Implementation

This page gives an overview of how Behaviors can be implemented.

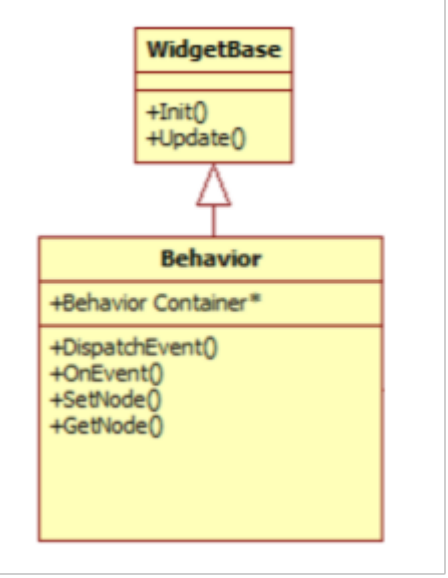
How to implement a Behavior

There is very little difference between implementing a Widget and implementing a Behavior. Though there are some difference which will be explained below.

Base Class

The major difference is that Behaviors need to be derived from the Candera::Behavior base class instead of the Candera::WidgetBase class. Incidentally the Candera::Behavior base class is internally derived from Candera::WidgetBase further demonstrating the similarities between both.

Methods



From the Candera::WidgetBase the already known Init() and Update() methods are inherited.

- **Init:** Called once when the Widget/Behavior is created, after all Widget/Behavior properties have been set. This is used to initialize the Widget/Behavior.
- **Update:** Called once per iteration (frame). Here normally the Widget/Behavior performs its actual functionality.

Newly added are the methods from Candera::Behavior which are related to event handling and the attached node:

- **DispatchEvent:** Starts dispatching an event with a specified strategy
- **OnEvent:** Called when an event is received by a Behavior
- **SetNode/GetNode:** Allows access to the node this behavior is attached to. Note that this uses a new data type called Candera::AbstractNodePointer which abstracts between 2D and 3D nodes. This allows creating behaviors that work und 2D and 3D without duplicating code.

META INFO

The Meta-Info is essential to provide generic access to all properties of the widget/behavior as well as special information for Scene-Composer about the available properties, their types and functionality. The available macros to define the Meta-Info are very similar to those used by widgets.

The first set of Macros defines a Behavior either for 3D, 2D or both (mixed):

```
...
CdaBehaviorDef(<<ClassName>>, <<BaseClassName>>)
    [Information][Properties][Events]
CdaBehaviorDefEnd()
...
```

```
...
CdaBehavior2DDef(<<ClassName>>, <<BaseClassName>>)
    [Information][Properties][Events]
CdaBehaviorDefEnd()
...
```

```
...
CdaBehaviorMixedDef(<<ClassName>>, <<BaseClassName>>)
    [Information][Properties][Events]
CdaBehaviorDefEnd()
...
```

All other macros are the same as where used by Widgets. This includes the macros to add additional information:

```
...
CdaReadableName("Name")
CdaCategory("Category")
CdaDescription("Description")
...
```

As well as the macros to define properties:

```
...
CdaProperties()
...     [Property List]
CdaPropertiesEnd()
...
```

```
...
CdaProperty(<<Name>>, <<Type>>, <<Setter>>, <<Getter>>)
CdaDescription("Column used by this Item inside the Area")
CdaCategory("Area")
CdaPropertyEnd()
...
```

The new macros for events will be described in the Event Handling chapter.

For further information about the Meta Info please also see chapter Behavior Definition.

RUNTIME TYPE INFORMATION

CGI-Studio uses its own RTTI system for major parts instead of relying on the RTTI of the compiler. For this a set of macros is added to each class that uses this system. In case of Behaviors only a single macro is required to be added to the code:

```
...
CGI_BEHAVIOR_RTTI_DEFINITION(<<ClassName>>)
...
```

Event Handling

Events are one of the major new features introduced via Behaviors. This chapter shows how they can be utilized from within the Behavior code.

Meta Info

The first requirement is to add all events that should are either received or sent to the Meta-Info via the following macros:

```
...
CdaEvents()
   CdaEvent(CdaOutputEvent, <<EventType>>, "Description")
   CdaEvent(CdaInputEvent, <<EventType>>, "Description")
CdaEventsEnd()
```

```
...
```

Sending Events

An Event can be sent by multiple different approaches.

The first is achieved by calling the DispatchEvent() method of the Behavior or any of the other available alternatives. It requires providing the dispatch strategy and event itself. When using this method the dispatching will start from the current node of the Behavior.

```
...
    CgiStudioControl::Event event;
    EventDispatchResult dispatchResult;
    EventDispatchStrategy dispatchStrategy;

    DispatchEvent(dispatchStrategy, event, dispatchResult);
...
```

> Note that the Event and EventDispatchStrategy need to be replaced with the concrete implementations.

The second options is to directly call the DispatchEvent() method of the Event-Dispatch-Strategy. This allows to define the node from which the dispatching should start:

```
...
    Candera::EventDispatchResult eventDispatchResult;
    Candera::BroadcastEventDispatchStrategy dispatchStrategy;
    CgiStudioControl::ChangeValueEvent valueChangedEvent(FeatStd::Variant
(listDataEvent->GetData()),
                                                         CgiStudioControl::ChangeValue::Absolute
    dispatchStrategy.DispatchEvent(GetNode(), valueChangedEvent, eventDispatchResult);
...
```

Receiving Events

Receiving of events is done via the OnEvent-Method. Here normally the incoming events are type-cast to check the Event-Type and processed if applicable. Further it is possible to stop further dispatching of the event via the Dispatch-Result. In case a custom Dispatch-Result is used (again type-casting can be used) then additional information can be returned to the sender.

```
...
void ListItemEventBehavior::OnEvent(const FeatStd::Event
& event, Candera::EventDispatchResult& dispatchResult)
{
    const ListDataEvent* listDataEvent = Candera::Dynamic_Cast<const ListDataEvent*>(&event);
```

```
    if (0 != listDataEvent) {
        if (m_index == listDataEvent->GetIndex()) {
            dispatchResult.StopDispatchingImmediately();

            // Do something with the data of the event
        }
    }
}
...
```

If derived from Behaviors that also do event processing also the Base-Class OnEvent method should be called.

```
...
Base::OnEvent(event, dispatchResult);
...
```

Custom Events

It is possible to define custom events by simply deriving from FeatStd::Event class. An Event can contain any number custom data.

```
...
class ListDataEvent : public FeatStd::Event
{
    public:
        FEATSTD_RTTI_DECLARATION();

        ListDataEvent(FeatStd::Int32 index, FeatStd::Int32 value) :
            m_index(index),
            m_value(value)
        {
        }

        FeatStd::Int32 GetIndex() const
        {
            return m_index;
        }

        FeatStd::Int32 GetData() const
        {
            return m_value;
        }

    private:
        FeatStd::Int32 m_index;
        FeatStd::Int32 m_value;
};
...
```

```
...
FEATSTD_RTTI_DEFINITION(ListDataEvent, FeatStd::Event)
...
```

FINDING BEHAVIORS

The second major advantage compared to widgets is that it is now possible to determine if any and which Behaviors are attached to a node. This can be done by iterating over all behaviors of a node via the code snipped shown below.

```
...
Candera::Behavior* behavior = Candera::Behavior::GetFirstBehavior(node);

while (0 != behavior) {
    // do something

    behavior = behavior->GetNextBehavior();
}
...
```

For a better understanding of how events work, please see chapter Predefined Behaviors.

Creating a custom Behavior

First to create a custom Behavior for most of them a Base Behavior has to be implemented. (If the Base Behavior is not needed the next few steps can be skipped)

To do this achieve that a <CUSTOMBEHAVIOR>Base.xhcdl file has to be created.
This file contains things like: includeName, behavior name, readableName, category, properties and the UUID.
These need to be filled out for the specifications of the Behavior.

Samples for this file can be found under: "...\cgi_studio_controls\src\Behaviors". Just select a BehaviorBase.xhcdl which has similarities with the custom behaviour copy it and modify it to the special needs.

If the file is finished add it to the "...\cgi_studio_controls\src\Behaviors\Behaviors.xhcdl" file in this way:

Behaviors.xhcdl

```
<include href="<BEHAVIORFOLDER>/<CUSTOMBEHAVIOR>Base.xhcdl" />
```

Next start the "...\cgi_studio_controls\src\Behaviors\GenerateBehaviors.bat". This file then generates the <CUSTOMBEHAVIOR>Base.h and <CUSTOMBEHAVIOR>Base.cpp files.

After this step is completed the actual Behavior logic has to be written. The <CUSTOMBEHAVIOR>.h/.cpp files will be made and the files of other already implemented Behaviors can serve as the Template for the custom Behavior.

Adding the Behavior to the entire CGI Studio:

If the Intention is to use the Behavior in every CGI Studio project it can be added to the whole Studio.

For this the <CUSTOMBEHAVIOR>Base.h/.cpp have to be added to "...\cgi_studio_controls\src\Behaviors\<BEHAVIORFOLDER>\generated\FileList.cmake" in this manner:

FileList.cmake

```
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.h"
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.cpp"
```

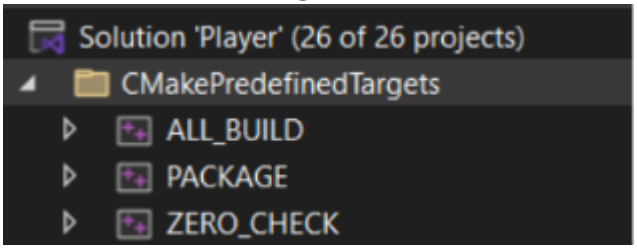
Next the <CUSTOMBEHAVIOR>.h/.cpp as well as <CUSTOMBEHAVIOR>Base.xhcdl have to be added to "...\cgi_studio_controls\src\Behaviors\<BEHAVIORFOLDER>\FileList.cmake" like this:

FileList.cmake

```
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.xhcdl"
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.h"
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.cpp"
```

After these two steps the Behavior can be build with CMake and then Visual Studio. (For a Guide how to do that have a look at the Wiki Page: Building-Player-and-Widgets)

After opening Visual Studio ALL_Build -> Build can be started which is found under CMakePredefinedTargets.



Now CGI Studio can be opened and the custom Behavior can be used.

Adding the Behavior to a specific Project:

To add a Behavior to a specific Project the "...\cmake\Apps\Courier\CourierSampleApp\CMakeLists.txt" of the Project has to be modified with this line. (CUSTOMBEHAVIOR_FOLDER has to be substituted with the actual folder name)

CMakeLists.txt

```
add_subdirectory("${CANDERA_ROOT_PATH}/<CUSTOMBEHAVIOR_FOLDER>"
"${CMAKE_BINARY_DIR}/<CUSTOMBEHAVIOR_FOLDER>")
```

> **Important:** This include has to be after all other includes, because some of them clear the Behavior list.

Next the <CUSTOMBEHAVIOR_FOLDER> has to be created in the Candera root Directory. This folder then needs to include CMakeLists.txt as well as a FileList.cmake.

CMakeLists.txt

```
# Creation of new dedicated behavior project
project(MyCustomBehaviors)
add_library(${PROJECT_NAME})

#------------------------------------------------------------------
# collecting all files
include("${CMAKE_CURRENT_LIST_DIR}/FileList.cmake")

# load the paths as sources into 'all_sources'
get_target_property(all_sources ${PROJECT_NAME} SOURCES)

# create a structured visualization inside the solution
source_group(TREE ${CMAKE_CURRENT_LIST_DIR} FILES ${all_sources})

#------------------------------------------------------------------
# this is the include path to the custom behaviors which shall be used.
target_include_directories(${PROJECT_NAME}
  PUBLIC
  ${CMAKE_CURRENT_LIST_DIR}
)

#------------------------------------------------------------------
# definition of which project must be linked
# if application parts like application lib is required, please link the project
accordingly.
target_link_libraries(${PROJECT_NAME}
  PUBLIC
  Candera
)

#------------------------------------------------------------------
# These two calls attach a Behavior project to the internal Behavior set.
# IMPORTANT: the 0 at the end tells the system that already existing behaviors should not be
cleared.
CgiBehaviorSet_AddTarget(${PROJECT_NAME} 0)
CgiCreateBehaviorPartSet("${PROJECT_NAME}" "BehaviorSet_${PROJECT_NAME}")
```

FileList.cmake

```
target_sources(${PROJECT_NAME}
  PRIVATE
  "${CMAKE_CURRENT_LIST_DIR}/FileList.cmake"
)
include("${CMAKE_CURRENT_LIST_DIR}/Behaviors/FileList.cmake")
```

After that we need to create another folder in our initial <CUSTOMBEHAVIOR_FOLDER>.

> **Important:** This folder has to be named "Behaviors".

Into this folder we paste our custom Behavior files. (<CUSTOMBEHAVIOR>Base.h/.cpp, <CUSTOMBEHAVIOR>.h/.cpp, <CUSTOMBEHAVIOR>Base.xhcdl)

Then we create another FileList.cmake where we add lines for all our custom Behavior files as seen below.

FileList.cmake

```
target_sources(${PROJECT_NAME}
  PRIVATE
  "${CMAKE_CURRENT_LIST_DIR}/FileList.cmake"

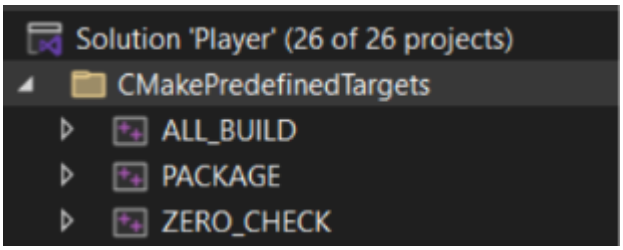
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.h"
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.cpp"

  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.xhcdl"
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.h"
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.cpp"

)
```

After these steps the Behavior can again be build with CMake and then Visual Studio. (For a Guide how to do that have a look at the Wiki Page: Building-Player-and-Widgets)

After opening Visual Studio ALL_Build -> Build can be started which is found under CMakePredefinedTargets.



Now the specific Project can be opened and the custom Behavior can be used.

Revision #12
Created 2023-02-24 01:51:16 UTC by Kanai Tomoaki
Updated 2026-08-19 02:06:38 UTC by Tsuyoshi.Kato