

ApsimX Design Document

Contents

| | | |
|-------|--|----|
| 1. | Introduction..... | 2 |
| 2. | The Model Framework (Model.exe)..... | 2 |
| 2.1 | Objectives | 2 |
| 2.2 | A specification for writing an ApsimX model..... | 3 |
| 2.3 | Core Interfaces | 4 |
| 2.4 | Core Attributes | 5 |
| 2.5 | Path Specification | 6 |
| 2.6 | Unit Specification..... | 6 |
| 2.7 | Scope | 8 |
| 3. | Running an ApsimX Simulation..... | 8 |
| 4. | User Interface Design (UserInterface.exe) | 10 |
| 4.1 | Objectives | 10 |
| 4.2 | Commands..... | 10 |
| 4.3 | Model View Presenter | 11 |
| 4.4 | ExplorerView/ExplorerPresenter..... | 12 |
| 4.4.1 | An example using the Axis model..... | 12 |
| 5. | Building, Testing and Deployment..... | 15 |

1. Introduction

The purpose of this document is to describe the design of ApsimX; the model framework (Model.exe), the user interface (UserInterface.exe) and the testing environment.

A model is defined as a software class that encapsulates a given piece of functionality and/or data. It may contain fields, properties, methods and may invoke events. A model can either be a computation for a real world object (e.g. crop) or it can be a data container for an entity in the user interface e.g. properties of a graph.

A zone is a model that manages a collection of other models. It also contains an area property.

A simulation is a top level zone that may contain other zones or models.

2. The Model Framework (Model.exe)

2.1 Objectives

- By default no toolbox, model or user interface XML files should be in the file system. This will make deployment easier. Default constant values should be explicitly stated in the model source code. A facility will be made available in the user interface to change/override these defaults.
- The .apsimx files are to precisely define the model objects and their configuration as it should be set out in memory. This will allow serialization to/from XML and binary files.
- Models are written to allow XML and binary serialisation and deserialisation. Binary serialisation will allow us to do check pointing as it can capture the exact state of a model at any point. XML serialisation will be used for serialization to/from the .apsimx files. It will also be possible to allow the model and user interface to work with binary serialisation files.
- The framework shall allow models to be loaded / unloaded at runtime.
- All source files (in the framework and user interface) shall have namespaces that match the folder names.

Q: Do we need a legacy model wrapper so that existing models can be run in ApsimX?

Q: Do we provide the ability to script a simulation or series of simulations (outside of a running simulation) so that rewinding to a checkpoint is possible and then run forwards again. I'm thinking of POAMA style climate scenarios i.e. having 99 possible endings to a season. This might be hard to do from within a manager2 script within in a simulation.

2.2 A specification for writing an ApsimX model

All models are written as normal .NET classes and need to implement the IModel interface. (see [section 2.3](#) for a listing of this interface).

Models must be binary and XML serializable. Normally this doesn't require any extra work by the model developer providing the types used are serializable (all .net types are).

If models need to communicate with other models, they may do so by declaring a private field with a *Link* attribute (see [section 2.4](#) for a detailed description of this attribute). ApsimX will provide a valid reference for the field at initialization time by finding another model that is in scope and has the same type as the field declaration. Once found, models can then call properties and methods of the reference model as normal. e.g.

```
[Link]
private Clock Clock = null;

public void OnInitialised()
{
    Simulation.Completed += OnCompleted;
    Clock.StartOfDay += OnStartOfDay;
}

private void OnStartOfDay()
{
    if (Clock.Today == SowDate)
        // do something
}
private void OnCompleted()
{
    Simulation.Commenced -= OnCommence;
    Simulation.Completed -= OnCompleted;
}
```

In order to decouple models from other models, it may be necessary to create interfaces (e.g. IClock) that specify what the public interface for that type of model is. This would then allow a different type of model to be swapped in. This would be particularly important for models where we have different implementations e.g. SoilWater.

Models may also optionally have an OnInitialised method that will be called to allow the model to initialise itself every time a simulation is run. By the time this method is called, ApsimX will have satisfied all link references. If a model wishes to trap events from other models, then code needs to be written to enable this. The above example shows code that traps the Clock's *StartOfDay* event.

When a simulation completes running, the Simulation object will invoke a 'Completed' event. This event can be trapped to perform cleanup. Note that models may not be unloaded from memory between consecutive simulation runs. It is therefore important that models perform cleanup at the end of a simulation. This includes unsubscribing to events when the simulation terminates, otherwise when OnInitialised is called for a second simulation run, the event will be subscribed to again, resulting in multiple calls to the event handler. The code in OnCompleted above shows how two events are unsubscribed.

In all ApsimX simulations, 'models' are run within a parent 'Zone' (a core model that looks after a collection of models). This zone model is itself contained within a top level 'Simulation'. Interfaces for these are provided in [Section 2.3](#). If a model needs to communicate with its Zone or the top level Simulation (also a Zone) it may do so via the normal link mechanism.

```

[Link]
private IZone Field = null;

private void OnStartOfDay()
{
    double[] sw = Field.Get("sw") as double[];
}

```

The Zone interface provides several methods for dynamically getting references to other models, getting the value of variables in other models, adding a new model to a zone and removing a model from a zone.

To flag a fatal error and immediately terminate a simulation, a model may simply throw an exception.

2.3 Core Interfaces

```

namespace Model.Core
{
    public interface IModel
    {
        string Name { get; set; }
    }
}

```

```

namespace Model.Core
{
    public delegate void ModelAddedDelegate(string NodePath);
    public delegate void PathDelegate(string Path);

    public interface IZone : IModel
    {
        event ModelAddedDelegate ModelAdded;
        event PathDelegate ModelDeleted;

        double Area { get; set; }
        IModel[] Models { get; }
        string FullPath { get; }
        object Find(System.Type ModelType);
        object Get(string NamePath);
        string Add(string ModelXml);
        bool Remove(string ChildFullPath);
    }
}

```

```

namespace Model.Core
{
    public enum CriticalEnum { Information = 1, Warning = 2, Critical = 3 };
    public interface ISimulation : IZone
    {
        event NullTypeDelegate Commenced;
        event NullTypeDelegate Completed;
        void WriteProperty(string Name, string Value);
        void WriteMessage(string Message, CriticalEnum WarningLevel);
    }
}

```

```

namespace Model.Core
{
    public interface ISimulations
    {
        event NullTypeDelegate Completed;
        string FileName { get; }
        bool Run();
        bool Run(ISimulation Simulation);
    }
}

```

2.4 Core Attributes

| Tag Name | Applies To | Description |
|--|-----------------------|--|
| <p>[Link]</p> <p><i>Named parameters:</i> NamePath:</p> <p>IsOptional:</p> | Private fields | <p>When applied to a private field, ApsimX will locate an object (in scope – Section 2.7) of the specified type and store a reference to it in the field. Will throw an exception if not found.</p> <p>When a NamePath is specified, ApsimX will locate the object using this path and store a reference to it in the field. NamePath must conform to the specification in Section 2.5. Will throw an exception if the NamePath isn't valid.</p> <p>When IsOptional = true, ApsimX will not throw an exception when an object cannot be found.</p> |
| [Units (string UnitString)] | Fields and properties | Specifies the units of the related field or property. ApsimX does not do anything with the units, however other models may (e.g. Report). Units must conform to the specification in Section 2.6 . |
| [PresenterName (string ClassName)] | Class | When applied to the model class, this attribute specifies the name of the User Interface presenter class that should be used when showing the model in the 'Right hand panel'. Class names need to include the namespace. |
| [ViewName (string ClassName)] | Class | When applied to the model class, this attribute specifies the name of the User Interface view class that should be used when showing the model in the 'Right hand panel'. Class names need to include the namespace. |

2.5 Path Specification

Paths are structured similarly to directory paths in Windows and Unix, using a '.' character instead of slashes. All paths are relative to the zone they are being applied to. e.g.

ModelA

Paddock1.ModelB

2.6 Unit Specification

Units are expressed as character strings. Text and Boolean quantities do not have units. The set of valid unit strings is generated by the grammar set out below. The principles on which this grammar is based are as follows:

- In general, SI units are used. Non-SI metric units are also permitted.
- Units are constructed from a restricted number of "base" SI units by applying scaling prefixes and powers and by combination into products and ratios.
- Integer, decimal and rational representations of powers are permitted. Where an exact integer representation exists, it is used. Otherwise, where an exact decimal representation exists it is used (e.g. 0.75, not 3/4). All powers are positive; negative powers are denoted by representing the unit in the denominator of a ratio.
- Where a unit is a ratio, all terms in the denominator must follow all terms in the numerator.
- Either "-" or "%" or "0-1" (where appropriate) are used to denote all dimensionless quantities.
- The null or empty string is a valid unit. It is reserved to denote situations where the unit is unknown or any unit is acceptable.
- The grammar may only be extended in future by expanding the set of "base" units.

```

<unit>          ::= [<term>{'.'<term>}]{ '/'<term>} | '-' | '%' | '0-1'
<term>          ::= [<scale>]<scalable-unit>['^'<power>]
                  | <non-scalable-unit>['^'<power>]
<scalable-unit> ::= 'g' | 'm' | 's' | 'K' | 'A' | 'mol' | 'cd'

| 'rad' | 'sr' | 'Hz' | 'N' | 'Pa' | 'J' | 'W' | 'C' | 'V' | 'F' | 'ohm'

| 'S' | 'Wb' | 'T' | 'H' | 'oC' | 'lm' | 'lx' | 'Bq' | 'Gy' | 'Sv' | 'kat'
| 't' | 'l' | 'min' | 'h' | 'd' | 'y'
<non-scalable-unit> ::= 'rad' | 'sr' | 'deg' | 'ha'
<scale>      ::= 'p' | 'u' | 'm' | 'c' | 'd' | 'D' | 'h' | 'k' | 'M' | 'G' | 'T'
<power>     ::= <integer> | <decimal> | <integer> '/' <integer>
<decimal>   ::= [<digit>{<digit>}] '.' <digit>{<digit>}
<integer>   ::= <digit>{<digit>}
<digit>     ::= '0'...'9'

```

- Unit strings are case-sensitive; whitespace is permitted (and ignored)
- Base units and their dimensions are as follows (see the appendix for more detail on dimensions and SI units):

| | | | | | |
|-----|-----------|-------------------------|-----|----------------|-----------------------|
| g | gram | M | Wb | weber | $L^2 M T^{-2} i^{-1}$ |
| m | metre | L | T | tesla | $M T^{-2} i^{-1}$ |
| s | second | T | H | henry | $L^2 M T^{-2} i^{-2}$ |
| K | kelvin | θ | °C | degree Celsius | θ |
| A | ampere | i | lm | lumen | I |
| mol | mole | n | lx | lux | $L^{-2} I$ |
| cd | candela | I | Bq | becquerel | T^{-1} |
| rad | radian | — | Gy | gray | $L^2 T^{-2}$ |
| sr | steradian | — | Sv | sievert | $L^2 T^{-2}$ |
| Hz | hertz | T^{-1} | kat | katal | $T^{-1} n$ |
| N | newton | $L M T^{-2}$ | t | tonne | M |
| Pa | pascal | $L^{-1} M T^{-2}$ | l | litre | L^3 |
| J | joule | $L^2 M T^{-2}$ | min | minute | T |
| W | watt | $L^2 M T^{-3}$ | h | hour | T |
| C | coulomb | $T i$ | d | day | T |
| V | volt | $L^2 M T^{-3} i^{-1}$ | y | year | T |
| F | farad | $L^{-2} M^{-1} T^4 i^2$ | deg | degree | — |
| ohm | ohm | $L^2 M T^{-3} i^{-2}$ | ha | hectare | L^2 |
| S | siemen | $L^{-2} M^{-1} T^3 i^2$ | | | |

- The "u" scaling factor denotes "micro" (10^{-6}). Other scaling prefixes have their usual meanings.
- The tokens "/" and "." are each used with two different meanings, but the meaning can always be determined from the following token.
- "%" must be used alone; for example, "%/d" is not a valid unit.

Examples of valid unit strings:

hPa scaled unit

MJ/m²/d ratio with two terms in the denominator

/s no numerator

kg^{0.75} or kg^{.75}

m^{1/3} but not m^{1/2}, which is grammatically correct but should be given as m^{0.5}

g.m/s² but not m/s².g as numerator terms must precede denominator terms

2.7 Scope

When trying to find models that are in scope (either by links or the 'get' method), the ApsimX will look for matches hierarchically in the simulation. Firstly it will try and match the entity in the same zone. If not found it will look in the zone above. If not found there, then the zone above that will be searched etc. It won't look for a match in a sub 'zone of the parent e.g.

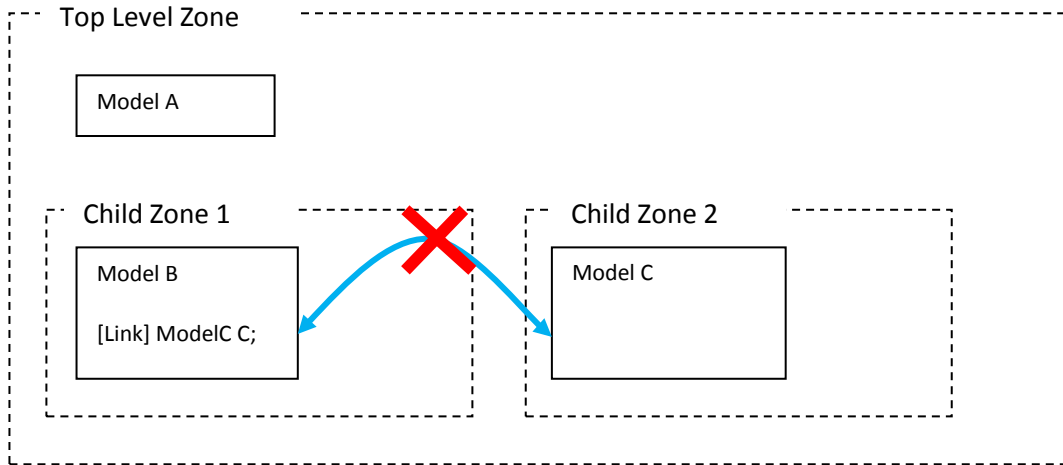


Figure 1: A schematic that demonstrates how Model B's link will not be connected to Model C because the model is not in the same zone as Model B and it also isn't in the parent zone (top level system).

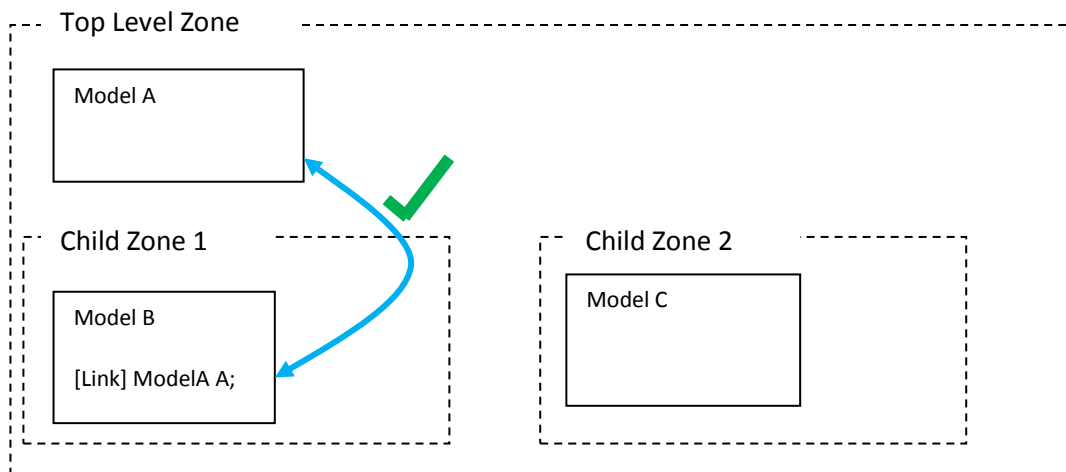


Figure 2: A schematic that demonstrates how Model B's Link will be connected to Model A.

3. Running an ApsimX Simulation

The ApsimX framework reads the .apsimx file (xml) and deserialises it to a hierarchical collection of zones / models in memory. During this process, values found in the .apsimx file are pushed into the newly created models when public properties are found in the classes. e.g. for the clock model:

```
<Clock>
  <Name>Clock</Name>
  <StartDate>1940-01-01T00:00:00</StartDate>
  <EndDate>1989-12-31T00:00:00</EndDate>
</Clock>
```



```
// Parameters serialised
public string Name { get; set; }
public DateTime StartDate { get; set; }
public DateTime EndDate { get; set; }
```

Once all models have been constructed, all private fields in all models are examined for [Link] attributes. ApsimX (Zone.cs) will supply a reference for each [Link] field found.

The framework will then iterate through all models and call any private OnInitialised methods found. Finally, the top level Simulation will invoke a commenced event. The clock component, during its OnInitialised method will add a handler to this event. Figure 3 shows a UML sequence diagram describing this startup sequence and the execution of a simulation. Once a simulation is completed, The Simulation object will invoke its Completed event.

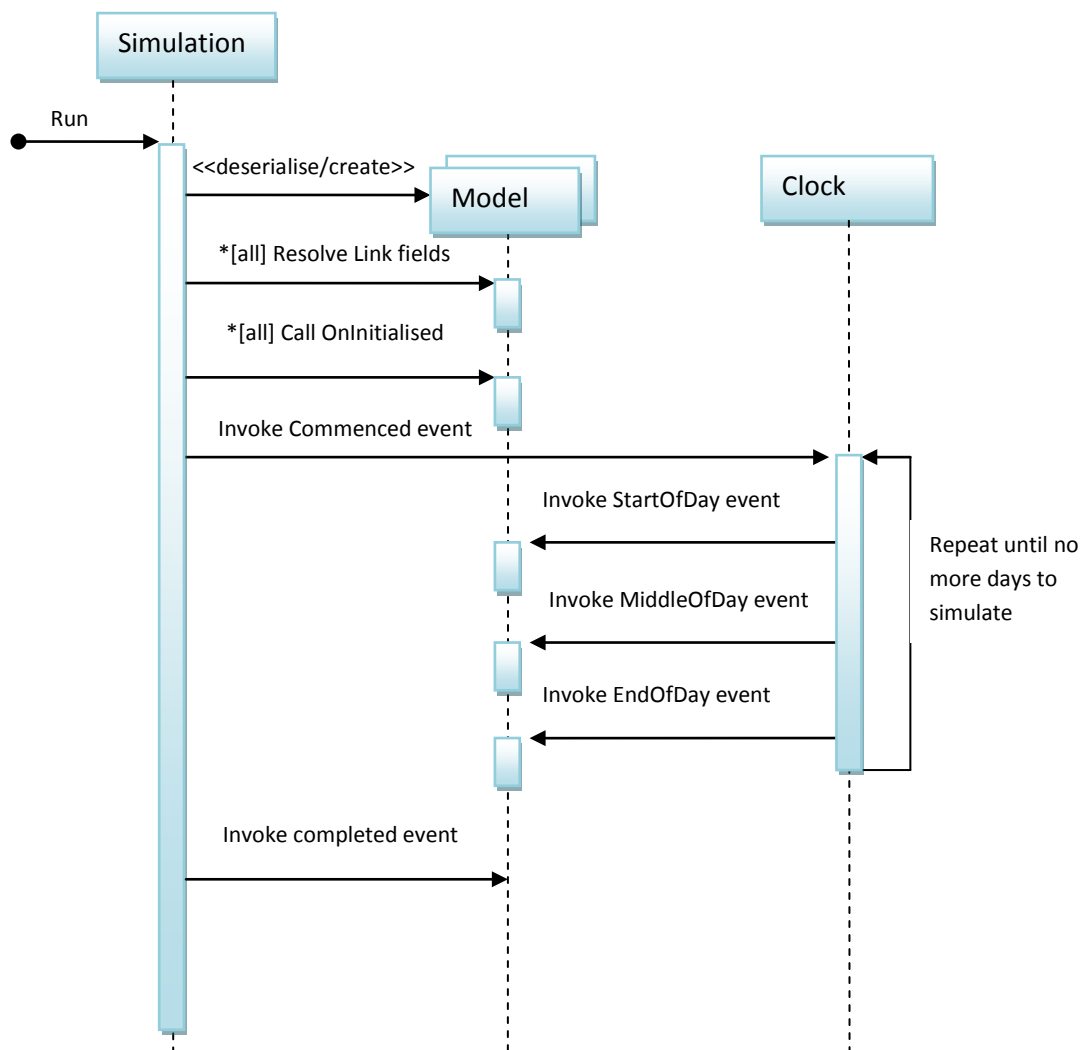


Figure 3: A UML sequence diagram showing the execution of a simulation.

4. User Interface Design (UserInterface.exe)

4.1 Objectives

- The user interface shall work with model objects in memory using reflection and not via XML as the current APSIM 7.5 does.
- The 'Command pattern' shall be used for all user interface actions (e.g. select a node, edit the clock start date, change the format of a graph). A history of commands will be stored for each .apsimx file loaded. This will allow undo/redo as each command will have the ability to undo itself. This also allows a script of commands to be sent to the user interface via the command line, allowing some automated user testing of the user interface.
- The user interface shall run on Linux (and Mac?)
- The simulation tree will populate its nodes on demand as nodes are expanded. This should allow quicker display for .apsimx files that have a large number of simulations.
- Perhaps we should explore the option to allow binary files (from binary serialisation). This should load much quicker. It will also mean that a simulation can be run to some point, a checkpoint made (serialise the simulation to a binary file or memory stream) and then this checkpoint loaded into the user interface.
- The user interface shall look for reflection tags in the models to determine what UI to display and where models can be dropped in the simulation tree.
- A 'Model-View-Presenter' pattern shall be used to disconnect views (forms) from the models. In theory, this should allow a different view layer to be built (e.g. System.Web.Forms) while keeping the majority of the code the same.

Q: Rather than have zones nested within parent zones in the simulation tree, should we be thinking about zones laid out on a grid i.e. each zone having an x/y coordinate?

Q: Should we be thinking about a web version of APSIM that allows users (with a browser) to load simulations from their DropBox, edit and configure them and then click run to send the simulation to our cluster?

4.2 Commands

The requirement for Undo/Redo has led to the adoption of the 'command' pattern in the user interface. This pattern dictates that all changes to all 'Model' classes must be done via a command. Commands are also used by the user interface when the user interacts with tree nodes. Each command has two methods, one for performing the command, another for undoing the command. The interface (ICommand.cs) looks like this:

```
namespace UserInterface.Commands
{
    public interface ICommand
    {
        object Do();
        object Undo();
    }
}
```

If the command alters the state of a model during a 'Do' or 'Undo', it should return the altered model to the caller (CommandHistory). The CommandHistory will then invoke a 'ModelChanged' event that the views can subscribe to and update their screens. As an example of a concrete command, the 'ChangePropertyCommand' is given below. This command is used to change a property value in a model. Before doing this though, it will retrieve the original value so that it can reapply this value during an Undo operation.

```

namespace UserInterface.Commands
{
    class ChangePropertyCommand : ICommand
    {
        private object Obj;
        private string Name;
        private object Value;
        private object OriginalValue;

        public ChangePropertyCommand(object Obj, string PropertyName, object PropertyValue)
        {
            this.Obj = Obj;
            this.Name = PropertyName;
            this.Value = PropertyValue;
        }

        public object Do()
        {
            // Get original value of property so that we can restore it in Undo if needed.
            OriginalValue = Utility.Reflection.GetValueOfFieldOrProperty(Name, Obj);

            // Set the new property value.
            if (Utility.Reflection.SetValueOfFieldOrProperty(Name, Obj, Value))
                return Obj;
            return null;
        }

        public object Undo()
        {
            if (Utility.Reflection.SetValueOfFieldOrProperty(Name, Obj, OriginalValue))
                return Obj;
            return null;
        }
    }
}

```

4.3 Model View Presenter

A 'Model' in this context is self explanatory. It is the class that holds the problem domain data (deserialised from the XML files) that is editable by the user and executes during a simulation run. Some examples include SoilWater, Clock and Graph.

A 'View' is a form that allows user interaction. It doesn't have any functionality beyond the display of information and receiving user input. It does not have any functionality that determines what data gets put on the screen. i.e. it doesn't talk to the model. A 'view' does not have a reference to a model or presenter. It is essentially a very passive (humble) form that is told what to do by the presenter. It does not contain any logic that describes what to do when the user interacts with it. In short, the idea is to keep it as simple as possible.

A 'Presenter' is an class that tells the view what to display, asking the model for that data. It acts as a go-between between a view and a model. It is also responsible for determining what to do when the user does something. A Presenter should not have code that assumes a particular display technology i.e. no using System.Windows.Forms or System.Web.Forms.

In theory, the user interface should be able to be recoded from a windows app to a web app by just recoding the 'views' and keeping everything else the same. It should also be noted that a view could have multiple presenters in different situations. For example, a 'GridView' (form with a grid on it) may have 1 presenter that populates the grid with property type info (like what the user sees when they click on a manager component). It might have another presenter that displays soil profile information. A third presenter might display the contents of an APSIM output file.

For more info on the Model/View/Presenter pattern visit here:

<http://codebetter.com/jeremymiller/2007/07/26/the-build-your-own-cab-series-table-of-contents/>

4.4 ExplorerView/ExplorerPresenter

The central concept in the user interface is the 'ExplorerView', a form with a simulation tree on the left and a right hand panel where model views are displayed. The associated 'ExplorerPresenter' is responsible for populating the controls on the view and for responding to input from the user. When the user selects a model in the simulation tree, an event handler is called in the presenter, which will in turn look for two reflection tags in the model.

```
[ViewName("UserInterface.Views.GridView")]
[PresenterName("UserInterface.Presenters.PropertyPresenter")]
```

The ViewName tag tells the presenter the full name (including the namespace) of the 'view' class to display on the screen. Each view class needs a corresponding presenter class and the PresenterName specifies this. With these two class names, the ExplorerPresenter can create instances of these and tell the ExplorerView to display the view in the right hand panel.

The presenter also maintains a 'CommandHistory' containing all executed commands and this is passed to each presenter that it creates so that they can create commands as required. This is done via the 'Attach' method in IPresenter.

```
namespace UserInterface.Presenters
{
    public interface IPresenter
    {
        void Attach(IModel Model, object View, CommandHistory CommandHistory);
    }
}
```

4.4.1 An example using the Axis model

The Axis model is a simple data container for storing properties associated with an axis on a graph. The data deserialised from the XML looks like this:

```
<Axis>
  <Type>Left</Type>
  <Title>Y axis title</Title>
</Axis>
```

The axis model looks like this:

```
namespace Model.Components.Graph
{
    public class Axis
    {
        public enum AxisType { Left, Top, Right, Bottom };

        /// <summary>
        /// The 'type' of axis - left, top, right or bottom.
        /// </summary>
        public AxisType Type { get; set; }

        /// <summary>
        /// The title of the axis.
        /// </summary>
        public string Title { get; set; }
    }
}
```

This model has 2 properties, type and title. The view for this model looks like this:

Title: 

The Axis view is very simple with a single text box that displays the axis title. The AxisPresenter that connects the model to the view looks like this:

```

using Model.Components.Graph;
using UserInterface.Views;

namespace UserInterface.Presenters
{
    /// <summary>
    /// This presenter connects an instance of a Model.Graph.Axis with a
    /// UserInterface.Views.AxisView
    /// </summary>
    class AxisPresenter : IPresenter
    {
        private Axis Axis;
        private IAxisView View;
        private CommandHistory CommandHistory;

        /// <summary>
        /// Attach the specified Model and View.
        /// </summary>
        public void Attach(object model, object view, CommandHistory commandHistory)
        {
            Axis = model as Axis;
            View = view as AxisView;
            CommandHistory = commandHistory;

            // Trap change event from the model.
            CommandHistory.ModelChanged += OnModelChanged;

            // Trap events from the view.
            View.OnTitleChanged += OnTitleChanged;

            // Tell the view to populate the axis.
            View.Populate(Axis.Title);
        }

        /// <summary>
        /// The 'Model' has changed so we need to update the 'View'.
        /// </summary>
        private void OnModelChanged(object Model)
        {
            if (Model == Axis)
                View.Populate(Axis.Title);
        }

        /// <summary>
        /// The user has changed the title field on the form. Need to tell the model this via
        /// executing a command.
        /// </summary>
        void OnTitleChanged(string NewText)
        {
            CommandHistory.Add(new Commands.ChangePropertyCommand(Axis, "Title", NewText));
        }
    }
}

```

In the Attach method, the Axis presenter traps the model's OnChanged event (caused by an Undo) and the views OnTitleChanged event (caused by the user). It also tells the view to populate the text box with the value of the title property from the model. When the title changes in the model (OnChanged), the presenter tells the view the new title. When the view changes the title (in OnTitleChanged), the presenter tells the model the new title via a command so that this can be undone at a later time.

All views should have an interface (IAxisView) to decouple the view from the presenter that calls into it. This allows a presenter to use different implementations of a view.

5. Building, Testing and Deployment

There is an opportunity to rework the Bob automated testing process to something that is smarter than what we currently have. We need to move to a system where all tests provide a pass/fail acceptance response. The new build and test procedure needs to look something like this (borrowed heavily from Jo Sharp's visit to Tmba):

Unit tests: We move from NUnit to the Microsoft Visual Studio testing framework. This has the advantage that no extra 3rd party installs are needed. It is also nicely integrated into the IDE. These tests, by their nature, either pass or fail. As much of the user interface and model framework as possible should be unit tested.

Sensitivity tests: Every science model must have a set of sensitivity tests that define sensible ranges or mean output values (a task for the model developer), stretch the limits of models and test the module under a range of scenarios/managements/environments (range of PAWC x nitrogen x environment responses). The tests provide a pass / fail response after comparing various statistics (e.g. mean, median, high, low etc values) using tolerances (relative or absolute or ANOVA).

Validation tests: Every science model must have a suite of validation tests. The automated testing process needs to be written to provide a pass/fail response after comparing the following stats to known good stats.

- Standard regression (linear relationship between measured and simulated)
 - Slope (relationship between measured and modelled) and y-intercept (lag or lead between measured and modelled)
 - Slope of 1, y-intercept of 0 indicates perfect fit
 - Assumes measured and modelled are linearly related
 - Assumes measured data are free of error
 - r (Pearson's correlation coefficient) and R^2 (coefficient of determination)
 - Describe degree of co-linearity between measured and modelled
 - r indicates the degree of linear relationship and ranges from -1 (perfect negative) to 1 (perfect positive), with 0 indicating no linear relationship
 - R^2 indicates the proportion of variance in measured data explained by the model and ranges from 0 to 1, with greater values indicating less variance in error
- Dimensionless (relative model evaluation assessment)
 - Index of agreement (d) indicates the degree of model prediction error
 - Ratio of mean square error and 'potential error'
 - falls between 0 (no agreement) and 1 (perfect agreement)
 - detects additive and proportional differences in the measured and modelled means and variances
 - Nash-Sutcliffe efficiency
 - Determines the relative magnitude of the residual variance compared to the measured variance, i.e. how well the measured v modelled fits 1:1 line
 - Value of 1 is optimal, between 0 and 1 acceptable, < 0 unacceptable
- Error index (deviation in the units of the data)
 - RMSE (and other forms) indicate error in units of interest
 - A value of 0 is a perfect fit
 - Suggested that $RMSE < \text{half standard deviation of measurements}$ is appropriate for model evaluation
 - If standard errors of the measurements are available the statistical significance of RMSE can be assessed by comparing with the 95% confidence interval of the measurements
 - Measures of bias indicate average tendency of modelled data to be larger or smaller than measured data

- A value of 0 is a perfect fit, positive values are model underestimation bias and negative values are model overestimation bias
- What proportion of modelled points fall within a certain tolerance of the measured points (such as 95% confidence interval of the measurement)

The above tests will need to be approved by a statistician to ensure that they are appropriate for the data available. The data may be non-parametric or time series, so we'll need to make sure the tests don't require normally distributed data.

To make this testing easier, R will probably be the preferable option as it could reuse a lot of R's current functionality. At the end of the testing process, the developer is notified with a pass / fail and a link to a meta-analysis of the above testing in a single, easy to read (and print) pdf file with graphs. An ability to compare two reports would be really nice as the Reference Panel and model developers are interested in how the model has improved (or otherwise) its performance. This 'test report' can then be tabled at Reference Panel meetings and stored on the web at release time.

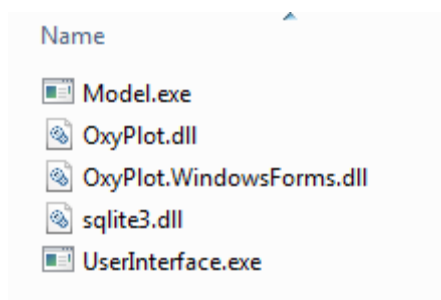
Upon every successful (green) build of APSIM, the testing report is stored in the build database along with the individual binaries. ApsimX (user interface) will contain an upgrade tool that updates ApsimX from the web site on a user agreed interval (e.g. once a month). Individual binaries may be upgraded independently from each provided that versioning and dependencies are known. For example, UserInterface.exe may be upgradable without the need to upgrade the models.

Q: Should all models be in single exe (Model.exe)? This would have the advantage of easy deployment on cluster.

Q: Should we move from SVN to Git (perhaps GitHub for hosting)? This will provide distributed version control with developer commits to their local copy. Only when merging with the master branch will this trigger a Bob build.

Q: Should we be looking at an open source continuous integration system (e.g. CruiseControl)?

Currently the deployment of ApsimX looks like this:



Model.exe needs sqlite3.dll and the user interface requires the 2 OxyPlot dlls although we can probably statically link these into UserInterface.exe.

Currently the user interface and model run on Windows, Ubuntu, and OSX Mountain Lion without any recompiling.