Chapter 4

System Implementation

In the preceding chapter we have seen an overview of MLDFA. Now, we are ready to consider how to incorporate a DFA model into its architecture. In the first section, we represent DFA domain knowledge explicitly by an object-oriented framework. Next, after succeeding in identifying objects and operations within the DFA domain, we integrate these elements into different architectural components based on consideration of commonality and variant. In the last section, we illustrate how a well-designed user interface could help users in working with MLDFA.

4.1 DFA Object-Oriented Framework

A DFA model represents a business entity's view of the world. In the narrowest sense of the term, it is used to represent the forecasting accounting reports in an insurance organization. Given this situation, two things must be defined in an object-oriented framework to let accounting reports can be simulated. They are accounts, and transactions which post to those accounts.

Three kinds of accounts common to all balance sheets are asset, liability and equity. So it follows that these are the three primary objects in the framework. On the side of transactions, although it is common to use transaction object in the real world accounting systems, we implement these transaction events within the methods of each account objects. The main reason is that the transactions in a DFA model are quite simpler than these in the real world. A DFA system is not necessary to have some means for carrying out day-to-day accounting activities. It simply projects balance sheet by stochastic simulation, and assumes that all cash flows take place at end of each time period or an infinitesimal time at the beginning of the next period.

Another feature of a DFA system is its ability to provide reasonable time series outcomes of future global economies risk factors (e.g. interest rates, inflation, stock market returns). These outcomes are then used to drive both the asset and liability sides of the balance sheet. Thus, the top-level hierarchy in our object-oriented framework is as showed in Figure 4-1.

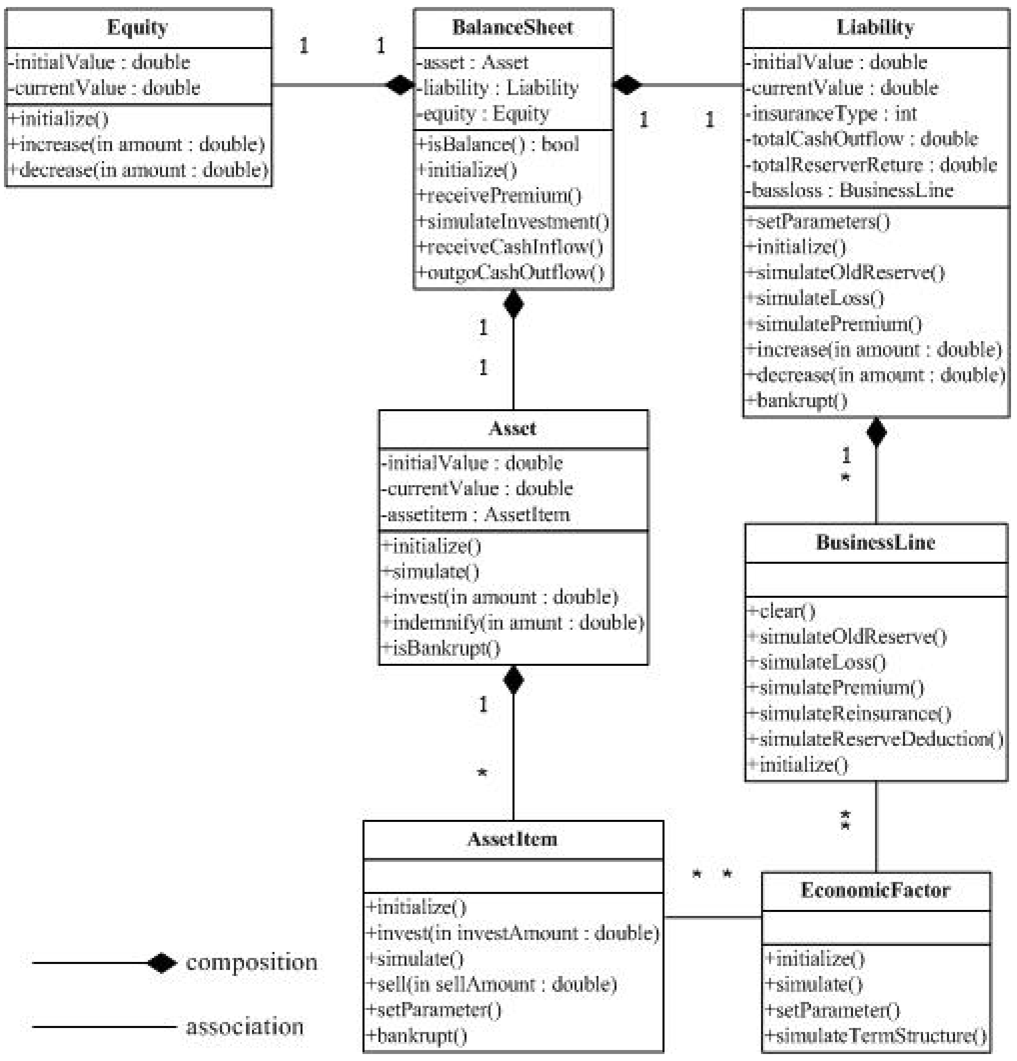


Figure 4-1 BalanceSheet hierarchy

The top-level object is our framework is BalanceSheet. As in the real world, BalanceSheet is composed of Asset, Liability and Equity. Most object-oriented programming languages, including C#, haven't defined special language support for the composition relationship. In MLDFA, we implement this by including an instance of the account object in BalanceSheet.

The same concept applies to both Asset and Liability. Although there are many possible ways to set up an account structure, we set Asset and Liability as composite objects which represent total assets and liabilities of an insurance company. References to each AssetItem and BusinessLine instance are created and saved in an array within these two objects. EconomicRisk deals with the projection of future economic environments. AssetItem and BusinessLine may use EconomicFactor in its simulation. Notation in Figure 4-1 shows this relationship.

AssetItem is an abstract class which defines characteristics common to all asset items in an insurance company. These characteristics are then available by inheritance to more specialized subclasses such as Bond or RealEstate (complete AssetItem hierarchy is presented at Appendix A). These subclasses share the characteristics of AssetItem, and probably add new behaviors.

Unlike AssetItem, liability risk is better described through the line of business. It is important to recognize that each line of business has its own particular business model. Liability projections need to be performed on a line-by-line basis before being aggregated to a total company level. Abstract class BusinessLine describes common characteristics to all business lines. The variable features for each business line are captured in its inheriting classes (complete BusinessLine hierarchy is presented at Appendix B).

Transactions between account objects must be controlled to fit the real case in the insurance company. Asset and liability values are combined with their corresponding cash flow patterns to arrive at new balance at next period. All of this must be done within an accounting framework. For example, when incurred losses are developed, losses paid and losses outstanding must be developed together. A mismatch would generate an inappropriate result in the subsequent financial activities. In our implementation, we utilize the composition hierarchies to deal with this problem. Table 4-1 explains the transaction process of cash receipt activity in each run of simulation. The other transaction processes are summarized in Table 4-2.

Table 4-1 Processes of cash inflow

1. BalanceSheet.receiveCashInflow() invokes Asset.decrease() to decrease some assets value.		
↳	1.1 Asset.decrease() determines which assets (ex. Account Receivable and Reinsurance Receivable) should decrease its value. 1.2 Asset.decrease() computes decreasing amount for each specified AssetItem. 1.3 Asset.decrease() invokes AssetItem.decrease() in each specified AssetItem instances.	
	↳	1.3.1 Specified AssetItem instances execute decrease() to update its value.
2. BalanceSheet.receiveCashInflow() invokes Asset.invest() to allocate new money.		
↳	2.1 Asset.invest() determines which assets are to be bought based on the rebalancing strategy. 2.2 Asset.invest() decides investment amount for each specified AssetItem. 2.3 Asset.invest() invokes AssetItem.invest() in each specified AssetItem.	
	↳	2.3.1 Specified AssetItem instances execute decrease() to update its value.
3. BalanceSheet.receiveCashInflow() invokes Equity.decrease() for bad debts.		
↳	3.1 Equity.decrease() updates its value based on bad debts amount.	

Table 4-2 Summary of transaction processes in framework

Seq.	Control method in BalanceSheet	Subtask in account objects	Description
1	receivePremium() (Carry out premiums written cycle)	1.1 liability.simulatePremium()	Simulate premiums
		1.2 asset.invest()	Invest unearned premium on assets
		1.3 liability.increase ()	Increase unearned premium reserves
2	simulateInvestment() (Simulate investment results)	2.1 asset.simulate()	Simulate capital gains (or losses) from assets
		2.2 equity.increase() (or equity.decrease())	Increase (or decrease) equity from capital gains (or losses)
3	receiveCashInflow() (Perform cash inflows)	3.1 asset.decrease()	Decrease some asset accounts due to premium receipts or reinsurance recovery
		3.2 asset.invest()	Reinvest cash on assets
		3.3 equity.decrease()	Adjust equity for bad debts
4	outgoCashOutflow() (Perform cash outflows)	4.1 liability.simulateLoss()	Simulate loss cycle
		4.2 asset.increase()	Increase Reinsurance Recoverable
		4.3 asset.indemnify()	Make loss payment
		4.4 liability.decrease ()	Decrease Loss Reserves
		4.5 equity.decrease()	Decrease equity when payment amount is greater than the loss reserves

4.2 Arrangement of Commonalities and Variants

The object-oriented framework we provide in previous section is adequate if we implement programs manually. It incorporates engineering knowledge necessary to produce a DFA application. The low level implementation details could be left to the framework users including both Model Managers and programmers to decide the application they need. Once the application meets their requirements, it is released.

In this traditional object-oriented development, there is generally no activity to identify potential, but unspecified needs of the users. To properly address these problems, DFA application must be developed with the expectation of future reuse and change. The main distinction between DSSDSA and other DFA applications is the ability to handle multiple variants of a DFA model.

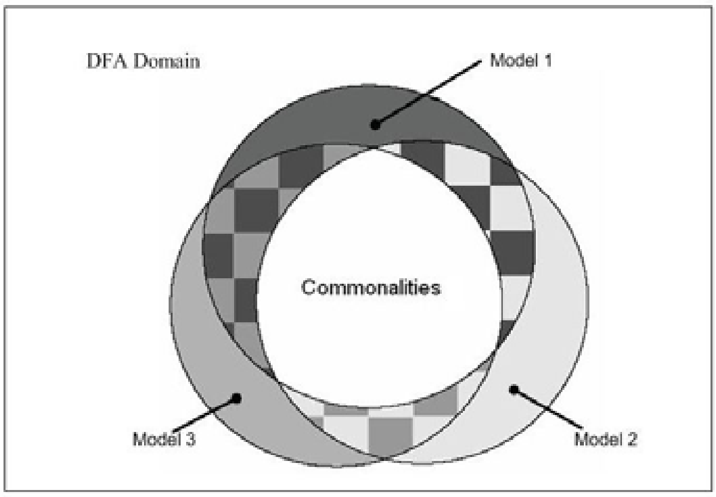


Figure 4-2 Commonalities and variants

As Figure 4-2 indicates, a DFA model may change over time; we call these differences **variants**. Different DFA models may have a common core of characteristics; we call such things **commonalities**. To identity and to organize all the commonalities and variants, the framework we propose seems ideal to be a starting point.

The class hierarchies describe the common relationships among all elements that arise in a set of similar DFA models. A particular model is created by defining an instance of this framework, i.e., supplying concrete subclasses of AssetItem and BusinessLine to provide the necessary customizations. In Figure 4-3, we add a special symbol <V> in our framework to represent variability, and elements that are not tagged by a <V> as a commonality.

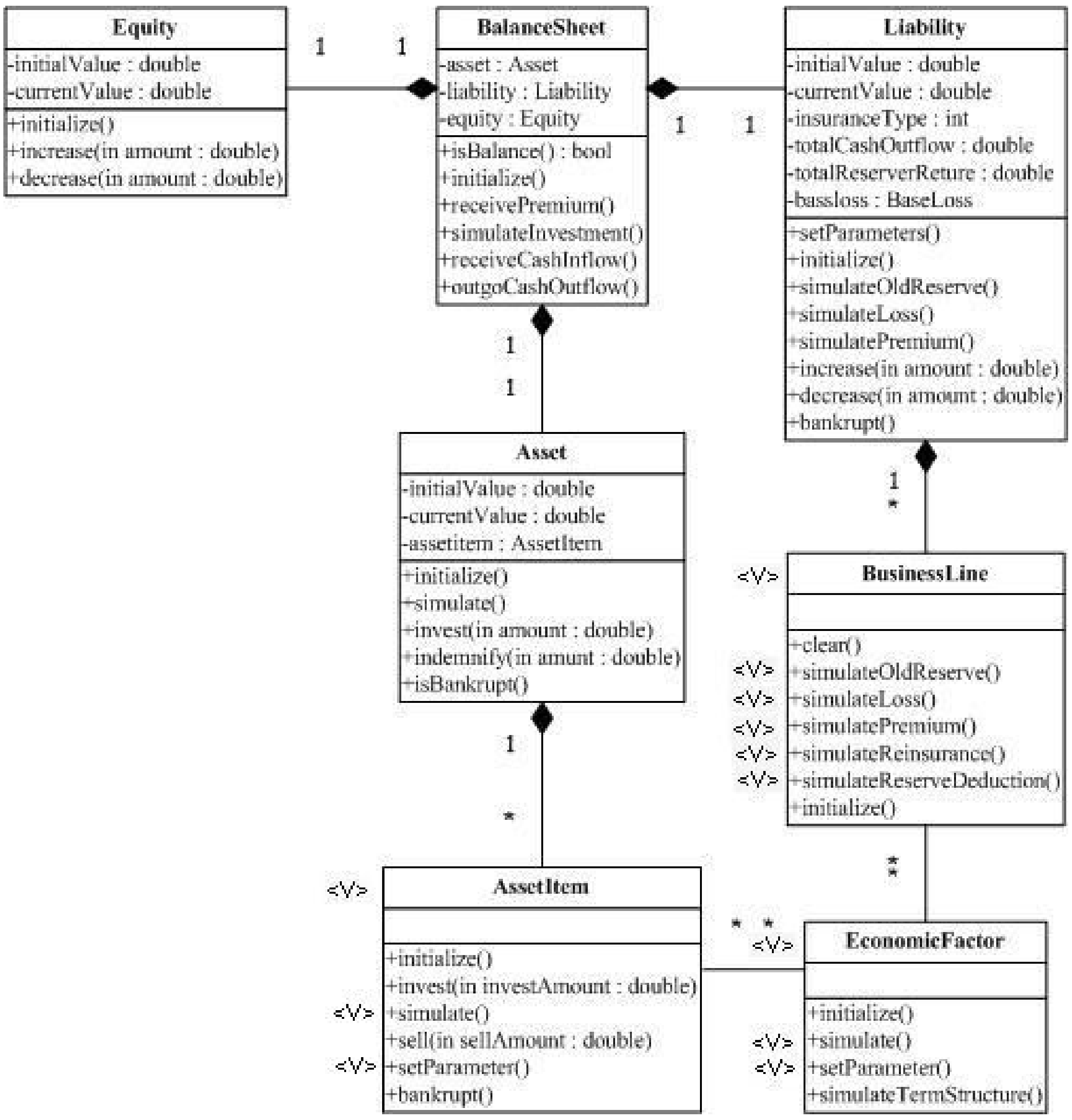


Figure 4-3 Commonalities and variants in framework

Figure 4-3 shows that all DFA models are based on balance sheet (BalanceSheet) simulation. Asset, Liability and Equity are common to all balance sheets. AssetItem, BusinessLine, and EconomicFactor are tagged by a <V>, which means they vary in different models. In each variable class, the same variability concept applies to its methods. Method AssetItem.sell() is always the same even in different instances (new balance = old balance - sell amount). In contrast, Model Manager may use different stochastic differential equations in AssetItem.simulate() to define how the value of an asset item is developed within each time period.

After identifying the common parts and variable parts, the next step is to decide when and how these parts are integrated into MLDFA. Figure 4-4 depicts our implementation strategy.

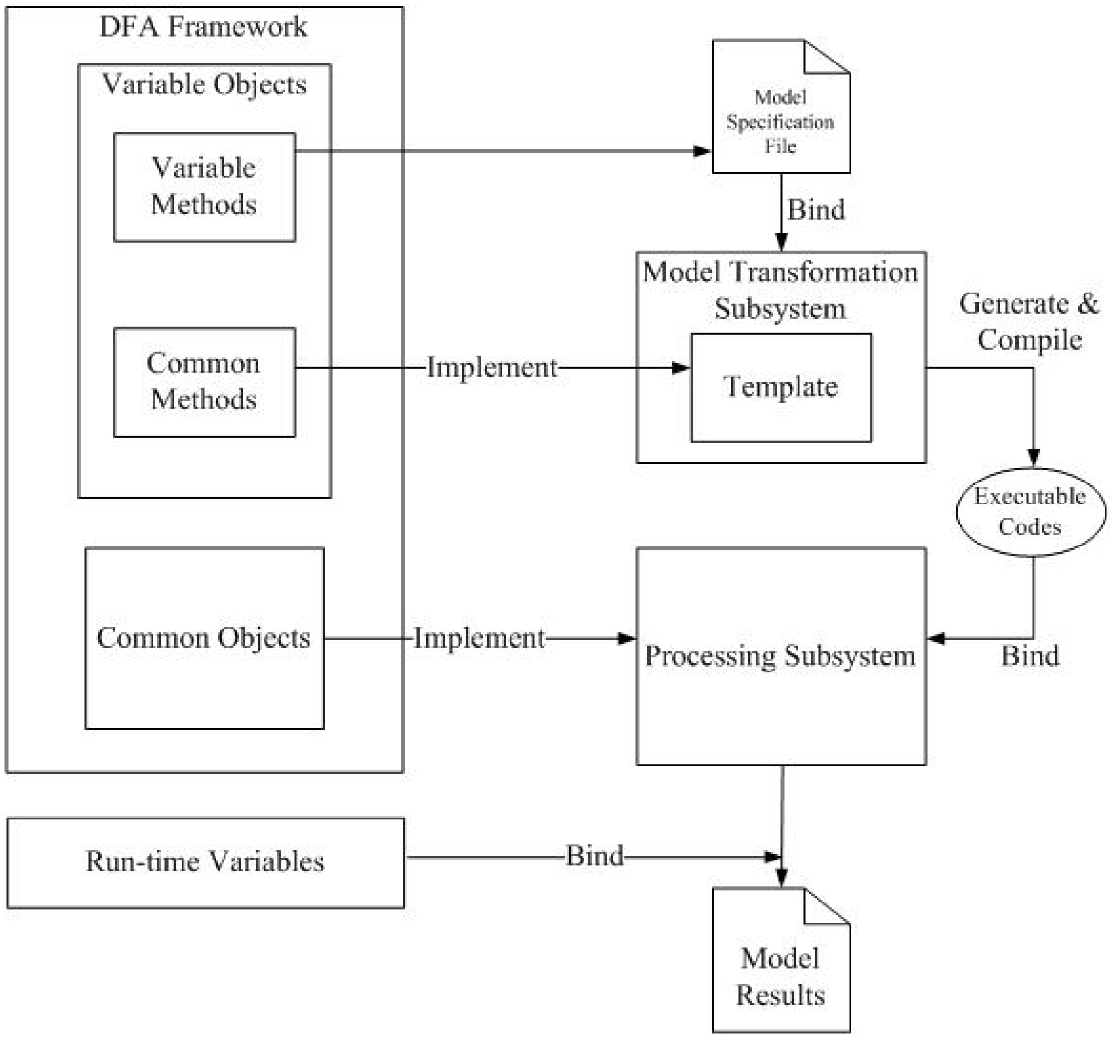


Figure 4-4 Commonalities and variants in MLDFA

Common Objects are shared across a set of DFA models. For increasing productivity, this shared part is implemented once and reused each time in Processing Subsystem. On the contrary, Run-time Variables, such as initial positions for risk factors or some simulation settings, are decisions determined by end users at simulation time. This kind of variables can only be bound into application directly from a user at run-time of Processing Subsystem. Variable Objects reach a certain level of complexity. These objects contain both common parts and variable parts. It leads to a fail in binding them totally at compile-time or run-time.

Code generation mechanism is adopted to solve this problem. The essential idea is to directly implement Common Methods as Templates in Model Transformation Subsystem. Each kind of specialized subclass (ex. account receivable, personal automobile) has its own template which contains most of the behavior for subclasses of that type. Typically, templates in MTS usually generate custom-made classes by following steps:

1. Getting the data: Specifications about Variable Methods are recorded in a XML file. Templates first parse this XML file and store its information into an equivalent data structure. (In C#, a standard XML parser is used to create the DOM(Document Object Model) data structure.)
2. Analyzing and transforming the data: The information is validated by checking for completeness or consistencies. Some transformations are made to change the original specifications into legal named variables or permitted expressions.
3. Generating the codes: A typical template in MTS is simple a series of print statements (`StreamWriter.Write()` in C#) with occasional flow of control statement. This is straightforward in principle, but becomes messy with details. A

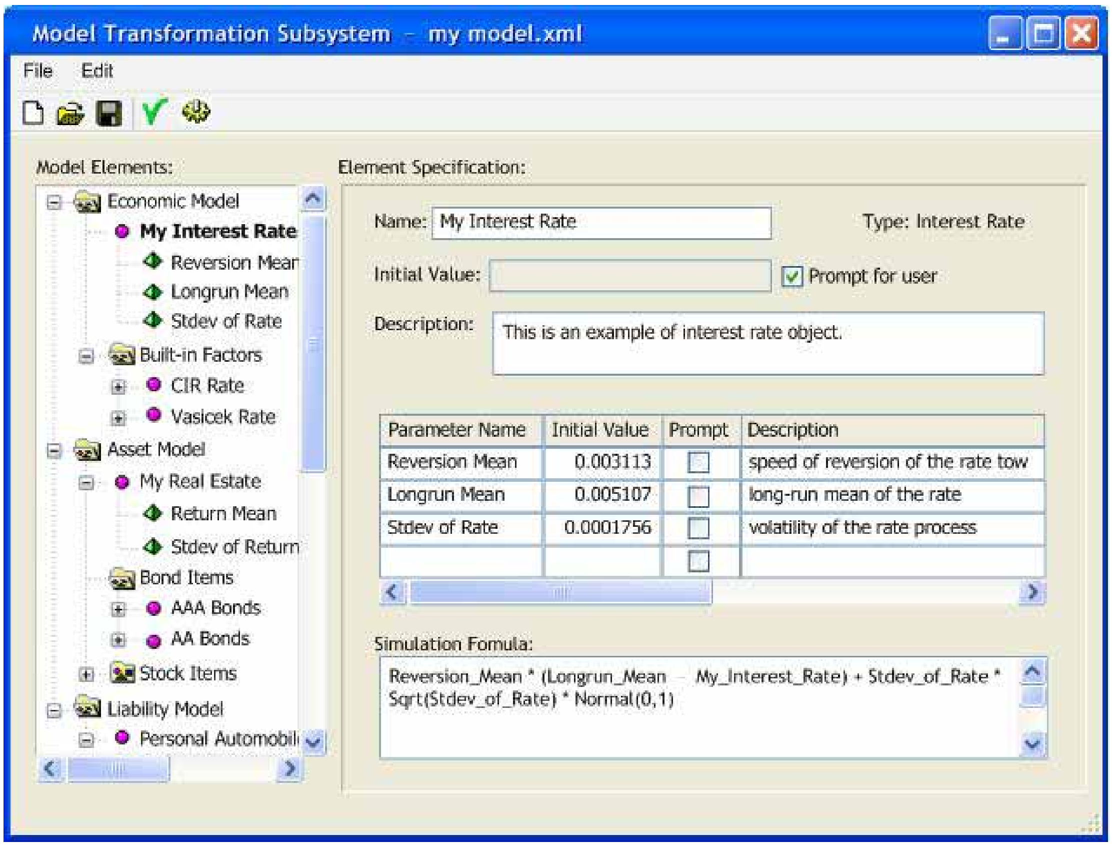
simple example is given in next section alongside the demonstration of user interface design and specification file.

After these generated codes are compiled into Executable Codes, Processing Subsystem binds them at run-time to deal with the simulation run itself. This binding process is simple. As described earlier, a generated class inherits a well-defined interface from an abstract class (AssetItem or BusinessLine). This well-defined interface could be easily composed with Asset and Liability in Processing Subsystem. Asset and Liability do not need to know about the implementation details in generated classes. While simulation runs, they simply instantiate these generated classes and send requests to them.

4.3 User Interface Design

Decision models are given a visual presentation in modern decision support systems, and it is through this visual representation that users can manipulate models. As an ongoing project, MLDFA has not finalized its user interface design. The purpose of this section is to show the possible solution that ensures the concepts addressed in the previous chapters are carried forward into the UI design stage. In order to keep the presentation concise, we do not provide the detailed operational manual here, but focus on demonstrating how a proper UI could help users to create and to manipulate an instance of model concepts.

Perhaps the most interesting advantage of using object-oriented framework plus code generation is that they provide an object-oriented style user interface. Many object-oriented systems only look object-oriented to the programmer. The user does not think in terms of objects, inheritance, and reuse. In MLDFA, the advantages available to the object-oriented programmer are also made available to the user. A prototype of UI for Model Transformation Subsystem is presented in Figure 4-5.



Economic Model
- My Interest Rate
  - Reversion Mean
  - Longrun Mean
  - Stdev of Rate
- Built-in Factors
  - CIR Rate
  - Vasicek Rate

Asset Model
- My Real Estate
  - Return Mean
  - Stdev of Return
- Bond Items
  - AAA Bonds
  - AA Bonds
- Stock Items

Liability Model
- Personal Automobil

Name: My Interest Rate

Type: Interest Rate

Initial Value:

[✓ Prompt for user]

Description: This is an example of interest rate object.

Parameter Name	Initial Value	Prompt	Description
Reversion Mean	0.003113	☐	speed of reversion of the rate tow
Longrun Mean	0.005107	☐	long-run mean of the rate
Stdev of Rate	0.0001756	☐	volatility of the rate process
		☐	

Simulation Fomula:

Reversion_Mean * (Longrun_Mean - My_Interest_Rate) + Stdev_of_Rate * Sqrt(Stdev_of_Rate) * Normal(0,1)

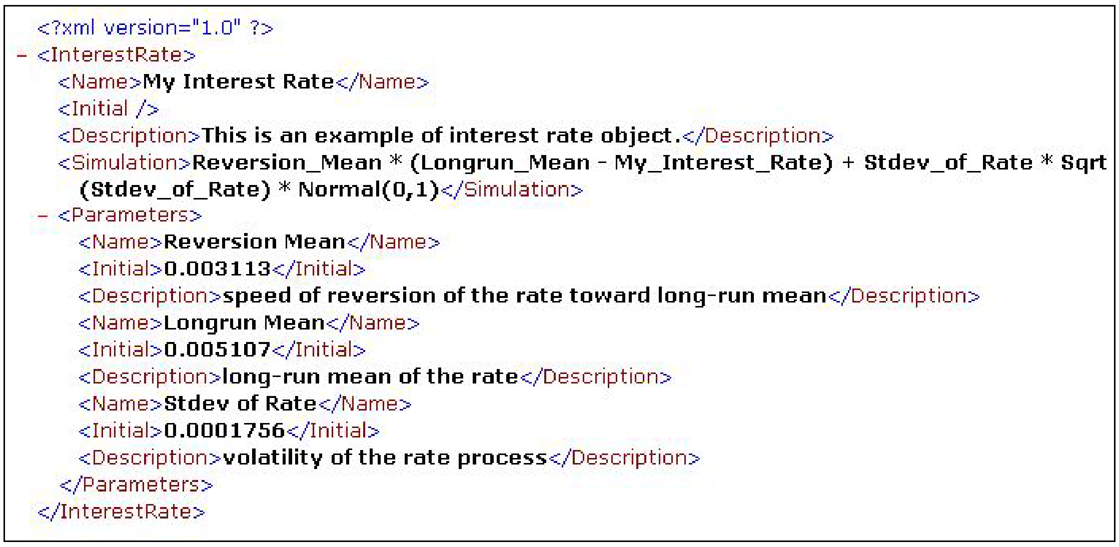
Figure 4-5 Screenshot of Model Transformation Subsystem

The key idea of this design is to explicitly represent knowledge that had been embedded in our framework. Multiple graphical presentation schemes are possible, but an intuitive solution is to use a tree widget to display the class hierarchy in our objected-oriented framework. Based on the tree-like representation, users can see the whole structure of the model that is being generated. Another advantage is tree-like representation is also wildly used by other applications such as Windows Explorer. This explorer-style interface allows users who understand the DFA domain to work with the MTS without having to receive additional training.

Just as working with Windows Explorer, users could use the mouse over the tree widget to display a dynamic pop-up menu that provides commands such as “Add Interest Rate”, “Add Parameter”, “Add Real Estate”, and “Delete”, etc. Under each hierarchy, users create a new element (object), and its specification can be edited on the right panel. Each type of elements has its own contents view, which depends on its characteristics. Figure 4-5 shows Interest Rate contents view allows users to define element’s name, initial position, description, parameter specifications, and simulation formula.

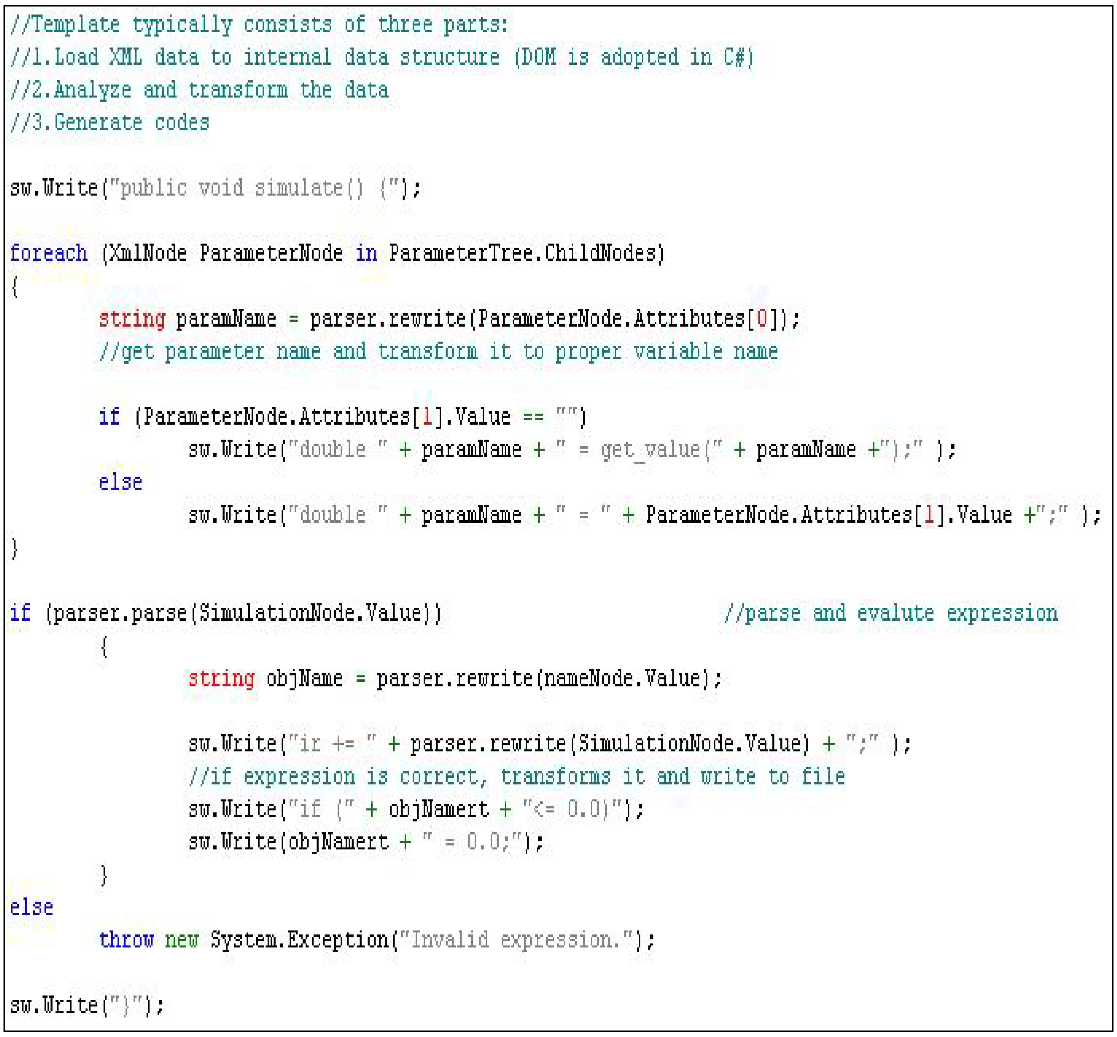
The point to be made is that some mechanisms may help users to define simulation formula appropriately. It is common practice to let users drag and drop necessary elements from tree widget to formula text editor, and automatically make some wording adjustments (ex. replace blank between words with underline symbol). Dynamic pop-up menu which displays available mathematic functions could also aid users.

The model specifications obtained through the user interface will be described in XML file. The paragraph corresponding to the example in above screenshot is presented in Figure 4-6. Once Model Manager completes the whole DFA model specification, it is time for code generation. Concerning above example, the template for interest rate object and its output source codes is presented in Figure 4-7 and Figure 4-8.



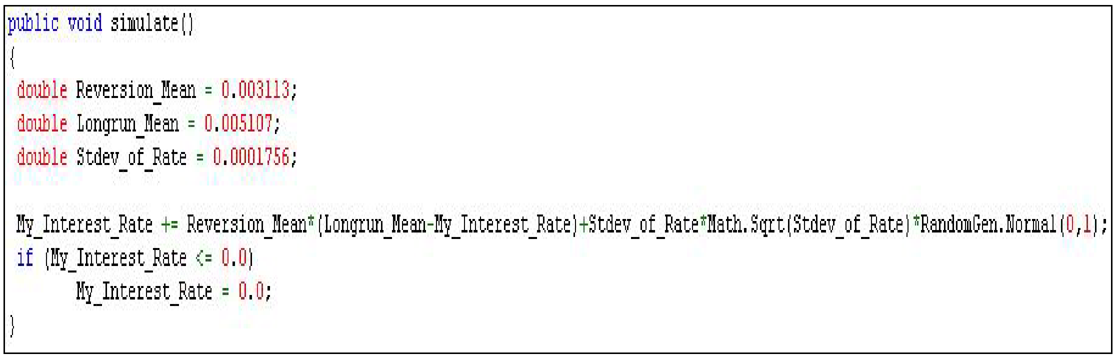
```
  <?xml version="1.0" ?>
- <InterestRate>
    <Name>My Interest Rate</Name>
    <Initial />
    <Description>This is an example of interest rate object.</Description>
    <Simulation>Reversion_Mean * (Longrun_Mean - My_Interest_Rate) + Stdev_of_Rate * Sqrt
      (Stdev_of_Rate) * Normal(0,1)</Simulation>
  - <Parameters>
      <Name>Reversion Mean</Name>
      <Initial>0.003113</Initial>
      <Description>speed of reversion of the rate toward long-run mean</Description>
      <Name>Longrun Mean</Name>
      <Initial>0.005107</Initial>
      <Description>long-run mean of the rate</Description>
      <Name>Stdev of Rate</Name>
      <Initial>0.0001756</Initial>
      <Description>volatility of the rate process</Description>
    </Parameters>
  </InterestRate>
```

Figure 4-6 Example: Partial Content of XML



```
//Template typically consists of three parts:
//1.Load XML data to internal data structure (DOM is adopted in C#)
//2.Analyze and transform the data
//3.Generate codes

sw.Write("public void simulate() {");

foreach (XmlNode ParameterNode in ParameterTree.ChildNodes)
{
        string paramName = parser.rewrite(ParameterNode.Attributes[0]);
        //get parameter name and transform it to proper variable name

        if (ParameterNode.Attributes[1].Value == "")
                sw.Write("double " + paramName + " = get_value(" + paramName +");" );
        else
                sw.Write("double " + paramName + " = " + ParameterNode.Attributes[1].Value +";" );
}

if (parser.parse(SimulationNode.Value))                          //parse and evalute expression
        {
                string objName = parser.rewrite(nameNode.Value);

                sw.Write("ir += " + parser.rewrite(SimulationNode.Value) + ";" );
                //if expression is correct, transforms it and write to file
                sw.Write("if (" + objNamert + "<= 0.0)");
                sw.Write(objNamert + " = 0.0;");
        }
else
        throw new System.Exception("Invalid expression.");

sw.Write("}");
```

Figure 4-7 Example: Template for Interest Rate Object (Partial)



```
public void simulate()
{
 double Reversion_Mean = 0.003113;
 double Longrun_Mean = 0.005107;
 double Stdev_of_Rate = 0.0001756;

 My_Interest_Rate += Reversion_Mean*(Longrun_Mean-My_Interest_Rate)+Stdev_of_Rate*Math.Sqrt(Stdev_of_Rate)*RandomGen.Normal(0,1);
 if (My_Interest_Rate <= 0.0)
        My_Interest_Rate = 0.0;
}
```

Figure 4-8 Example: Output codes for Interest Rate Object (Partial)