3.5 Definition of Questionnaires III: Expressions

The system includes an expression parser as a kind of stopgap between the two basic scoring approaches of **contributing question scores** and **severity scores** and the option of resorting to special-purpose programming as is used for the Kirby system described in section 3.7 below.

The **Expression Builder** that makes this possible is described at some length in the User Documentation and this should be read first.

The entire expression system is handled by the **Expression Builder** form frmExpressionBuilder, and the module xpParser.

The underlying objects on which the expression handling runs are defined in xpParser as these:

```
public enum xpState { Any, Number, Statement, InBetween}
public class xpBits { public xpElemType typ; public int pos; }

public enum xpElemType { Op, Opnd }
public enum xpOpType { Plus, Minus, Mul, Div, EQ, NEQ, GT, LT, GTE, LTE, And, Or, Xor, Close, Between,
      parseOpen }
public enum xpOpndType { Open, Number, Boolean, Score, Question, Option }
public class xpElemBase
{
   public string text = ""; //never null
   public int posi;
   public xpElemType type;
   public virtual xpElemBase Clone() { return (xpElemBase)this.MemberwiseClone(); }
}
public class xpOp : xpElemBase
{
   public xpOpType optype;
   public xpState state;
   public xpOp() { type = xpElemType.Op; }
   public xpOp Clone() { return (xpOp)this.MemberwiseClone(); }
}
public class xpOpnd : xpElemBase
{
   public xpOpndType opndtype;
   public double value = 0, leftval = float.MinValue, rightval = float.MaxValue;
   public bool boolvalue;
   public xpState state;
   public bool IsIntermedResult;
   public int Qix; //safeguard as Qs text may not be unique
   public long QSCHash, QHash, QOHash; //Hash codes of the various items this can point to: note that
      for an option must store the question Hash as well
   [XmlIgnore]
   public QuestionSumCategory QSC;
   [XmlIgnore]
   public QuestionDef qd;
   [XmlIgnore]
```

```
    public QuestionOption QO;
    public xpOpnd() { type = xpElemType.Opnd;  }
    public xpOpnd Clone() { return (xpOpnd)this.MemberwiseClone(); }
    //the QSC, qd and QO options are never saved anyway
}
```

They basically fall into the two groups of **Ops** and **Opnds** or operators and operands, although as you can see the two use a common base. **Operands** or **Opnds** will either be numerical constants or boolean values, or, and more commonly, they'll be *actual objects within the system*, either **questions** (QuestionDef), **options** (QuestionOption) or **scores** (QuestionSumCategory).

Because it would be inordinately clumsy to store copies of all these things with all their subsidiary bits to disc, these are all marked [XmlIgnore], and what is actually put out to disc are their **Hash codes** (as I call them): public long QSCHash, QHash, QOHash;.

I had problems trying to serialize the full expression code as a list of xpElemBase objects, so I disassemble the list into a list of **Ops** and a list of **Opnds** with an index list giving the structure of the original expression.

The true expression has to be re-assembled on loading back from disc, and all the **Hash codes** have to be "filled in" by assigning the actual objects referred to.

This is done by a special class xpDissembler:

```
public class xpDissembler
{
    [XmlIgnore]
    public List<xpElemBase> XP = new List<xpElemBase>();
    public List<xpOp> Ops = new List<xpOp>();
    public List<xpOpnd> Opnds = new List<xpOpnd>();
    public List<xpBits> Bits = new List<xpBits>();
    public void Assemble()
    {
        if (Bits != null)
        {
            XP = new List<xpElemBase>();
            foreach (xpBits bit in Bits)
                if (bit.typ == xpElemType.Op) XP.Add(Ops[bit.pos]); else XP.Add(Opnds[bit.pos]);
        }
    }
    public void ResetRefs(QuestionnaireDef QD)
    {
        foreach (xpOpnd opnd in Opnds)
        {
            switch (opnd.opndtype)
            {
                case xpOpndType.Question:
                    if (opnd.Qix > 0) opnd.qd = QD.questions[opnd.Qix - 1];
                    else foreach (QuestionDef qd in QD.questions)
                        if (qd.QuestionText == opnd.text) { opnd.Qix = qd.ix; opnd.qd = qd; break; }
                    break;
```

```
            case xpOpndType.Score:
               foreach (QuestionSumCategory qsc in QD.categories)
               {
                  if (qsc.Name == opnd.text) {opnd.QSC = qsc; break;}
               }
               break;
            case xpOpndType.Option:
               if (opnd.Qix > 0)
               {
                  opnd.qd = QD.questions[opnd.Qix - 1];
                  foreach (QuestionOption qo in opnd.qd.options)
                  {
                     if (qo.Text == opnd.text) { opnd.QO = qo; break; }
                  }
               }
               break;
         }
      }
   }
   public void Disassemble()
   {
      Ops.Clear();
      Opnds.Clear();
      Bits.Clear();
      foreach (xpElemBase eeb in XP)
      {
         xpBits bit = new xpBits();
         if (eeb.type == xpElemType.Opnd)
         {
            xpOpnd opnd = (xpOpnd)eeb;
            Opnds.Add(opnd);
            bit.pos = Opnds.Count - 1;
            bit.typ = xpElemType.Opnd;
            Bits.Add(bit);
         }
         else
         {
            xpOp op = (xpOp)eeb;
            Ops.Add(op);
            bit.pos = Ops.Count - 1;
            bit.typ = xpElemType.Op;
            Bits.Add(bit);
         }
      }
   }
}
```

All expresions are actually stored ONLY as these xpDissembler objects. At present, such objects can be set for a **QSC** or QuestionSumCategory, or for a SeverityScore ONLY.

The actual evaluation of expressions is entirely handled by the special class xpParser after which this module takes its name, and after earlier attempts to do things another way, this now evaluates expressions strictly using an **expression tree**, the basic constituent of which is defined as the first child class within xpParser:

```
public class xpParser
{
   public List<xpElemBase> Expression = new List<xpElemBase>();
   public xpState state;
   public bool ExpectOp = false, betweenHadAnd = false;
   public int bracketcount = 0, betweenbracketcount = 0;
   Stack<xpOp> OpStack = new Stack<xpOp>();
   Stack<xpTreeElem> OpndStack = new Stack<xpTreeElem>();
   Stack<xpState> StateStack = new Stack<xpState>();

   class xpTreeElem
   {
      public xpTreeElem Left,  Right;
      public xpElemBase Item;
      public xpTreeElem(xpTreeElem l, xpElemBase i, xpTreeElem r)
      {
         Left = l; Item = i; Right = r;
      }
   }
```

After this declaration, the class defines a couple of functions to check operator precedence, and then comes the main Evaluate() routine to evaluate the effect of the various **ops** on their **operands**.

Being one big **switch** this should be fairly easy to follow:

```
   public xpOpnd Evaluate(xpOp op, xpOpnd opnd1, xpOpnd opnd2, List<xpError> errors, bool isbetween)
   {
      xpOpnd opnd = new xpOpnd();
      opnd.opndtype = xpOpndType.Number;
      opnd.text = op.text; //so as to give some hint in the error messages
      switch (op.optype)
      {
         case xpOpType.Plus: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype ==
        xpOpndType.Boolean)
               errors.Add(new xpError("Cannot add boolean results ("+opnd1.text +", "+ opnd2.text +")"));
            else opnd.value = opnd1.value + opnd2.value;
            break;
         case xpOpType.Minus: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype ==
        xpOpndType.Boolean)
               errors.Add(new xpError("Cannot subtract boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
            else opnd.value = opnd1.value - opnd2.value;
            break;
         case xpOpType.Mul: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype ==
        xpOpndType.Boolean)
               errors.Add(new xpError("Cannot multiply boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
            else opnd.value = opnd1.value * opnd2.value;
            break;
         case xpOpType.Div: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype ==
        xpOpndType.Boolean)
```

```
        errors.Add(new xpError("Cannot divide boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
    else opnd.value = opnd1.value / opnd2.value;
    break;
case xpOpType.GT: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype == xpOpndType.Boolean)
        errors.Add(new xpError("Cannot take > on boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
    else { opnd.boolvalue = opnd1.value > opnd2.value; opnd.opndtype = xpOpndType.Boolean; }
    break;
case xpOpType.GTE: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype == xpOpndType.Boolean)
        errors.Add(new xpError("Cannot take >-= on boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
    else { opnd.boolvalue = opnd1.value >= opnd2.value; opnd.opndtype = xpOpndType.Boolean; }
    break;
case xpOpType.LT: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype == xpOpndType.Boolean)
        errors.Add(new xpError("Cannot take < on boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
    else { opnd.boolvalue = opnd1.value < opnd2.value; opnd.opndtype = xpOpndType.Boolean; }
    break;
case xpOpType.LTE: if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype == xpOpndType.Boolean)
        errors.Add(new xpError("Cannot take <= on boolean results (" + opnd1.text + ", " + opnd2.text + ")"));
    else { opnd.value = opnd1.value * opnd2.value; opnd.opndtype = xpOpndType.Boolean; }
    break;
case xpOpType.EQ: if (opnd1.type != opnd2.type)
        errors.Add(new xpError("Operand types don't match: (" + opnd1.text + ", " + opnd2.text + ")"));
    else
    {
        if (opnd1.opndtype == xpOpndType.Boolean) opnd.boolvalue = (opnd1.boolvalue == opnd2.boolvalue);
        else opnd.boolvalue = (opnd1.value == opnd2.value);
        opnd.opndtype = xpOpndType.Boolean;
    }
    break;
case xpOpType.NEQ: if (opnd1.type != opnd2.type)
        errors.Add(new xpError("Operand types don't match: (" + opnd1.text + ", " + opnd2.text + ")"));
    else
    {
        if (opnd1.opndtype == xpOpndType.Boolean) opnd.boolvalue = (opnd1.boolvalue != opnd2.boolvalue);
        else opnd.boolvalue = (opnd1.value != opnd2.value);
        opnd.opndtype = xpOpndType.Boolean;
    }
    break;
case xpOpType.Or: if (opnd1.opndtype != xpOpndType.Boolean || opnd2.opndtype != xpOpndType.Boolean)
        errors.Add(new xpError("Cannot take 'OR' except on boolean results (" + opnd1.text + ", " + opnd2.text +
                    ")"));
    else { opnd.boolvalue = (opnd1.boolvalue || opnd2.boolvalue); opnd.opndtype = xpOpndType.Boolean; }
    break;
case xpOpType.Xor: if (opnd1.opndtype != xpOpndType.Boolean || opnd2.opndtype != xpOpndType.Boolean)
        errors.Add(new xpError("Cannot take 'XOR' except on boolean results (" + opnd1.text + ", " + opnd2.text
                    + ")"));
    else { opnd.boolvalue = (opnd1.boolvalue ^ opnd2.boolvalue); opnd.opndtype = xpOpndType.Boolean; }
    break;
case xpOpType.And:
    if (isbetween)
    {
        if (opnd1.opndtype == xpOpndType.Boolean || opnd2.opndtype == xpOpndType.Boolean)
            errors.Add(new xpError("Cannot use boolean results in a BETWEEN (" + opnd1.text + ", " + opnd2.text
                    + ")"));
        else { opnd.leftval = opnd1.value; opnd.rightval = opnd2.value; }
    }
    else
    {
        if (opnd1.opndtype != xpOpndType.Boolean || opnd2.opndtype != xpOpndType.Boolean)
            errors.Add(new xpError("Cannot take 'AND' except on boolean results unless in BETWEEN (" +
                    opnd1.text + ", " + opnd2.text + ")"));
        else { opnd.boolvalue = (opnd1.boolvalue && opnd2.boolvalue); opnd.opndtype = xpOpndType.Boolean; }
    }
```

```
            break;
        case xpOpType.Between:
            if (opnd2.text.ToLower() != "and"  || opnd2.opndtype != xpOpndType.Number)
                errors.Add(new xpError("Wrong argument types for BETWEEN (" + opnd1.text + ", " + opnd2.text + ")"));
            else { opnd.boolvalue = (opnd1.value >= opnd2.leftval && opnd1.value <= opnd2.rightval); opnd.opndtype =
                    xpOpndType.Boolean; }
            break;
    }
    opnd.IsIntermedResult = true;

    return opnd;
}
```

Note that I allow for an unusual operator **between**, e.g.:

```
if X Between(Y and 4)
```

The real business of parsing an expression is handled by:

```
public List <xpElemBase> Parse(ref bool IsValid, out xpOpnd result, List<xpError> Errors)
{
    List<xpElemBase> Polish = new List<xpElemBase>();
    Errors = new List<xpError>();

    IsValid = true;

    OpStack.Clear();
    OpndStack.Clear();
    xpOp op;
    xpOpnd opnd;
    bracketcount = 0;

    int ii = -1;
    foreach(xpElemBase eb in Expression)
    {
        ii++;
        switch (eb.type)
        {
            case xpElemType.Op: op = (xpOp)eb;
                if (op.optype == xpOpType.Close)
                {
                    bracketcount--;
                    if (bracketcount < 0)
                    {
                        Errors.Add(new xpError("Missing Opening Bracket for Closing ')' at " + ii.ToString()));
                        IsValid = false;
                    }
                    ClearStacks(op, ii, Polish, Errors);
                    if (Errors.Count > 0) IsValid = false;
                    continue;
                }
                else
                {
                    if (OpStack.Count == 0) { OpStack.Push(op); continue; }
                    while (Precedence(op, OpStack.Peek()) <= 0)
```

```
            {
               if (OpndStack.Count < 2)
               {
                  //Error
                  Errors.Add(new xpError("Missing Operands for operator "+OpStack.Peek().text+"
                              at " + ii.ToString()));
                  IsValid = false;
               }
               else
               {
                  PopStacks(op, ii, Polish, Errors, op.text);
                  if (Errors.Count > 0) IsValid = false;
               }
               if (OpStack.Count == 0) break;
            }
            OpStack.Push(op);
            continue;
         }
         //break;
      case xpElemType.Opnd: opnd = (xpOpnd)eb;
         if (opnd.opndtype == xpOpndType.Open)
         {
            bracketcount++;
            xpOp opener = new xpOp();
            opener.optype = xpOpType.parseOpen;
            opener.text = "(";
            OpStack.Push(opener);
 //Wegner says put it on the OpStack() although I've never really been happy with it!
 //turns out if out it on the opnd stack it can get "hidden" by the intermediate result of the
           expression in brackets
         }
         else
         {
            xpTreeElem TE = new xpTreeElem(null, opnd, null);
            OpndStack.Push(TE);
         }
         break;
   }
}
ClearStacks(null, ii, Polish, Errors);
if (bracketcount > 0)
 { Errors.Add(new xpError("Missing Closing Bracket  at end of expression" +  ii.ToString())); }
//end of evaluation - value is in Opnd Stack
result = new xpOpnd();
if (OpndStack.Count == 0)
{
   Errors.Add(new xpError("No Operand left at end"));
}
else if (OpndStack.Count > 1)
{
   Errors.Add(new xpError("Multiple Operands left at end"));

}
else
{
```

```
            xpTreeElem TE = OpndStack.Pop();
            bool isbetween = false;
            if (TE.Item.type == xpElemType.Op)
            {
               op = (xpOp)TE.Item;
               if (op.optype == xpOpType.Between) isbetween = true;
            }
            EvaluateTree(TE, Polish, ref result, Errors, isbetween);
         }
         //doing it this way seems to be the only way to guarantee that the order of arguments for -, /, >, <,
                  etc is right
         if (Errors.Count > 0) IsValid = false;

         return Polish;
      }

      void EvaluateTree(xpTreeElem TE, List <xpElemBase> polish, ref xpOpnd result, List<xpError>Errors,
                           bool isBetween)
      {
         xpOpnd opnd1 = null, opnd2 = null;
         xpElemBase eb = TE.Item;
         if (eb.type == xpElemType.Opnd) { polish.Add(eb); result = (xpOpnd)eb; return; }
         else
         {
            xpOp op = (xpOp)eb;
            EvaluateTree(TE.Left, polish, ref opnd1, Errors, false);
            EvaluateTree(TE.Right, polish, ref opnd2, Errors, (op.optype == xpOpType.Between)? true:false);
            polish.Add(op);
            result = Evaluate((xpOp)eb, opnd1, opnd2, Errors, isBetween);
         }
      }
```

This is called only from public xpOpnd EvaluateExpression(xpDissembler XP) in Scores.cs, apart from a test call in the **ExpressionBuilder**.

I would hope that the working of the **Expression Builder** itself, being entirely driven by user choices, should be clear simply from the accompanying User Documentation.