The OpenMI Document Series

Migrating Models

For the OpenMI (Version 2.0)

Grant agreement number LIFE06 ENV/UK/000409

Title	OpenMI Document Series: Migrating Models for the OpenMI (Version 2.0)
Editor	The OpenMI Association Technical Committee (OATC)
Authors	The OpenMI Association Technical Committee (OATC)
Current version	v1.0
Date	30/11/2010
Status	Final © The OpenMI Association
Copyright	All methodologies, ideas and proposals in this document are the copyright of the OpenMI Association. These methodologies, ideas and proposals may not be used to change or improve the specification of any project to which this document relates, to modify an existing project or to initiate a new project, without first obtaining written approval from those of the OpenMI-LIFE participants who own the particular methodologies, ideas and proposals involved.

Preface

The OpenMI stands for Open Modeling Interface, which aims to deliver a standardized way of linking environment-related models.

This is part of the OpenMI report series providing examples, how to migrate models to be OpenMI complients

Titles in the series include:

- Scope
- The OpenMI ‘in a Nutshell’
- OpenMI Standard 2 Reference
- OpenMI Standard 2 Specification
- **Migrating Models** (this document)

The OpenMI is maintained by the OpenMI Association and this document, along with other more detailed documentation, can be obtained from www.openmi.org.

The official reference to this document is:

The OpenMI Association (2010) *Migrating Models for the OpenMI (Version 2.0)*. Part of the OpenMI Document Series

Disclaimer

The information in this document is made available on the condition that the user accepts responsibility for checking that it is correct and that it is fit for the purpose to which it is applied.

The OpenMI Association will not accept any responsibility for damage arising from actions based upon the information in this document.

Acknowledgement

The OpenMI Association's members would like to acknowledge the contribution of the European Commission in co-funding the HarmonIT and OpenMI-LIFE projects. In particular, we would like to thank the Commission's staff for their sustained encouragement and support over many years.

Further information

Further information on OpenMI-LIFE can be found on the project website, www.OpenMI-Life.org.

Information on the OpenMI Association and the Open Modelling Interface (OpenMI) can be found on www.openmi.org.

Migrating models

MIGRATING MODELS **6**

1. Introduction 7
1.1. OpenMI compliance 7
1.2. Planning the migration 9

2. .... A general wrapping pattern 11

2.1. The LinkableEngine – wrapping time stepping engines 11

3. .... Migration – step by step 12

3.1. Making the engine core a shared DLL 12
3.2. MyEngineWrapper 13
3.3. Adding input and output items 15

4. .... Migrating an unmanaged engine 18

4.1. Creating the .NET assemblies 18
4.2. MyEngineDLLAccess – Accessing the engine core 19
4.3. MyEngineDotNetAccess – using C# conventions 20
4.4. Adding input and output items 21

5. .... Testing the component 25

5.1. Unit testing 25

6. .... The OMI file 29

6.1. Structure of the OMI file 29

7. .... Performance issues 31

7.1. Memory consumption 31
7.2. System processes 31

8. .... The Simple River example 32

9. .... Use cases 34

9.1. Connecting to other rivers 34
9.2. Inflow from geo-referenced catchment database 35

10. .... Migration of the Simple River 38

10.1. The Simple River wrapper 38
10.2. Implementation of the Initialize method 38
10.3. Implementation of getting/setting values in input and output items 39
10.4. Implementation of the remaining methods 40

1. Introduction

The OpenMI standard was designed to allow easy migration of existing model engines. The standard is implemented in C#/.NET and Java. Many existing model engines are implemented in other programming languages, such as Fortran, Pascal, C and C++. This article describes a *wrapping pattern* that tries to minimize the amount of changes needed to be made to the engine core in order to make the model engine OpenMI compliant, regardless of whether the model engine is implemented in .NET, java or any other programming language.

Although it may appear a huge challenge to turn a model engine into an OpenMI-compliant linkable component, it may not be as difficult as it seems. The OpenMI Environment provides a large number of utilities that make migration easier. These utilities can be used by anyone who is migrating a model. Note that you are not required to use these utilities in order to comply with the OpenMI standard, they are just there to ease your work. The utilities can be used as a whole or you can pick and choose from them; alternatively, you can use the utilities as the basis for your own implementations.

This article will show you how to migrate using the OpenMI utilities to the full extent. Step-by-step instructions are given for the whole migration process, from defining the requirements for an OpenMI component, through design and implementation to testing.

1.4 note

Notes that are relevant for those familiar with version 1.4 of the OpenMI standard will be presented like shown here. Readers that do not know anything about version 1.4 can skip over any paragraph of this type.

1.1. OpenMI compliance

The requirements for a model to be OpenMI compliant are given in detail in **OpenMI Standard 2 Specification** document. In short, there are three basic requirements for compliance:

1. The component must implement the OpenMI.Standard2.IBaseLinkableComponent interface.

2. The component must be able to handle specified state-transitions, and invocation of certain methods in each state/phase.

3. The component must have an associated XML file containing information of the components, its capabilities and availability.

The IBaseLinkableComponent is the key interface in the OpenMI standard. For a time progressing engines, which is what we will consider here, the ITimeSpaceComponent is the key interface. It defines exactly what is required from a time progressing component. It provides lists of input and output items that specifies what kind of data the model can exchange.

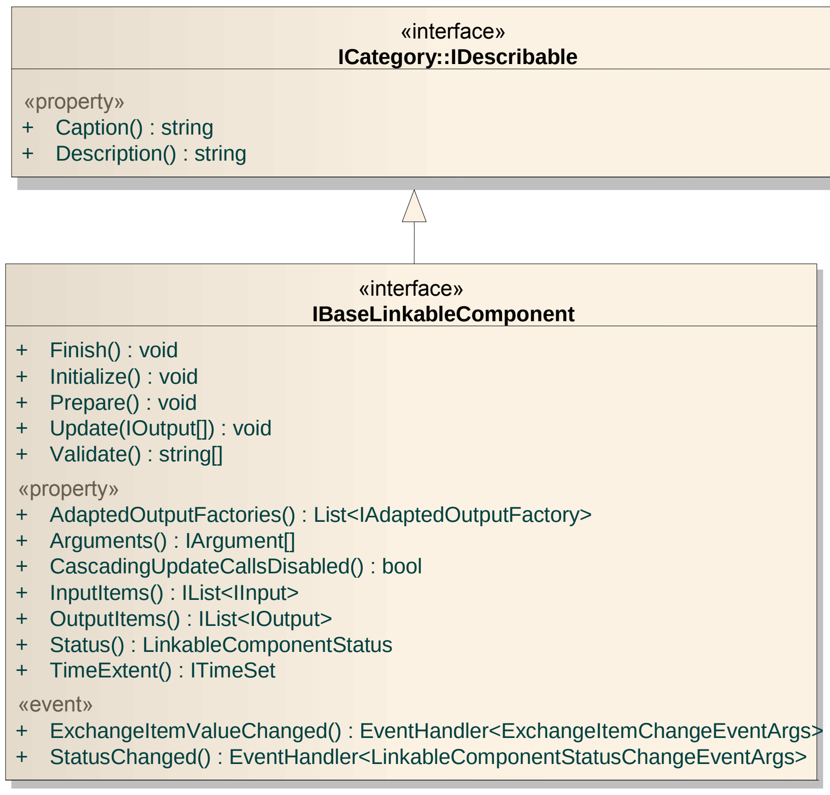

«interface»
ICategory::IDescribable
«property»
+ Caption() : string
+ Description() : string
«interface»
IBaseLinkableComponent
+ Finish() : void
+ Initialize() : void
+ Prepare() : void
+ Update(IOutput[]) : void
+ Validate() : string[]
«property»
+ AdaptedOutputFactories() : List<IAdaptedOutputFactory>
+ Arguments() : IArgument[]
+ CascadingUpdateCallsDisabled() : bool
+ InputItems() : IList<IInput>
+ OutputItems() : IList<IOutput>
+ Status() : LinkableComponentStatus
+ TimeExtent() : ITimeSet
«event»
+ ExchangeItemValueChanged() : EventHandler<ExchangeItemChangeEventArgs>
+ StatusChanged() : EventHandler<LinkableComponentStatusChangeEventArgs>

Figure 1 ILinkableComponent interface

The component must also behave in a way that makes it possible, during runtime, to discover its capabilities, to link with it to components, and during simulation-time exchange values between the components. Therefor, a set of states/phases has been defined, that a component must handle. The standard interface specification gives an overview of the phases and the most important methods that must be available in each phase.

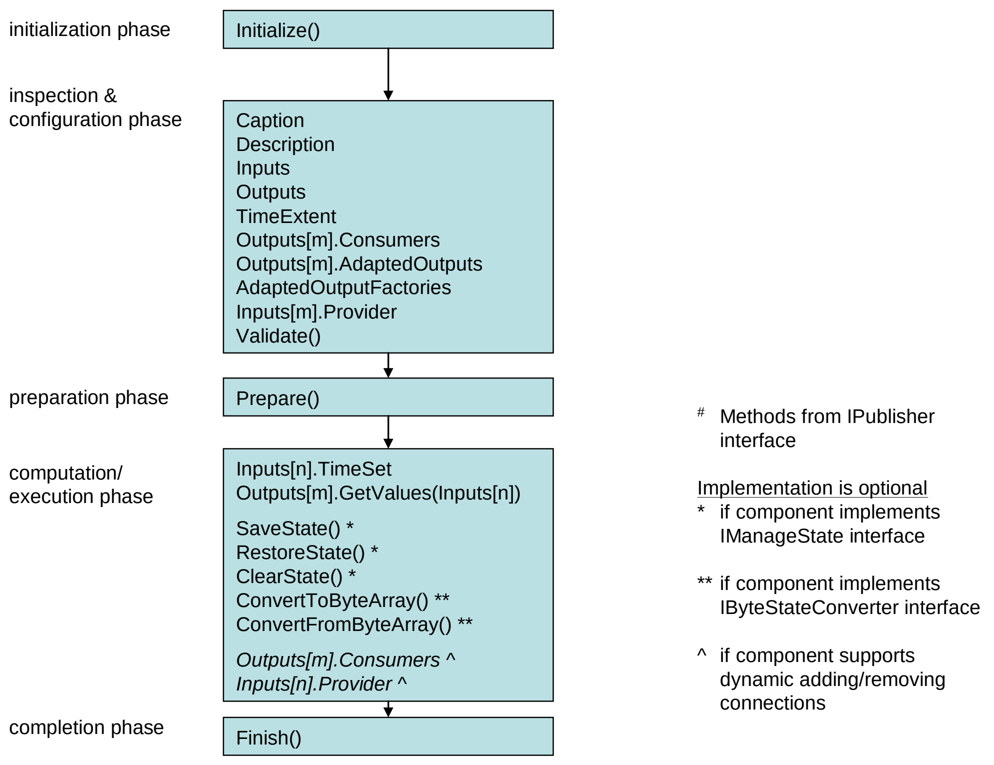

initialization phase
Initialize()
inspection & configuration phase
Caption
Description
Inputs
Outputs
TimeExtent
Outputs[m].Consumers
Outputs[m].AdaptedOutputs
AdaptedOutputFactories
Inputs[m].Provider
Validate()
preparation phase
Prepare()
computation/ execution phase
Inputs[n].TimeSet
Outputs[m].GetValues(Inputs[n])
SaveState() *
RestoreState() *
ClearState() *
ConvertToByteArray() **
ConvertFromByteArray() **
Outputs[m].Consumers ^
Inputs[n].Provider ^
completion phase
Finish()
# Methods from IPublisher interface
Implementation is optional
* if component implements IManageState interface
** if component implements IByteStateConverter interface
^ if component supports dynamic adding/removing connections

Figure 2 Required phases, their sequence and available methods

In essence,

- The model must be structured in such a way that initialization is separate from computation, with inputs as boundary conditions being collected in the computation phase and not during initialization.
- The model must be able to submit to run-time control by an outside entity.
- The model must be able to expose information to the outside world on the modelled quantities it can provide.
- The model must be able to provide the values of the modelled quantities and specify the point/extent in time and space that the values belong to.

1.2. Planning the migration

Before you start migrating a model it is important that you have a precise idea about how your model is intended to be used when it is running as an OpenMI component. Think about any situation where it will be useful to run your model linked to other OpenMI components. Such components could be other models, data providers, optimization tools or calibration tools. You may even find it useful to run two instances of your model component in the same configuration.

The goal is to indentify potential inputs and outputs to/from your model.

Chapter 3 shows an example, suggesting ways in which you can plan the migration of a model, including the development of use cases and the definition of exchange items.

2. A general wrapping pattern

Wrapping means that you create a C# class that implements the key LinkableComponent interfaces, and that accesses the engine core through some user specified API. The wrapper will appear to the users as an OpenMI LinkebleComponent, handling your engine core as a 'black box'.

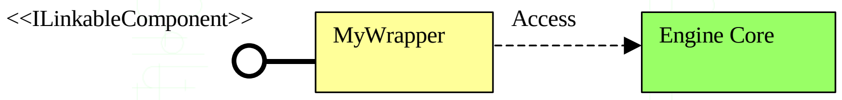


Figure 3 OpenMI wrapping

The main advantage of using a wrapping pattern is that we can keep the OpenMI-specific implementations separated from the engine core. Typically, the engines will also be used as standalone applications where OpenMI is not used and OpenMI should not interfere with that.

2.1. The LinkableEngine – wrapping time stepping engines

Model engines that are doing timestep-based computations have many things in common. The OpenMI association has therefore tried to develop a wrapping component that can be used for these types of engines. This wrapper is part of the provided SDK, under the name *LinkableEngine* and is located in the package

OATC.OpenMI.SDK.ModelWrapper.LinkableEngine

The LinkableEngine is an abstract class with a default implementation of the ITimeSpaceComponent interface. The specific behaviour of your model engine must be implemented by overriding the abstract methods in the LinkableEngine.

Remember that when using the SDK, the namespace should be change to one that identifies your company/component. Remember also that the SDK implementation is just one example of how this can be accomplished. Modifications to the SDK classes/replacement of classes in the SDK may be necessary/appropriate in certain cases.

3. Migration – step by step

This chapter will go through the steps required to wrap a time stepping engine using the LinkableEngine class. It is assumed that the engine functionality is accessible from .Net. For engines not accessible from within .Net, Chapter 4 will describe one way of doing that for a Fortran based model.

3.1. Making the engine core a shared DLL

Model engines are typically compiled into an executable file (EXE). Such executable files are not accessible by other components and as such are not applicable when running in an OpenMI environment. It is therefore necessary for an engine core to be compiled into a shared library/a dynamic link library file (DLL).

Ideally we should make modifications to the engines so that the same engine core can be used both when running in the OpenMI environment and when running as a standalone application. The preferable approach is to make a new application (EXE) that calls a function in the engine core DLL which, in turn, makes the engine perform a full simulation.

The Figure 4 Running an engine as a standalone applicationillustrates the software components required to run an engine as a standalone application, using a shared engine core dll. The MyEngineApplication.EXE is a simple program calling into the MyEngineDLL.dll. The application is never used when running in OpenMI environments.

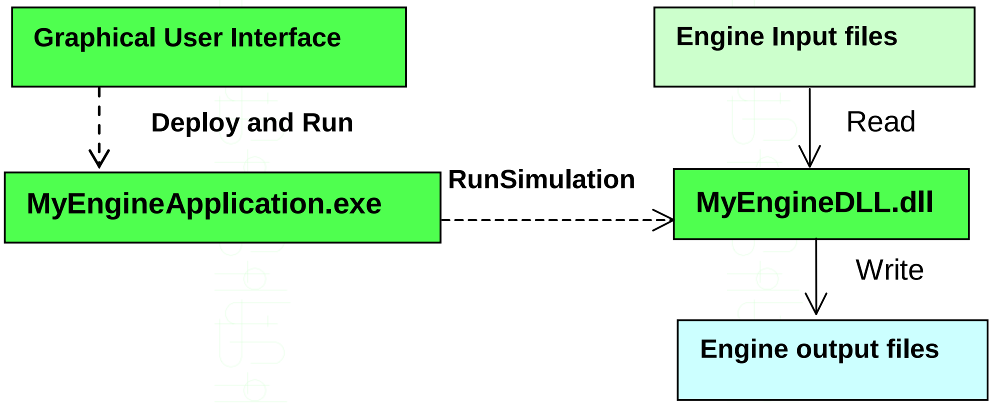


Figure 4 Running an engine as a standalone application

The following steps are required in the conversion of the engine core:

1. Change the engine core so that it compiles into a shared DLL.
2. Add a function to the engine core DLL that will run a full simulation, alike:

```
logical function RunSimulation(...)
```

3. Create a new engine application (EXE) that calls the RunSimulation function in your engine core DLL.
4. Run your new engine application and check that the engine is still producing correct results.

When an engine is running in the OpenMI Environment it must be able to initialize, perform single timesteps, finalize etc. as separate operations. The engine core must support this behaviour, and that may require the engine core code to be reorganized.

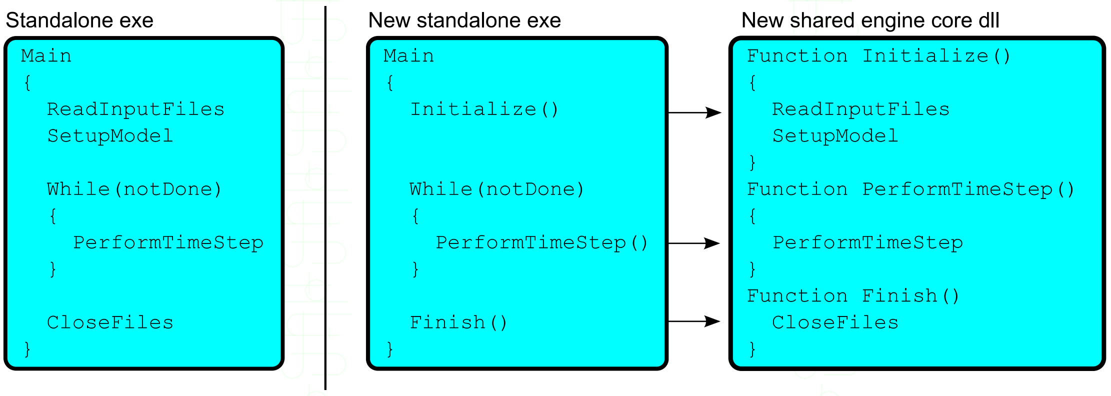
One approach is to add the following functions to the API of the engine core:

```
logical function Initialize()
```
(Read input files and setup model)

```
logical function PerformTimeStep()
```
(Perform a single timestep)

```
logical function Finish()
```
(Close files, deallocate memory)

The main function can now be changed so that it calls the Initialize function, then repeatedly the PerformTimeStep function until the simulation has completed, and finally calls the Finish function. At this point remember to run the application again and check that the engine is still producing the correct results.



Standalone exe

```
Main
{
  ReadInputFiles
  SetupModel

  While(notDone)
  {
    PerformTimeStep
  }

  CloseFiles
}
```

New standalone exe

```
Main
{
  Initialize()


  While(notDone)
  {
    PerformTimeStep()
  }

  Finish()
}
```

New shared engine core dll

```
Function Initialize()
{
  ReadInputFiles
  SetupModel
}
Function PerformTimeStep()
{
  PerformTimeStep
}
Function Finish()
  CloseFiles
}
```

We have now completed the restructuring of the engine core. We will need more functions in the engine core API; however, the additions need not affect existing code and/or structure of existing code. For now, we will move on to creating the wrapper code, and add functions to the engine core API along the way.

3.2. MyEngineWrapper

The next step is to implement the MyEngineWrapper class

The MyEngineWrapper we will extend from the LinkableEngine, which implements the ITimeSpaceLinkableEngine interface. The abstract functions of the LinkableEngine must be implemented. The easiest way to get started is to make the development environment auto-generate the stub code for these abstract functions.

Assume we have the shared engine core functionality in a class called MyEngineCore. We give the MyEngineWrapper a private field (_myEngine) that holds a reference to the MyEngineCore class.

The first step is to implement the Initialize, PerformTimeStep and Finish methods. The Initialize method will instantiate the MySharedEngineCore object and assign this object to the _myEngine variable.

Example code for this is shown in Figure 5.

```
using System;
using System.Collections
namespace MyOrganisation.OpenMI.MyModel
{
  public class MyEngineWrapper : Oatc.OpenMI.Sdk.ModelWrapper.LinkableEngine
  {
    private MyEngineCore _myEngine;

    public void Initialize (string filePath)
    {
      _myEngine = new MyEngineCore();
      _myEngine.Initialize(filePath);
    }
    public void PerformTimeStep()
    {
      _myEngine.PerformTimeStep();
    }
    public void Finish()
    {
      _myEngine.Finish();
    }
    [...]
  }
}
```

Figure 5 Example code for the wrapper classes

The basic structure of the engine and wrapper code is now in place. The task is now to go through the MyEngineWrapper class and complete the implementation of the methods that had stub code auto-generated. Some of these methods can be completed only by changing the code in the MyEngineWrapper; for others, additions need to be made to the engine core DLL. After completion of each method remember to create and update the test classes and run the unit test.

For each method you must decide if the bulk of implementation should be located in the MyEngineWrapper class or in the engine core (MyEngineDLL). There is no general answer to this question. Placing the bulk of implementation in the engine core could be advantageous from the perspective of maintenance because you have most things located in one place. On the other hand, you may want to keep the engine core as free as possible of OpenMI-related code and therefore put the bulk of the implementation into the MyEngineWrapper class. Finally, there may also be considerations about the preferred programming language; the engine core may be programmed in Fortran, C or Pascal, whereas the MyEngineWrapper class is programmed in C#.

Implementation of the LinkableEngine methods depends on the engine core, so it is not possible to give a general explanation of how each individual method should be implemented. The methods that must be implemented is:

```
//== The Oatc.OpenMI.SDK.ModelWrapper.LinkableEngine methods ==
// -- Execution control methods --
void Initialize(IArgument[] arguments);
void OnValidate();
void Prepare();
void PerformTimestep(ICollection<EngineOutputItem> requiredOutputItems)
void Finish();
//-- Time methods  --
ITime StartTime
ITime EndTime
ITime GetCurrentTime(bool asStamp)
ITime GetInputTime(bool asStamp)
```

Figure 6 The LinkableEngine methods

Notice that the MyEngineCore may end up looking more or less like the MyEngineWrapper. In some cases it will be easier just to let the MyEngineCore extent directly from the LinkableEngine, making the wrapper redundant. However, that introduces OpenMI relevant code in the core engine dll, which is not always desirable. This is also only possible when the MyEngineCore is a .Net component.

3.3. Adding input and output items

The model now implements the ITimeSpaceComponent interface and can run in the OpenMI environment. It is not much worth yet though; As long as it does not have any input and output items, it can neither link to other models nor exchange data.

We need to add the input items and the output items to the component. These are usually set up in the Initialize method of the MyEngineWrapper. When using the LinkableEngine, we must use the EngineInputItem and EngineOutputItem.

EngineInputItem and EngineOutput item are abstract classes, which require an implementation of the SetValuesToEngine and GetValuesFromEngine methods respectively. You can extend and implement the set and get value functionality and add them to the component.

A number of classes exists that implements the EngineInputItem and the EngineOutput item, utilizing different techniques:

- EngineDInputItem and EngineDOutputItem: When created, these must be provided with a delegate (function pointer) of a certain type, that does the setting and getting of the values from the engine.
- EngineIInputItem and EngineIOutputItem: When created, these must be provided with an object implementing the IValueSetter or IValueGetter interface. The SetValues and GetValues of these interfaces must then set and get the values from the engine.
- EngineEInputItem and EngineEOutputItem: These work together with the LinkableGetSetEngine. When created they are provided a LinkableGetSetEngine, and they pass on the set and get value functionality to the SetEngineValues and GetEngineValues of the LinkableGetSetEngine. It is not the recommended approach to use this, because it is often difficult to optimize and have shown to produce bottle necks.

1.4 note

One important difference in the version 2.0 is that the responsilibity for the data transfer from an input/output item to the engine core now is on the input/output item, not the linkable engine. The `LinkableGetSetEngine` has been implemented to mimic the 1.4 way of setting/getting values, but its use is not recommended.

3.3.1. Delegate version

In the engine core API you can make a method available that gets/sets a value from/to the engine. It is an easy and safe implementation.

```
public void AddInflow(int nodeIndex, double value)
public double GetFlow(int branchIndex)
```

Then we need to create an input item and an output item that utilizes this. The EngineDInputItem and EngineDOutputItem are input items and output items that can do that.

```
EngineDInputItem inflow = new EngineDInputItem("someId", quantity, elmtSet, this);
int nodeIndex = i;
inflow.ValueSetter =
  delegate(IValueSet values)
    {
      _myEngine.AddInflow(nodeIndex, (double)values.GetValue(0,0));
    };

EngineDOutputItem flow = new EngineDOutputItem("someId", quantity, elmtSet, this);
int branchIndex = i;
flow.ValueGetter =
  delegate()
    {
      IList res = new List<double>(1) { _myEngine.GetFlow(branchIndex) };
      return new ValueSet(new List<IList>{res});
    };
```

Alternatively, if the wrapper has direct access to the internal variables of the core engine, the delegate implementation can set the values directly:

```
double[] _flow ...
EngineDOutputItem flow = new EngineDOutputItem("someId", quantity, elmtSet, this);
int branchIndex = i;
flow.ValueGetter =
  delegate()
    {
      IList res = new List<double>(1) { _flow[branchIndex] };
      return new ValueSet(new List<IList>{res});
    };
```

An example of use can be found in

```
Oatc.OpenMI\Examples\SimpleCSharpRiver\RiverModelDelegateLC.cs
```

3.3.2. Interface version

The EngineIInputItem and EngineIOutputItem is very alike the delegate versions, apart from that they must be initialized with a class implementing the IValueSetter or IValueGetter interface.

```
double[] _flow ...
EngineIOutputItem flow = new EngineIOutputItem("someId", quantity, elmtSet, this);
flow.ValueGetter = new ValueToVectorGetSetter<double>(_flow, branchIndex);
```

The ValueToVectorGetSetter is a class implementing the IValueGetter interface, and for any vector can set and get the value at a given index.

An example of use can be found in

```
        Oatc.OpenMI\Examples\SimpleCSharpRiver\RiverModelInterfaceLC.cs
```

4. Migrating an unmanaged engine

When migrating an unmanaged engine, for example a Fortran engine, access to the engine core API from C#/.Net must also be implemented. The steps of the previous chapter are still valid; however, a number of intermediate layers between the unmanaged engine core and the LinkableEngine wrapper must be inserted.

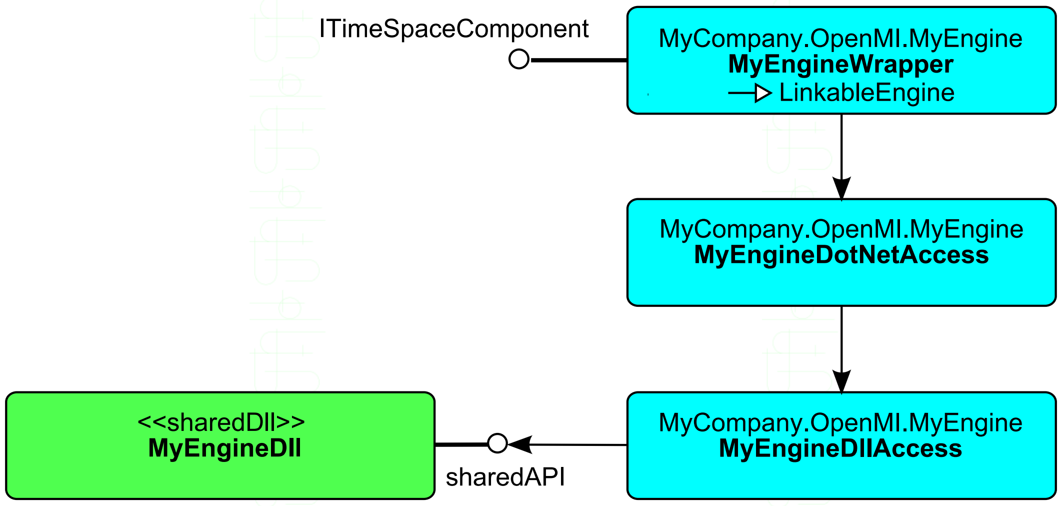


Figure 7 pictures the recommended wrapping pattern for a fortran model engine when using the LinkableEngine.

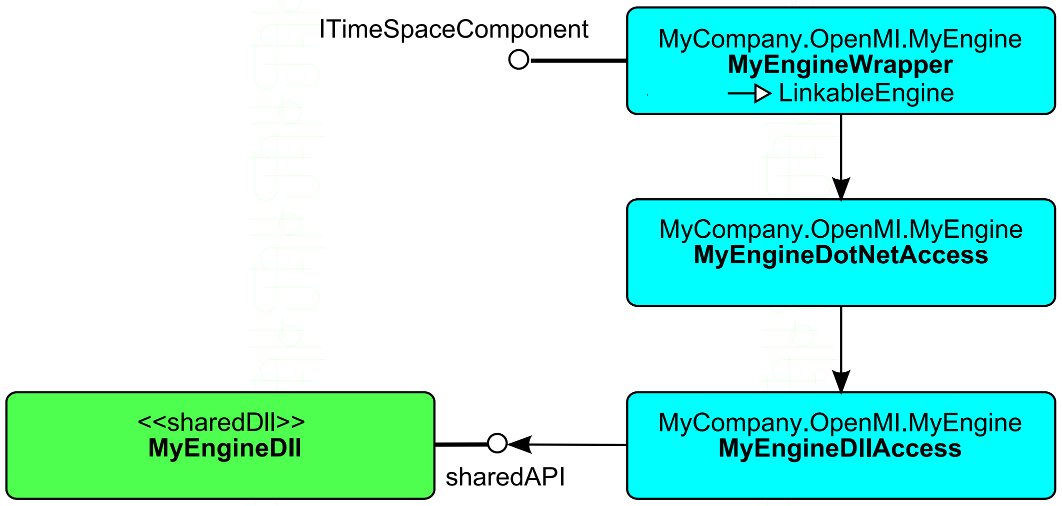


Figure 7 Wrapping pattern with classes and engine core DLL

MyEngineDLL is the unmanaged core engine code (Fortran, Pascal, C++, C), exposing its functionality as a shared dll. This can be created alike step 1 from previous chapter, though the resulting dll is not a .Net dll, but an umnaged dll produced by the Fortran compiler (adding dll export directives on shared methods).

MyEngineDLLAccess is responsible for translating the unmanaged shared API of MyEngineDLL to a .NET (C#) API, using dll import (P/Invoke) from within .NET.

MyEngineDotNetAccess changes calling conventions and adds .NET type of error handling. Calling conventions and exception handling are different for .NET and Fortran.

MyEngineWrapper extends the LinkableEngine and implements the ILinkableComponent interface. The implementation of this is very alike step 2 and 3 from the previous chapter.

The following sections describe how to create the intermediate assemblies.

4.1. Creating the .NET assemblies

After step 1, and before step 2, we must create access classes exposing engine core API to .NET. For this stage, the OpenMI Environment must be installed on your PC.

In our .NET development environment we create one assembly for the access and wrapper classes and it is strongly recommended that one assembly for unit tests is created.

We use the following naming conventions for the access/wrapper assembly:

Assembly name:	MyOrganization.OpenMI.MyModel
Assembly DLL name:	MyOrganization.OpenMI.MyModel.DLL
Namespace:	MyOrganization.OpenMI.MyModel
Class names:	MyModelEngineWrapper MyModelEngineDotNetAccess MyModelEngineDLLAccess

Naming conventions for the test assembly:

Assembly name:	MyOrganization.OpenMI.MyModelTest
Assembly DLL name:	MyOrganization.OpenMI.MyModelTest.DLL
Namespace:	MyOrganization.OpenMI.MyModel
Class names:	MyModelOpenMIComponentTest MyModelEngineWrapperTest MyModelEngineDotNetAccessTest

To execute the tests, install the NUnit test software (see http://www.nunit.org/)

To the wrapper assembly, add the following references:

```
OpenMI.Standard2
Oatc.OpenMI.SDK
Oatc.OpenMI.SDK.ModelWrapper
```

To the test assembly, add also:

```
NUnit.framework
MyOrganisation.OpenMI.MyModel
```

Details are given in the following sections on the implementation of each class.

4.2. MyEngineDLLAccess – Accessing the engine core

The third step is to implement the MyEngineDLLAccess class. The task of the MyEngineDLLAccess class is to make a one-to-one conversion of all exported functions in the engine core DLL, making them available as public .NET methods.

The specific implementation of the MyEngineDLLAccess class depends on the compiler you are using. Start by implementing export methods for the Initialize, PerformTimeStep, Finish and Dispose functions.

Figure 8 shows an example of such an implementation. Note that this implementation corresponds to a particular Fortran compiler; the details may vary between compilers.

```
using System;
using System.Run-time.InteropServices;
using System.Text;
namespace MyOrganisation.OpenMI.MyModel
{
  public static class MyEngineDLLAccess
  {
    [DLLImport(@'C:\MyEngine\bin\MyEngine.DLL',  EntryPoint = 'INITIALIZE',
      SetLastError=true, ExactSpelling = true,
      CallingConvention=CallingConvention.Cdecl)]
    public static extern bool Initialize(string filePath, uint length);

    [DLLImport(@'C:\MyEngine\bin\MyEngine.DLL',  EntryPoint = 'PERFORMTIMESTEP',
       SetLastError=true, ExactSpelling = true,
       CallingConvention=CallingConvention.Cdecl)]
    public static extern bool PerformTimeStep();

    [DLLImport(@'C:\MyEngine\bin\MyEngine.DLL',  EntryPoint = 'FINISH',
       SetLastError=true, ExactSpelling = true,
       CallingConvention=CallingConvention.Cdecl)]
    public static extern bool Finish();
  }
}
```

Figure 8 Implementing the MyEngineDLLAccess

Note that the MyEngineDLLAccess class can not be instantiated. The class and all its methods are static and are accessed directly by referencing the class.

4.3. MyEngineDotNetAccess – using C# conventions

The next step is to implement the MyEngineDotNetAccess class. The MyEngineDotNetAccess has two purposes: to change the calling conventions to C# conventions and to handle errors, i.e., in case of an error, put the message into an exception and throw that.

Figure 9 shows example code for a MyEngineDotNetAccess class that implements the Initialize method, the PerformTimeStep method and the Finish method. In each of these methods the corresponding method in the MyEngineDLLAccess class is called. If the method returns false, the error message from the engine is queried through a

GetNumberOfMessages and GetMessage method (which is assumed to be a part of the shaped egine API), followed by creation of an exception.

The normal convention for Fortran DLLs is that values are returned from the function through reference parameters. The normal convention for C# is that values are returned with the Return statement. In Fortran arrays usually start from index 1 whereas in C# the convention is to start from zero. These differences can be handled in the MyEngineDotNetAccess class.

When the implementation of the methods shown below is completed, we can create and implement the corresponding test class and run the unit test.

```
using System;
using System.Text;
namespace MyOrganisation.OpenMI.MyModel
{
  public class MyEngineDotNetAccess
  {
    public void Initialize(string filePath)
    {
      if(!(MyModelDLL.Initialize(filePath, ((uint)filePath.Length))))
        CreateAndThrowException();
    }
    public void PerformTimeStep()
    {
      if(!(MyModelDLL.PerformTimeStep()))
        CreateAndThrowException();
    }
    public void Finish()
    {
      if(!(MyModel.Finish()))
        CreateAndThrowException();
    }
    private void CreateAndThrowException()
    {
      int numberOfMessages = MyModelDLL.GetNumberOfMessages();
      string message = 'Error Message from MyModel ';
      for (int i = 0; i < numberOfMessages; i++)
      {
        int n = i;
        StringBuilder messageFromCore = new StringBuilder('                 ');
        MyModelDLL.GetMessage(ref n, messageFromCore,
                              (uint) messageFromCore.Length);
        message +='; ' + messageFromCore.ToString().Trim();
      }
      throw new Exception(message);
    }
  }
}
```

Figure 9 Code for the MyEngineDotNetAccess class

4.4. Adding input and output items

Assume we want to be able to add an inflow to a branch in the model engine. We add a function to the engine core API that adds the inflow, and propagate that up to the MyEngineWrapper through the MyEngineDLLAccess and MyEngineDotNetAccess. And we can create an input item in the MyEngineWrapper that uses this new API function whenever inflow is to be added.

Below is an example; parts of source code from the Simple Fortran River

MyEngineDLLAccess – exposing shared engie DLL function to .NET:

```
[DllImport(engineDllFilePath, EntryPoint = "ADDINFLOW",
    SetLastError = true, ExactSpelling = true,
    CallingConvention = CallingConvention.Cdecl)]
public static extern bool AddInflow(ref int branchIndex, ref double inflow);
```

MyEngineDotNetAccess – changing from 0-based to 1-based indices:

```
public void AddInflow(int index, double inflow)
{
  int n = index + 1; // Fortran: 1 based, C# 0 based
  if (!(SimpleRiverEngineDllAccess.AddInflow(ref n, ref inflow)))
    CreateAndThrowException();
}
```

MyEngineWrapper – creates a new input item:

```
public override void Initialize(IArgument[] arguments)
{
  [... other code ...]
  // Create a flow quantity with flow unit and dimension
  Dimension flowDimension = new Dimension();          // m^3/s dimension
  flowDimension.SetPower(DimensionBase.Length, 3);
  flowDimension.SetPower(DimensionBase.Time, -1);

  Unit flowUnit = new Unit("m3/sec", 1, 0, "m3/sec");
  flowUnit.Dimension = flowDimension;

  Quantity inFlowQuantity = new Quantity(flowUnit, "Inflow description", "InFlow");
  inFlowQuantity.ValueType = typeof(double);

  int numberOfNodes = _myEngine.GetNumberOfNodes();
  for (int i = 0; i < numberOfNodes - 1; i++)
  {
    ElementSet branch = new ElementSet("description", "Branch:" + i,
                                        ElementType.PolyLine, "");
    branch.AddElement(new Element("Branch:" + i.ToString()));
    branch.Elements[0].AddVertex(new coordinate( _myEngine.GetXCoordinate(i),
                                         _myEngine.GetYCoordinate(i), 0));
    branch.Elements[0].AddVertex(new Coordinate(_myEngine.GetXCoordinate(i + 1),
                                         _myEngine.GetYCoordinate(i + 1), 0));

    int branchIndex = i;
    EngineDInputItem inflowToBranch = new EngineDInputItem(
                     "Branch:" + i + ":InFlow", inFlowQuantity, branch, this);
    inflowToBranch.ValueSetter = delegate(IValueSet values)
      {
        _simpleRiverEngine.AddInflow(branchIndex, (double)values.GetValue(0,0));
      };

    EngineInputItems.Add(inflowToBranch);
  }
  [... other code ...]
}
```

Above example shows how a delegate calls the AddInflow function to set the value in the engine core.

As mentioned before, adding a method for setting each type of value to the engine can make the core engine API quite verbose. For unmanaged code, there are a number of options to keep the API smaller:

You can return a pointer to the unmanaged data:

```
public IntPtr GetDataPointer(int quantityIndex)
```

With that pointer at hand, you can get/set values directly in the unmanaged memory of the engine core, using unsafe code in C#. Below is example of setting one value of a unmanaged array:

```
IntPtr dataPtr = GetDataPointer(...)
Unsafe {
  double* ptr = (double*) dataPtr
  ptr[index] = some value
}
```

This is only possible for in-process engine cores. Be aware of the unsafe context and that now we must check that array indices are correct ourselves.

Another alternative is to return a pointer to an unmanaged structure/class and use that again when getting/setting values

```
public IntPtr GetStructurePointer(int quantityIndex)
public double[] GetStructureValues(IntPtr structurePointer)
public void SetStructureValues(IntPtr structurePointer, double[] values)
```

5. Testing the component

It is important to test the component to check that it is working correctly. This chapter discusses how to create unit tests of the model and wrapper.

5.1. Unit testing

The testing procedure described here is based on the NUnit test tool. You can download the NUnit user interface and libraries from http://www.NUnit.org. The web page also gives more information about NUnit. Basically, you create a test class for each of the wrapper classes; in the test classes you implement a test method for each public method in the class.

This section describes how to create test assemblies. The test classes used for the Simple River example are shown in Figure 10.

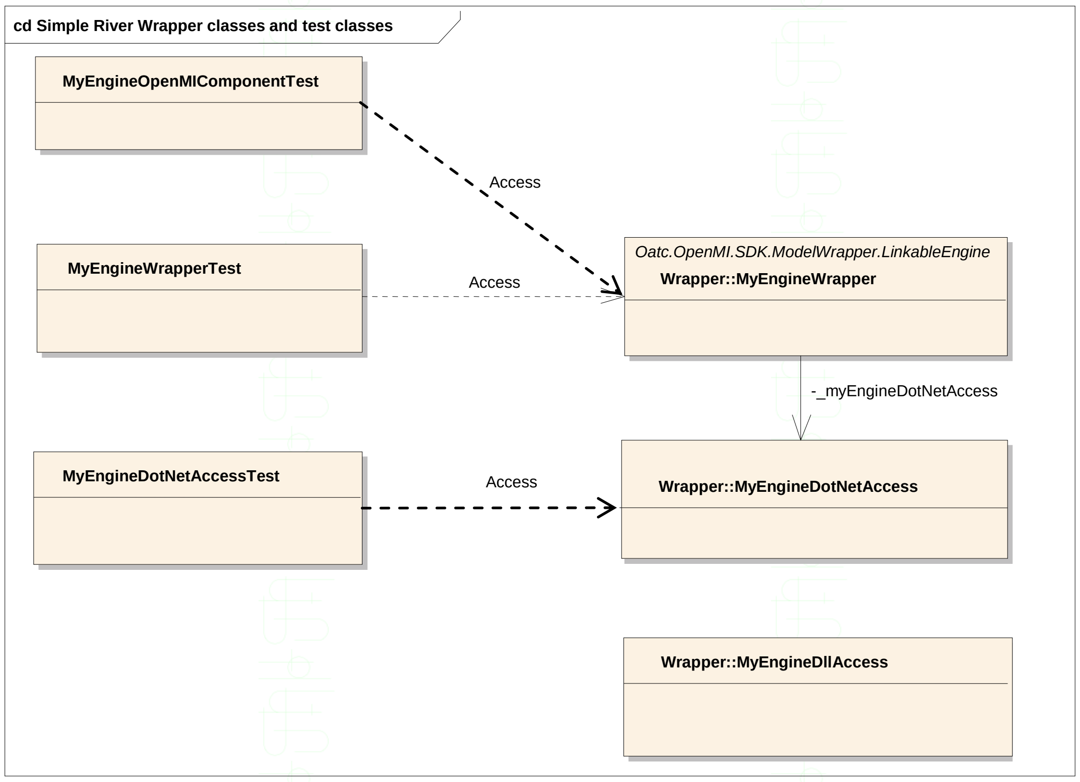

cd Simple River Wrapper classes and test classes
MyEngineOpenMIComponentTest
Access
MyEngineWrapperTest
Access
Oatc.OpenMI.SDK.ModelWrapper.LinkableEngine
Wrapper::MyEngineWrapper
-_myEngineDotNetAccess
MyEngineDotNetAccessTest
Access
Wrapper::MyEngineDotNetAccess
Wrapper::MyEngineDllAccess

Figure 10 Wrapper and test classes for the model

There is a one-to-one relation between the wrapper classes and the test classes, with two exceptions. There is no test class for the SimpleRiverEngineDLLAccess class and there is one additional class called MyEngineOpenMIComponentTests. The test class for the SimpleRiverEngineDLLAccess class was left out because every method in the SimpleRiverEngineDotNetAccess class will invoke the corresponding method in the SimpleRiverEngineDLLAccess class; therefore testing all methods in the SimpleRiverEngineDotNetAccess class will be sufficient.

The main idea of unit testing is to create very simple code that will test each method in the classes. However, it can also be useful to make some more advanced tests that are actually running full simulations. This can be done in an additional UseCaseTests class or similar, which can test and verify that the defined use cases are fullfilled.

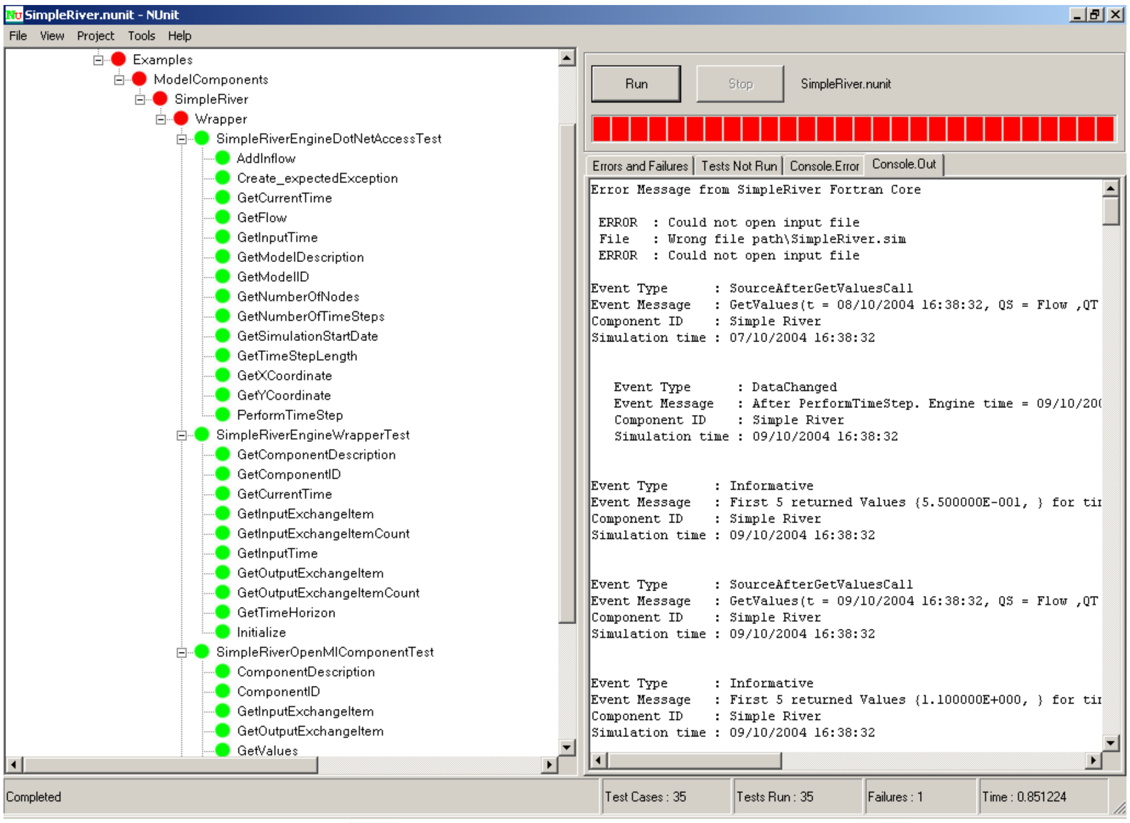
Figure 11 contains sample test code for the GetModelID method implementation in the Simple Fortran River model. Figure 12 shows the NUnit interface.

```
using System;
using Oatc.OpenMI.Examples.SimpleFortranRiver.Wrapper;
using NUnit.Framework;
namespace Oatc.OpenMI.Examples.ModelComponents.SimpleRiver.Wrapper.UnitTest
{
  [TestFixture]
  public class SimpleRiverEngineDotNetAccessTest
  {
    SimpleRiverEngineDotNetAccess _simpleRiverEngineDotNetAccess;
    string _filePath;
    string _simFileName;

    [SetUp]
    public void Init()
    {
      _simpleRiverEngineDotNetAccess = new SimpleRiverEngineDotNetAccess();
      _filePath = @"..\..\..\..\Data";
      _simFileName = @"SimpleRiver.sim";
      _simpleRiverEngineDotNetAccess.Initialize(_filePath, _simFileName);
    }

    [Test]
    public void GetModelID()
    {
      _simpleRiverEngineDotNetAccess.Initialize(_filePath, _simFileName);
      Assert.AreEqual("The river Rhine", _simpleRiverEngineDotNetAccess.GetModelID());
      _simpleRiverEngineDotNetAccess.Finish();
    }
  }
}
```

Figure 11 Test code for the GetModelID method implementation



```
Error Message from SimpleRiver Fortran Core

 ERROR  : Could not open input file
 File   : Wrong file path\SimpleRiver.sim
 ERROR  : Could not open input file

Event Type      : SourceAfterGetValuesCall
Event Message   : GetValues(t = 08/10/2004 16:38:32, QS = Flow ,QT
Component ID    : Simple River
Simulation time : 07/10/2004 16:38:32


   Event Type      : DataChanged
   Event Message   : After PerformTimeStep. Engine time = 09/10/200
   Component ID    : Simple River
   Simulation time : 09/10/2004 16:38:32


Event Type      : Informative
Event Message   : First 5 returned Values (5.500000E-001, ) for tir
Component ID    : Simple River
Simulation time : 09/10/2004 16:38:32


Event Type      : SourceAfterGetValuesCall
Event Message   : GetValues(t = 09/10/2004 16:38:32, QS = Flow ,QT
Component ID    : Simple River
Simulation time : 09/10/2004 16:38:32


Event Type      : Informative
Event Message   : First 5 returned Values (1.100000E+000, ) for tir
Component ID    : Simple River
Simulation time : 09/10/2004 16:38:32
```

Completed | Test Cases : 35 | Tests Run : 35 | Failures : 1 | Time : 0.851224

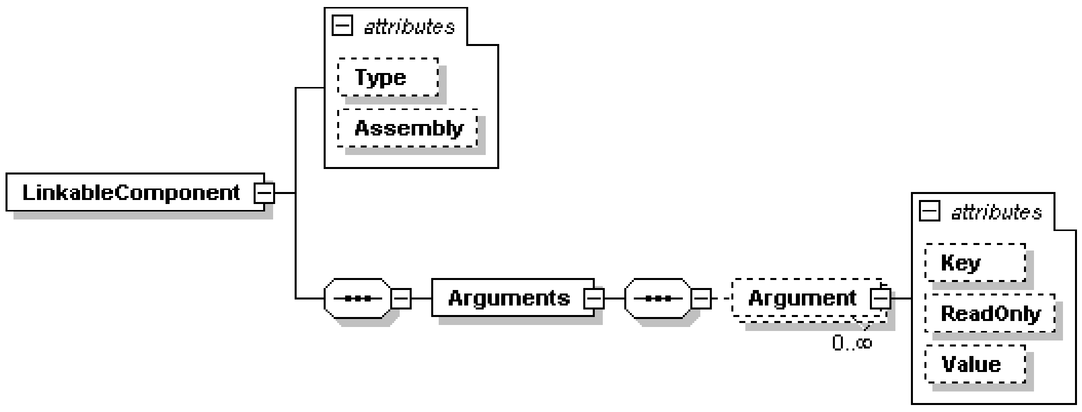
Figure 12 NUnit User interface with Simple River wrapper classes loaded

6. The OMI file

The OMI file defines the entry point to a LinkableComponent. It contains information on the software unit to instantiate and the arguments to provide at initialization. This file makes it possible for a user interface to deploy your model, for example.

6.1. Structure of the OMI file

The structure of the OMI file is defined by the LinkableComponent.XSD.



The Xml Schema Definition of an OMI-file

```
<?xml version="1.0" ?>

<xs:schema id="LinkableComponent" targetNamespace="http://www.openmi.org/LinkableComponent.xsd"

  xmlns:mstns="http://www.openmi.org/LinkableComponent.xsd"
xmlns="http://www.openmi.org/LinkableComponent.xsd"

  xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"

  attributeFormDefault="qualified" elementFormDefault="qualified">

  <xs:element name="LinkableComponent">

    <xs:complexType>

      <xs:sequence>

        <xs:element name="Arguments" minOccurs="1" maxOccurs="1">

          <xs:complexType>

            <xs:sequence>

              <xs:element name="Argument" minOccurs="0" maxOccurs="unbounded">

                <xs:complexType>
```

```
                  <xs:attribute name="Key" form="unqualified" type="xs:string" />

                  <xs:attribute name="ReadOnly" form="unqualified" type="xs:boolean" use="optional" />

                  <xs:attribute name="Value" form="unqualified" type="xs:string" />

                </xs:complexType>

              </xs:element>

            </xs:sequence>

          </xs:complexType>

        </xs:element>

      </xs:sequence>

      <xs:attribute name="Type" form="unqualified" type="xs:string" />

      <xs:attribute name="Assembly" form="unqualified" type="xs:string" use="optional" />

    </xs:complexType>

  </xs:element>

</xs:schema>
```

XML-look of the OMI file

```
<?xml version="1.0"?>

<LinkableComponent                                      Type="Deltares.OpenMI.Wrapper.DLinkableComponent
     Assembly="..\..\bin\Deltares.OpenMI.Wrapper.dll"
         xmlns="http://www.openmi.org/LinkableComponent.xsd">

  <Arguments>

    <Argument Key="Model" ReadOnly="true" Value="CF" />

    <Argument Key="Schematization" ReadOnly="true" Value=".\CmtWork\sobesim.fnm" />

  </Arguments>

</LinkableComponent>
```

7. Performance issues

This chapter discusses some of the issues that may affect performance when migrating models to the OpenMI.

7.1. Memory consumption

When running a set of linked models on one computer it is important to realize that several models will be kept in memory at the same time and that the overall computation time is the sum of the computation time for the individual models. Therefore it is crucial that the individual models should consume as little memory as possible. When the amount of memory used by all programs exceeds the amount of physical memory in the computer, the computer starts swapping chunks of memory to disk. This is very slow and can lead to severe performance degradation.

7.2. System processes

Although all linkable components run in one system process, the actual computation can either be run in the same process or in a different process. Running the computation in the same process is preferable because communication between processes can be up to a hundred times slower than in-process communication. However, this is not always possible, especially when Fortran code with many global variables is used. In that case there is no other option than to run the Fortran code in a separate process

8. The Simple River example

A Simple River model engine was developed as an example of model migration. The model engine is programmed in Fortran and is a very simple conceptual river model.

The Simple River consists of nodes and branches, as shown in Figure 13. For each timestep, the inflow to each node is obtained from a boundary-input file. These flow rates are multiplied by the timestep length and added to the storage in each node. Then, starting from the upstream end, for each node some of the water is leaked while the remainder of the water is moved to its downstream node and the flow rate in each branch is calculated.

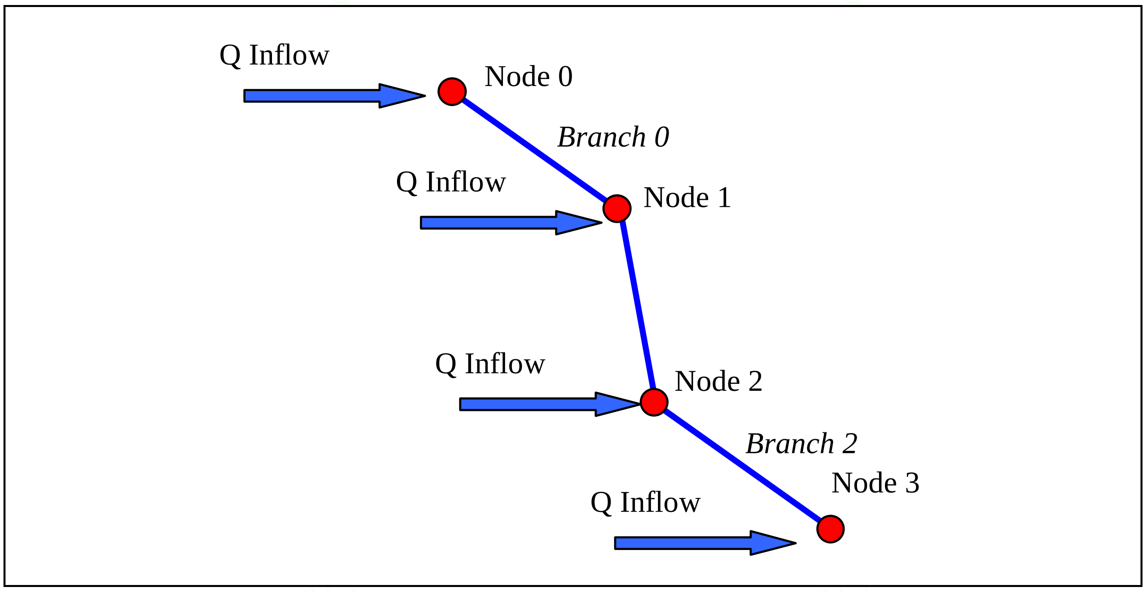


Figure 13 Simple River network

The Simple River engine reads data from three input files, which contain information about the inflow to the river nodes (boundary file), the simulation period and timestep length (simulation file) and the river network (network file) – see Figure 14 .

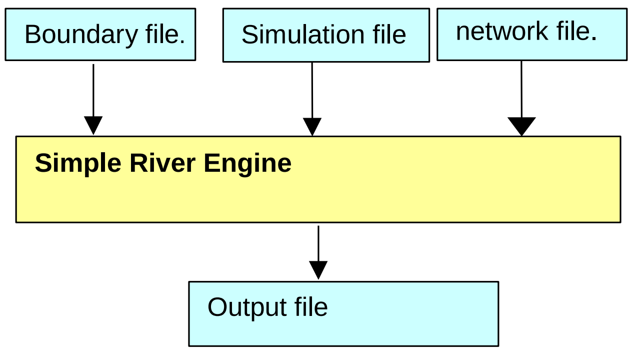


Figure 14 Simple River input and output files

The full source code for the Simple River model, the associated wrappers and the test classes used to migrate the model is available at www.OpenMI.org.

9. Use cases

Use cases (examples of how software is used) have become very popular in software development. There are no formal requirements for defining a use case. However, what makes a use case different from an example is that a use case is more detailed and well defined. Most importantly, a use case must be formulated in such a way that, after completion of software development, you can unambiguously determine whether the use case is fullfilled or not. The advantage of use cases is that they are easily understood both by the software developer and the software user.

At the beginning of the development process it is important to define a number of use cases. It is also important that the set of use cases at any time, in all areas of the software development, reflects the current target. If a particular use case cannot be fulfilled it should be modified or removed.

Two use cases for the migrated Simple River model are given below. The use cases give a step-by-step description of how a user will use the models.

9.1. Connecting to other rivers

In the first use case, the Simple River model is connected to another OpenMI-compatible river model, as show in Figure 15.

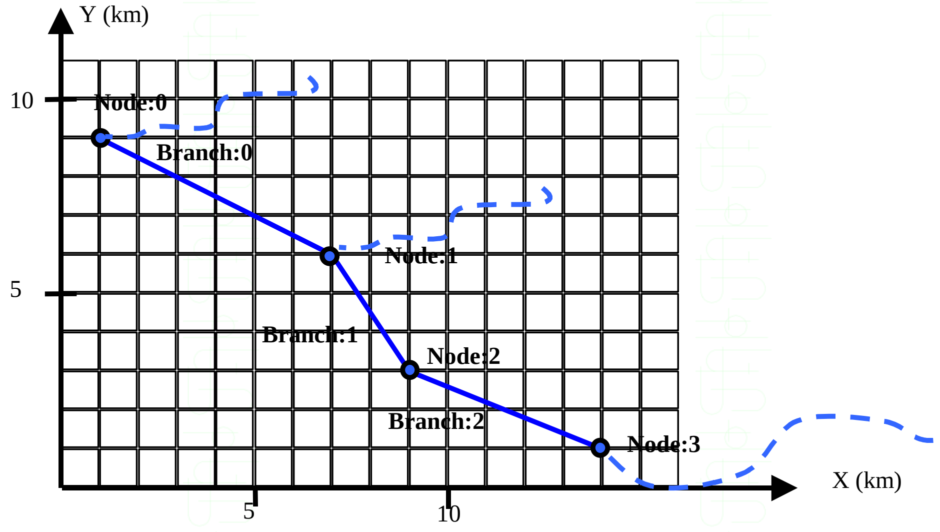


Figure 15 Use case Connecting to other rivers

Preconditions:

- The model user has the OpenMI-compliant Simple River model installed on his PC.
- The model user has input files for the Simple River model available on his PC.
- The model user has an OpenMI configuration user interface installed on his PC.
- The model user has another OpenMI-compliant river model (including required data files) available on his PC.

Success guarantee (postconditions):

- All models have generated correct results.

Main success scenario:

1. The model user loads the OpenMI GUI on the PC.
2. The model user uses the GUI to browse for available LinkableComponents.
3. The model user finds the Simple River OMI file and the OMI file for the other river model.
4. The model user loads the two files (components) into the GUI.
5. The model user creates a unidirectional and ID-based link from the downstream node in the other river model to the upstream node in the Simple River.
6. The model user selects input and output exchange items for the link (input quantity for the Simple River is 'Inflow').
7. The model user defines the simulation period.
8. The model user runs the simulation.

Extensions to the use case provide alternative flows. Here, the flow splits from step 5 into two alternatives.

First alternative, switching up/downstream order of the two models:

5a. The model user creates a unidirectional and ID-based link from the downstream branch in the Simple River model to the upstream node in the other river model.
6a. The model user selects input and output exchange items for the link (output quantity for the Simple River is 'flow').

Second alternative, linking to an internal node in the Simple River:

5b. The model user creates a unidirectional and ID-based link from the downstream branch in the other river model to an internal node in the Simple River model.
6b. The model user selects input and output exchange items for the link (input quantity for the Simple River is 'Inflow').

9.2. Inflow from geo-referenced catchment database

In the second use case, the inflow for the Simple River model comes from an OpenMI-compliant runoff database Figure 16.

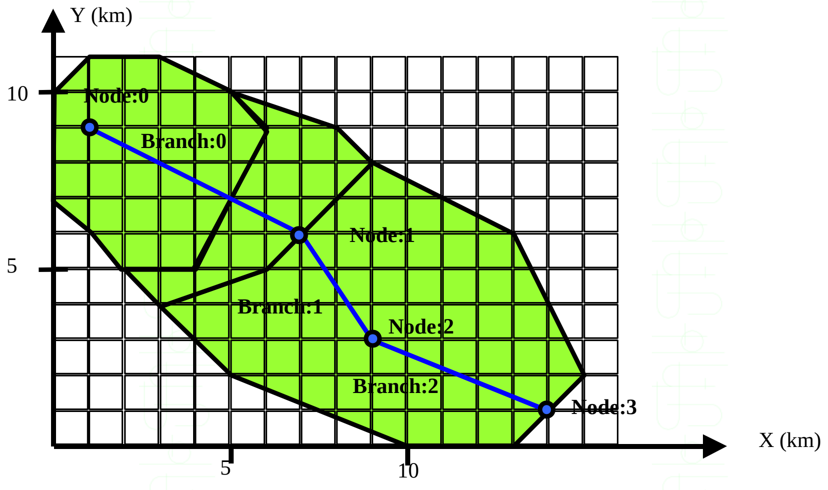


Figure 16 Use case: Inflow from catchments

Preconditions:

- The model user has the OpenMI-compliant Simple River model installed on his PC.
- The model user has input files for the Simple River model available on his PC.
- The model user has an OpenMI configuration user interface installed on his PC.
- The model user has an OpenMI-compliant runoff database (including required data files) available on his PC.

Success guarantee (postconditions):

- All models have generated correct results.

Main success scenario:

1. The model user loads the OpenMI GUI on the PC.
2. The model user uses the GUI to browse for available LinkableCompnents.
3. The model user finds the Simple River OMI file.
4. The model user finds the OMI file for the runoff database.
5. The model user loads the two files (components) into the GUI.
6. The model user creates a unidirectional and geo-referenced link from the runoff database to 'All Branches' input exchange item in the Simple River model.
7. The model user selects input and output exchange items for the link (input quantity for the Simple River is 'Inflow').
8. The model user defines the simulation period.
9. The model user runs the simulation.

Note that the runoff for a particular polygon is distributed on the river branches depending on how large a portion of a branch is included in each polygon. This type of boundary condition, where water is added to branches, was not possible in the original Simple River engine. The Simple River engine is (as a result of the migration) extended with this feature, simply because such a boundary condition becomes a possibility when running in the OpenMI environment.

9.2.1. Defining exchange items

Exchange items are combined information about what can be exchanged and where the exchanged item applies. An input exchange item could define that inflow can be accepted on nodes or river branches. An output exchange item could specify that flow can be provided on branches. The Quantity identifies what can be exchanged (e.g. ‘Flow’) and the ElementSet identifies where this quantity applies (e.g. ‘Node:1’).

The next step is to define input and output exchange items. The exchange items that are required in order to run the use cases are listed in Table 1.

Table 1 Required exchange items for use cases 1 and 2

ElementSet.ID	Type	Quantity.ID	Unit	IsInput	IsOutput	Use case
‘Branch:0’	Polyline	Flow	M3/sec	No	Yes	1
‘Branch:1’	Polyline	Flow	M3/sec	No	Yes	1
‘Branch:2’	Polyline	Flow	M3/sec	No	yes	1
‘Node:0’	IDBased	Inflow	M3/sec	Yes	No	1
‘Node:1’	IDBased	Inflow	M3/sec	Yes	No	1
‘Node:2’	IDBased	Inflow	M3/sec	Yes	No	1
‘Node:3’	IDBased	Inflow	M3/sec	Yes	No	1
‘Branch:0’	Polyline	Inflow	M3/sec	Yes	No	
‘Branch:1’	Polyline	Inflow	M3/sec	yes	No	
‘Branch:2’	Polyline	Inflow	M3/sec	yes	No	
‘All Branches’	Polyline	Inflow	M3/sec	yes	No	2

Naturally, the exchange items should not be limited to a particular network, but for the purpose of planning the migration it is easier to start out with a specific case and then generalize this case when it come to the more detailed design.

10. Migration of the Simple River

The previous chapter described the steps involved in migrating a model to the OpenMI. This chapter shows how the migrated code is implemented for the Simple River example.

To view the source code you can visit the OpenMI repository web interface at:

http://openmi.svn.sourceforge.net/viewvc/openmi/trunk/src/csharp/Oatc.OpenMI/Examples/SimpleFortranRiver/

or you can download the source code from the source repository, see

http://public.deltares.nl/display/OPENMI/How+to+download+the+most+recent+source+code

10.1. The Simple River wrapper

The Simple River model uses the migration pattern described in this document. The following gives a detailed explanation of how the Simple River wrapper works in terms of the wrapper classes.

10.2. Implementation of the Initialize method

The SimpleRiverEngineWrapper has two internal fields:

```
IList<EngineInputItem> EngineInputItems;
IList<EngineOutputItem> EngineOutputItems;
```

These list of input and output items are populated in the Initialize method. The EngineInputItem and EngineOutputItem contain functionality for getting/setting values, and automatically update the model and the items when required.

The implementation of the Initialize method requires that a number of methods are added in the MyEngineDotNetAccess class, the MyEngineDLLAccess class and the engine core DLL.

The sequence diagram in Figure 17 illustrates the communication with the other wrapper classes when the Initialize method is invoked. The EngineDLL is not included in the diagram since there is a one-to-one relation between the EngineDLL and the EngineDLLAccess classes. In other words, each time a method is called in the EngineDLLAccess the corresponding function is called in the EngineDLL.

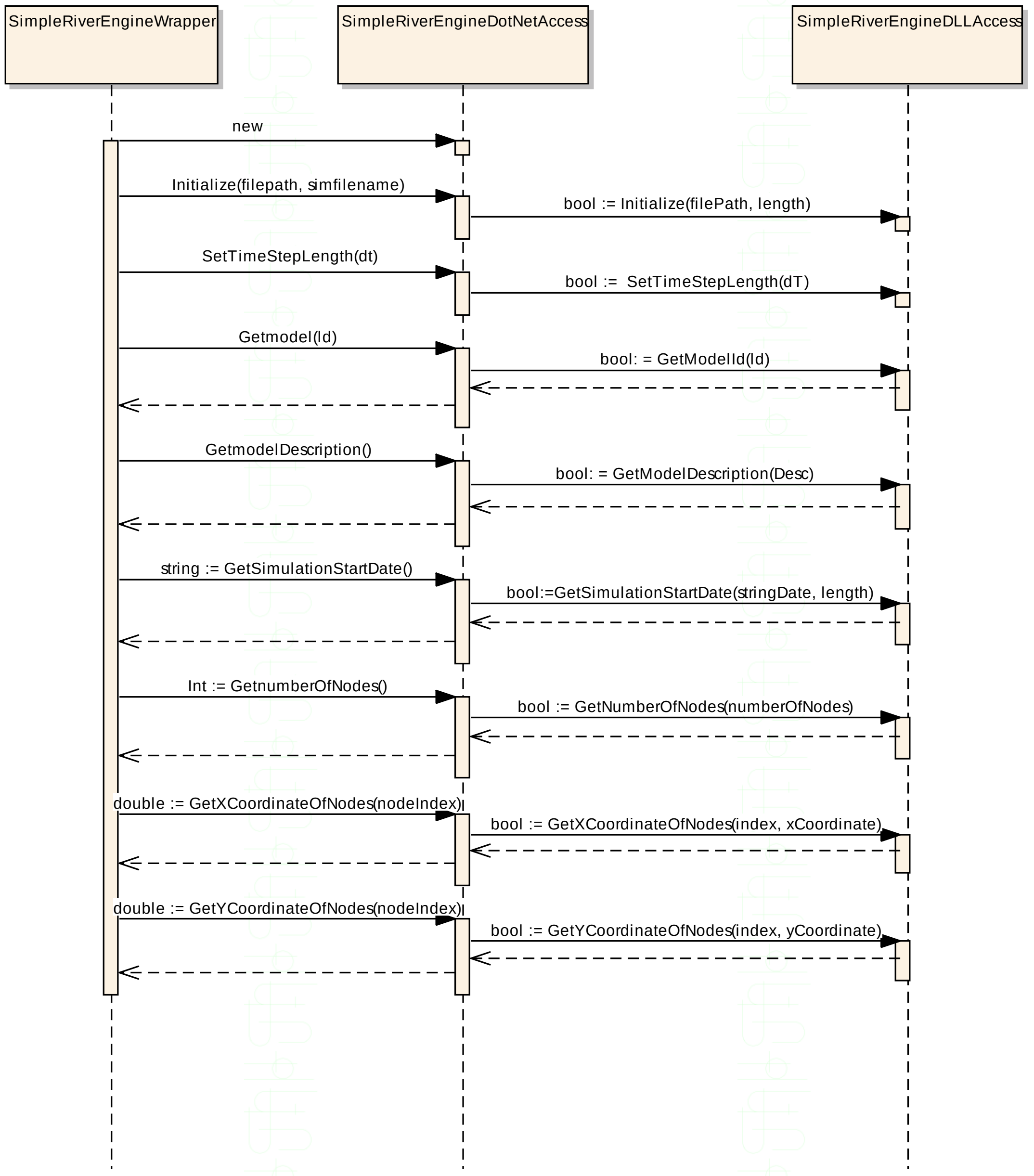

SimpleRiverEngineWrapper
SimpleRiverEngineDotNetAccess
SimpleRiverEngineDLLAccess
new
Initialize(filepath, simfilename)
bool := Initialize(filePath, length)
SetTimeStepLength(dt)
bool := SetTimeStepLength(dT)
Getmodel(Id)
bool: = GetModelId(Id)
GetmodelDescription()
bool: = GetModelDescription(Desc)
string := GetSimulationStartDate()
bool:=GetSimulationStartDate(stringDate, length)
Int := GetnumberOfNodes()
bool := GetNumberOfNodes(numberOfNodes)
double := GetXCoordinateOfNodes(nodeIndex)
bool := GetXCoordinateOfNodes(index, xCoordinate)
double := GetYCoordinateOfNodes(nodeIndex)
bool := GetYCoordinateOfNodes(index, yCoordinate)

Figure 17 Calling sequence for the initialize method

10.3. Implementation of getting/setting values in input and output items

The calling sequence for updating an input item is shown in Figure 18, for adding additional inflow.

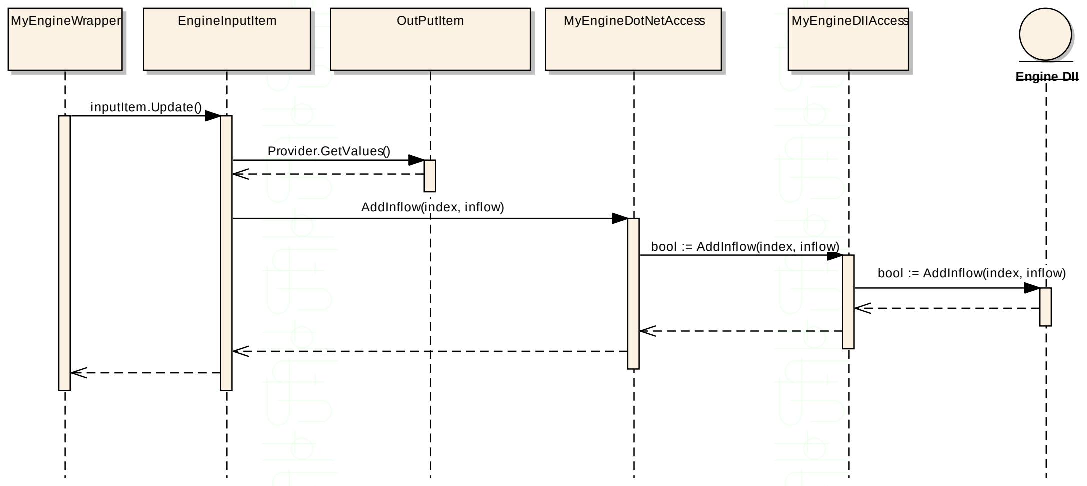


Figure 18 Calling sequence for the InputItem.Update method

For the Simple River model, inflow is interpreted as additional inflow, which means that the inflow already received from other sources (the boundary inflow) is not overwritten. The inflow is added to the current storage in the nodes.

If the inflow is going to the branches, the water is added to the downstream node for each branch. If the inflow is going to the nodes, the water is simply added to the storage of the node.

1.4 note

The ids of the Quantity and the ElementSet have lost their role in 2.0: In 1.4 the id combination defined which data that was to be exchanged and the ModelWrapper handled the data transfer based on the ids. In 2.0 the data exchange are initiated on the exchange items directly, and each exchange item is responsible for the data exchange. To mimic 1.4 behaviors in 2.0, you can use the `LinkableGetSetEngine` instead of the `LinkableEngine`: When using the `EngineEInputItem` and `EngineEOutputItem`, they will delegate the work back to the linkable engine `GetEngineValues(...)` and `SetEngineValues(...)` methods, which then can handle the data exchange based on the quantity and elementset ids, as in 1.4.

10.4. Implementation of the remaining methods

Implementation of the remaining methods is not complicated. On the sequence diagram in Figure 19 you can see how generally each method is accessing the other engine wrapper classes.

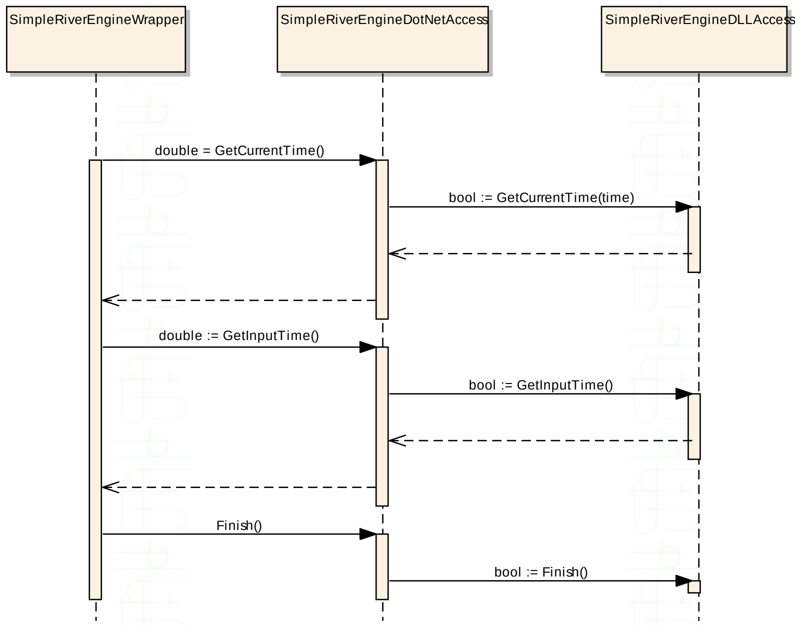

SimpleRiverEngineWrapper
SimpleRiverEngineDotNetAccess
SimpleRiverEngineDLLAccess
double = GetCurrentTime()
bool := GetCurrentTime(time)
double := GetInputTime()
bool := GetInputTime()
Finish()
bool := Finish()

Figure 19 Calling sequence for Simple River Initialize

Note that for some of the methods the full implementation is done in the SimpleRiverEngineWrapper class. The methods GetCurrentTime and GetInputTime are all invoking the GetCurrentTime method in the SimpleRiverDotNetAccess class. The returned time is the engine local time. This time is converted to the ModifiedJulianTime in the SimpleRiverEngineWrapper, see Figure 20.

```
public ITime GetCurrentTime()
{
  double time = _simulationStartTime
            + _simpleRiverEngine.GetCurrentTime() / ((double)(24 * 3600));
  return new Time(time);
}
```

Figure 20 Handling of time