OpenMI Wrapping Models

Rob Brinkman
WL | Delft Hydraulics

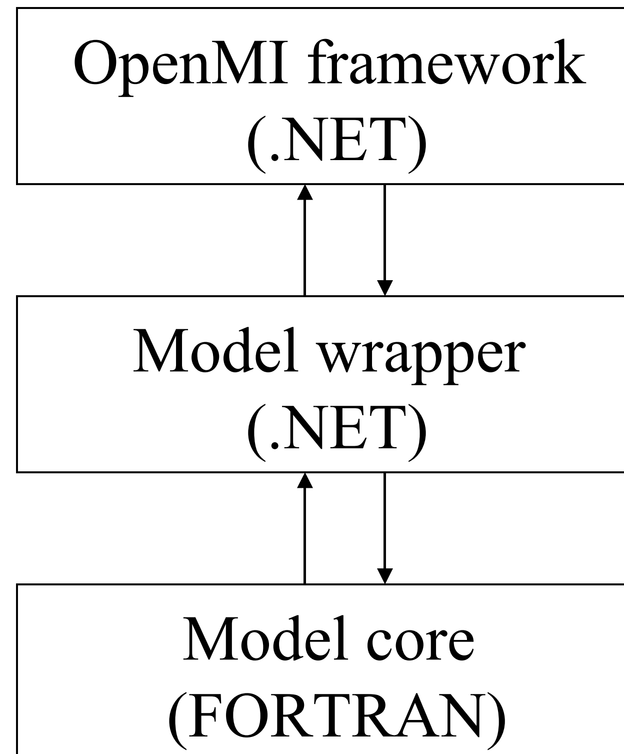
Why migrate?

- Many hydrological models result of years of development
- Amount of money invested in models is significant
- Migrating an existing model is often the only choice

Why not rewrite everything from scratch?

- Very time-consuming
- Risk of introducing new bugs in the code
- The investment is usually not in the code but in the schematisations
- It is easier and cheaper to “wrap” existing code with a HarmonIT wrapper

Typical scenario



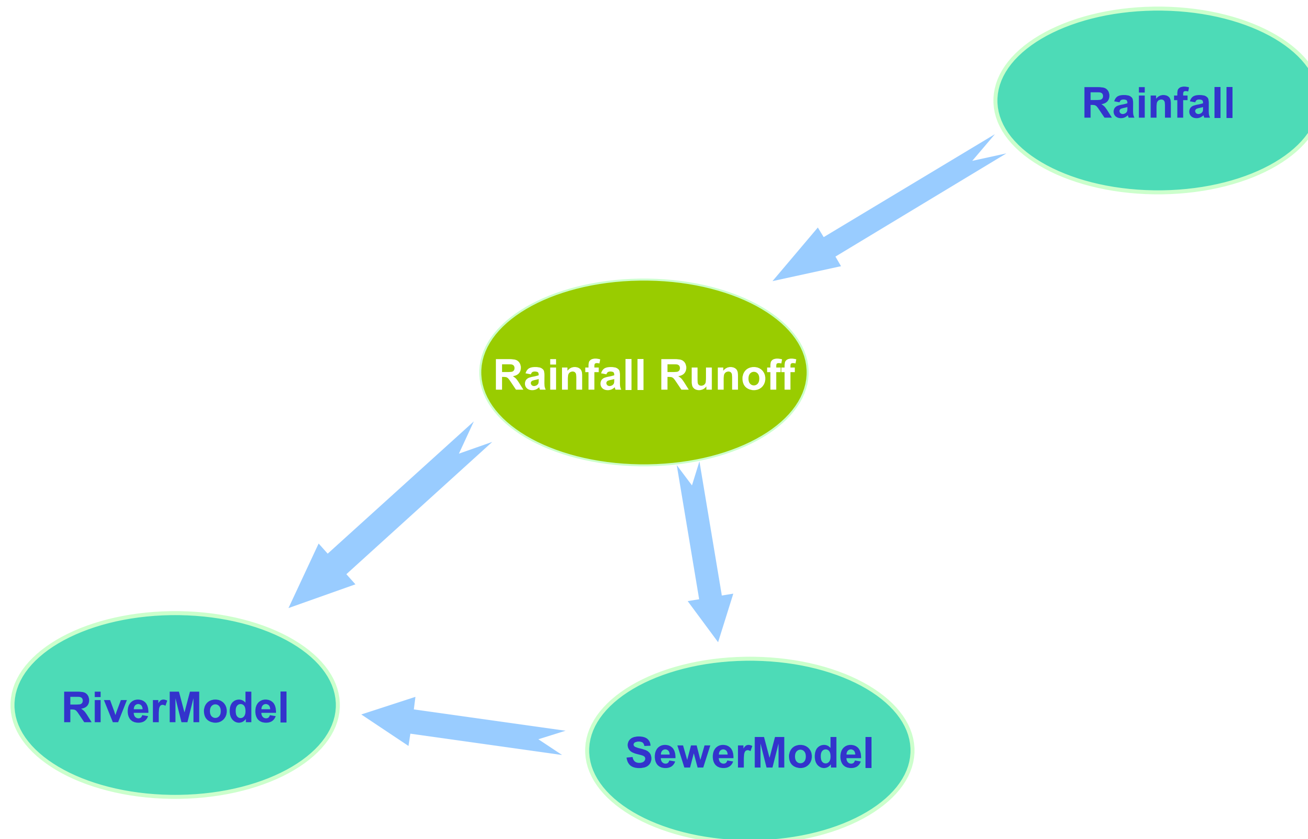
Typical wrapper

- Many tools available to help developer
  - Buffering
  - Temporal interpolation
  - Spatial interpolation
  - Unit conversion
  - Handling links
    - Adding links
    - Retrieving values from other models

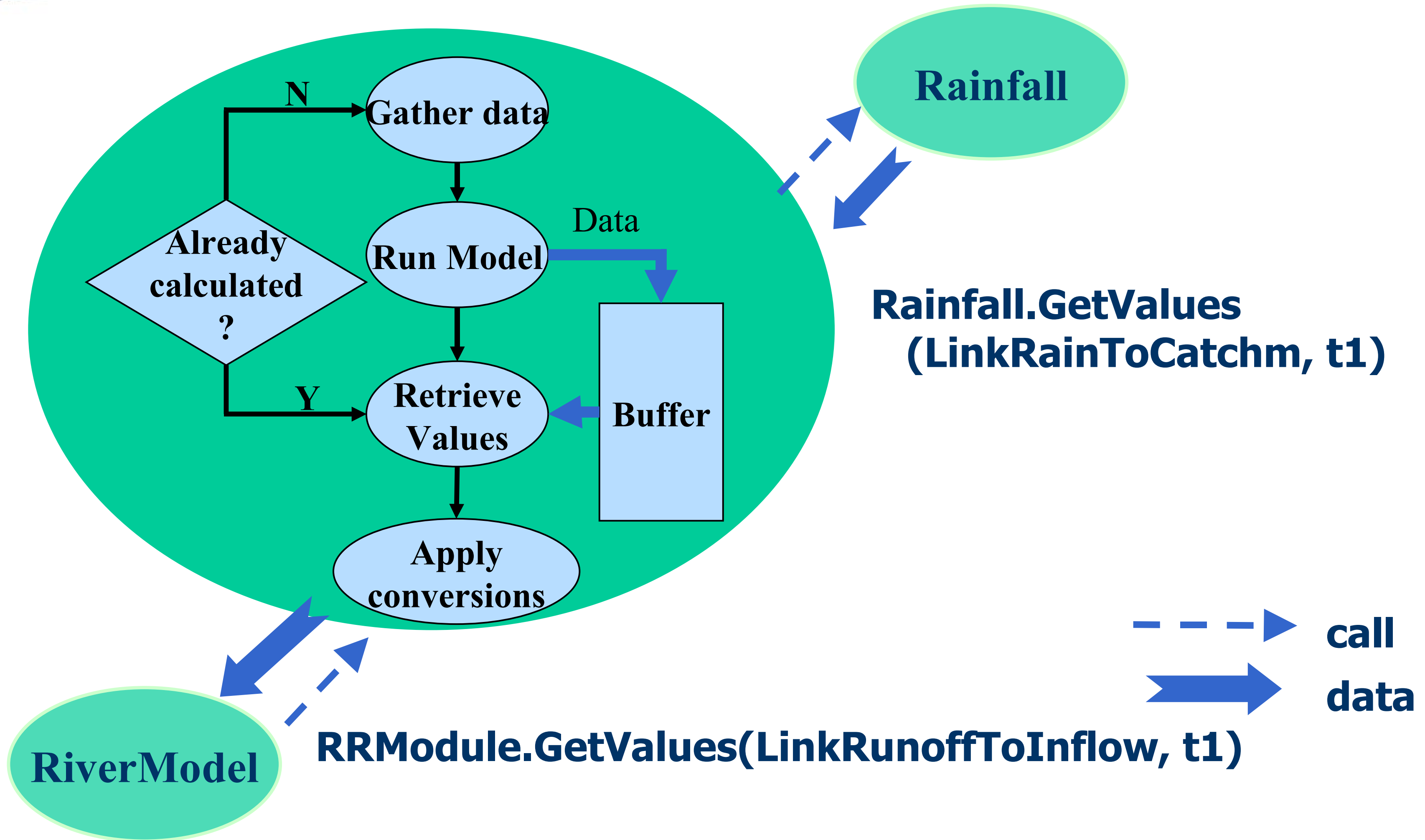
Different types of components

- Algebraic / Time independent
  - compute required valueSet
- Database
  - find (/ interpolate) valueSet
- Time dependent (numerical) models:
  - perform computational step, store externally required results, return valueSet

Implementing GetValues



LinkableEngine class



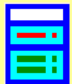
Rainfall.GetValues (LinkRainToCatchm, t1)

RRModule.GetValues(LinkRunoffToInflow, t1)

Org.OpenMI.Utilities

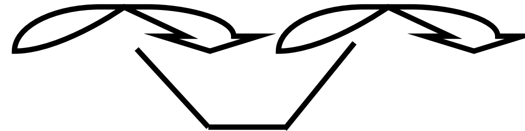
HarmoniT

Org.OpenMI.Utilities.Buffer

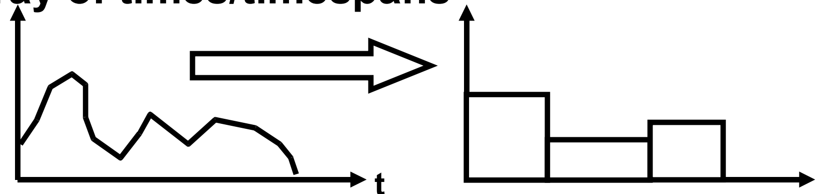


Buffer

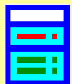
Buffers results from the engine core



Mapping of values associated to one array of times /timespans to values represented on another array of times/timespans

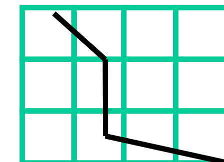


Org.OpenMI.Utilities.Spatial

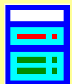


ElementMapper

Mapping of values associated to one ElementSet to be represented on another ElementSet



Org.OpenMI.Utilities.Wrapper



Wrapper

Generic wrapper suited for time stepping model engines

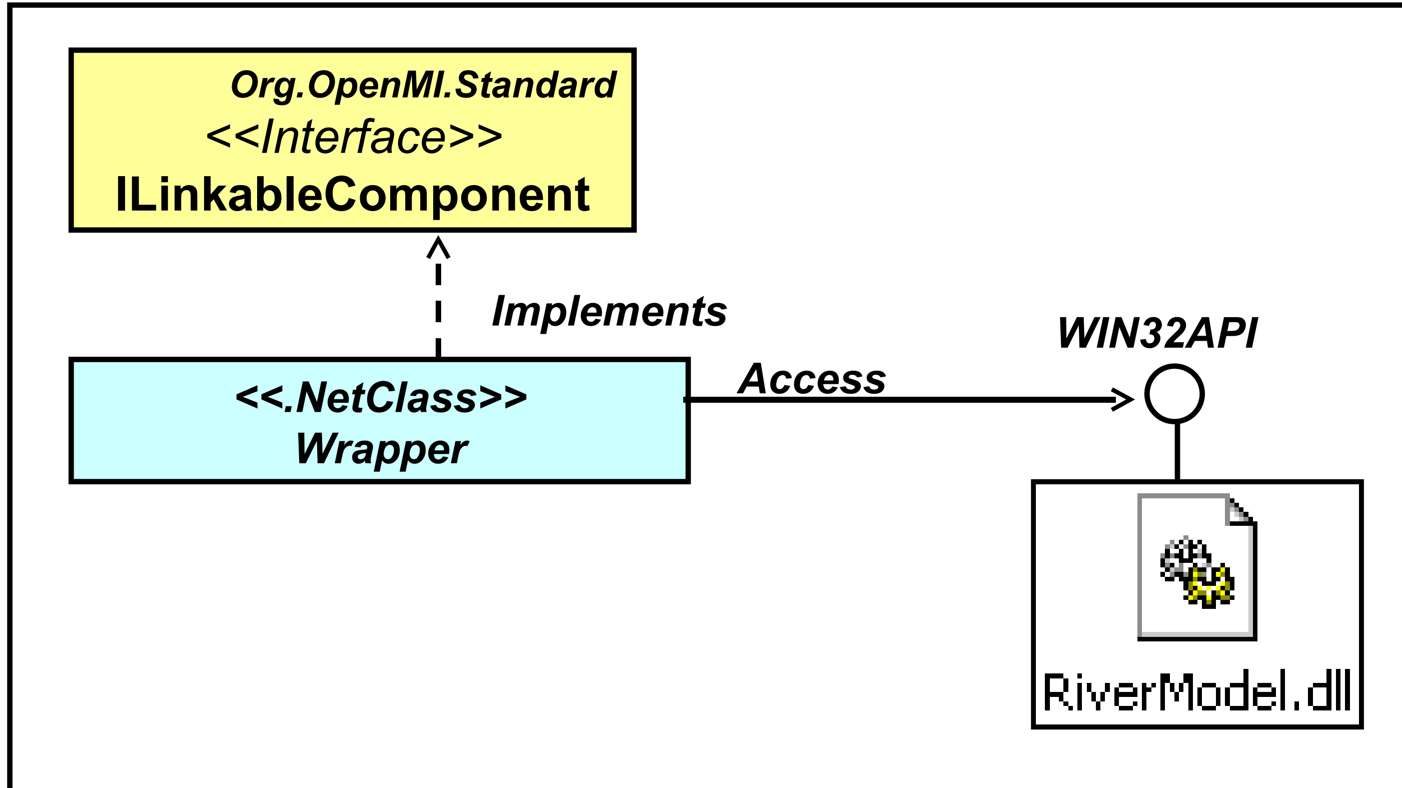
org.OpenMI.Utilities.Wrapper

- Provides a default implementation of the ILinkableComponent interface

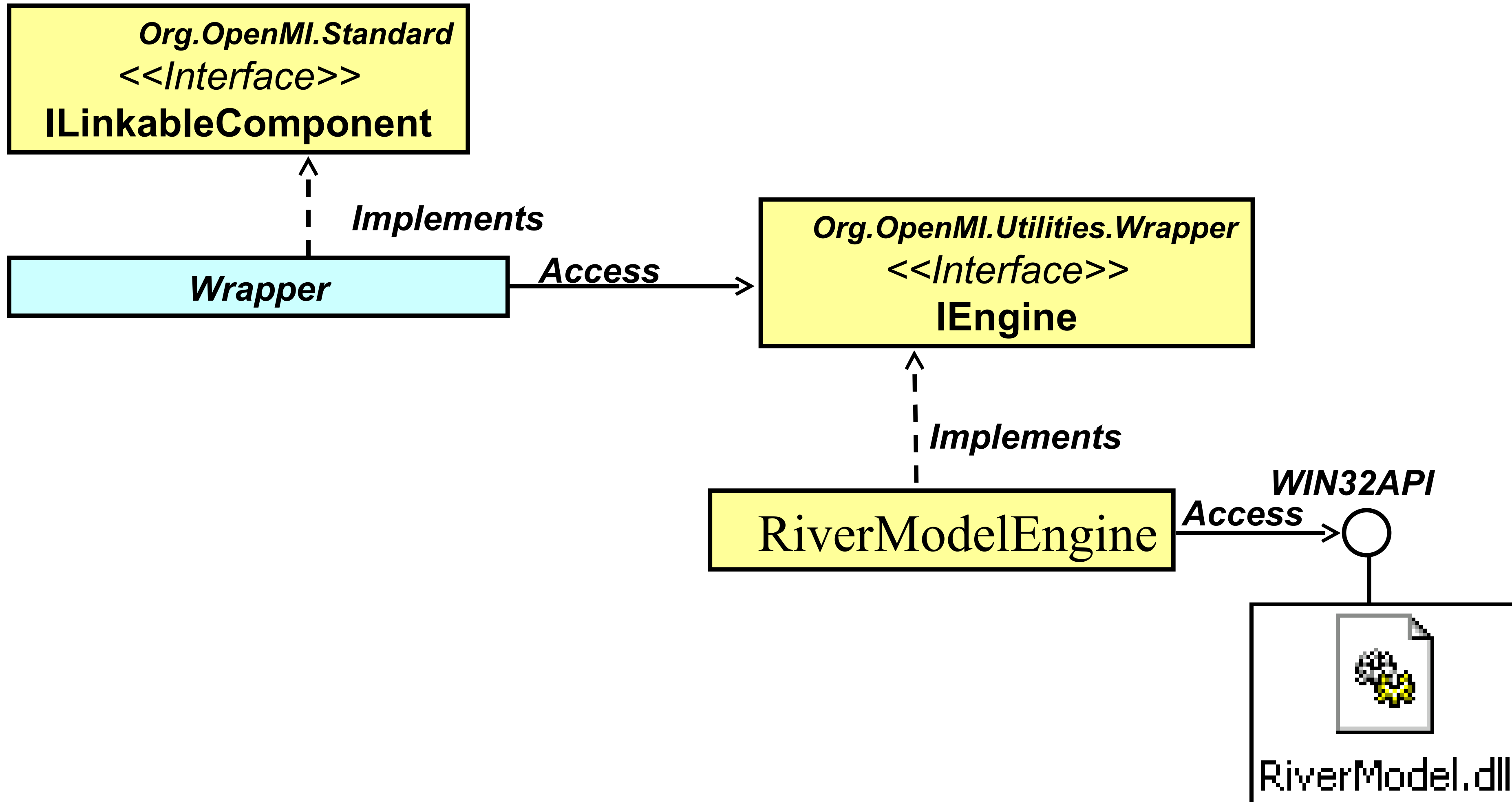
- Administration of incoming and outgoing links
- Event handling

- Buffering
- Temporal interpolations, aggregations, extrapolations
- Spatial interpolations, aggregations, extrapolations

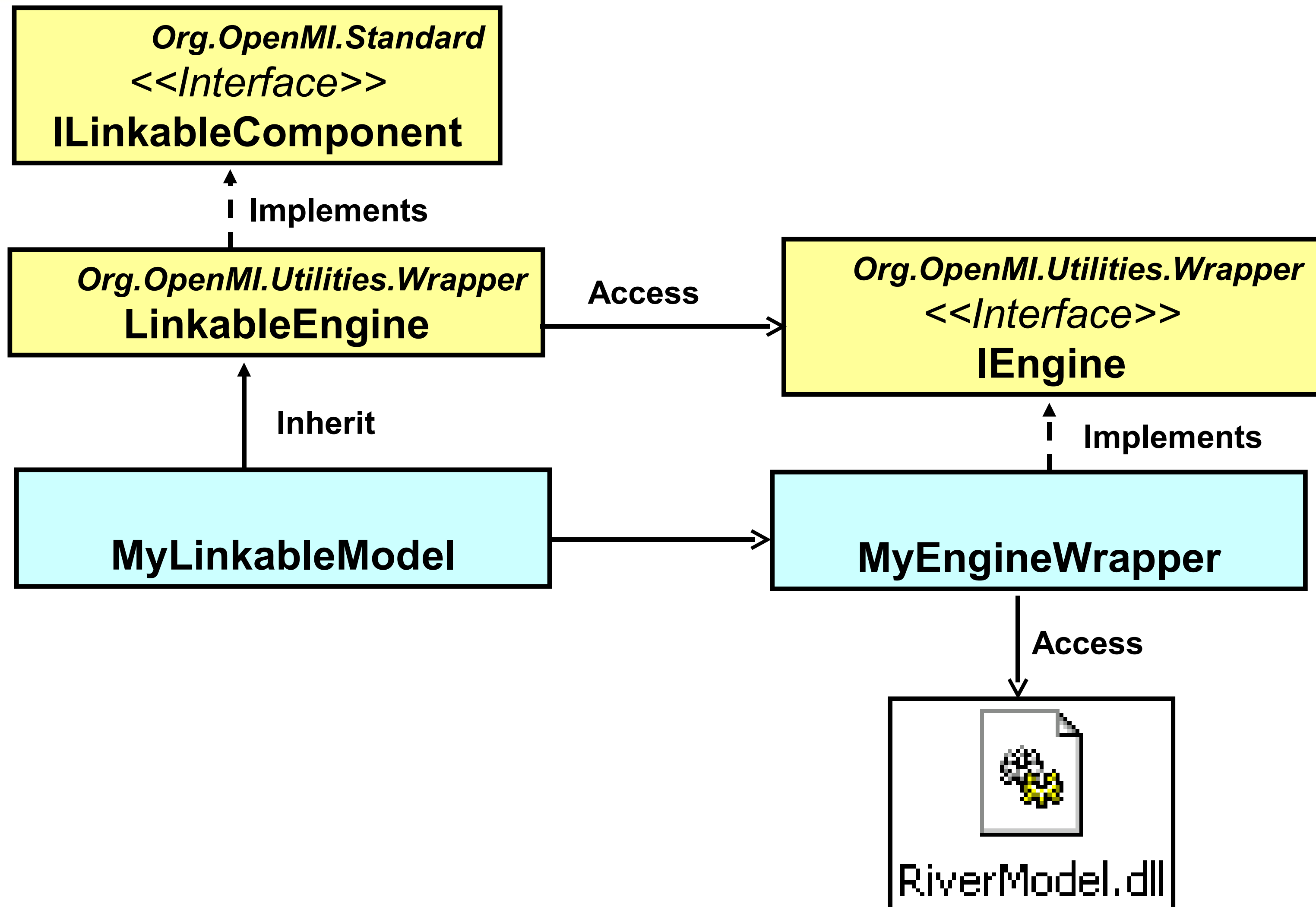
Wrapping



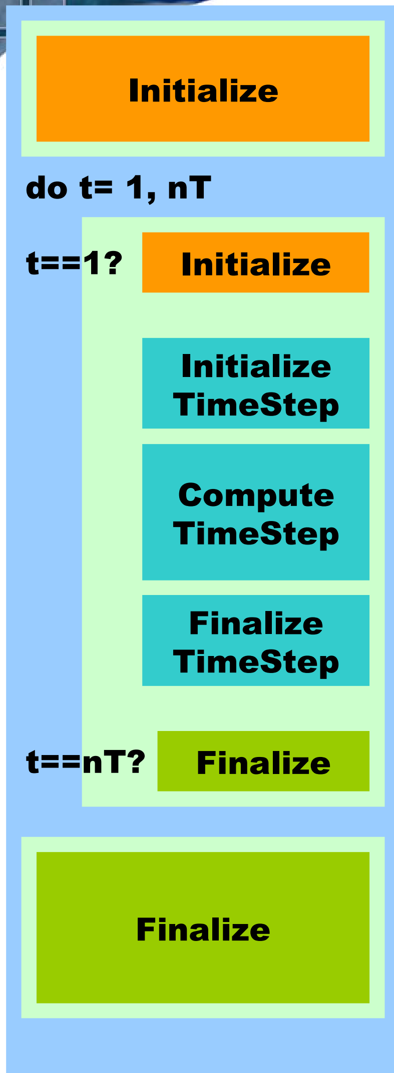
Wrapping

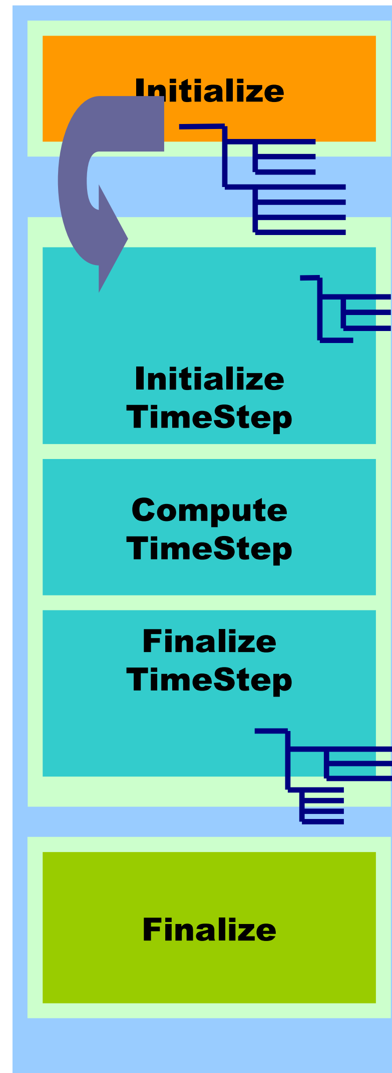


OpenMI.Utilities.Wrapper



Typical computational flow

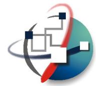




? How to provide input to the engine

? How to retrieve output from the engine

Engine Access



HarmoniT

	Fortran Buffer	IEngine

ExchangeItem definitions
ExchageItem Values
InputExchangeItems OutputExchangeItems
SetValues
GetValues

Fortan.dll	IEngine
Initialize	Initialize
PerformTimestep	PerformTimestep
GetInputtime	GetInputtime
GetCurrenttime	GetCurrenttime
GetError	GetError
Finalize	Finalize

Fortan.dll

IEngine

ILinkableComponent interface

HarmonIT

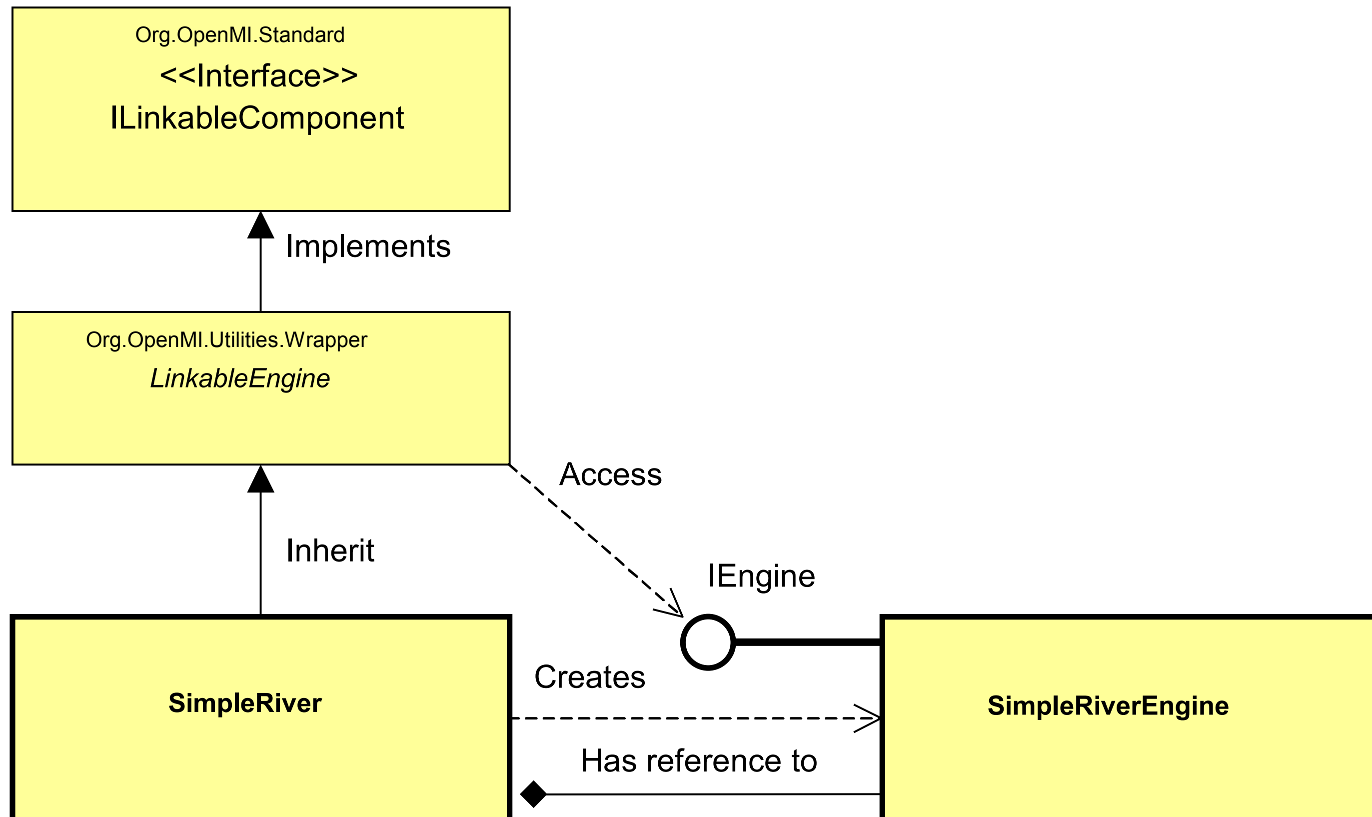
<<*Interface*>> ILinkebleComponent	
void	**Initialize**(IArgument[] properties)
string	**ComponentID**
string	**ComponentDescription**
string	**ModelID**
string	**ModelDescription**
ITimeSpan	**TimeHorizon**
int	**InputExchangeItemCount**
IInputExchangeItem	**GetInputExchangeItem**(int inputExchangeItemIndex)
int	**OutputExchangeItemCount**
IOutputExchangeItem	**GetOutputExchangeItem**(int outputExchangeItemIndex)
void	**AddLink** (ILink link)
void	**RemoveLink**(string linkID)
string	**Validate**()
void	**Prepare**()
IValueSet	**GetValues**(ITime time, string linkID)
ITimeStamp	**EarliestInputTime**
void	**Finish**()
void	**Dispose**()

IEngine interface

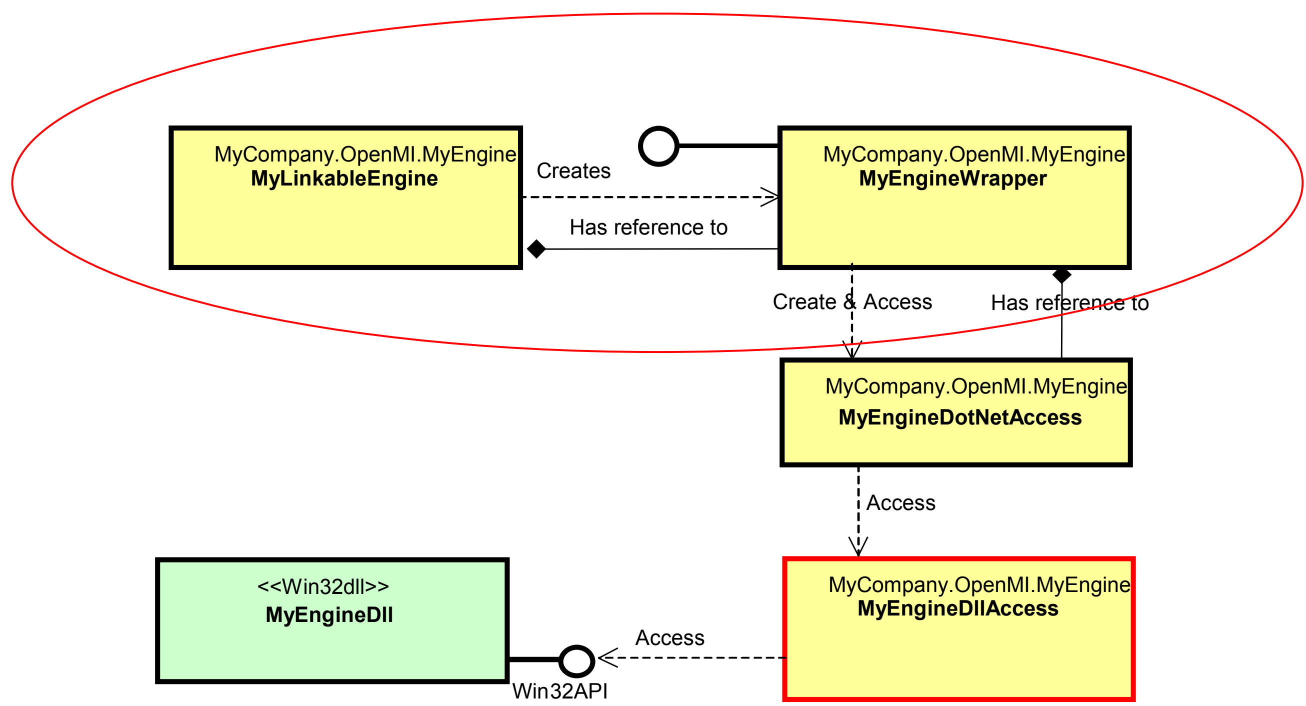
HarmonIT

<<IEngine>>	
void	Initialize(Hashtable properties);
bool	PerformTimeStep();
void	Finish();
ITime	GetCurrentTime();
ITime	GetInputTime(string QuantityID, string ElementSetID);
ITimeStamp	GetEarliestNeededTime();
void	SetValues(string QuantityID, string ElementSetID, IValueSet values);
IValueSet	GetValues(string QuantityID, string ElementSetID);
double	GetMissingValueDefinition();
string	GetComponentID();
string	GetComponentDescription();
string	GetModelID();
string	GetModelDescription();
double	GetTimeHorizon();
int	GetInputExchangeItemCount();
int	GetOutputExchangeItemCount();
org.OpenMI.Backbone	GetInputExchangeItem(int exchangeItemIndex);
org.OpenMI.Backbone	GetOutputExchangeItem(int exchangeItemIndex);

Simple river model design



Remaining work



MyLinkableModel

```
namespace MyOrganisation.OpenMI.MyModel
{
  public class MyLinkableModel: org.OpenMI.Utilities.Wrapper.LinkableEngine
          {
                    protected override void SetEngineApiAccess()
                    {
                              _engineApiAccess = new MyEngineWrapper();
                    }
          }
}
```

Exercise 6: Linkable Engine

Step 1

Create a new linkable component called SimpleRiver and inherit from LinkableEngine

Step 2

Create a class SimpleRiverEngine, which implements IRunEngine.

Step 3

Implement all methods of IRunEngine and make a simple computation in PerformTimeStep

Step 4

Use SetValues to provide input for PerformTimeStep and use GetValues to retrieve the results

Step 5

Implement SetEngineAPIAccess in SimpleRiver

Step 6

Create a test program in NUnit and test your implementation

Work on computational core

HarmoniT

- Typical computational core (Sobek-RR)
  - Written in FORTRAN
  - Uses files for input and output
  - Reads input, computes all time-steps, writes output
  - Is not object-oriented
  - Uses many common variables (globals)

First step: input/output metadata

- Use input and output exchange items to describe:
- What data can be exchanged
- Where
- In what units
- Combination of location (ElementSet) and quantity

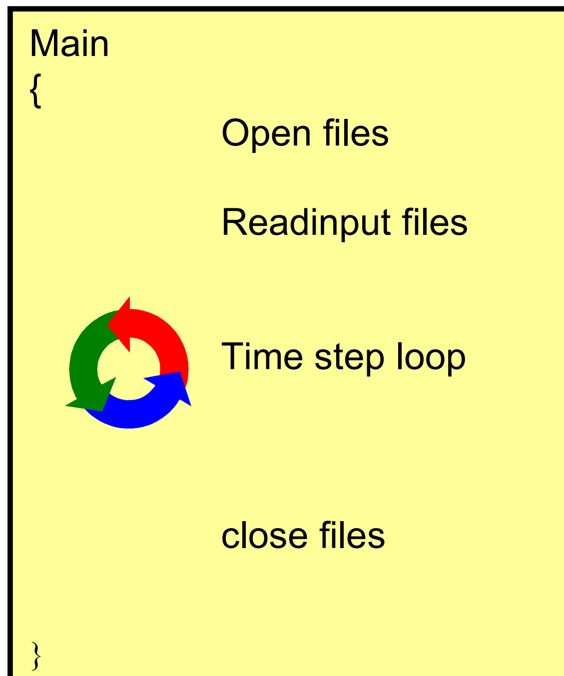
Second step: runtime

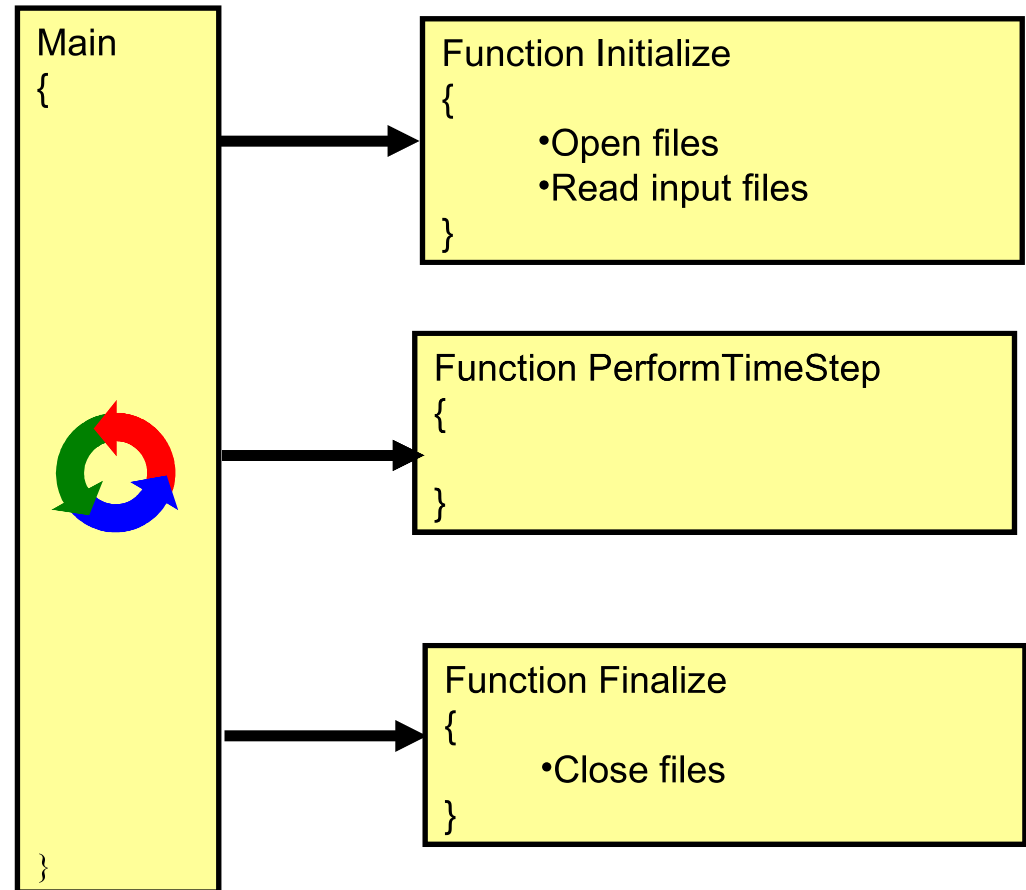
- Where the actual computation takes place
- Input values are received
- Output values for this time-step are computed
- Output values returned
- And so on for all time-steps

Typical FORTRAN simulation code

```
subroutine flow
initialise
do i=1,n
   calculate timestep
end do
write results
end
```

Changes to the engine core





Suggested Fortran functions

HarmoniT

```
logical function Initialize()
```

- Open files and populate your engine with initial data.

```
logical function PerformTimeStep()
```

- Perform a single time step

```
logical function Finish()
```

- Close files

```
logical function Dispose()
```

- De-allocate memory

Main changes needed

HarmoniT

- Separate main routine into initialisation and computation
- “Break up” main loop, make subroutine that can execute one time-step at a time
- Make local variables static by adding a “save” command to the FORTRAN code

Updated code

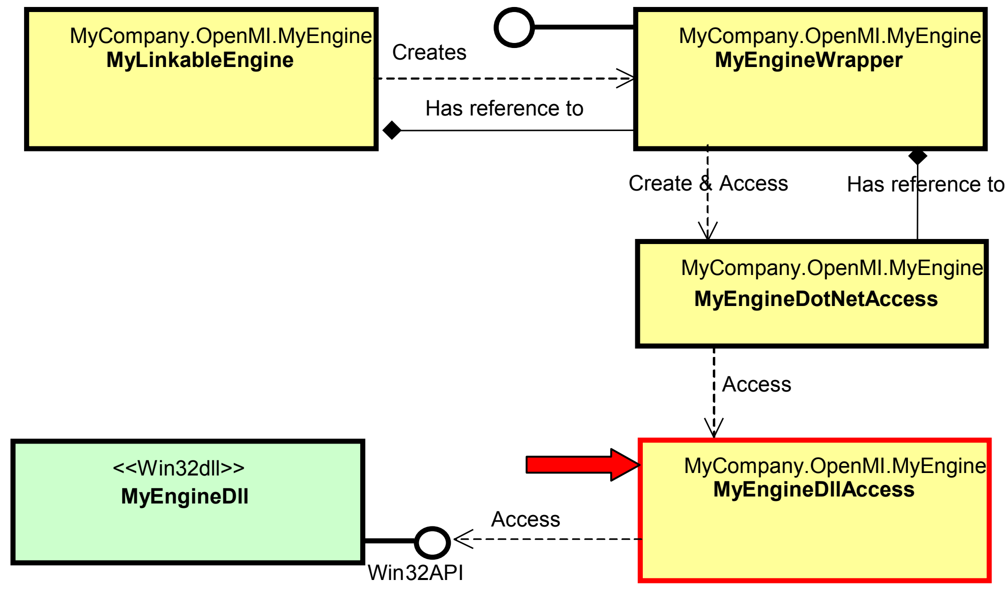
```
subroutine flow (mode)
save
if (mode.eq.1) then
      initialise
end if
if (mode.eq.2) then
      calculate timestep
      return results
end if
end
```

Changes to Fortran code

HarmoniT

- Change your engine core in order to be compiled into a dll.
- Add a function to you engine core that will run a full simulation:
  `logical function RunSimulation()`
- Create a engine application (exe) that from its main program calls the RunSimulation functions in your Engine core dll.
- Try to run your engine by deploying the engine application and check if your engine still is producing correct results.

Accessing functions in your engine core



Fortran DLL

- Your Fortran code should be exported as a DLL, otherwise C# can't access it
- Static linking of Fortran code to C# is not possible
- Please consult the Fortran Compiler manual to find out how to make a DLL
- Fortran .NET compilers are starting to become available
- You could write the whole wrapper in Fortran .NET

Fortran – C# communication

HarmoniT

Export function in Fortran:

(may be different in your compiler, this is for Visual Fortran 6.6)

```
logical function GetModelID(ID)
!dec$ attributes dllexport::GetModelID
```

In C# import from Fortran DLL:

```
[DllImport("SimpleRiverFortranEngineDlld.dll", EntryPoint = "GETMODELID",
 SetLastError=true, ExactSpelling
    =true,CallingConvention=CallingConvention.Cdecl)]
public static extern bool GetModelID(StringBuilder id,uint length);
```

Most things are passed by reference in Fortran

Strings are passed as an array of chars followed by the length (by value)

Passing strings between Fortran and C#

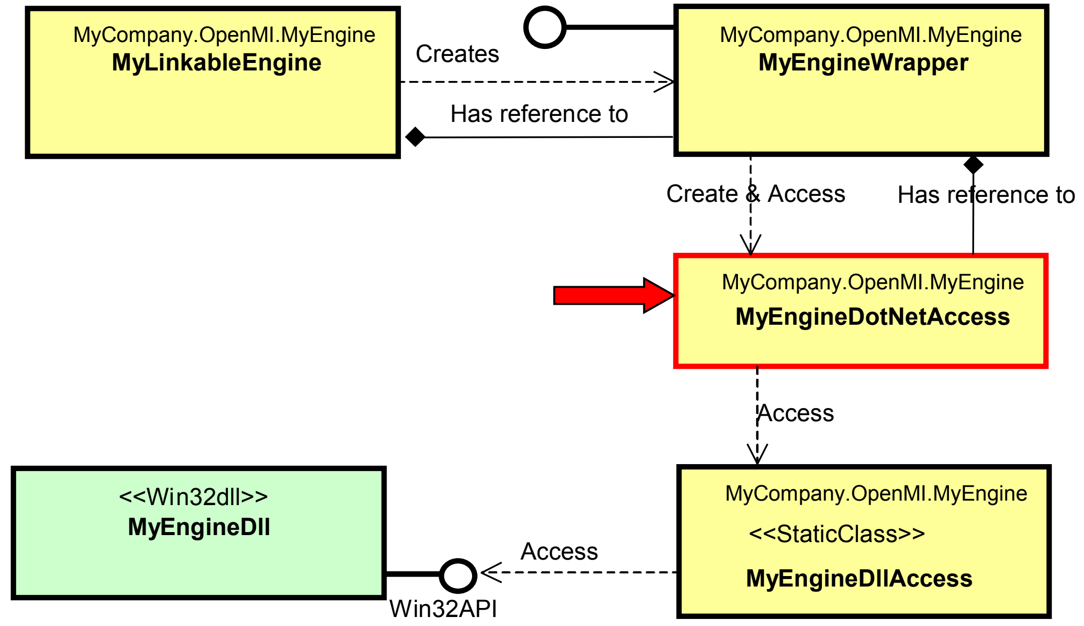
HarmonIT

```
public string GetModelID()
{
        StringBuilder id = new StringBuilder("                              ");
        if(!(SimpleRiverEngineDllAccess.GetModelID(id,
                (uint) id.Length)))
        {
                CreateAndThrowException();
        }
        return id.ToString().Trim();
}
```

Numbers are passed by reference

```
public double GetInputTime()
{
        double time = 0;
        if(!(SimpleRiverEngineDllAccess.GetInputTime(ref time)))
        {
                CreateAndThrowException();
        }
        return time;
}
```

Implementing MyEngineDotNetAccess



Translating error codes into exceptions

HarmoniT

- Fortran has no prescribed error handling mechanism, many people use return codes
- In .NET, exceptions are used for error handling
- The main purpose of MyEngineDotNetAccess is translation of error codes into exceptions
- See the code for more details

Exercise 7: Access FORTRAN code

HarmonIT

Step 1

Open SimpleRiver.dsw in the Fortran IDE

Step 2

Run SimpleRiver and make sure you understand the code. Have a look at the output file. Read the Guidelines book 4

Step 3

Convert the code to a dll with the following methods: PerformTimestep, GetValues, SetValues, Initialize and Finish

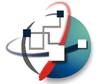
Step 4

Adjust SimpleRiverEngine in .Net so that it calls the dll

Step 5

Test your dll with the NUnit test of exercise 6. Make sure the output file is still produced

Exercise 8: Get Exchange Items from FORTRAN



Step 1

Open SimpleRiverEngine and let it inherit from IEngine

Step 2

All needed information should be retrieved from the FORTRAN dll. Make extra methods in the dll and implement them

Step 3

Create an omi file for SimpleRiver

Step 4

Import SimpleRiver in the user interface. Make sure the exchange items are populated as you expect them

Step 5

Create a composition where you use AsciiReader as input for SimpleRiver. Run the composition and check the output file