Chapter 3

System Analysis and Design

In this chapter, we first discuss some of the issues which arise in designing and implementing a DFA system; then, we suggest the possible ways to construct an improved DFA decision support system.

3.1 System Analysis

In this section, we describe the requirements for building a DFA system. Although we begin with an overview of the DFA process based on the experience of a real DFA project, we have the viewpoint of information system development as the object of this section; therefore, the detail operations from the viewpoint of DFA domain has been kept to a relative few. For a more complete discussion of DFA models see references mentioned in Chapter 2.

The process of DFA spans 8 steps:

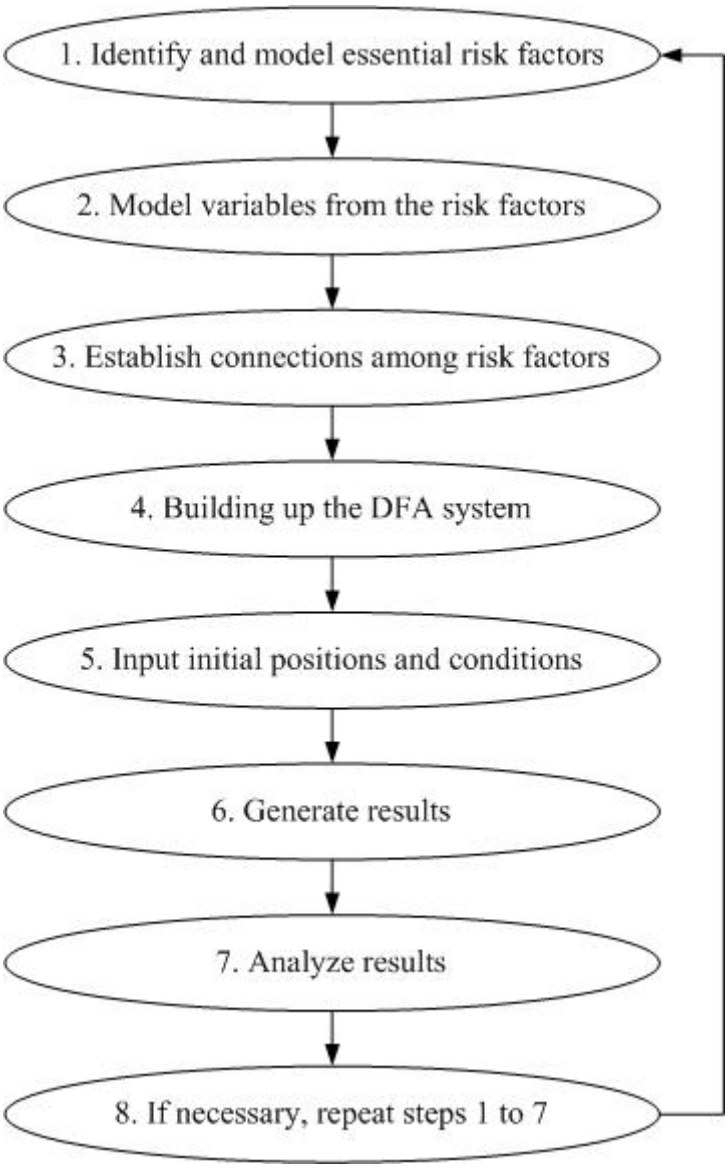

Figure 3-1 DFA process

Step 1: Identify and model essential risk factors

The first step in building a DFA model is to decide the risk factors faced by an insurance enterprise. Since the DFA model is going to project the balance sheet and operating statement of the insurer over the planning horizon, the most relevant risks that affect assets, liabilities, underwriting, or investment income need to be considered. Table 3-1 gives an overview of risk factors typically included in DFA models.

Table 3-1 Overview of risk factors

Global Economic	Asset	Liability	Business
Interest rates Exchange rates (etc.)	Cash Account receivable Bonds Stocks Real estate Foreign investment (etc.)	Loss reserves Loss development (etc.)	Underwriting cycle Reinsurance cycle (etc.)

Step 2: Model variables from the risk factors

This step is to decide how the risk factors are assumed to behave over the forecast horizon. To define individual risk factors, many possible models from actuarial science, finance and economics are available. Take interest rate risk for example, Vasicek Model and Cox, Ingersoll, Ross (CIR) Model are widely adopted. (Ahlgrim, Arcy, and Gorvett, 1999)

Step 3: Establish connections among risk factors

Many of the important risk factors of DFA model are complexly interrelated. This step focuses on consolidating different risk factors together to reflect its internal structure. For example, a change in interest rates would lead to a movement in the value of existing assets.

Step 4: Building up the DFA system

In this step, the conceptual DFA model derived from above steps is coded into a computer-recognizable form. There are several ways used for constructing a computer program. One method is using programming language to implement a simulation model. Yet some degree of expertise in the language is needed. The other way is to

construct a simulation program by the use of the DFA software packages. However, no matter which way is adopted, this step typically involves sophisticated computer modeling techniques.

Step 5: Input initial positions and conditions

Necessary data are collected and fed into the software. These inputs include:

1. The financial state of the company at the time of the simulation.
2. The assumptions to be tested by the DFA model, such as investment strategy and reinsurance strategy.
3. Statistical and economical information derived from the examination of historical data or by using the public data.

Step 6: Generate results

Perform the simulation to generate results. Simulation results include the outcome from a single execution of a model or those results which might require several executions with different initial positions and conditions.

Step 7: Analyze results

Given the stochastic simulation, a large number of output values and entire distribution of these outcomes are available for analysis. Therefore, sophisticated analysis becomes necessary for extract information from the output.

Step 8: If necessary, repeat steps 1 to 7

Figure 3-1 implies that a DFA model is not only defined and developed but is continually refined, updated, modified, and extended. The modeling process is evolutionary. As the regulatory and competitive environment changes, management

will be interested in incorporating changes to stay along with the times. The DFA models need to be continually refined to respond to these requests.

DFA represents a new area requiring new tools and new expertise. We use Model Manager, who has a good working knowledge of DFA modeling, to represent a power user of a DFA system. A DFA system is not only software; it is a process of gaining knowledge through the combination of model, data and analysis. As Figure 3-2 indicates, DFA processes mentioned above can be separated into three phases which focuses on constructing model, inputting data, and analyzing results, respectively. Close interaction among these three phases is required when formulating a problem and building a model.

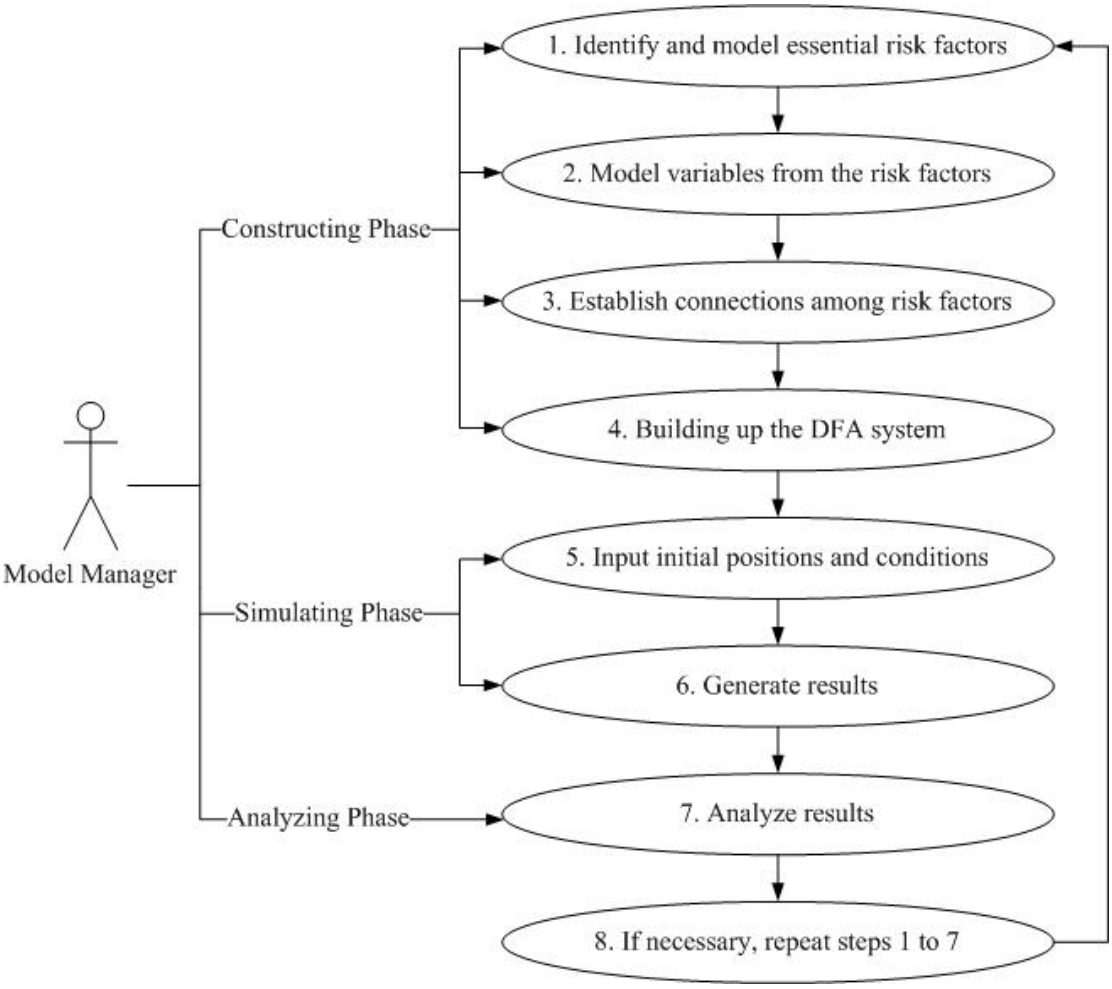

Figure 3-2 Three Phases in DFA Processes

3.2 Design Philosophy for a New DFA System

The main challenge of building a DFA system is that, after constructing the conceptual model which exists in the mind of the Model Manager, next step is to transform these DFA model specifications into the programmed model that admits execution by a computer to produce simulation results. In general, the model specifications are much easier to read, write and understanding than the code that implements the specification. The implementation code may contain detailed knowledge about the programming techniques or algorithms. Often, the implementation process involves the efforts of IT personnel as Figure 3-3 indicates.

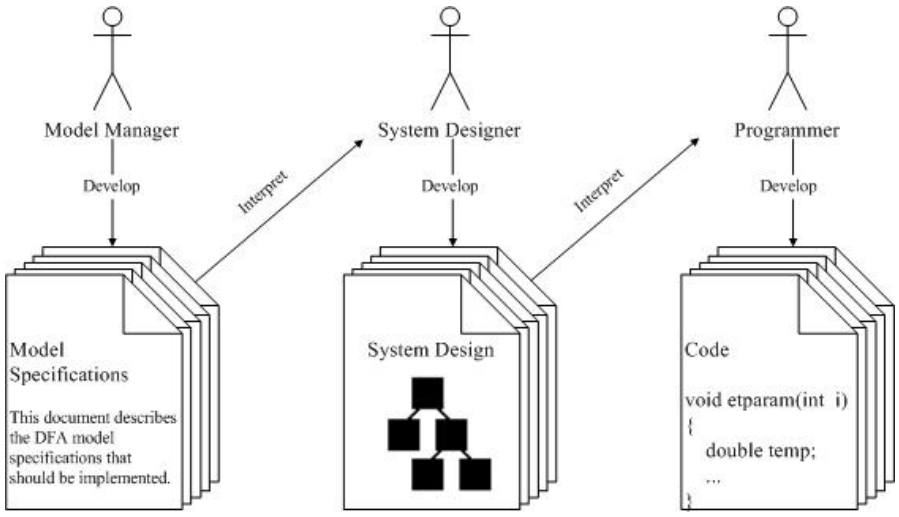

Figure 3-3 Conventional DFA computer system development process

Obviously, a DFA system developed should help users to streamline all processes discussed previously. In other words, a truly flexible solution is to offer a mechanism of performing model transformation automatically. Thus, the Model Manager could build and refine DFA model without burdening the IT resource. The resulting system

is referred to as Modeling Language for Dynamic Financial Analysis (MLDFA). The high-level overview of system components is depicted in Figure 3-4.

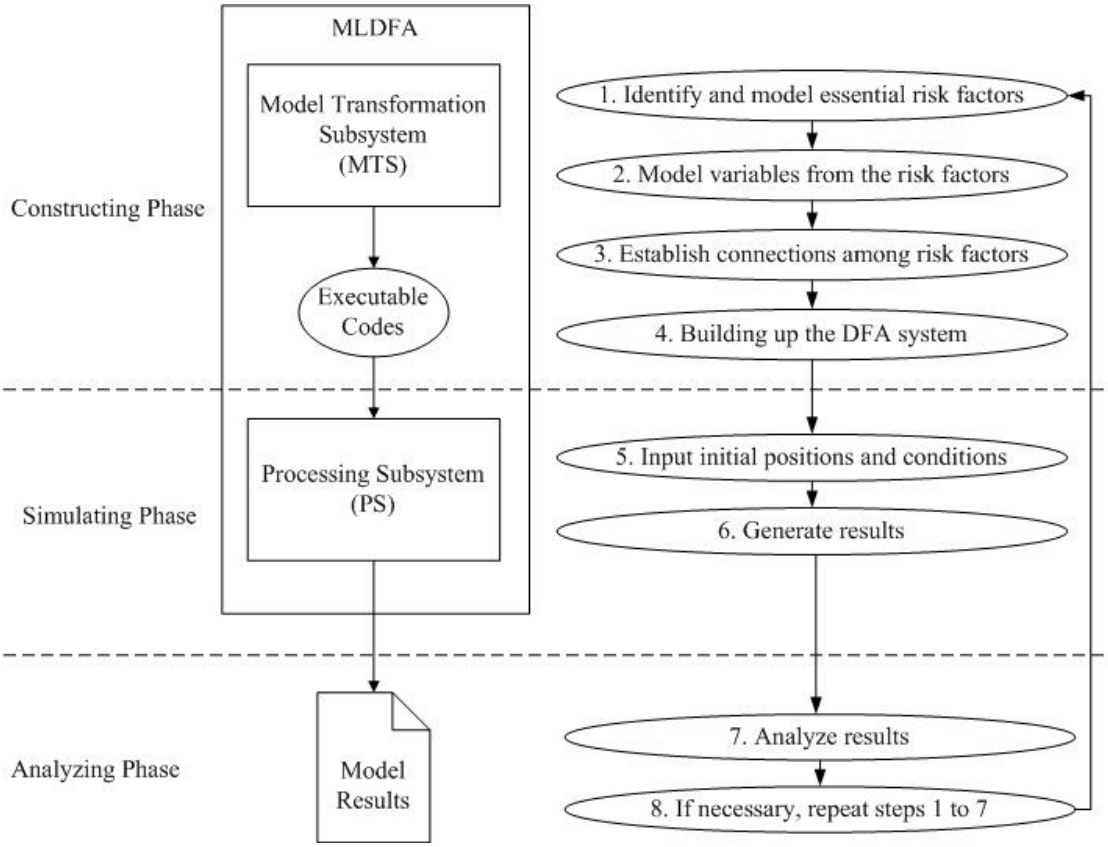

Figure 3-4 High level overview of MLDFA

MLDFA has two subsystems that are built separately to ensure that the required features are presented.

- Model Transformation Subsystem (MTS): The task of building model is accomplished through the MTS. Working with MTS is fairly simple as MTS offers graphical user interface that let Model Managers use their familiar business language to create the model. After creating the whole DFA model, MTS generates Executable Codes based on the specification defined by Model Managers.

- Processing Subsystem (PS): Processing Subsystem is a simulation execution platform that provides interactive control for starting, stopping, and monitoring the simulation. It loads Executable Codes generated by MTS and display the corresponding interface for inputting initial positions and simulation preferences.

Rather than seeking an answer in additional technical functionality, we propose a new approach to make a transformation from the conceptual model into programmed model as easy as possible. The basic idea is that, by combining conceptions of Object Oriented Programming (OOP) and Code Generation, MTS could automatically generate the correct target codes.

- Object Oriented Programming: From the viewpoint of OOP, all risk factors in a DFA model could be described in terms of ‘objects’. For example, by investing, selling or market movements, the value of asset items is affected. The ‘value’ could be viewed as the attribute/property of the asset object, and the behaviors such as investing, selling and market movements could map to the different methods.
- Code Generation: The model specifications defined by the Model Manager eventually should be implemented. For each risk factor, Code Generation concept provides a simple and powerful mechanism for creating a new object stemming from its specifications. By the use of templates, which are sequence of programming language containing ‘holes’ in place of some values, target codes are generated at run time by simply copying templates and filling ‘holes’ with values computed at run time.

For illustration purposes, let’s take a very simple example. The Model Manager considers the risk factor, Real Estate, has the following characteristics:

1. The current value of the real estate is determined at simulation time.
2. At next period:

   2.1 The value of this investment is estimated by the formula:

   new value = old value + old value*(avg. of growth rate + std. of growth rate* normal(0,1))

   2.2 The rent income from the real estate is estimated by the formula:

new rent = old rent + old rent*(avg. of rent growth rate + std. of rent growth rate * normal(0,1))

3. The total amount of this investment increases by the allocation of new cash.
4. The total amount of this investment decreases by being sold out for liability payments.

The object-oriented representation of above specifications and partial output source codes are as follows:

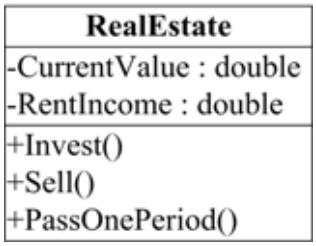

Figure 3-5 Example: Object entity

```
public class RealEstate
{
	public double CurrentValue;
	public double RentIncome;

	public void Invest(double amount)
	{
		CurrentValue += amount;
	}

	public void Sell(double amount)
	{
		CurrentValue -= amount;
	}

	public void PassOnePeriod(double AvgGrowth, double StdGrowth,double AvgGrowthRent, double StdGrowthRent)
	{
		CurrentValue = CurrentValue + CurrentValue*(AvgGrowth + StdGrowth*UserFunction.gaussian(0,1));
		RentIncome = RentIncome + RentIncome*(AvgGrowthRent + StdGrowthRent*UserFunction.gaussian(0,1));
	}
}
```

Figure 3-6 Example: Source codes

There are several advantages of this approach.

- Separation of concerns: Model Manager focuses on the DFA model specifications. By given GUI with a proper graphical representation of object-oriented concept, Model Manager is able to express their specific concept, directly in their everyday terms and independently of resolution and implementation concerns.
- Correctness of transformations: Based on the model specifications, MTS may generate thousands of lines of code that are far more reliable than if they were hand crafted. If solving this problem by handing the model specifications to some programmers and asking them to write out the codes by hand, it is difficult to make sure the job was done correctly.
- Reuse of codes: Reusability is a central means to improve software development. It is inefficient to implement an application by “copying and pasting” duplicate codes. On the contrary, template is a basis of shared codes. Any number of

copies can be created and being modified in the course of the generation process. It leads to shorter development times because similar objects are not built from scratch each time.

Taking above advantages and working together with other components in MLDFA, a flexible platform for building DFA solutions is comprised. In the next section, we will give the details of how this is done.

3.3 System Architecture

Figure 3-7 shows the entire MLDFA architecture. The following sections concentrate on each single component in the system and explain the way they are related to one another.

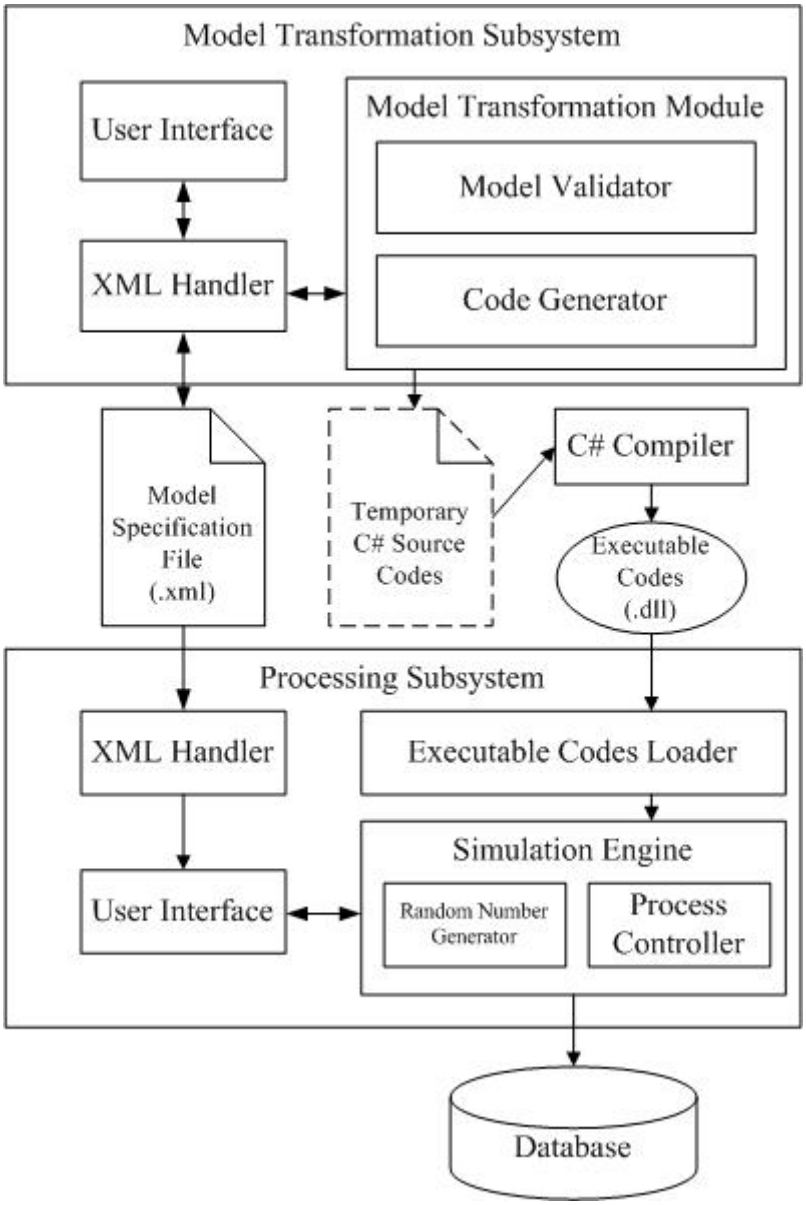

Figure 3-7 System architecture of MLDFA

3.3.1 Model Transformation Subsystem

User Interface

A component allows Model Manager to define specifications of each DFA model in a graphical environment. These specifications are sent to the XML Handler for saving in Model Specification File. The main goal of this component is to provide a presentation structure which Model Managers feel compatible with their cognitive world. Fuller discussion of the interface design will be present in the next chapter.

XML Handler

The specifications of a DFA Model should be saved to an independent file for possible modifications in later days. We advocate the use of eXtensible Markup Language (XML) as a representation for the specifications. Each DFA model consists of several risk factor categories, and each category has many elements with particular attributes and behaviors. This hierarchy could easily map to the tree structure of XML. The standard XML Parser in .Net Framework is implemented in XML Handler. After reading an XML document and parsing its contents into a tree data structure, XML Handler supplies this information to the User Interface and Model Transformation Module.

Model Transformation Module

Model Transformation Module is responsible for generating Executable Codes based on the model specifications. To avoid the consistent-updated problem, it asks user to save any modification to the Model Specification File before starting transforming. In the beginning, it generates Temporary C# Source Code in the memory. Then C# Compile is triggered to compile source codes into Executable Codes. Model Transformation Module can be split into two main components: Model Evaluator and Code Generator

Model Validator

A mathematical expression defined by users is validated for completeness and consistency. Model Validator looks for model specifications errors. For example, it makes sure that there are no undeclared variables. A finite state machine is implemented in Model Validator to identify each individual token in the expression.

Once a list of tokens has been generated, and each assigned an appropriate type (operator, number, etc), the syntax of expression is checked. If any error is found, Model Evaluator asks users to correct mistake and stop the code generation process. The accurate expression is restructured in conformity with C# and passed to the Code Generator for further manipulation.

Code Generator

Code Generator converts high-level model specifications into actual source codes. Once model specification is analyzed, and perhaps transformed, it is time for code generation. By a template-based approach, Code Generator generates Temporary C# Source Codes organized in the objected–oriented style. For each risk factor, a new class is created, and custom codes may be generated in its method. More information about the object model and code generation process is presented in the next chapter.

3.3.2 Processing Subsystem

User interface

User Interface provides GUI to define simulation settings. It displays a dynamic form for inputting model parameters (ex. initial positions) and simulation preferences (ex. time horizons). It also allows users to monitor simulation status and results.

Executable Codes Loader

Executable Codes created by Model Transformation Subsystem is loaded at run-time of Processing Subsystem. Assembly and Reflection API in C# are used to find out the contents of Excusable Codes. Such information, including what classes and interfaces Executable Codes implement, is sent to Simulation Engine for later execution.

Simulation Engine

As implied by the name, Simulation Engine plays a main role of executing simulation. It consists of two components, Random Number Generator and Process Controller.

Random Number Generator

When the computational side of stochastic simulation is considered, there seems to be a need for a random number generator. It is able to produce many kinds of independent and identically distributed random variables (ex. normal, uniform, exponential, and Poisson distributions). Based on its outputs, forecasting values for each risk factor are calculated.

Process Controller

In a sense, Process Controller could be viewed as the accounting framework for a DFA model. It projects various kinds of accounting cycles by invoking different methods in Executable Codes in proper order. The details will be presented in the next chapter.