# Writing and extending model interpreters for C2WT

## Structure

├── blob-local-storage  //local storage for the generated artifacts and etc. (icons, resources). Not under version control  
├── config //WebGME Configuration Files. Any changes made here requires restart of WebGME  
├── node\_modules //local cache of dependent node modules. Not under version control  
├── seeds //Folder containing the json files that can be used to seed new projects   
└── src   
 ├── decorators //Folder for model decorators  
 │   └── HLADecorator //Decorator code for showing Interactions and Object with a UML class like notation  
 └── plugins //Folder for the model plugins (aka. interpreters or generators)  
 ├── C2Core //Folder for shared code components  
 │ ├── MavenPOM.js //Class to represent Maven POM files  
 │   ├── ModelTraverserMixin.js //Mixin of the model Visitor/Traverser framework  
 │ └── xmljsonconverter.js //Simple function to convert json to xml (developed by Patrik Meier)   
 ├── C2Federates //Folder for type specific model visitors to be used by the exporters  
 ├── DeploymentExporter //Source of the Exporter that generates the deployment and simulation execution artifacts  
 │ └── Templates //Folder for the ejs code templates for the DeploymentExporter  
 └── FederatesExporter //Source of the Exporter that generates the federate base and RTI (interactions and objects) code  
 └── Templates //Folder for the ejs code templates for the FederatesExporter

## Visitor framework

The visitor framework was inspired by the Antlr4 parse tree visitor. The idea is to separate the domain specific logic from the programmatic traversal of the model. To learn more about the motivation watch this [presentation](https://www.youtube.com/watch?v=q8p1voEiu8Q). Beyond separating the traversal and domain specific logic this framework also alleviates some problematic aspect of asynchronous javascript development, such as the deeply nested callback functions and proper finishing of the tree traversals. The framework uses a depth-first traversal of the model.

### Initialization

The visitor framework is implemented as [Mixin](https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/) so it can be easily integrated into any WebGME plugin. To include the visitor in a plugin it has to be specified as dependent require module and called at initialization time.

:::javascript  
define([  
 'plugin/PluginConfig',  
 'plugin/PluginBase',  
 'C2Core/ModelTraverserMixin'  
], function (  
 PluginConfig,  
 PluginBase,  
 ModelTraverserMixin  
) {  
 /\*\*  
 \* Initializes a new instance of Plugin.  
 \* @constructor  
 \*/  
 var Plugin = function () {  
 // Call base class' constructor  
 PluginBase.call(this);  
 // Call to mix in the ModelTraverser functions  
 ModelTraverserMixin.call(this);  
 }  
});

### Invocation

To start the visiting of the model invoke the visitAllChildrenFromRootContainer method with the root node and the callback function that will be invoked when the processing of the whole model is done or when an terminal error has been raised.

:::javascript  
this.visitAllChildrenFromRootContainer(rootNode, /\*callback\*/ function(err){  
 if(err){  
 self.logger.error(err);  
 }else{  
 //finish the plugin execution;  
 }  
 });

### Visitors

Visitors are the functions that are invoked to process each of the model nodes. The visitors are tied to the meta type of the nodes. Each node will be exactly once per parent. Children and descendants of a node is not visited if any of its parent containers are excluded from the visit.

All of the visitor have to return an object that is the result of the visit.

:::javascript  
{  
 context:{}, //To be passed to the children and to the post-visitor  
 error: undefined // Should be undefined or not specified unless there is an error in the processing the node  
}

Specifying the error key of the object will cause the traversal to quit and return the value of the error key to the callback of invocation

#### Root visitor

This is the start of the model traversal. Its name is set and cannot be changed as the root itself does not have a meta type

:::javascript  
FederatesExporter.prototype.ROOT\_visitor = function(node){  
 return {context:{}};  
}

#### Node visitor

The node visitor is called when the traversal reaches the node in its progress down the model. The visitor of the parent node (if registered) has already been called. The context object provided by the parent visitor is passed down this visitor. Each visitor gets its own shallow copy of the context to provide separation.

:::javascript  
this.visit\_FCO = function(node, parent, context){  
 return {context:{}};  
}

#### Node post visitor

The node visitor is called when the traversal finished visiting all the children (and transitively all descendants) of the node.

:::javascript  
this.post\_visit\_FCO = function(node, context){  
 return {context:{}};  
}

### Control

The following control methods can be overwritten in the plugin class to customize the traversal logic.

#### Visitor name per meta type

:::javascript  
Plugin.prototype.getVisitorFuncName = function(nodeType){return /\*string\*/ visitorName;}  
Plugin.prototype.getPostVisitorFuncName = function(nodeType){return /\*string\*/ visitorName;}

#### Excluding types from the traversal

:::javascript  
Plugin.prototype.excludeFromVisit = function(node){return /\*bool\*/ exclude;}

#### Providing order for visiting the children of a type

:::javascript  
Plugin.prototype.getChildSorterFunc = function(nodeType, self){return /\*function(a,b)\*/ sorter;}

## Federate Code Generators

The FederateExporter (/src/plugins/FederateExporter) generates the source code from the Federates, Objects and Interactions from the model. The FederateExporter is constructed from several parts that can be categorized as generic or federate type specific. The generic parts include the above mentioned Visitor framework, configuration parameters, file artifact generators and the Object and Interaction (Real Time Interface - RTI) visitors.

### File artifact generators

The FederateExporter provides a callback mechanism to the visitors that are to generate files inside a WebGME artifact that is created as the result of the interpretation. The visitor can register a function that will be called after all of the model has been traversed and the artifacts has been created and ready for population.

:::javascript  
//The use of fileGenerators attribute to create file in an artifact post traversal  
self.fileGenerators.push(function(artifact, callback){  
 artifact.addFile(/\*string\*/ filename, /\*string\*/content, function (err) {  
 if (err) {  
 callback(err);  
 return;  
 }else{  
 callback();  
 }  
 });  
});

### Federate type specific components

The federate type specific components are in separate files in the src/plugins/C2Federates folder. These components are also mixins that are added to the FederateExporter object at initialization time.

:::javascript  
define([  
 'plugin/PluginConfig',  
 'plugin/PluginBase',  
 'common/util/ejs',  
 'C2Core/ModelTraverserMixin',  
 'C2Core/xmljsonconverter',  
 'C2Core/MavenPOM',  
 'FederatesExporter/PubSubVisitors',  
 'FederatesExporter/RTIVisitors',  
 'C2Federates/Templates/Templates',  
 'C2Federates/GenericFederate',  
 'C2Federates/JavaFederate',  
 'C2Federates/MapperFederate',  
 'C2Federates/CppFederate',  
 'C2Federates/OmnetFederate',  
 'C2Federates/CPNFederate'  
], function (  
 PluginConfig,  
 PluginBase,  
 ejs,  
 ModelTraverserMixin,  
 JSON2XMLConverter,  
 MavenPOM,  
 PubSubVisitors,  
 RTIVisitors,  
 TEMPLATES,  
 GenericFederate,  
 JavaFederate,  
 MapperFederate,  
 CppFederate,  
 OmnetFederate,  
 CPNFederate  
 ) {  
 var FederatesExporter = function () {  
 // Call base class' constructor.  
  
 this.federateTypes = this.federateTypes || {};   
  
 PluginBase.call(this);  
 ModelTraverserMixin.call(this);  
   
 //Mixins of the generic parts of the exporter  
 PubSubVisitors.call(this);  
 RTIVisitors.call(this);  
 GenericFederate.call(this);  
  
 //Mixins of the federate type specific part exporter  
 JavaFederate.call(this);  
 MapperFederate.call(this);  
 CppFederate.call(this);  
 OmnetFederate.call(this);  
 CPNFederate.call(this);  
  
 //...  
 };  
}

Registering a federate type specific component is done by adding a descriptor of it to the federateTypes map and mixing in its visitor functions into the FederateExporter object. The descriptor also contains an init function that can be used to initialize variables used the visitors before the actual traversal. The init function of all federate types is called early in the execution of the plugin right after the user provided parameters are processed.

:::javascript  
define([  
 'common/util/ejs',  
 'C2Core/MavenPOM',  
 'C2Federates/Templates/Templates', //The compiled Templates.js file  
 'C2Federates/JavaRTI' //Dependent component  
], function (  
 ejs,  
 MavenPOM,  
 TEMPLATES,  
 JavaRTI  
) {  
 var SpecialFederateExporter = function () {  
 var self = this,;  
  
 JavaRTI.call(this); //Make sure dependent component is added  
 this.federateTypes = this.federateTypes || {};  
   
 //Register the new SepcialFederate type  
 this.federateTypes['SpecialFederate'] = {  
 includeInExport: false,  
 longName: 'SpecialFederate',  
 //init function to initialize variable needed by the visitors  
 init: function(){  
 self.initJavaRTI(); //Make sure dependent component is initalized  
 self.specialfed\_myVariable = "Hello "; //define a variable to be used in the visitor  
 }  
 }  
   
 //Add the visitor of the SpecialFederate  
 this.visit\_SpecialFederate = function(node, parent, context){  
 //use the variable defined in the init  
 var msg = self.specialfed\_myVariable + self.core.getAttribute(node, 'name');  
 return {context:{message: msg}};  
 };  
 }   
 return JavaFederateExporter;  
});

### Templates

The Templates directory under src/plugins/FederatesExporter contains all the ejs templates that are used by the code generators. To help deliver these templates to the client all the files in the directory are escaped and appended into the Templates.js file. The Templates.js is a require module that can be added to the exporter as a dependency. After making any changes to any of the templates make sure to recombine the Templates.js file by invoking

:::bash  
node combine\_templates.js

#### directoryNameTemplate

To promote unified naming of directories and artifacts in the main Federate exporter the self.directoryNameTemplate is defined as an ejs template. This template will generate names with following structure {FederationName}-{ArtifactType}-{Language} for example HelloWorld-base-java.

:::javascript  
//Exampel use of directoryNameTemplate   
var simDirSpec = {federation\_name: self.projectName, artifact\_name: "rti", language:"java"},  
 simDirPath = simDirBasePath + ejs.render(self.directoryNameTemplate, simDirSpec);

## Simulation Execution Generator

## Things to be careful with

### Name clash in Federate visitors

As the type specific (C++, Java, Omnet etc.) federate exporters are also implemented with mixins they share a common parent object (this, self). Care has to be taken in creating new attributes under that object. The naming convention that has been used is to prepend the variables with the name of the federate type. If variable is to be shared between different exporters than it is advised to check whether it already exist before redefining it.

:::javascript  
var self = this;  
//Type specific Examples:  
this.cppRTIInitDone = true;  
self.cpp\_federateBasePOM = new MavenPOM();  
  
//Shared between different java based federate types  
if(!self.javaPOM){  
 self.javaPOM = new MavenPOM(self.mainPom);   
}  
  
//"Global" variable used in all exporters  
self.mainPom = new MavenPOM();