepare the material. Before printing, the editor checks the subject, then he sends the material to the corrector. Page-proofs and design follow after that. And again everything goes back to the editor for the last check.

Note that the processing of the event **Printing** begins with the editor. Then we go down the hierarchy, involving into the process new people until we reach the source of the event (Printery) . After the completion of the event, the process control returns again to the main element of the hierarchy (our Editor).



Thus, subscriptions processing inside the hierarchy is performed in three stages:

1. **Tunneling** (from the parent to the source of the event)
2. **Target** (event processing at its source)
3. **Bubbling** (from the source to its parent)

The same we can observe with the events: sometimes there is a need to keep track of their occurrence within the hierarchy of the scene Unity. Let's suppose that the object "Window" has got a plenty of nested elements, among which there is a "data table" and a "close button". So, when you press the button "close the window", the event **Close the Window** is created. Before closing it is necessary to save the data from the table, then to play the animation effect on the button itself, and after that to start the animation of window closing. If you open more than one windows, then the event should be processed only where the button "close the window" is pressed.

The tests will be carried out in a scene Sample/HierarchicalEvent. This scene is a little different from the previous tests. Here we see the suppositive hierarchy tree. The main planet is located in the center of the top, and all the rest which are in her submission are connected with it by trade lines. The subordinate planets in the scene are the children of the planets located in the scene higher.

Subscription Registration

The priority which is already familiar to us will help us to determine the order of subscriptions calling. This priority should be set from the menu before starting the subscription. For the event inside the scene of the hierarchy, the priority plays a dual role:

-with a positive value of the priority the subscription is made at the stage **Tunneling**

- with a zero or negative value of the priority the subscription is registered at the stages **Target** and **Bubbling.**

We should keep in our mind that the spreading of the event first happens within the hierarchy, and only then by priority within the object of the hierarchy.

Registration is performed in the same way as we already know in class **Planet4.**

```
public override void OnMouseDown()
{
	if (Store.Regime == Store.TestRegime.Record)
	{
		Sender.AddEvent(getEventName(), Store.Priority, getEventHandler());
		CreateEventReaction();
	}

}
```

As you can see, subscription to the event is of no difference from the usual subscription with the specified priority. And if you are interested in the stages of **Target** and **Bubbling**, you can subscribe without specifying the priority (do not forget that in this case it will be considered a zero (0) by default).

Calling of the event in the hierarchy

As the distribution of an event in the hierarchy of the scene is not possible without pointing out to an element of the scene, in our test there is another difference on the generation of the event. Now we cannot call an event directly from the menu, we need to specify the source object. Therefore, using the button “Play” we will no longer send the events, but instead we’ll change "Test” mode into “Play” mode in our system and will wait for the user until he chooses a planet that will be the source of the event. So, our handler for the MouseEvent will be changed again.

```
public override void OnMouseDown()
{
	if (Store.Regime == Store.TestRegime.Record)
	{
		Sender.AddEvent(getEventName(), Store.Priority, getEventHandler());

		CreateEventReaction();
	}
	if (Store.Regime == Store.TestRegime.Play)
	{
		Sender.SendEventHierarchy(Planet.getEventName(), this);
	}
}
```

Now by clicking the planet in the “Play” mode , we’ll send the current event (Planet.getEventName ()) according to the hierarchy: first, from the root element down the hierarchy to the source of the event (this), and then vice versa in the same way back to the root element.

In this case, as with the usual sending of the event, you may specify additional parameters by specifying them in a bunch of key-value as described above.

Now let's test it. First of all you should run the test, then change the priority to a value> 0 and add a few subscribers on different nodes. Now change the priority to a negative value, and add a few more subscribers.

Now, let's turn to the "Play" mode and choose one of the planets of the scene. We'll see how the branch of hierarchy (where the object is located), will react to the event. First of all, we'll observe the **Tunneling** stage with all the subscriptions in this branch from the root element to the source with a positive priority. Then all the subscriptions at the source will take place. And finally all the remaining subscription in the branch of the source object will take place, climbing up the hierarchy.

That's all about the main points of the Plugin. Now you can use it in order to facilitate some of your developments. When using the Plugin, you will find detailed documentation for each instrument and parameters.

Results:

The main steps:

Action	Method
Handler (a method that takes one argument of the type **SimpleEvent**)	`Action<SimpleEvent>`
Subscribe to the event with a priority	`Sender.AddEvent(string nameEvent, int priority, Action<SimpleEvent> function)`
Subscribe to the event with the default priority	`Sender.AddEvent(string nameEvent, Action<SimpleEvent> function)`
Unsubscribe from the event with a priority	`Sender.RemoveEvent(string nameEvent, int priority, Action<SimpleEvent> function)`
Unsubscribe from the event with the default priority	`Sender.RemoveEvent(string nameEvent, Action<SimpleEvent> function)`
Send the event	`Sender.SendEvent(string nameEvent, object target, paramsobject[] args)`
Send the event according to the branch of hierarchy	`Sender.SendEventHierarchy(string nameEvent, object target, paramsobject[] args)`

When processing the received event, you should extract the parameters using the built-in functions and variables in the object **SimpleEvent** passed to the handler.

When the event takes place, the object **SimpleEvent** is passed to the handler. Using the built-in functions of this object you can receive the parameters passed to the event. Using the values of the properties of this object, you can get all the necessary information about the event: event name, source, etc.

Now you may use the plugin **SimpleEvent** in your work and make it easier!