Modeler's Workbench User Guide

1. INTRODUCTION: WHAT IS THE WORKBENCH? 3

2. BACKGROUND YOU SHOULD HAVE 4

2.1. Accounting 5
2.1.1. The chart of accounts 5
2.1.2. Double-entry accounting with debits and credits 6
2.1.3. Non-cash accruals 10

2.2. Basics of asset-liability modeling 11
2.2.1. Invested asset background 11
2.2.2. Liability product background 11
2.2.3. Principle-based liability valuation 12

2.3. Software background 13
2.3.1. C# language and object-oriented design concepts 13
2.3.1.1 Obect-oriented design concepts 14
2.3.1.2 Advantages over C++ and Visual Basic 15
2.3.2. Object-oriented implementation of a model 15
ContractClass 18
InvestmentClass 20
InforceBlockClass 21
InvPortfolioClass 23
ControllerClass 24
AsmpFileClass 27
CashFlowClass 27
InvStrategyClass 27
LedgerClass 27
ResultsDictionaryClass 28
RunLog 28
RunParmsClass 28
ScenarioManagerClass 28
TableManager 28

3. HOW TO USE THE WORKBENCH 29

3.1. The model development screens – the user interface 29

3.2. Understanding file types 33

3.3. Developing your first model **35**

3.4. Ideas and approaches for more complex models **51**
Adding formulaic liability calculations........ 51
Separating results by year of issue 54
Contract changes such as annuitization 55

4. HOW TO USE MODELS PRODUCED BY THE WORKBENCH **56**

4.1 The top level user interface **56**

4.2 Files used by the model **57**
4.2.1 Input files 58
4.2.2 Output files 58
4.2.3 Table files 59
4.2.4 Model support files 59

4.3 Working with scenarios **61**
4.3.1 Defining risks 61
4.3.2 Generating scenarios 63

4.4 Running the asset / liability simulations **64**

4.5 Viewing the output **65**

5. APPENDICES **69**

Defining and using assumptions **69**

Actuarial tables and the Table Manager **71**

Chart of accounts and the ledger **72**

Report writer **75**

Risk definition files **76**

Principle-based valuation and the inner loop **79**
How the model controller handles inner loops 79
Requesting that the inner loop be included in a model run 81
Generating the inner loop scenario(s) 81
Adding margins in the inner loop assumptions 83
Capturing and using results of inner loop scenarios 84

Principle-based margins and the Representative Scenarios Method (RSM) **84**

The source code generator **86**

Statutory reserves and BFReserveLib.dll **92**

1. Introduction: What is the Workbench?

The Modeler's Workbench is a software platform for developing asset-liability projection models of the kind commonly used by actuaries. Such models are used in financial enterprises of all kinds, but especially in insurance companies and pension funds. In recent years the use of such models has become a staple in activities including risk management, regulatory compliance, financial reporting, and valuation.

Comparison to other asset-liability projection models

Asset-liability models vary widely in their complexity, from simple worksheets in Microsoft Excel, to full-blown enterprise-level systems for large institutions. The Modeler's Workbench fills a niche between those extremes. The Workbench provides a platform for model development with all the flexibility of a worksheet, but with a software structure designed for dynamic asset-liability modeling like the major enterprise modeling systems. The Workbench is an open-code platform, making it easy to change and experiment with a model. This contrasts with enterprise platforms that are often closed-code and burdensome to change. As a result, the Workbench is most suited to prototyping models for new products, exploring the effect of proposed new accounting rules, alternate investment strategies, and other possibilities. It is also useful for research into model efficiency techniques.

Because its structure is parallel to the structure of enterprise models, it can also be used for independent validation of enterprise model results.

While the Workbench can be used to develop an enterprise model, such use requires addition of governance and other controls that are not a part of the Workbench. In creating the Workbench the emphasis was on modeling flexibility and open-code, characteristics that are at odds with the controls desired in enterprise model implementations. Enterprise models generally come with lots of (expensive) support from a large consulting organization. With the Workbench you are largely on your own. You can add controls and governance yourself, of course, and there is at least one large insurer that has done so with a home-grown system like the Workbench. But that company has a permanent paid professional staff dedicated to maintenance and use of the model (instead of external consultants).

Primary output and analysis capabilities

The primary output of models developed using the Workbench is projected financial statements. Such statements encapsulate all future financial activity including cash flows, investment returns, profitability, and capital levels. Projections can also include activity counts and volume of business, depending on the type of liability being modeled. A great variety of analyses are enabled by having such projections.

The workbench includes a stochastic economic scenario generator, and automates the process of running sets of stochastic scenarios and viewing the distribution of any projected result.

The workbench also includes a random number generator, so stochastic techniques can be applied to simulation of claims, other actuarial decrements and contractholder behavior.

Importantly, any model developed using the workbench is capable of cash-flow-based valuation of liabilities on future dates, under assumed future economic conditions. This requires "model-within-a-model" capability, where the state of the simulation is stopped at a future valuation date, and a new projection (or stochastic projections) for valuation purposes is done starting from that date. After the valuation is completed, the model restarts from where it was stopped. This capability provides for realistic projection of financial results under accounting frameworks that require cash-flow-based valuation of liabilities, including IFRS and in some cases US statutory accounting.

Ultimate flexibility

While the Workbench is an asset-liability model platform, the liability side of the model is completely open. Figuratively, you start with a blank sheet of paper when defining and insurance contract, a pension plan, or any other sort of financial liability that is backed by invested assets. You must define the records that are kept for the liability contract, the way transaction amounts are calculated, and the way transactions are recorded for accounting purposes. A structure for such definitions is provided, but the definitions themselves are completely open. That means that the liability could be anything including but not limited to the following:

- Insurance contracts of any kind
- Pension liabilities, including retiree health care benefits
- Bank deposits (i.e. checking accounts, time deposits)
- Continuing care retirement community obligations
- ESOPs (Employee Stock Ownership Program liabilities for liquidation upon separation from employment)

Asset-liability models of the kind developed using the Workbench are useful in connection with any financial arrangement where money is collected in advance and invested in order to provide payments of some kind in the future. Uncertainties regarding investment returns and the amount and timing of future payments represent risks that must be managed. The Workbench can be used to develop models that simulate those risks.

2. Background you should have

An asset-liability projection model for a financial enterprise is inherently complex. To develop such a model using the Workbench, you need not only a deep professional background in actuarial aspects of financial contracts and their liabilities and risks, but also in fundamentals of accounting and computer

software coding. Subsections below summarize the background required in the areas of accounting, asset / liability modeling, and software coding. Readers can skip any sections covering background they already have. All users will, however, need to become familiar with the material in section 2.3.2 – The Object-oriented Implementation of a Model.

2.1. Accounting

The primary output of an asset-liability model is projected financial statements, which are assembled from projected accounting records. The model developer needs to understand both how the accounting records are maintained and how financial statements are assembled from those records.

2.1.1. The chart of accounts

The chart of accounts is simply a list of every quantity that is to be recorded for each accounting time period (normally each month in this context). Each quantity corresponds to a ledger account, and the list of such accounts is called the chart of accounts.

There are five categories of ledger accounts. The table below lists the categories and gives some examples of each:

Category	Category description	Example ledger accounts
Assets	Cash and invested assets Amounts owed to us	Cash on hand Investments (bonds, stock) Occupied real estate Tax refunds not yet received
Liabilities	Amounts owed to others	Insurance claims not yet paid Potential future insurance claims Taxes owed but not yet paid
Income	Cash taken in, change in amounts owed to us.	Insurance premiums Interest earned on investments Capital gains
Disbursements	Cash paid out, change in amounts owed to others.	Insurance claims payments Operating expenses Taxes Increase in value of liabilities
Capital and surplus	Net worth (=Assets – Liabilities)	Paid in capital Retained earnings
Other (not traditionally part of the chart of accounts, but included here as used in the Workbench)	Inventory counts, activity counts, required disclosure items	Number of new contracts sold Number of insurance claims Total face amount of insurance

A ledger can be envisioned as a list of account balances, that is, a list of numbers where each number corresponds to the amount recorded in a particular account.

The line-items on financial statements are calculated based on the amounts (balances) in ledger accounts. For example, net profit is calculated as (Total Income) minus (Total Disbursements). There is no item in the chart of accounts for Net Profit; it is calculated based on items that do appear in the chart of accounts.

In the Workbench, you can define your own chart of accounts. This allows flexibility to simulate any set of accounting rules, because different sets of accounting rules sometimes define the items to be reported in different ways.

In the Workbench, the formulas for line-items on financial statements (such as net profit) are also defined in the chart of accounts. Instead of containing an account balance, each such item contains a formula based on other items in the chart of accounts. This is discussed further in the Appendix.

2.1.2. Double-entry accounting with debits and credits

What are debits and credits? A debit is an addition to an account balance. A credit is a subtraction from an account balance. A debit balance is a positive balance. A credit balance is a negative balance.

In a double-entry accounting system, every transaction is recorded by a set of debits and credits that add to zero. As a result, if you add all balances in the chart of accounts (excluding the “Other” category), you should always get exactly zero. That gives a simple but profound assurance that the accounts are in balance.

So how do we produce financial statements when all the account balances always sum to zero? Let’s work through a simple example involving one year of activity for a block of life insurance contracts.

Here is the chart of accounts, along with the balances in each account at the beginning of the year:

Account	Category	Acct Type	Beginning Balance
Insurance premium income	Income	Credit	0
Investment income	Income	Credit	0
Death claims	Disbursement	Debit	0
Expenses	Disbursement	Debit	0
Change in liability for future claims	Disbursement	Debit	0
Cash and invested assets	Asset	Debit	1000
Liability for future claims	Liability	Credit	(900)
Capital and surplus	Capital	Credit	(100)

Note that the balances add to zero. This occurs because of a sign convention based on the category of each account. The asset and disbursement categories are “debit” accounts which normally have a positive balance. The liability, income, and capital categories are “credit” accounts that normally have a negative balance. When the balance of a “credit” account is quoted, the sign is flipped. After reflecting this sign flipping, the beginning balances above indicate 1000 of assets, 900 of liabilities, and 100 of capital and surplus.

Every time a transaction occurs, it is recorded by a set of debits and credits that add to zero. Let’s go through a list of potential transactions for the year. For each transaction we will review the debits and credits and the cumulative balances in the chart of accounts.

Transaction: Premium income received in cash

Assume premium income of $300 is received in cash. We debit cash and credit premiums for $300.

Account	Entry amount	Acct Type	Cumulative Balance
Insurance premium income	(300)	Credit	(300)
Investment income		Credit	0
Death claims		Debit	0
Expenses		Debit	0
Change in liability for future claims		Debit	0
Cash and invested assets	300	Debit	1300
Liability for future claims		Credit	(900)
Capital and surplus		Credit	(100)

Transaction: Payment of expenses in cash

Assume expenses of $75 are paid in cash. We credit cash and debit expenses premiums for $75.

Account	Entry amount	Acct Type	Cumulative Balance
Insurance premium income		Credit	(300)
Investment income		Credit	0
Death claims		Debit	0
Expenses	75	Debit	75
Change in liability for future claims		Debit	0
Cash and invested assets	(75)	Debit	1225
Liability for future claims		Credit	(900)
Capital and surplus		Credit	(100)

Transaction: Payment of claims in cash

Assume claims of $150 are received paid in cash. We credit cash and debit claims for $150.

Account	Entry amount	Acct Type	Cumulative Balance
Insurance premium income		Credit	(300)
Investment income		Credit	0
Death claims	150	Debit	150
Expenses		Debit	75
Change in liability for future claims		Debit	0
Cash and invested assets	(150)	Debit	1075
Liability for future claims		Credit	(900)
Capital and surplus		Credit	(100)

Transaction: Investment income received

Assume investment earnings of $50 are received in cash. We debit cash and credit investment income for $50.

Account	Entry amount	Acct Type	Cumulative Balance
Insurance premium income		Credit	(300)
Investment income	(50)	Credit	(50)
Death claims		Debit	150
Expenses		Debit	75
Change in liability for future claims		Debit	0
Cash and invested assets	50	Debit	1125
Liability for future claims		Credit	(900)
Capital and surplus		Credit	(100)

Transaction: End of year liability valuation

A valuation at the end of the year indicates our liability for future claims has increased to $1000 from $900 at the beginning of the year. We debit the “change in liability” and credit the liability for $100.

Account	Entry amount	Acct Type	Cumulative Balance
Insurance premium income		Credit	(300)
Investment income		Credit	(50)
Death claims		Debit	150
Expenses		Debit	75
Change in liability for future claims	100	Debit	100
Cash and invested assets		Debit	1125
Liability for future claims	(100)	Credit	(1000)
Capital and surplus		Credit	(100)

Transaction: Closing entry for the year

At the end of the year we prepare for the beginning of next year when the balance of all income and disbursement accounts must again be zero. This is called closing the ledger and is done by making a balancing entry to Capital and surplus. The amount of the balancing entry is our net income for the year. In this case the net income is positive $25. It shows up as a $25 credit to the Capital and surplus account. Since Capital and surplus is a credit account, the sign is flipped and the $25 represents and addition to capital and surplus.

Account	Entry amount	Acct Type	Cumulative Balance
Insurance premium income	300	Credit	0
Investment income	50	Credit	0
Death claims	(150)	Debit	0
Expenses	(75)	Debit	0
Change in liability for future claims	(100)	Debit	0
Cash and invested assets		Debit	1125
Liability for future claims		Credit	(1000)
Capital and surplus	(25)	Credit	(125)

The financial statements for the year are easily assembled from these account balances. We use the income and disbursement balances from before the closing entry and the balance sheet items from after the closing entry.

Income statement

Premium income	300
Investment income	50
Total income	350
Claims	150
Expenses	75
Increase in liabilities	100
Total disbursements	325
Net income	25

Balance sheet

Assets	1125
Liabilities	1000
Capital	125
Total liabilities and capital	1125

2.1.3. Non-cash accruals

Sometimes accounting rules require the recording of income or expense items at some time other than when cash is exchanged. To balance the entry for income or expense, an account other than cash must be included in the chart of accounts for this purpose. Such accounts are often called accruals, and they are either assets (used for recording income) or liabilities (used for recording disbursements).

One example of such an account is "Investment income due and accrued". This asset account is used to record investment income that has been earned but not yet collected in cash. For example, corporate bonds often pay interest in semi-annual coupons, but the income needs to be recorded monthly. To accomplish this, the following entries are made:

Account	Entry amount	Acct Type	Cumulative Balance
In month 1:			
Investment income	(100)	Credit	(100)
Income due and accrued	100	Debit	100
In month 2:			
Investment income	(100)	Credit	(200)
Income due and accrued	100	Debit	200
In month 3:			
Investment income	(100)	Credit	(300)
Income due and accrued	100	Debit	300
Similar entries for months 4, 5, and 6 not shown			
In month 6, to record receipt of cash:			
Cash	600	Debit	600
Income due and accrued	(600)	Credit	0
Investment income	None	Credit	(600)

Note that the entry made when cash is received in month 6 does not involve investment income. It simply records conversion of the accrual asset "income due and accrued" into a cash asset.

As an aside, the value of "income due and accrued" is typically included as part of the value of invested assets when computing investment yield rates from financial statements. That reflects the idea that accrual assets are real assets.

Such accruals are common in the context of asset-liability models. The modeler needs to be familiar with those to be used in any model to be developed using the workbench. While the workbench provides accounting for assets, the modeler is responsible for defining all accruals and cash transactions for liabilities and expenses.

2.2. Basics of asset-liability modeling

It should go without saying that the user of the Workbench needs to understand what an asset/liability model is, how it works, and why it is useful. Such basic background is beyond the scope of this user guide.

Beyond the basics, the user needs to understand some specifics about invested assets, including accounting for them, and about liability product design and administration. Since the Workbench includes features that facilitate principle-based liability valuation, the user needs to understand what is meant by principle-based valuation of liabilities. An overview of the required asset-liability modeling background is provided in the subsections below.

2.2.1. Invested asset background

The current version of the Workbench includes only basic types of invested assets: Cash, Bonds, Mortgages, and Stock (equities). By default, the accounting for these assets follows US statutory rules for insurance companies. That means that fixed income assets (cash, bonds and mortgages) are valued at amortized cost and stocks are valued at market. The user is assumed to understand the difference between amortized cost valuation and market valuation for fixed income investments, and the risks posed by each type of invested asset.

The Workbench provides market values for fixed income invested assets. If the user wishes to use an accounting framework where fixed income assets are held at market value, it is the user's responsibility to appropriately create financial statements from the available accounting information. The appendix covering the report writer provides details as to how this can be done.

2.2.2. Liability product background

The Workbench provides complete flexibility to model any liability based on the contract with a customer. Doing so requires the user to have a deep understanding of the contract to be modeled. In particular, the user needs to understand not only how contract values are calculated, but also the details of the recordkeeping process involved in ongoing administration of the contract.

The user must define:

- The list of data items that are maintained for each contract (often called the master record)

- The list of possible transactions that can occur in connection with a contract. For a life insurance contract this would include premium payment, death claims, contract surrender, payment of expenses for sales and administration, and others. For each kind of transaction the user must define:
    - The calculation of any monetary amounts
    - The accounting debits and credits to be made when any payment (or accrual of future payment) is to be recorded.
    - The updates to the contract master record resulting from the transaction
- The list of contract liabilities to be valued at the end of each period. For each liability associated with a contract:
    - The calculation of the liability value (tools are provided for principle-based valuation)
    - The accounting debits and credits to be made to record changes in the liability value.

The Workbench provides a structure in which all of these definitions can be specified. With those specifications, most of the computer code for a model can be automatically generated by the Workbench. The calculations of monetary amounts, however, cannot be automatically generated and must be coded by the user.

Sample contract definitions will be made available for some products. Nevertheless, the burden is on the user to completely specify all of these aspects of any product for which a model is to be developed.

Many life insurance actuaries have prepared "asset share" pricing worksheets using Microsoft Excel. The amount of effort needed to prepare an asset share worksheet calculation is comparable to the amount of effort needed to define a product within the Workbench. But the end result is much different. With the Workbench you get a fully integrated and dynamic asset-liability model with ability to handle whole blocks of business at once. With a worksheet you get something that looks at one contract at a time, usually in a static economic environment.

2.2.3. Principle-based liability valuation

While any appropriate valuation is based on some principles, the term "principle-based valuation" has taken on a specific meaning in the USA in the 21st century. Essentially, a principle-based valuation is one where the value of a liability is set equal to the value of the assets required in order to pay it off, and includes a margin that is related to the size of the risk.

The principle here is that the liability is valued with reference to the value of the assets. Since assets can be valued in different ways (e.g. amortized cost vs. market) this means that a principle-based valuation can be done in different ways.

Mechanically, a principle-based valuation is calculated as the present value of future liability cash flows, discounted at the projected future rate of return on the assets, where the asset return is measured

under a specified accounting regime. The Workbench automates this calculation, and gives the user control over the way the discount rate is calculated.

In US statutory accounting, the asset return is measured with fixed income investments valued at amortized cost. In IFRS, the asset return is measured with all assets valued at market. Different asset valuations lead to different investment returns and therefore different discount rates and different liability valuations. The principle is that the assets and liabilities are valued consistently together as a package. The value of the liability means nothing without comparison to the value assigned to the assets.

As mentioned earlier, the Workbench automates principle-based valuations. On any valuation date in a scenario, the model pauses and "clones" itself. The cloned copy is used to project cash flows and investment returns from that date forward for valuation purposes, using a scenario or set of scenarios whose future path is determined without foreknowledge of the scenario being used in the original model that was cloned. When the valuation is complete, the original model records the results, discards the "cloned" model, and resumes its simulation from where it paused.

Obviously principle-based valuation is very time-consuming so it is only done on user request.

The Workbench's open-code framework exposes the calculation of projected investment returns to the user, so you can change the way discount rates are calculated based on different asset valuations. Also, the amount of margin to be included in principle-based reserves is the subject of much debate and research. The Workbench's open-code approach allows the user flexibility to define the margin. This is discussed further in an appendix.

2.3. Software background

The Workbench is an open-code model development platform built on Microsoft .NET and the C# language. While the Workbench provides most of the modules and code for a completed model, the user must write some of the code. Doing so requires an understanding of the C# language and object-oriented design concepts as they are used in this kind of model.

2.3.1. C# language and object-oriented design concepts

The C# language was developed by Microsoft in connection with the .NET framework for software development. It is very similar to other general-purpose languages such as C++ and Visual Basic, so someone familiar with either of those languages will find it easy to learn C#.

A complete exposition of the C# language is beyond the scope of this user guide. Here we briefly summarize the object-oriented design concepts that are embedded in the language, and the advantages of C# over C++ and Visual Basic.

2.3.1.1 Obect-oriented design concepts

General-purpose programming languages all allow the definition of variables that are integers, floating point numbers, alphanumeric characters, and other simple types. They also allow the definition of functions (a formula that returns a variable) and subroutines (procedures that carry out many steps with a single call).

Object-oriented languages go beyond that to allow the definition of more abstract items called object classes. In doing so, such languages have syntax to convey the concepts of encapsulation, inheritance, and polymorphism.

Encapsulation refers to the idea of packaging (or encapsulating) a group of variables, functions, and subroutines together. The packaged items define an object class. In Visual Basic this is called a class module. In C++ and C# it is simply called a class. The developer can then declare variables that represent items of that class, where every such declared item has its own internal copy of the variables, functions, and subroutines that define the class. The declared item is referred to as an instance of the class.

An insurance contract is one example of an object class that can be encapsulated this way. The variables could be the premium paid, the amount of coverage, the dates coverage begins and ends, and so on. The functions and subroutines can provide things like premium rates for fractional periods or liability for coverage past a particular date. In a model, each insurance contract is an object, that is, an encapsulated item that has specific values for all the variables in the class. Each such item is called an instance of the class.

Inheritance refers to the idea of expanding the definition of an existing class by adding more variables, functions, and subroutines. A new class can be defined starting from, or inheriting, everything in an existing class, without the need to redefine all those contents. For example, when a new insurance contract is developed which simply adds an additional kind of coverage, a class for this new contract can be defined by inheriting from an existing contract definition that does not refer to that additional kind of coverage.

Polymorphism refers to the idea of defining a class in an abstract way, so that there can be more than one kind of thing in the class. To implement polymorphism, one must be able to define names of the functions and subroutines in a class (and their arguments) without defining how those functions and subroutines are implemented. This kind of definition is called an interface, and one can then implement

more than one class based on that interface by implementing the named functions and subroutines in more than one way.

For example, the interface to an insurance contract may define variables like the date of issue and amount of insurance which are common to all contracts, and name a subroutine called simulateOneMonth(). One can then implement that subroutine in many ways, one for each kind of insurance contract to be modeled. Each kind of contract is defined by a class that implements the interface. The advantage of polymorphism is that this allows the developer to process insurance contracts as abstract items. The abstraction happens when the code declares an instance of an insurance contract named myContract, and then calls myContract.simulateOneMonth(). Even though the call to simulateOneMonth()does not specify what kind of contract myContract is, the appropriate version of simulateOneMonth() will be called automatically based on the kind of contract. In the Workbench, this vastly simplifies the coding used to process lists of dissimilar contracts.

2.3.1.2 Advantages over C++ and Visual Basic

For actuarial work, C# has advantages over both C++ and Visual Basic.

Visual Basic does not fully implement the object-oriented design concepts listed above. In particular, Visual Basic does not implement inheritance, and the only way to imitate polymorphism involves syntax that is very verbose and duplicative. Since inheritance and polymorphism play important parts in the implementation of a model, a language that fully implements these concepts is preferred.

C++ was developed with the purpose of fully implementing the object-oriented design concepts listed above, and C++ is a very powerful language. However, some of the power of C++ comes at the cost of adding work for the developer. In particular, the developer must write code to manage memory. When debugging and testing software, problems associated with memory management are often the toughest to track down. C# eliminates this issue by relieving the developer of the need to write code that manages memory. The .NET framework includes an automated memory manager like the one associated with Visual Basic[1].

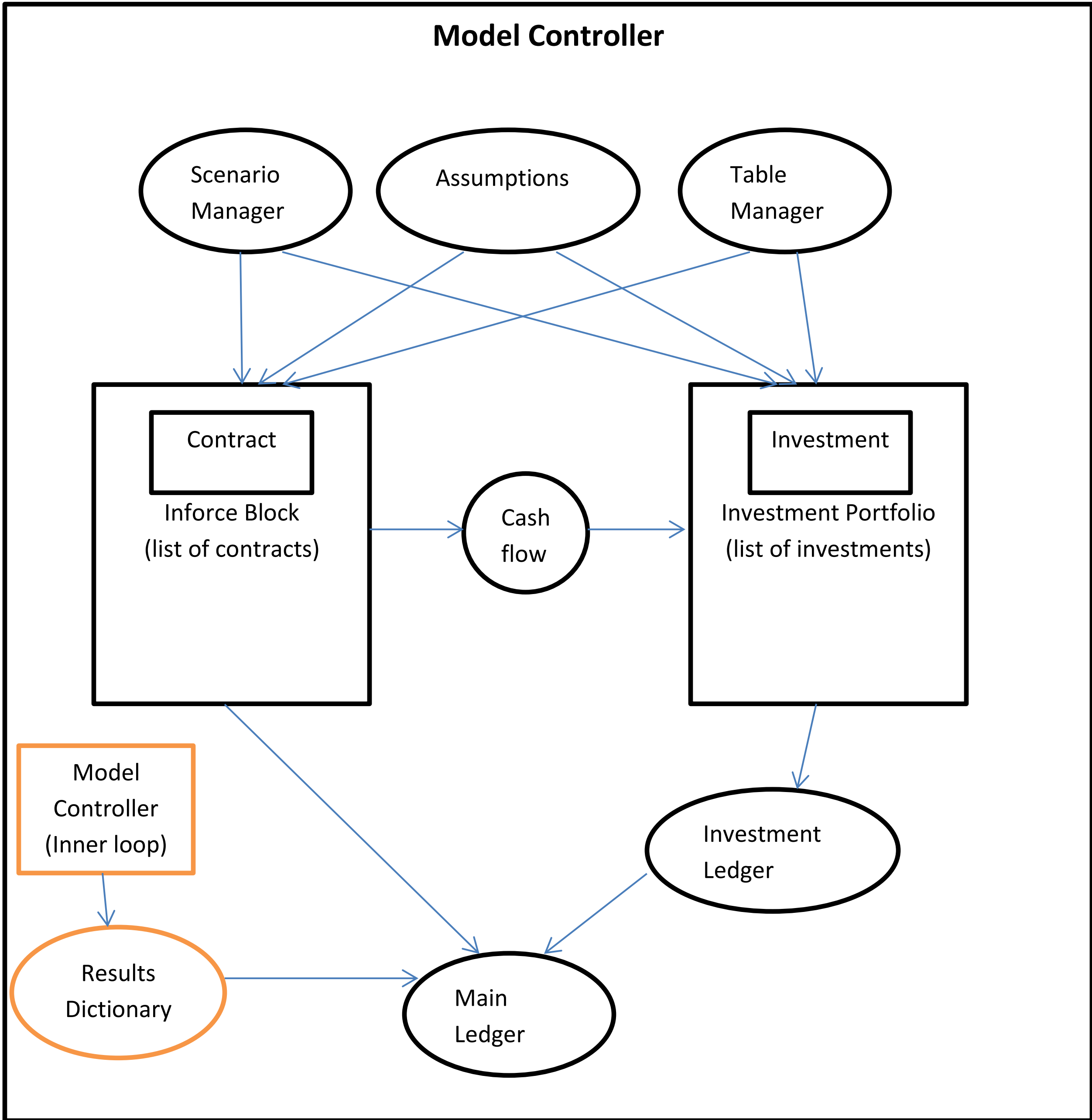
To summarize, the developer writing code in C# .NET avoids the time-consuming obligation to write memory management code as in C++, while gaining the object-oriented design capability that is missing from Visual Basic.

2.3.2. Object-oriented implementation of a model

[1] The trade-off between the use of an automated memory manager and the ability to completely control memory management is hotly debated among software developers. The choice of C# for the Workbench, which uses an automated memory manager, is based on the expectation that most users will be actuaries that prefer to focus on model processing and have little interest or technical background in optimizing memory management. The opportunity to avoid spending time on memory management has significant value for such users.

This section describes the object classes included in a model developed using the Workbench, and provides an overview of how they interact. Keep in mind that most of the code used to implement these classes is either included as pre-built program modules or is auto-generated based on the contract definitions you provide. In many models, the only code you must write is that which calculates the payments and other monetary amounts associated with the contracts you define.

The classes fall into two general categories – those that carry out the simulation and those that serve to provide input data or record results. The diagram below uses rectangles for the simulation processing and ovals for data management. As shown in the diagram, the model controller contains all the other items and orchestrates their behavior.



Classes involved in the simulation

The classes involved in the simulation work at three different levels of abstraction.

- **The lowest level** is the individual contract (e.g. an insurance contract) or individual investment (e.g. a bond). The classes that represent contracts and investments are each polymorphic, meaning there can be many kinds of contracts and many kinds of investments in a model.
- **The next level** is a list or collection of contracts or investments. A list of investments is called a portfolio, and a list of contracts is called an "inforce", a term borrowed from insurance terminology. The object that represents a portfolio or an inforce can be used to treat all of its content as a single entity, thereby abstracting a long list of objects into a single object.
- **The highest level of abstraction** is the model controller. The controller manages the entire simulation process, that is, the entire business enterprise. The controller contains one or more inforces and one or more investment portfolios, along with the objects that provide input data and record the results. At this level of abstraction, the code for virtually any asset-liability model with just one inforce and one portfolio is the same. This makes it possible to auto-generate the code for most simple models. The algorithm for running the model for one economic scenario can be summarized as follows:

1. Perform an initial valuation of all inforces and portfolios to fill in the starting balance sheet.

2. Start a loop through all future time periods in the simulation. For each time period:
    a. Simulate and record all liability transactions (process all the inforces)
    b. Capture the generated cash flow and pass it to the investment portfolios
    c. Simulate and record all investment portfolio transactions
    d. Perform an end-of-period valuation of all inforces and portfolios[2]
    e. Simulate any desired end-of-period transactions such as taxes, capital transfers, etc.
    f. Save a snapshot of the accounting records as of the end of the period.

3. At the end of the last period, save an output file containing all the end-of-period snapshots of accounting records.

When the model is run for a set of stochastic scenarios, the above process is carried out separately for each scenario. On machines with multiple processors, one scenario at a time is assigned to each processor.

Here is a summary of the classes that carry out the simulation. This summary is intended only to help the user understand the basic working of each class. Some less important parts are not described here. Detailed software documentation is provided outside this user guide.

[2] The end-of-period valuation for liabilities can be either calculated by user-specified formula or it can involve a principle-based valuation of liabilities. The latter involves a model-within-a-model to calculate the discounted present value of cash flows projected forward from the valuation date. An appendix covers principle-based valuation.

ContractClass

Description: This is the base class for all liability-side contracts to be simulated by a model. In the insurance context, this represents a type of insurance contract. Classes are derived from ContractClass to represent different types of contracts and are named by pre-pending a name for the type of contract, as in TermLifeInsuranceContractClass. The properties and methods below are common to all kinds of ContractClass and define the interface IContractClass. Each specific kind of contract has additional properties and methods that are defined by the model developer using the contract definition input dialogs.

Properties:

- **InforceBlock**

  A reference to the list of inforce contracts in which this contract is included. Since the inforce block has a reference to the assumption file, this reference can be used when obtaining the values of assumptions.

- **Controller**

  A reference to the model controller for the model in which this contract is included.

- **ScenarioManager**

  A reference to the scenario manager for the model in which this contract is included.

Methods:

- **void ProcessMonth(int aDate)**

  Carries out all simulation processing for one month. Since this class is polymorphic, the actual processing done here can be different for every different kind of ContractClass. There is a pattern that is followed, however. The pattern has three main steps:

  1. Calculate all the monetary amounts that are involved in all transactions for the month.
  2. For each transaction, record the transactions in the ledger using debits and credits to the appropriate accounts. The amount of each debit or credit is based on the calculations done in step 1.
  3. For each transaction, update the master record (the properties of the contract) to reflect the transaction.

Most of the code for this is auto-generated based on user-supplied definitions in the .ctype file. However, the formulas used in step 1 above are coded by the user.

- **void PerformValuation(int aDate)**

  Records any required formulaic valuation amounts in the appropriate balance sheet accounts. This method is called at the end of the month, after all transactions for the month have been simulated. The calculation of the valuation amounts to be recorded here actually takes place in step 1 of ProcessMonth(), described above.

  This valuation processing is separate from the principle-based valuation, which is carried out in the ControllerClass.

- **void SetLgrSeg()**

  Sets the ledger segment number in which all transactions for this contract will be recorded. By default segment 1 is used, but the model developer can change this so that different kinds of contracts are recorded in different ledger segments.

- **double SalesMeasure(int aDate)**

  When a model simulates future new contracts (i.e. sales), one of the input assumptions will be the amount of sales in each future month. The amount of sales might be in a unit of measure other than the number of contracts (e.g. total insurance premiums), so this method provides a way to translate the amount of sales into a number of contracts. For example, if the sales are expressed as total insurance premium sold, this method would return the per-contract amount of premium. The number of contracts sold is always the assumed sales amount divided by this value.

- **IContractClass Clone()**

  This method makes a copy of this object in the computer's memory. It is used when cloning a model-within-a-model for purposes of principle-based valuations on future dates.

InvestmentClass

Description: This is the base class for all investments (other than cash) to be simulated by a model. Classes are derived from InvestmentClass to represent different types of investments such as bonds and stocks. The properties and methods below are common to all kinds of InvestmentClass and define the interface IInvestment. Each specific kind of investment has additional properties and methods.

In most cases, the user will not need to do any coding for investments because code for basic kinds of investments is already provided.

Properties:

- **Portfolio** A reference to the portfolio in which this investment is held
- **LedgerSegment** The number of the ledger segment where transactions are recorded
- **TypeCode** A value used to indicate what kind of investment this is
- **ID** A unique identifier string for this investment
- **PurchasePrice**
- **PurchaseDate**
- **MarketValue(aDate)**
- **StatementValue(aDate)**
- **TaxValue(aDate)**
- **BookValue(aDate)**
- **BookYield**
- **IsFixedIncome** True or false. Indicates whether this investment is a fixed income type.

Methods:

- **ProcessPurchase(aDate, aTiming)**

  Simulates the transaction of purchasing the investment by paying cash from the portfolio and adding the investment to the portfolio.

- **ProcessMonth(aDate)**

  Simulates all transactions associated with this investment for one month. The transactions are simulated using the same basic 3-step process as is used to simulate contract transactions.

- **PerformValuation(aDate)**

  Records the market value of the investment in the ledger.

- **Clone()**

  This method makes a copy of this object in the computer's memory. It is used when cloning a model-within-a-model for purposes of principle-based valuations on future dates.

InforceBlockClass

Description: This is the base class for all blocks of liability contracts to be simulated by a model. An object of this class is essentially a list of similar contracts such as life insurance or annuities. Classes are derived from this class to represent blocks of different types of contracts and are named by pre-pending a name for the type of contract, as in TermLifeInsuranceInforceBlockClass. The properties and methods below are common to all kinds of InforceClass and define the interface IInforceBlockClass. Each specific kind of inforce may have additional properties and methods that are defined by the model developer.

In many (but not all) cases, the user will not need to do any coding for a class derived from InforceBlockClass because generic code for handling a list of contracts as a single object can be auto-generated by the workbench. Additional coding is needed only when the entire group of contracts shares some property or characteristic that is best handled once in the aggregate rather than separately for each contract.

Properties:

- **Contracts** An object containing the list of contracts included in this InforceBlock

- **References to other objects in the model but outside this class:**
  - **Assumptions** A reference to the contents of an assumption file
  - **Controller** A reference to the controller of this model
  - **Lgr** A reference to the Ledger where transactions are recorded
  - **ScenarioManager** A reference to the scenario manager of this model

- **References to objects constructed and used internally by this class:**
  - **AuditFileWriter** A utility that writes information to an audit file in .csv form
  - **CashFlow** An object that accumulates cash flow from all contracts at the beginning, middle, and end of the month
  - **ContractFactories** A utility that creates new contracts in the computer's memory
  - **SalesMix** A list that defines the kind of new contracts to be created to simulate new sales

- **Range of ledger segments used by this InforceBlock**
  - **FirstLgrSeg**
  - **NumLgrSegs**

- **State variables stored once for the block rather than for each contract**
  - **CumulativeInflationFactor** A cumulative inflation factor.

Methods:

The three methods AddNewSales(aDate), ProcessMonth(aDate), and PerformValuation(aDate) handle the bulk of the model simulation, with each carrying out its named function for the entire list of contracts in the block. The code for these methods can be auto-generated in many cases, based on the definition of the contract. Other methods included in this class are utilities that often do not depend on the type of contracts in the block.

- **Methods central to the simulation:**
  - **AddNewSales(aDate)** Adds new contracts to the list of those in this block
  - **PerformValuation(aDate)** Carries out a period-end valuation[3]
  - **ProcessMonth(aDate)** Simulates transactions for all contracts for one month
  - **Clone()** Creates a cloned copy of this inforce block. It is used when cloning a model-within-a-model for purposes of principle-based valuations on future dates.

- **Methods to read and write .csv files containing the list of contracts:**
  - **ReadFromCsv(filename)**
  - **WriteToCsv(filename)**

- **Methods to handle an audit file used to save interim results on request**
  - **OpenAuditFile(filename)**
  - **CloseAuditFile()**

- **Methods used to record information in the ledger**
  - **Debit(accountNum, segmentNum, amount)**
  - **Credit(accountNum, segmentNum, amount)**
  - **SetBalance(accountNum, segmentNum, amount)**

- **Method used to retrieve assumptions**
  - **GetAsmpValue(aDate, assumptionNumber)**

[3] The period-end valuation carried out here is a formulaic valuation, not the principle-based valuation that requires cloning the entire model. The formulaic valuation is carried out in three steps. Two ledger accounts are involved: a balance sheet account for the liability and an income statement account for the change in liability. The three steps are: 1) Capture the old valuation amount (liability account balance before valuation update) and zero out the liability account. 2) Loop through all contracts in the inforce block, retrieve the value of each one and accumulate the total in the liability account. 3) Determine the change in account balance (new total minus the old amount captured in step 1) and make a ledger entry for the change in liability. This ledger entry balances the net effect of steps 1 and 2. All of the code for these steps is auto-generated by the Workbench except the formula for the valuation of a single contract.

InvPortfolioClass

Description: This is the base class for all investment portfolios to be simulated by a model. Typically there is just one portfolio in a model, but classes can be derived from InvPortfolioClass to represent different types of investment portfolios. The properties and methods below are common to all kinds of Portfolios and define the interface IInvPortfolioClass.

In most cases, the user will not need to do any coding for investments because code for basic kinds of investments is already provided and is sufficient.

Properties:

- **Investments**

  The list of investments included in this portfolio.

- **Ledger**

  A reference to the ledger in which activity for this portfolio is recorded.

- **ScenarioManager**

  A reference to the scenario manager for the model in which this portfolio is included.

- **Strategy**

  A reference to an object containing parameters of an investment strategy.

Methods:

- **ProcessMonth(CashFlowClass cf, int aDate)**

  Simulates all transactions for one month for this portfolio. Processing involves adding the amount of cash flow given as the first argument, simulating all transactions on investments already in the portfolio, and then buying and selling investments based on a strategy.

- **PerformValuation(int aDate)**

  Records the market value of the portfolio in the ledger. Note that book value is not updated here because it is normally updated as part of normal transaction processing in ProcessMonth().

- **Clone()**

  This method makes a copy of this object in the computer's memory. It is used when cloning a model-within-a-model for purposes of principle-based valuations on future dates.

ControllerClass

Description: This is the object that encapsulates the entire model, and which works at the highest level of abstraction. The controller contains all of the objects that represent parts of a model and orchestrates their activity to carry out the asset / liability projection.

The power of this level of abstraction is illustrated by the algorithm below for running the model for one economic scenario. This algorithm is completely generic and can be used without modification in a great many models:

1. Perform an initial valuation of all inforces and portfolios to fill in the starting balance sheet.

2. Start a loop through all future time periods in the simulation. For each time period:
    a. Simulate and record all liability transactions (process all the inforces)
    b. Capture the generated cash flow and pass it to the investment portfolios
    c. Simulate and record all investment portfolio transactions
    d. Perform an end-of-period valuation of all inforces and portfolios[4]
    e. Simulate any desired end-of-period corporate-level transactions such as taxes, capital transfers, etc.
    f. Save a snapshot of the accounting records as of the end of the period.

3. At the end of the last period, save an output file containing all the end-of-period snapshots of accounting records.

Different models differ mainly in which inforces and portfolios are included, and which end-of period transactions (step 2e above) are simulated. The generic ControllerClass that is provided can easily be modified to implement such differences.

Properties:

- **AsmpFile** Provides access to assumptions
- **Inforce** Provides interface to an InforceBlock
- **InvPortfolio** Provides the interface to an InvPortfolio
- **IsOuterLoopModel** True or False, set to False for controllers cloned for inner loops[5]

[4] The end-of-period valuation for liabilities can be either calculated by user-specified formula or it can involve a principle-based valuation of liabilities. The latter involves a model-within-a-model to calculate the discounted present value of cash flows projected forward from the valuation date. An appendix covers principle-based valuation.

[5] The models cloned for inner loops (model-within-a-model) use exactly the same program code as the models from which they were cloned. However, they are prevented from cloning their own inner loops by setting this property to false. This property can also be used in calculations, for example, to modify the assumptions used in inner loop models to add explicit margins for valuation purposes.

- **ModelName** A character string containing the name of the model
- **ModelResults** A ResultsDictionaryClass object used to capture specific named results
- **RunParms** A set of runtime parameters such as file names, start date, etc.
- **ScenarioManager** Manages access to scenarios of economic and other data
- **UseInnerLoop** True or false, indicates whether this controller carries out principle-based valuations in an inner loop.

Note that while every model controller has a ledger inside it, the ledger is not accessible from outside so it is not listed here as a property of the controller.

Also note that while every model also includes a RunLog and a TableManager, these items are independent of the model controller so they are not properties of the controller. In object-oriented terminology, they are "static".

Methods:

The main purpose of a model controller is to run a single scenario over time. When that scenario involves principles-based valuations on future dates, the model controller must clone itself for each scenario involved in that valuation, thereby managing potentially a large number of inner-loop models.

- **RunScenario()**

  This is the routine that runs a scenario. Before this is called, SetupObjects() and SetupScenario() must have been called. First steps in running a scenario include reading in the initial list of contracts in force and the starting investment portfolio, then performing an initial valuation of each to fill in the starting balance sheet. After that, the simulation loop through future calendar periods begins.

- **RunInnerLoop()**

  This is called from RunScenario() when needed to run the inner loop scenarios required for a principle-based valuation on a future date.

- **SetupObjects()**

  This initializes all of the objects in the controller that are not scenario-specific. These objects only need to be set up once at the start of a run, not once for each scenario. The objects set up here include the RunLog, Table Manager, Assumption file, and the chart of accounts and list of ledger segments.

- **SetupScenario(scenarioID)**

This initializes the state of the controller to the start of a scenario run. This requires initialization of objects that are scenario-specific or change state over the course of a scenario. The objects set up here include setting the state (current scenario) of the Scenario Manager, creation of a new blank ledger, and reading the starting investment portfolio and starting inforce from input files.

There are many additional methods inside a Controller in order to provide calculation and processing support. Complete software documentation is provided elsewhere.

Classes used primarily for data management

The following objects are included in a model, many as public or private properties of the model controller. Each is implemented as a separate class in its own right, encapsulating all of the data and processing needed to carry out its part in the model simulation. Full documentation of each of these classes is voluminous so all that is included here is a general description of each one. Separate software documentation provides more details.

AsmpFileClass

Description: This class handles access to assumptions. It provides methods to read and write a file of assumptions and retrieve assumption values by date and assumption code. Actuarial tables that are used as assumptions are managed separately by the TableManager, however the assumption file may define which table is to be retrieved by the TableManager for a specific purpose.

CashFlowClass

Description: This class is used to accumulate total cash flow for a month in three parts; beginning of month, middle of month, and end of month. Cash flow is generated during monthly processing of each contract in an InforceBlock. The Controller passes the accumulated cash flow to an Investment Portfolio (as an argument to the ProcessMonth() method) for simulation of investment processing.

InvStrategyClass

Description: This class provides data to define a simple investment strategy for a portfolio consisting of cash, bonds, and stocks. Cash management is defined by a maximum percentage of the total portfolio to hold in cash, the maximum percentage to borrow before forcing sales to generate cash, and the borrowing spread. Stock management is defined simply by a minimum and maximum percentage of the portfolio to hold in stocks. Fixed income portfolio management can be either to purchase specified items or to maintain a target duration. When purchasing specified items, a list of items specifies the maturity, credit quality, and portion of available cash to purchase each item in the list.

LedgerClass

Description: The ledger contains the accounting records that form the primary output of any model. One can visualize the data in the ledger as a three-dimensional array of account balances.

One dimension corresponds to the chart of accounts, that is, the list of accounts for which balances are stored in the ledger. The chart of accounts is described in detail in an Appendix.

The second dimension is ledger “segment”. A ledger “segment” is a part of the business for which results are to be recorded separately. One must sum across all ledger segments to get the total enterprise balance in any account in the chart of accounts. For many models there is just one ledger segment.

The third dimension is calendar date. A ledger implemented as an object of the LedgerClass contains snapshots of the state of all ledger account balances as of each date during the time period of a model

run. Normally that includes the model start date and each future month that is modeled. Each snapshot can be visualized as a page in a book.

ResultsDictionaryClass

Description: This is a storage facility for results from a set of scenarios, such as those that are used for stochastic principle-based valuation on a future date within a model. This is a two-level dictionary. The first level lookup is by scenario name; there is a separate dictionary of results for each scenario. The second-level lookup, within each scenario-specific dictionary, allows items can be stored and retrieved by name. A stored item can be of any type, including for example the array of monthly cash flows to be used when calculating a scenario present value in a principle-based valuation. The monthly cash flows and discount rates needed for principle-based valuation are automatically stored for Workbench models that perform such valuations on future dates. The user can define additional items to be stored in this dictionary and use them in any desired way.

RunLog

Description: This is simply a list of one-line text messages to indicate progress during model processing, such as the starting or completion of one processing step. If errors occur, the intent is that they be caught and written to the log. Each model contains just one RunLog, and it is generally cleared at the start of each model run.

RunParmsClass

Description: This is a list of "runtime parameters" that define the scenario or set of scenarios to be run. Example runtime parameters include the name of the input data files containing all contracts in force, the name for the output ledger file, the model start date, and so on. The user can define the names of runtime parameters and how they are used. A standard set of parameters is provided to start from.

ScenarioManagerClass

Description: The Scenario Manager is used to manage the economic scenarios used by the model. All access to scenario data is provided by the Scenario Manager. When principle-based valuation is carried out on a future date using stochastic or deterministic scenarios, the Scenario Manager generates the scenarios to be used for valuation starting from conditions on the valuation date, and keeps valuation scenarios separate from the scenario in use for the outer loop.

TableManager

Description: The Table Manager provides speedy access to values from actuarial tables. Every model uses a folder containing actuarial tables in permanent (disk) storage. The Table Manager reads tables into memory on first request and provides fast access to table values thereafter. The Table Manager in the WorkBench provides the same capabilities as the TableManager add-in to Excel which has been used by many actuaries. Each model contains just one TableManager.

3. How to use the Workbench

3.1. The model development screens – the user interface

When the Workbench starts up, it displays a main window with multiple parts. This section explains each part and its purpose.

Window title bar

The title bar displays the path of the project file for the model currently being developed.

Main menu and toolbar

The menu lists many of the commands the user can give while developing a model.

- **File menu**
  - **New project**… Creates a new project. The user is asked to specify the project folder. The folder is created and several files needed to run a model are copied into the folder.
  - **Open project**… Allows the user to specify a project file to open.
  - **Close project**… Closes the currently active project.

- **Add file to project**… Allows the user to specify a file to be added to the project. The chosen file will be copied into the project folder and added to the list of project contents in the top left frame. This can be useful for copying files from other projects into the current project.
- **Remove file from project**… Allows the user to specify a file to be removed from the project.
- **Create Visual Studio project**… Allows the user to create a project file for use by Visual Studio. The file will have the same name as the Workbench project file, but the extension .csproj. The file will be created in the project folder. Be aware that after the file is created, there is no connection between Visual Studio and the Workbench. Any changes made in Visual Studio (other than source code changes to existing files) will not be reflected in the Workbench project.
- **Exit**… Closes the Workbench.

- **Model Assembly**
  - Create data files
    - Runtime parameters
    - Assumptions
  - **Generate code**…Brings up a dialog allowing the user to specify which parts of the model code to generate. A full discussion of the code generator appears in a dedicated appendix to this User Guide.
  - **Build**… Compiles all the source code to create an executable (*.exe) file. When done, a popup window indicates either success or errors were found. If errors were found, they are listed in the messages area at the bottom of the Workbench screen. Clicking on an error in that list can bring up the source code with the cursor positioned at the spot of the error.
  - **Run**… Runs the model that has been built (the *.exe file in the project folder).
- **Utilities**
  - **Update libraries**… Copies several .dll files from the Workbench to the project folder. This can be useful when a new version of the Workbench is created with upgrades (or bug fixes) to some of the underlying model functionality.
  - **View calculation dependencies**… This is only active when the current file editing tab contains source code for a *ContractClass.cs file. The user must edit the formulas used to calculate some variables in such a file. Each calculated variable depends on other variables, so there are always dependencies among variables. Circular dependencies can be created by mistake. This utility helps spot them.
  - Other items may be added to the Utilities menu in later versions of the Workbench.
- **Help**
  - **About:** Provides copyright, version number, and license information about the Workbench software.
  - **Contents:** Provides detailed software documentation.

The toolbar provides quick access to several of the menu commands and adds a few, including the editing commands Undo, Redo, Cut, Copy, and Paste.

Project Contents

This area on the top left of the Workbench lists all of the files used in the current project. The files are grouped in categories:

- Contract types — Each file defines a type of contract
- Assumption groups — Each file defines a group of assumptions
- Runtime parameters — Defines the list of filenames and other parameters that set up a run
- Ledger definition — Defines the list of accounts and segments in the ledgers
- Reports — Each file defines a report to be generated from a ledger
- Program code — Each file contains C# program code. Many of the file names are standardized and contain the project name or the name of a contract type. But you can add any others you wish.

<u>To create a new file in any category, right-click on the category name.</u> You will not be able to create new files under “Runtime parameters” or “Ledger definition” because the Workbench will not use additional files in those categories.

<u>To take an action with any file, click on the file name.</u> This will bring up a popup menu allowing you to Open, Save, Close, Rename, or Delete the file. Of course the file must already be open before you can save or close it.

File editing tabs

This large area in the center of the Workbench is where you modify the contents of the project files. You can have several files open at once; there is a tab at the top of the area for each file that is open. To switch to a different file, click on its tab. To close a file, click on the small “x” that appears on its tab. To save a file, use the File | Save menu option or click on the save icon in the toolbar.

Each type of file has its own editor that is tailored to the type of data in the file. Computer source code files (*.cs) use a text editor with special features common to many editors of that sort, including the availability of Ctrl-F as a shortcut key for “Find” to locate specific text in the file.

Contract details

This area on the top right of the Workbench lists the names of the items defined by the user in each contract type that has been defined. This is a useful reference when writing code that uses these names.

There are three categories of names that the user defines for each contract:

- Fields — The information items on the master record for the contract
- Functions — Items that must be calculated each month to process transactions
- Transactions — Transactions that can occur such as Premiums or Claims

When you have the source code for a *ContractClass.cs file open in the editing tab, you can click on any of the names in the contract details list and the cursor will be moved to the place in the source code where that item is declared. For Functions, this will be at the code for the calcXXXX() function, where XXXX is the name of the calculated item. For Transactions, this will be at the processXXXX() method where XXXX is the name of the transaction.

Assumptions list

This area on the bottom left of the Workbench lists the names of all the assumptions that have been defined in the Assumption Groups files in this project.

When you have the source code for a *ContractClass.cs file open in the editing tab and you want to enter the code to retrieve an assumption value, double click on the name of the assumption to be retrieved and the appropriate code will be inserted at the cursor position. For example, if you double click on the assumption name "ExpenseRate", the following code will be inserted:

```
block.GetAsmpValue(aDate, A.ExpenseRate)
```

Message area

This area on the bottom center of the Workbench has two tabs, "Notes" and "Compiler messages".

The "Notes" tab is a free-form text area for the user to write notes or reminders to themselves while working on the workbench.

The "Compiler messages" tab is filled by the compiler when the user issues the Build command. If the compiler finds errors in the source code, they will be listed here. Double-clicking on a listed error will take you to the spot in the source code where that error occurred.

Ledger accounts list

This area on the bottom right of the Workbench lists the names of all the ledger accounts and ledger segments used in the model. This is a useful reference both when editing a contract definition (where the accounts used to record each transaction are specified) and when editing source code that needs to retrieve or set an account balance. When editing source code on the editor tab, one can double click on a name in the ledger accounts list and the appropriately prefixed name corresponding to the account number will be entered into the source code.

There is a dedicated appendix in this user guide to explain the chart of accounts and the means available to define and change it.

3.2. Understanding file types

Several different filename extensions are used by the Workbench to identify different types of files. The table below summarizes the filename extensions in use. All files are stored in XML format unless otherwise indicated.

Filename extension	Usage			Description
	Workbench Model Design	Model Input	Model Output	
.af		X		**Assumption file.** Contains all assumption values for a model run.
.asgrp	X			**Assumption group.** This defines a list of assumptions (not the values of the assumptions) that are grouped under one heading. Each model normally contains several assumption groups.
.cs	X			**C# source code.** Each model contains several source code files.
.csv		X	X	**Comma-separated values file.** Used for compatability with Microsoft Excel and other third party software. Can contain any data laid out in rows and columns. The file name should indicate the content.
.ctype	X			**Contract type.** This file contains all of the definitions provided by the user regarding a single type of contract. Such definitions include the data for the master record, the list of amounts to be calculated for each time period, the transactions to simulate, and the ledger entries and master record updates associated with each transaction. This data is used by the Workbench to auto-generate most of the code for simulating a contract in a model. The main task left for the user is to define the formulas for the list of amounts to be calculated for each time period.
.inf		X		**Inforce.** Contains a list of contract master records.
.inv		X		**Investment portfolio**. Contains a list of investments and other information about a portfolio.
.invst		X		**Investment strategy.** Contains information that defines an investment strategy.
.lgract	X	X		**Ledger chart of accounts.** Defines a chart of accounts for a ledger. Each model has two of these, MainLgrAccts.lgract for the main ledger and InvLgrAccts.lgract for the investment portfolio ledger.
.lgrseg	X	X		**Ledger segments.** Defines a list of subdivisions or segments in a ledger. A full chart of accounts is kept for each "segment". Each model has two of these, MainLgrSegments.lgrseg for the main ledger and

				InvLgrSegments.lgrseg for the investment portfolio ledger.
.lgmap	X	X		**Ledger map.** Since detailed information in the investment portfolio ledger is summarized in the main ledger, this file provides the required mapping between the investment portfolio ledger and the main ledger.
.lgr			X	**Ledger.** At least one of these is produced for each scenario in a model run. A ledger contains all accounting records of the model simulation. Internally, the model uses two ledgers, one for the investment portfolio and one for the enterprise as a whole (the "main" ledger). Since summarized investment results are included in the main ledger, the detailed results in the investment ledger are often not saved.
.rp		X		**Runtime parameters.** Contains the list of runtime parameters used to set up a run of the model. Example items include the starting date and names of input and output files.
.rpdef	X			**Runtime parameters definition.** This file defines the items that must be contained in the list of runtime parameters used to set up a run of the model. There should be only one such file for each model.
.rptdef	X	X		**Report definition.** This file contains the list of line items to appear in a pre-defined financial statement or other report from the ledger.
.rsdef		X		**Risk definition.** This file is used to define the list of risk drivers in connection with the Representative Scenarios Method, wherein a small number of "representative" scenarios for each risk driver are used instead of fully stochastic scenarios.
.xml		X		**XML data.** This can be any XML data file, but is used specifically by the model for economic scenarios. Scenario files are coded using EconSML, a dialect of XML defined for this purpose. The file name should indicate the content of the file.

Some suggestions for file naming

As you use your model, you will generate lots of files. A good naming convention can help you keep them straight. Some suggestions include:

- Use folders to hold different types of files. Each model is automatically created with three sub-folders named Input, Output, and Tables. It is suggested that you use those folders as their

names imply, and perhaps create subfolders under Input and Output to organize your own work.

- Create a newly named input file every time you change the contents of an input file. You can do this by using the File | Save as… option after making your changes. This applies to:
    - Assumptions (*.af)
    - Investment strategies (*.invst)
    - Investment portfolios (*.inv)
    - Inforce contracts (*.inf)
    - Runtime parameters (*.rp)
    - Scenarios (*.xml)

- Use the name of the runtime parameters file (excluding the extension) as the name for the main ledger output file. That way, if you need to check which files were used when creating a ledger, you can check the corresponding runtime parameters file. That will also force you to save a separate runtime parameters file for each set of model results that you want to save.

- Include the as-of-date in the name of files that are date-specific. For example, the investment portfolio as of December 31 2013 might be named "InvPortfolio201312.inv". This applies to:
    - Investment portfolios
    - Inforce contracts
    - Scenarios

  An alternative to this is to create a new folder for each model start date. The name of the folder can be the start date in YYYYMM form. You can then create separate \Input and \Output subfolders under that date-named folder.

3.3. Developing your first model

This section provides a step-by-step guide to creating your first asset/liability model using the Workbench. The contract to be modeled is a 20-year term life insurance contract which provides a fixed death benefit if the insured person dies during the 20 years after the contract starts. Insurance premiums are payable in equal amounts at the beginning of each year.

1. Start up the Workbench and select "New project…" from the File menu.

    a. In the dialog box, specify the folder for the project. Suggestion: create a new folder for this purpose.

    b. Another dialog appears – specify the name for the project file. Do not include spaces in the name. Suggestion: TermModel

2. Explore the workbench screen. The top title bar of the Workbench now shows the name of the project file being worked on. The top left panel of the workbench now lists files contained in the project. The bottom right panel shows the lists of ledger accounts that are provided by default.

    a. In the top left panel, click on the plus signs in the tree to expand it. Right click on one of the file names that are shown, and in the pop-up menu click “open”. The file is opened for editing in the central window, with a tab showing the file name. Close the file by clicking on the “x” in the tab.

3. Explore your hard drive. Using the Windows Explorer, navigate to the project folder. Notice that three sub-folders have been created under the project folder. These are \Input, \Output, and \Tables. These folders were created when the project was created, and they are intended for your use in organizing the data files to be created later. While you are free to create a more complex structure of folders, bear in mind that the project folder itself and the \Tables subfolder are required for the model to run. The project folder will contain the model program and all source code, and the \Tables subfolder will be used by the Table Manager to retrieve actuarial tables as needed.

4. Define an insurance contract for use in the model. In the top left panel, right-click on “Contract types”, then click “New...”. In the popup window, type the name of the product you are creating, e.g. “Term20”. Then click OK.

    a. In the central panel of the workbench, a file tab appears with the name “Term20.ctype”. This panel has its own set of tabs: Fields, Functions, Transactions, and Valuation.

In the Fields tab, enter the following list of master record fields for this contract type:

Name	Type	Description
ID	string	Contract number or identifier
NumContracts	double	Number of contracts in this cell
IssueAge	int	Age at issue
IssueDate	int	YYYYMM month of issue
FaceAmount	double	Face amount of insurance

In the Functions tab, enter the following list of items that need to be calculated each month in order to simulate transactions:

Name	Type	Description
PolicyYear	Int	Current policy year (first year = 1)
PolicyMonth	int	Current month within the policy year (1 = first month)

TotPremium	double	Total premium paid
TotClaims	double	Total claims paid
TotExpenses	double	Total expenses incurred
TotReserve	double	Total reserve liability
NumClaims	double	Number of death claims
NumExpiries	double	Number of contracts expiring

In the Transactions tab there are three tables. The two on the right are linked to the one on the left. In the left table enter the list of monthly transactions:

TransactionName	Description
Premiums	Premium payment
Claims	Payment of death benefits
Expenses	Payment of expenses

The top right “ledger entries” table is keyed to the transaction name selected in the top left table.

Select the “Premiums” transaction and enter the following in the “ledger entries” table:

Action	Account	Amount	Timing
Credit	RenewalPrem	TotPremium	Beg
Debit	Cash	TotPremium	Beg

Select the “Expenses” transaction and enter the following into the “ledger entries” table:

Action	Account	Amount	Timing
Credit	Cash	TotExpenses	Beg
Debit	MaintExp	TotExpenses	Beg

Select the “Claims” transaction and enter the following into the “ledger entries” table:

Action	Account	Amount	Timing
Credit	Cash	TotClaims	End
Debit	DeathBen	TotClaims	End

Note that in each of the above cases, the Account is either “Cash” or an item that appears in the list of main ledger accounts on the bottom right of the screen, and the Amount appears in the list of Functions on the top right.

While the “Claims” transaction is still highlighted, enter the following into the “data updates” table:

Field name	New value
NumContracts	NumContracts-NumClaims-NumExpiries

Go to the “Valuation” tab and enter the following in the table:

ItemDescription	CalculatedVariable	BalanceSheetAccount	IncomeStmtAccount
Statutory reserves	TotReserves	Reserves	IncRes

As with the transactions entered earlier, the BalanceSheetAccount and IncomeStmtAccount must be names that appears in the list of main ledger accounts at the bottom right of the screen. The CalculatedVariable must appear in the list of Functions on the top right.

Go to the “Sales mix” tab and click on the button “Copy contract fields”. The table is filled in with the list of fields defined earlier for the master record, plus an additional field named “PctSales” for percent of sales. A record with these fields is used to define a type of contract to be sold in the future, along with the fraction of total sales that will be of that type of contract. A set of these records is called a “sales mix”, and we will create a file defining a sales mix later.

After entering all these items, save the file. Either use the File|Save menu item or click on the blue disk icon on the toolbar. Then close the Term20.ctype file by clicking on the X in the tab at the top of the center panel.

The top right panel now shows the names of the fields, functions, etc. that you defined for your product.

5. <u>Define some assumptions for use by your model.</u> Assumptions are created in groups, so that related assumptions can be viewed and edited together. We will create two assumption groups, one for table numbers (mortality and lapse), and one for expense assumptions.

To create an assumption group, go to the top left panel, right-click on “Assumption Groups” and then click on “new…”. In the pop-up window, enter then name “TableNumbers” and click OK. The central panel now has a tab titled “TableNumbers.asgrp”.

In the tab, give the assumption group a name and a number. Leave the checkbox unchecked (about varying by date). Then enter the following description of the table numbers needed:

Number	Text	Mnemonic	Decimals	Description
1	Mortality rates table number	MortalityTableNum	0	Qx table
2	Lapse rates table number	LapseTableNum	0	Wx table
3	Premium rates table number	PremRateTableNum	0	Prem rate table

After these are entered, save the file by clicking on the blue disk icon on the toolbar, then close the file. Note that the Mnemonics of the assumptions you entered now appear in the lower left panel of the workbench.

Create a second new assumption group named “Expenses”, and fill in the name and a different number. Enter the following description of the expense assumptions needed, then save the file and close it.

Number	Text	Mnemonic	Decimals	Description
1	First year sales expense % premium	FirstYrPctPremExp	4	Distribution cost
2	Annual maint. expense per contract	AnnualMaintExp	2	Maintenance expense
3	Expense per claim	PerClaimExp	2	Claim processing exp

Create a third assumption group named “Sales”, and fill in the name and a different number. In this case, check the “Assumptions in the group vary by date” box. Enter the following description of the sales assumptions, then save the file and close it.

Number	Text	Mnemonic	Decimals	Description
1	Premium on new sales	Term20Sales	0	Total sales in each month

After saving all three assumption group definition files, go back and edit them to be sure the three groups have different group numbers and that all the assumptions in each group are numbered sequentially 1,2,3,… This numbering scheme significantly speeds up lookup of assumption values in the model.

6. Create a blank assumption file. Use the Workbench's top menu Model Assembly | Create data files | Assumptions menu item to create an assumption file.
    a. After doing so, note the presence of the file "Assumptions.af" in the "Input" folder. We will edit this file later – it has blanks for values at this point.
    b. In the top left panel of the Workbench, note that under "Program code" there is a new file named "AssumpCodes.cs". Open that file to observe that it defines a named constant for each assumption, based on the assumption group numbers and assumption numbers you entered.

7. Create a blank runtime parameters file. Use the Workbench's top menu Model Assembly | Create data files | Runtime parameters menu item to create the file.
    a. After doing so, not the presence of the file "RunParms.rp" in the "Input" folder. We will edit this file later – it has blanks for values at this point.

8. Generate program code files. Use the Workbench's top menu Model Assembly | Generate Code menu item to bring up a dialog box allowing specification of which files to generate. Click on the "select all" button, then the "Generate selected modules" button.
    a. After doing so, note the presence of several new files under "Program code" in the top left panel of the Workbench. These files are in the model project directory.

9. Build an executable model. Use the Workbench's top menu Model Assembly | Build menu item to compile the model. (The toolbar icon with a green downward arrow does the same thing.) If successful, a message box will appear announcing success. You now have a stand-alone executable model program. Of course it does no calculations. But it provides an interface for creating and editing input files.

10. Startup your new executable model. Use the Workbench's top menu Model Assembly | Run menu item. Alternatively, click on the toolbar icon with the blue forward arrow. Or, find the .exe file in your project directory using the Windows file explorer, and double click on the .exe.
    a. After doing so, your new model program should take over the full screen. Try out some of the menu items, including File | Edit and File | New. Don't try running the model calculations or viewing reports yet, because none of the product calculations are coded and we haven't created files with the required input data.

11. Get some tables for use by the model and put them in the \Tables subfolder. Keep your model open while doing this step. You may wish to minimize the model window to do step a. below, but that isn't necessary.
    a. First, obtain a table from the Society of Actuaries online database. Use your web browser to go to the site mort.soa.org. Search for table number 1076 (2001 CSO super select male). At the very right hand side of the search results there is a column labeled "Actions", inside which there is a link consisting of "XML" underlined. Click on that link to obtain table 1076 in xml file format, and save the file in your \Tables subfolder. Then

use the model's main menu Tables | Update index of tables menu option to update the index so it includes the table you added. After that you can view the table using the Tables | View a table… menu option.

b. Create a table of lapse rates by duration. Go back to the model program that was left open, and use the Tables | Create a table… menu option. This brings up a dialog allowing you to define the shape of the new table. In the top set of radio buttions, click "Duration" to indicate this will be a table by duration. Then in the bottom area, fill in 1 and 20 for the minimum and maximum tabulated durations. Then click the button labeled "Edit table values". This brings up a dialog that looks just like the table viewing dialog, but allows you to enter values. On the "Description" tab, enter a table name and the table number 10001. Then enter some lapse rates on the "Values" tab. When done, click the "Save" button at the bottom of the dialog.

c. Create a table of premium rates by issue age. Use the Tables | Create a table… menu option. This time, click the radio button for table structure "Aggregate by age". Then fill in the minimum and maximum age you wish for the table and click "Edit table values", just as before. Give this table a name and the table number 10002, enter some values, and save the table as before.

12. Enter values for the assumption file. In your model, use the File | Edit | Assumptions… menu option. A dialog appears allowing you to select an assumption file. Navigate to the \Input subfolder and select Assumptions.af. When the dialog appears showing file contents, not that there are two panes. On the left there is a list of the assumption groups you defined earlier, allowing you to select the group being edited. On the right there is a worksheet allowing you to enter values for assumptions in the selected group. To edit an assumption, first select the assumption group and then click inside the worksheet containing assumption values and navigate to the appropriate value. Enter the following assumptions values:
    a. In the "Table numbers" group:
        i. Mortality rates table number = 1076
        ii. Lapse rates table number = 10001
        iii. Premium rates table number = 10002
    b. In the "Expenses" group:
        i. First year sales expense % premium = 0.95
        ii. Annual main. Expense per contract = 100
        iii. Expense per claim = 150
    c. In the "Sales" group, which has a date heading for each row:
        i. In the first row enter the date 201701 and Premium on new sales = 10000.
        ii. If you wish the level of sales to change over time, enter a new row for each later date on which the level of sales should change. The model will use the value in each row for all months up to the date in the next row, or forever if there is no next row.

When done, save and close the assumption file.

13. <u>Create a new investment portfolio file.</u> In your model, use the File | New | Inv portfolio menu option. A dialog appears allowing you to provide a path name. Navigate to the \Input subfolder and enter a name; the extension ".inv" will automatically be added. Then click on "Open". Another dialog will appear allowing you to specify parameters to create an investment portfolio consisting of cash and a list of bonds. You must enter the "as of date" for this data, the amount of cash, and parameters that define a list of bonds. To define the list of bonds, provide the total book value, coupon rate, credit quality (normally 2 or 3 for A or BBB) and the longest maturity in years. The bonds in the list will all be identical except for their maturity date. One bond in the list will mature in each calendar month for the number of years until the longest maturity. Click on "Create portfolio" to create the file.

14. <u>Create a new investment strategy file.</u> In your model, use the File | New | Inv strategy menu option. Specify a name for the file; the extension ".invst" will automatically be added. A dialog appears allowing you to specify an investment strategy. There are three tabs:

    a. The "Cash" tab contains parameters for cash management
        i. The "Maximum cash ratio" is the portion of the portfolio that will be kept in cash at all times for minimum liquidity. New investments will not be purchased unless cash in excess of this ratio accumulates. To force no future re-investment, set this ratio to 1.0.
        ii. The "Maximum borrowing ratio" specifies the largest loan that is allowed as a portion of the total portfolio. When cash is needed but not available, loans are simulated up to this maximum, then assets are sold. As future cash becomes available, it is used to pay off loans first. To simulate no borrowing, set this ratio to zero.
        iii. The "Borrowing interest spread in basis points" is the spread over Treasuries used to simulate the interest rate on loans.

    b. The "Fixed income" tab has two radio buttons that correspond to different ways of managing a fixed income portfolio.
        i. The "Duration matching" radio button is disabled in this version of the workbench because it has not been adequately tested. In a future version it will activate the duration matching strategy. You must specify the target duration in years, the maximum difference from target, and the longest maturity allowed for new investments when lengthening the portfolio to maintain the target duration.
        ii. The "Specified allocation" radio button activates a strategy whereby available cash is used to purchase a specified allocation of new fixed income investments. The allocation is specified by creating a list of investments to purchase and the

portion of cash used to purchase each one. The list includes the following information for each item to be purchased:

1. Fraction. This is the fractional allocation to this investment type.
2. InvType. This must be "Bond". Bonds purchased are semi-annual non-callable coupon bonds with a lump sum maturity payment. Later versions of the Workbench will allow mortgages and other fixed income instruments.
3. MaturityMonths. This is the number of months to maturity as of the purchase date.
4. Credit rating. Credit ratings are 1-6. Normal values are 2 and 3 for A or BBB.

c. The "Equity" tab allows you to specify three parameters of a simple stock portfolio strategy. Stocks are purchased when the aggregate value of stocks falls below a specified minimum fraction of total portfolio value. Stocks are sold when their aggregate value rises above a specified maximum fraction of the portfolio. When the aggregate value of stocks falls between those levels, a regular annual fraction of the stock portfolio is traded each year as specified by the turnover rate.

15. Create a new file listing policies in force. In your model, use the File | New | Contracts menu option. A dialog appears allowing you to provide a path name. Navigate to the \Input subfolder and enter a name; the extension ".inf" will automatically be added. Then click on "Open". Another dialog will appear allowing you to enter a list of contracts by specifying values for each field on the master record of each contract. The fields correspond to those you specified when you defined the contract. Note that you can create a list of identical contracts quickly in the following way. First enter all the fields for one contract and then, with the cursor on that contract, press Ctrl-c to insert an identical contract into the list. You can press Ctrl-c repeatedly to create a long list of identical contracts, and then perhaps go back to vary the issue ages or other parameters of each contract in the list.

16. Create a new sales mix file. In your model, use the File | New | Sales mix menu option. A dialog appears allowing you to provide a path name. Navigate to the \Input subfolder and enter a name; the extension ".inf" will automatically be added. Then click on "Open". Another dialog will appear allowing you to enter a list of contract types to be sold. The first field in each row is "PctSales", the percent of sales on the type of contract indicated in that row. Values in the "PctSales" column should add to 100. Fill in the other columns with values that specify a type of contract to be sold.

    Note that there will be a column for the IssueDate. The value you put there will be overridden with the actual issue date when new contracts are simulated by the model.

17. Create a new report definition. In your model, use the File | New | Report definition menu option. A dialog appears allowing you to provide a path name. In this case, do not navigate to the \Input subfolder or any other subfolder. Report definitions should remain in the same folder as the model program itself – the root project folder. Enter a file name for the report definition; the extension ".rptdef" will automatically be added. Then click on "Save". A dialog will appear allowing you to define your report. There are three parts to the dialog:

    a. The list of accounts to be used as the basis of the report. Every report follows the same general format, where ledger account names appear as row headings and dates appear as column headings. One must specify the list of accounts from which the row headings will be taken. Normally this is MainLgrAccounts.lgract from your root project folder.
    b. A name for the report.
    c. Items in the report. This listbox, which appears on the bottom right, specifies the row headings in the report. You add items to this list by highlighting an item from the list of available items on the left, and then pressing one of the buttons in the center to add that item, insert a blank line, or place an underline under the previous item. Once items are in the list of items in the report, you can manipulate them using the buttons in the bottom right corner of the dialog (delete, move up , move down). Suggestion – create a report whose line items correspond to represent an income statement and/or balance sheet.

18. Create a risk definition file. In your model use the Scenarios | Define risks… menu option to bring up a dialog for defining the contents of a scenario. A dialog appears allowing you to provide a path name. Navigate to the \Input subfolder and enter a name; the extension ".rsdef" will automatically be added. Then click on "Open". Another dialog will appear allowing you to define the contents of a scenario. The defaults are just interest rates and equity returns. Accept the defaults and save the file. (Note that there is an appendix named "Risk definition files" with further information regarding risk definition files.)

19. Create some economic scenarios. In your model, use the Scenarios | Generate Scenarios… menu option to bring up a dialog that serves as the scenario generator.
    a. Click on the button to select the Risk definition file and select the file created n the previous step.
    b. Click on the button to select the file to generate and type in a name with the .xml extension, such as "MyScenarios.xml".
    c. Enter the date to start from (201612) and the number of months to include in the scenario. Note that values for the last month are treated as applying forever thereafter in any model run for a longer time period.
    d. In the box titled "Scenarios to generate:" click on the "Stochastic scenarios" radio button and leave the number of scenarios at the default of 100.

e. If you wish, type in new values for the initial yield curve, that is, the US treasury yield curve on the start date (201612). Default values are already in the form and can be used.
f. Click on the "Generate scenarios" button. You will get a message box saying "Scenario generation complete" when your file has been created. A file with the name you specified in b. above will then exist.
g. Close the scenario generation dialog. Use the Scenarios | View scenarios… menu option to bring up a dialog that allows reviewing the scenarios you generated. First you must select the file to view, so select the one you just created. Note that there will be two files with similar names, one with the extension .xml and the other with the extension .offsets.xml. Be sure to view the .xml file, not the .offsets.xml file as the latter is just an index of scenarios, not the scenarios themselves.

20. <u>Edit the runtime parameters file to put values in it.</u> In your model, use the File | Edit | Runtime parameters menu option. A dialog appears allowing you to select a runtime parameters file to edit. You should find one in the /Input folder. After you select a file, a dialog box will appear allowing you to edit the runtime parameters. Note that many of the parameters are file names or folder names. When the cursor is on the input line for one of those parameters, you can click on the appropriate button at the bottom, either "Browse for a file…" or "Browse for a folder…" to select an item without the need to type out its full path name.

    It is suggested that input files normally be found in the \Input subfolder, and output files be placed in the \Output subfolder. Also, the \Tables subfolder is normally used as the folder where tables are to be found.

    Here are sample values for your first runtime parameters file. Full path names should be used, so fill in your project folder name in place of the "…" that appears in names in the list below. If you named your input files differently, use the names of the input files you saved:

Start date	201612
Months to run	60
Scenario file	…\Input\TestScenarios.xml
Inv portfolio file	…\Input\MyPortfolio.inv
Inforce file	…\Input\MyInforce.inf
Sales mix file	…\Input\MySalesMix.smx
Audit file	…\Input\MyAudits.audit
Assumption file	…\Input\Assumptions.af
Risk definition file	…\Input\MyRiskDef.rsdef
Tables folder	…\Tables
Main ledger file	…\Output\Main.lgr
Inv ledger file	…\Output\Inv.lgr
Inv strategy file	…\Input\MyStrategy.invst

Calc det reserve	No
Calc stoch reserve	No
Calc RSM reserve	No
Calc RSM stoch reserve	No
Num stoch scenarios	50
Num months in valuation	240

There are some check boxes on the right hand side of the runtime parameters dialog:

- Save investment detail ledgers. If checked, the investment ledgers created during any outer loops will be saved, otherwise those ledgers will not be saved because the needed summary investment results are included in the main ledger.
- Accumulate surplus. This checkbox does nothing by default, but the model developer can retrieve its value and make it have some effect if desired.
- Save transaction detail files. If checked, an audit file will be written for each scenario. The file name will be named with the main ledger file name with the word "_Audit_" and the scenario number appended, and will be in .csv format. The file will contain one record for each month for each contract ID being audited. The record will contain the entire starting master record for the month plus all the calculated amounts for the month. The list of contract ID's being audited appear below this checkbox when it is checked and is saved in the "audit file" named in the list of runtime parameters.

21. Close your model and go back to the Workbench to define some calculations. You need to define the calculations required for the contract you are modeling. Open the program code for the "Term20ContractClass.cs" file and edit the code for the "calc" methods. Below is sample code that needs to be added to Term20ContractClass.cs to fill in the "calc" methods. Use the Workbench's source code editor to fill in this section of the file, and don't forget to save the changes.

    The following notes might be useful:

    - The name of the file to edit should appear in the Workbench's top left list box under "Program code". If you didn't name your product "Term20" then the file name will be …ContractClass.cs where the "…" corresponds to your product name.
    - Each required calculation routine already appears in the code that was auto-generated. However, the space between the beginning and ending braces is blank. You must fill in the code required to calculate the variable whose name appears after "calc" in the name of each calculation routine. For example, calcPolicyMonth() must include ccode that results in assigning a value to PolicyMonth.

- In places where an assumption value is needed, you can quickly add the code to retrieve the assumption by double-clicking on the assumption name in the list in the Workbench's lower left panel.
- In places where a field name or the result of a function is needed, you can quickly type the correct spelling and capitalization of the name by double-clicking on the name in the list in the Workbench's top right panel.

Sample code appears on the next page. Note that no code is filled in for calcTotReserves() as that calculation could be very long. For purposes of this tutorial we leave it out, thereby treating the variable TotReserves as zero for now. Clearly the model will not be complete until the calculation of reserves is specified.

```
// Calculation routines **********************************

public void calcPolicyYear(int aDate)
{
        // First policy year is 1
        int numMonths = BFDate.monthsBetween(IssueDate, aDate);
        PolicyYear = (int) (1 + Math.Truncate(numMonths / 12.0));
}

public void calcPolicyMonth(int aDate)
{
        // First policy month in each year is 1
        int numMonths = BFDate.monthsBetween(IssueDate, aDate);
        PolicyMonth = (numMonths % 12) + 1;
}

public void calcTotPremium(int aDate)
{
        TotPremium = 0;
        if(PolicyMonth == 1)
        {
                int tblNum = (int) block.GetAsmpValue(aDate, A.PremRatesTableNum);
                double premRate = TableManager.GetValue(tblNum, IssueAge);
                TotPremium = premRate * FaceAmount * NumContracts;
        }
}

public void calcTotClaims(int aDate)
{
        TotClaims = NumClaims * FaceAmount;
}

public void calcTotExpenses(int aDate)
{
        TotExpenses = NumContracts * block.GetAsmpValue(aDate, A.AnnualMaintExp)/12;
}

public void calcNumClaims(int aDate)
{
        int tblNum = (int)block.GetAsmpValue(aDate, A.MortalityTableNum);
        double qx = TableManager.GetValueSelect(tblNum, IssueAge, PolicyYear);
        qx = 1 - Math.Pow(1 - qx, 1.0 / 12.0);
        NumClaims = NumContracts * qx;
}

public void calcNumExpiries(int aDate)
{
        NumExpiries = 0;
        if((PolicyYear == 20) && (PolicyMonth == 12)) NumExpiries = NumContracts - NumClaims - NumLapses;
}

public void calcTotReserves(int aDate)
{
}

public void calcNumLapses(int aDate)
{
        int tblNum = (int) block.GetAsmpValue(aDate, A.LapseTableNum);
        double wx = TableManager.GetValue(tblNum, PolicyYear);
        wx = 1 - Math.Pow(1 - wx, 1.0 / 12.0);
        NumLapses = NumContracts * wx;
}
```

Be sure to save the file after making these changes.

22. Re-generate the contract source code. This step may seem out of place, because we previously generated the source code and just finished editing it. Nevertheless, THIS STEP IS IMPORTANT because it updates the order in which calculations take place for the month.

    Before re-generating the code, scroll down in the code for your ContractClass.cs to the method ProcessMonth(aDate). (Or you can use Ctrl-F to activate Edit|Find.) Note that ProcessMonth() calls each of the calcXXXX() methods in the order the calculated items were initially declared. It is possible, however, that the calculation of one of the early items in the list might now depend on the value of an item yet to be calculated later in the list. That would create an obscure error. When the code is re-generated after you make changes, the dependencies among the calculated items are reviewed and the order of the calculations in the code generated for ProcessMonth() may change as a result. Also, if any circular references are found, a message box will notify you of them.

    To regenerate the contract source code use the Model Assembly | Generate code… menu option. In the dialog, check the box for “Contract”. Then click “Generate selected modules”.

23. Update your model to reflect the revised calculations. Use the workbench’s top menu Model Assembly | Build... menu option. This will invoke the compiler and create the .exe file in your project directory. If successful, you will get a message dialog saying “Model build was successful”. If not, a list of compiler messages will appear at the bottom of the screen. If you double-click on any message in the list, the file with the offending code should open in the editor window with the cursor placed at the point of error. Often such errors will be simple syntax or spelling errors. It is up to you to correct the code, save the code file, and rebuild the model until all errors are eliminated and you get the “Model build was successful” message.

24. Run your model. In your model, use the Model | Run menu option or click on the blue arrow on the toolbar. A dialog appears allowing you to select a runtime parameters file. Once you specify the runtime parameters, a list of scenarios appears in a listbox on the left side. These are the scenarios contained in the scenario file named in the runtime parameters. You must select the scenarios you wish to run. If there are many scenarios and you wish to run them all, click on the checkbox labeled “Run all scenarios” and they will all be selected. However, we suggest running just one scenario at this time. After selecting a scenario (or scenarios), click on the “Start” button in the middle of the dialog. Messages indicating progress will appear. When done, the last message will be “Model run complete”.

25. Generate and view a report, and save it as an Excel workbook. In your model, use the Reports | Select ledger menu option to select a ledger file created in your model run – this would be the filename you specified for the main output ledger with the scenario ID appended to it. After

selecting the ledger file, note that its name appears on the toolbar labeled as "Reporting from". Any reports that are generated will use that file as their source.

In your model, use the Reports | User defined menu option. This brings up a dialog allowing you to choose one of the report definitions you created (these are .rptdef files in the root project folder). You can also specify which segment of the ledger to use, and you can specify the kind of dates to use as column headings. When all are specified, click the "View report" button.

A report window should appear containing your report in what appears to be and acts much like a Microsoft Excel worksheet. There is a toolbar at the top of the window that provides several functions including:

- save the worksheet as an Excel file (the blue disk icon)
- change the ledger segment being reported on (first drop-down list box)
- change the dates to use as column headings (second drop-down list box)

Use this approach to create and save one or more reports as an Excel file. To save as an Excel file, click the blue disk icon on the report window and you will be asked to provide an Excel file name to use.

26. Go back to change a calculation, rebuild the model, and run it to see the results of your changes. To do all this, follow these steps:
    a. Close your model. Use the File | Exit menu option or click the X in the top right corner of the window.
    b. Update the source code. In the Workbench, open the source code file Term20ContractClass.cs. Find the code for calcTotClaims(). Replace the line "*TotClaims = NumClaims * FaceAmount;*" with the line "*TotClaims = NumClaims * FaceAmount * 2;*" as if there were some sort of double-indemnity clause in the contract.
    c. Re-generate source code. In the Workbench, use the Model Assembly | Generate code… menu option. In the dialog, check the box for "Contract". Then click "Generate selected modules".
    d. Rebuild the model. In the Workbench, use the Model Assembly | Build… menu option.
    e. Start the model. In the Workbench, click on the toolbar's blue forward arrow.
    f. Update the runtime parameters. Use your model's File menu to edit a runtime parameters file. Change the name of the "Main ledger" and save the modified list of runtime parameters under a new name using File | Save As.
    g. Run the new model using your new runtime parameters.
    h. Select a ledger from the new model for reporting. Use the menu option Reports | Select ledger…
    i. Generate a report from the new model. Use the menu option Reports | User defined….
    j. Repeat steps h and i for a ledger from the previous model. You should then have two report windows open on the screen so that you can visually compare the results.

27. Add the calculation of reserves. This was left out until now because the calculation of reserves for insurance contracts can be complex, and is of no interest to Workbench users in other kinds of business. One example of adding formulaic liability calculations appears in section 3.4 below.

3.4. Ideas and approaches for more complex models

This section provides general approaches that can be used to modify the C# code generated by the Workbench to implement more complex models. Users that become familiar with the generated code will find that most common modifications can be done in a straightforward manner because the Blufftop Framework provides a clear structure within which to insert additional processing.

Adding formulaic liability calculations

The liability for a contract appears on the balance sheet and must be calculated. Often a formula-based value can be used. The library ReserveLib.dll provides some utilities for US statutory reserves for some kinds of insurance contracts. Here are two examples showing how one might use ReserveLib.dll or a similar library of your own creation.

Net premium reserves for life insurance

Net premium reserves for life insurance can be calculated from a set of "commutation functions". Life insurance actuaries in the US are familiar with these functions with the names Cx, Mx, Dx, and Nx. The x is a subscript denoting an attained age in years.

ReserveLib.dll contains a class named "CommutationFunctions". To use it, do the following:

- In the code for XXXInforceClass (the specialized inforce class code for your contract type) add a line to declare a class member that is an object of the "CommutationFunctions" class.

```
public CommutationFunctions commFunc;
```

- In the Initialize() method of XXXInforceClass, call the constructor for the object named above, and then set two of its properties, the mortality table number and interest rate. (Note that if the mortality table is a select table, you will need to set the issue age as well and also create a separate instance of "CommutationFunctions" for each issue age to be modeled.) For example:

```
commFunc = new CommutationFunctions();
commFunc.TableNum = 41;
commFunc.IntRate = 0.03;
commFunc.IssueAge = 35;
```

- In the contract definition, in the list of "functions", add an item for the reserve factor per dollar of face amount. Name it something like NetPremiumReserveFactor.

- Re-generate the code for your Contract type. The generated code in XXXContractClass.cs will contain an empty method named calcNetPremiumReserveFactor(). You must fill in the code for this method, and you can use the commutation functions in the code. To refer to the commutation functions object you put in the inforce class, use a reference like this:

```
((XXXInforceClass)block).commFunc
```

  where "block" is the normal reference to the inforce block in which the contract resides and in which you added the commutation functions object. In this case it must be typecast to the specific kind of inforce block for this product (shown as XXXInforceClass), because otherwise the interface to the block does not include a reference to the commutation functions you added. Once you have the reference to the object, you can use its Cx(age), Mx(age), Dx(age), and Nx(age) methods as needed to calculate the net premium reserve factor. For example:

```
CommutationFunctions cFunc = ((XXXInforceClass)block).commFunc;

double netPrem =
(cFunc.Mx(35) - cFunc.Mx(55)) / (cFunc.Nx(35) - cFunc.Nx(55));

…
```

  The formula above calculates the net level premium for a 20-year term contract issued at age 35. The actual NetPremiumReserveFactor formula would be more complex.

- <u>Build and test your model.</u> If everything goes well, you should see both the reserves and the increase in reserves in the output ledger. However, you may discover that the reserve for the model start date is shown as zero, as if it had not been calculated. It is important to understand the reason for this, and it is easily fixed.

  The reason the reserve (and other formulaic balance sheet items) are not calculated on the model start date is that ProcessMonth() is not called before the initial valuation and therefore all the calcXXX() methods are not called. In each future month, ProcessMonth() is called before PerformValuation() so all the calcXXX() methods have been called and values of all the calculated variables are available.

  To fix this, one must insert code at the beginning of ProcessMonth() in the ContractClass code to execute the required calcXXX() methods, but only if the date is the model start date. Code like that below could be inserted at the beginning of ProcessMonth(). The user would need to determine exactly which calcXXX() methods need to be called and in what order:

```
If (aDate == block.Ledger.StartDate)
{
       calcPolicyYear(aDate);
       calcNetPremiumReserveFactor(aDate);
       calcTotReserve(aDate);
}
```

Note that code inserted in XXXContractClass.cs for PerformValuation() is overwritten by the code generator, so if you ever regenerate the code for this ContractClass this inserted code will disappear. It is recommended that you keep a copy of this inserted code somewhere (such as in a text file) so that it can be easily re-inserted if it becomes useful to re-generate the code for other reasons such as the addition of more fields on the master record or more calculated variables.

The same situation applies to the code inserted above in XXXInforceClass.cs. The code generator knows nothing about the inserted member "commFunc".

The code generator does retain the code you put in all the calcXXX() methods in XXXContractClass.cs. More details on which changes the code generator retains and which get overwritten is included in the Appendix on the source code generator.

Reserves according to Actuarial Guideline 38

The calculations in Actuarial Guideline 38 are very complex and are specific to each contract. In this case, we need to add a number of arrays of calculated items to the data maintained in memory for each contract. These arrays are not part of the contract master record; they are created and calculated when the contract is initialized for modeling either as part of the initial inforce or as a new sale.

To accomplish this, one can create a new C# code file that extends the code for the ContractClass. Name this file XXXContractClassExtensions.cs where the XXX is the name of the contract type.

An example of such a file is ULSGContractClassExtensions in the \Examples\AG38 folder. This file provides extra code for ULSGContractClass; code that will not be touched by the Code Generator. The code does three main things:

- Declares the arrays of values by policy duration needed for AG38 calculations.
- Provides a method named Setup() to calculate the arrays.
- Provides a function named AG38Reserve() to calculate the reserve for a particular duration using values from the arrays and the current contract status.

Note that all of the data on the contract master record is available to the Setup() and AG38Reserve() routines since they are inside the same class with the rest of the contract.

These routines need to be called from the code for XXXContractClass.cs, and the user must insert the code to make the calls. Setup() should be called from SetLgrSeg() because that routine is already called when a contract is being created for modeling. AG38Reserve() should be called from within the calcXXX() routine where XXX is the name you gave to the AG38 liability value.

One further coding change is needed. The Clone() method in XXXContractClass.cs must be expanded so that it creates a copy of the arrays declared in XXXContractExtensions.cs for the cloned contract.

Important: The code generator knows nothing about these arrays, so it does not add them to the auto-generated Clone() method. If you run the code generator again later to generate XXXContractClass.cs, you must be sure to re-insert the code to copy these arrays because the code generator will have deleted it because it knows nothing about them. The code generator maintains any code previously written in SetLgrSeg() or the calcXXX() methods, but always rewrites the Clone() routine to clone the data declared in the contract definition file.

Separating results by year of issue

Some accounting frameworks require treating each year of new business separately. This is often done for capitalization and later amortization of business sales and acquisition expenses, but is sometimes done for other reasons as well.

It is not difficult to implement this treatment in the Workbench. The following changes would be required:

- Edit the list of ledger segments in the file MainLgrSegments.lgrseg. Create a ledger segment for each year of issue, and create a virtual segment containing the total. When done editing the file, generate the code to define the names of the main ledger segments; this will rewrite the code file MainLgrSegments.cs.
- In the XXXContractClass.cs code, modify the SetLgrSeg() method so that it sets the ledger segment number based on the year the contract was issued. The issue date is normally one of the data fields on the master record.
- In the XXXInforceClass.cs code, in the Initialize() method, change the values given to the variables FirstLgrSeg and NumLgrSegs to specify the range of ledger segment numbers used by this inforce block.
- At the end of the ModelControllerClass.cs code there is a method called MapInvResults(). This method allocates investment portfolio results (assets, income, capital gains, etc) to the ledger segments in proportion to the total liabilities in each segment. You may wish to use some other allocation basis, and if so you will need to edit the code in MapInvResults().
- Inside the generated code for XXXControllerClass.cs there is a method named PeriodEndProcessing(). This is called at the end of the simulation for each model month. If you wish to capitalize and amortize acquisition costs or perform other calculations that depend on the ledger totals for each year of issue, you could put the code to do so in PeriodEndProcessing().

Note that investment portfolio yields are calculated and stored in the ledger using the code in the routine PeriodEndProcessing(), which is contained in XXXControllerClass.cs. The code provided by default stores investment yields in ledger segment 1 only. This is so that if you create a report with total results across many ledger segments, the investment yield is added in only once. If you wish to record the investment yields elsewhere, you need to change the code in PeriodEndProcessing() to do so.

Contract changes such as annuitization

Some insurance contracts involve two distinct phases, an active or accumulation phase, and a payout or claims phase. Here are some examples:

- A deferred annuity contract involves accumulation of premiums for a period of time, after which the money may be withdrawn or used to fund future lifetime income payments. This is a contract with an accumulation phase potentially followed by a payout phase.
- A long-term disability insurance contract involves payment of premiums while healthy, and the potential for a claim to put the contract in claim payment status. This is a contract with an active phase and a claims payout phase. Long-term-care insurance is similar in this respect.

In the Workbench it can be appropriate to model the active or accumulation phase as one contract, and model the payout or claims phase as a separate contract. Two different contract definition files are created, one for the active or accumulation phase and one for the payout or claims phase. The XXXContractClass.cs for each contract in the active or accumulation phase is modified to contain a list of the payouts or claims contracts that have arisen from it.

The following kinds of changes are needed to implement a model of this sort.

- Create two different contract definition files, one for the active or accumulation phase and one for the payout or claims phase.
- When code is generated, there will be an YYYInforceClass for the payout or claims phase (where YYY is the name of the payout or claims contract definition). An instance of this inforce must be included in the contract code for the active or accumulation phase. To do so, create a separate code file containing extensions to that class, possibly named XXXContractClassExtensions.cs. Inside the code for XXXContractClass add an instance of the YYYInforceClass for payouts or claims that will arise from this contract. You may also wish to add other code to support creation of new claim or payout contracts.
- There is a method in the generated code for XXXContractClass.cs which is normally left blank. It is called ProcessContractChanges() and is called at the end of each month. This is where any new payout or claims contracts should be created and added to the list contained in the active or accumulation phase contract. Also, the list's processMonth() method must be called to process all of the payouts or claims in the list.
- The PerformValuation() method in the auto-generated code for XXXContractClass.cs does not know about the list of payouts or claims. You must insert a line of code to call the PerformValuation() method of the list.

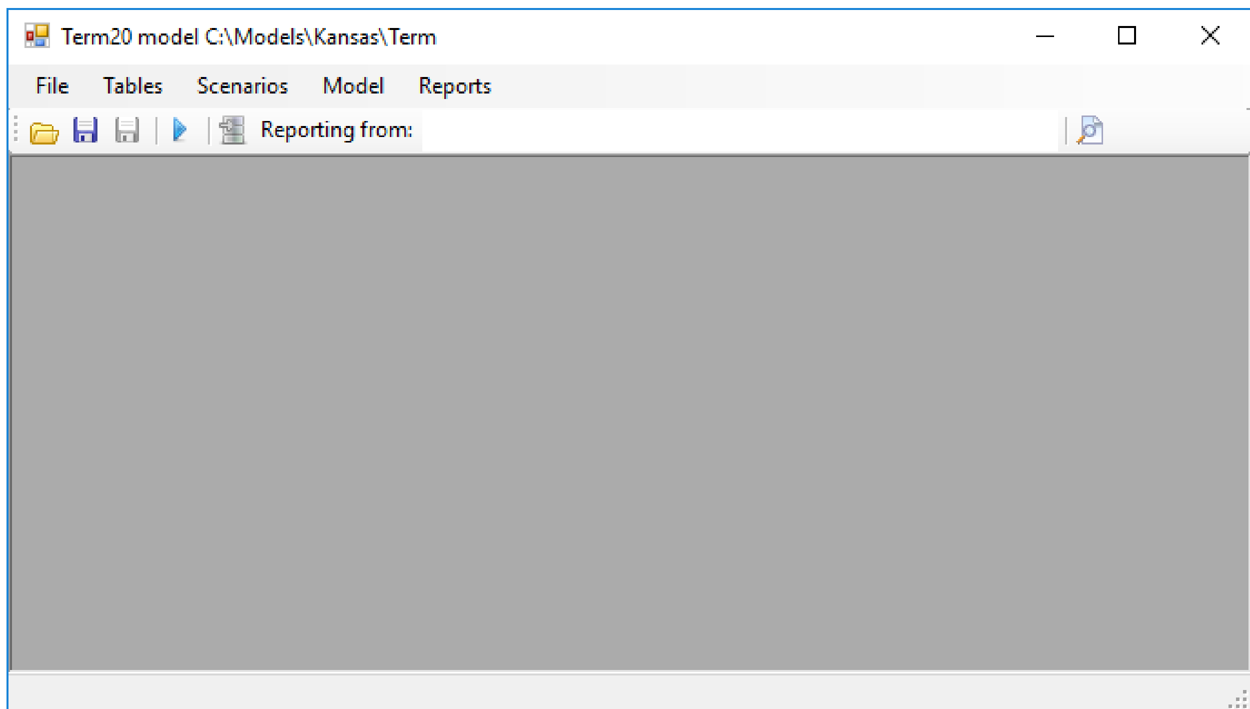
This is just a broad outline of the changes that are needed. As you get familiar with the object-oriented structure of the code, filling in the details will not be difficult. Careful testing is of course always necessary to ensure all changes work as intended.

4. How to use models produced by the Workbench

Models produced by the Workbench carry out asset / liability simulations using future economic scenarios, with financial statements as the primary output. This section describes how to run the simulations and how to view the output.

4.1 The top level user interface

Every model developed using the Workbench uses the same top level user interface. When the model is started, the main window appears:



The window title bar displays the name of the model and the working folder, which is normally the project folder.

The top menu provides the means to edit or create model input data, run the model, and view the results. The top menu options are:

- **File meu.** This provides options to create or edit any of the model input files, including
  - Assumptions
  - Runtime parameters
  - Inforce files (lists of contracts at the model start date)
  - Sales mix files
  - Investment portfolios

- Investment strategies
- Report definitions

All of the files listed above are stored in XML format. In some cases it is easier to prepare the list of contracts in .csv format using Excel spreadsheets. Therefore the file menu provides options to import from or export to that format.

- **Tables menu.** This provides the ability to create, view, or edit the actuarial tables used by the model. Tables are stored in files in the \Tables subfolder. Each table file has the name tNNN.xml where NNN corresponds to the table number. Creation of tables is the responsibility of the user; no tables are provided by default. The Society of Actuaries maintains an online library of standard tables that can be downloaded in .xml format and used here. In most cases the user will want to create some proprietary tables for use as well.

- **Scenarios menu.** This provides the ability to create or view files of scenarios to be used by the model. It also provides an option to create or edit a risk definition file – a file that defines the risk drivers used in the Representative Scenarios Method (RSM). More information is in the subsection below on "Working with Scenarios".

- **Model menu.** This provides the option to run the model, also provided by the forward arrow button on the toolbar. More information is in the subsection below on "Running the asset/liability simulations".

- **Reports menu.** After the model has been run, this menu option provides options to select ledger files and generate reports based on them. The path name of the currently selected ledger appears on the toolbar after the "Reporting from:" label. Reports appear in spreadsheet-style windows. More information is in the subsection below on "Viewing reports".

Many of the top menu options bring up dialog boxes or other windows that appear in the large grey area in model main window. When it is useful, several different windows can be displayed there at the same time; for example several different reports might be displayed side-by-side.

The toolbar provides buttons to open a file, save a file when editing it, save a file under a new name. The forward arrow icon brings up the dialog to start a model run.

The looking glass icon at the right end of the toolbar is a file viewer utility. Most files are in either XML or CSV form, so this utility allows viewing XML files in their native form as readable text, or viewing CSV files as worksheet tables.

4.2 Files used by the model

All of the files used by a model normally reside in either the project folder or sub-folders within the project folder. The files can be listed in three categories: input files, output files, table files, and model

support files. Each category normally resides in a separate designated folder, but the user can change the folder structure.

4.2.1 Input files

Input files are created by the user to provide data for the asset/liability simulation. They are normally kept by the user in the \Input sub-folder under the project folder. Files that are normally kept there include the following:

- Runtime parameters (*.rp) Specify all the file names and other inputs for a model run
- Assumption files (*.af) Contain all assumption values for a model run
- Inforce files (*.inf or *.csv) Contain the list of contracts at the start of a model run
- Sales mix files (*.smx) Specify the mix of new contracts to be sold in a model run
- Investment portfolio files (*.inv) Specify the list of investments owned at model start
- Risk definition files (*.rsdef) Define the risk drivers for use in RSM scenarios
- Scenario files (*.xml, *.offsets.xml) or (*.csv, *.index.csv)
    - Specify scenarios of future economic conditions
    - Each set of scenarios is in two files, one for data and one as an index

4.2.2 Output files

Output files are generated by running scenarios through the model. They normally are written to the \Output folder under the project folder.

The following output files are commonly produced:

- **Ledger files (*.lgr).** The model always saves the main ledger for every output loop scenario that is run. The file name is as given in the runtime parameters, but with the scenario ID appended before the file extension.
- **Stochastic summary files (*.lgrdb).** After a set of stochastic scenarios has been run, the Reports menu allows the user to create a summary of selected stochastic results. Such a summary is contained in this file, with the same name as the main ledger and the extension .lgrdb.
- **Audit file (*.csv).** When audits are requested, an audit file contains one line for each month and for each contract being audited. The data shows the full master record and all calculated amounts for that month. The file name is the name of the main ledger, but with the word "Audit" appended.
- **RSM calculation file (*_RSM*.csv).** When RSM liability calculations are performed in an inner loop, a summary of the PV of cash flows for each RSM scenario is saved with other summary data in a file for each valuation date. The file has the same name as the main ledger, but with "RSMYYYYMM" appended after the scenario ID. YYYYMM is the year and month of the valuation date. The file is in .csv format.

4.2.3 Table files

Table files contain actuarial tables. These can be tables of values by age or duration, select tables by age and duration, or continuance tables of the sort commonly used for disability insurance.

Tables are normally kept in the \Tables sub-folder under the model project folder. The user is entirely responsible for files that are placed there. Each table is stored in its own file in .xml format. The file name is "tNNN.xml" where NNN is the table number.

The \Tables sub-folder also contains an index to the tables in it. The index is in the file IndexOfTables.xml.

The Society of Actuaries keeps a library of actuarial tables available on the internet in the .xml format that is used by the Workbench. That format is called XtbML and was developed by the Technology Section of the Society of Actuaries in cooperation with ACORD. It is also the format used by the Table Manager software that has been used by many actuaries.

4.2.4 Model support files

Model support files are required for the model to run, but normally are not listed in the runtime parameters. They must be present in the same folder as the model executable file (*.exe), which is normally the project folder that contains all of the source code and other files that define the model.

The following model support files must be present in the same folder as the model executable file (*.exe)[6]

- Ledger definitions
  - MainLgrAccts.lgract
  - MainLgrSegs.lgrseg
  - InvLgrAccts.lgract
  - InvLgrSegs.lgrseg
  - InvLgrMap.lgrmap

- Standard default rates and credit spreads
  - VM20CreditRiskTables.csv

- Libraries of compiled code used by the model software
  - BFCoreLib.dll — Core software routines in the Blufftop Framework
  - BFFormLib.dll — User interface routines in the Blufftop Framework
  - BFInvLib.dll — Investment processing in the Blufftop Framework
  - BFReserveLib.dll — US statutory reserve calculations in the Blufftop Framework

---

[6] If the project is moved for development using Visual Studio, then all of the model support files must be placed in the same folder with the executable file (.exe). In Visual Studio that is normally in a \bin\Debug or \bin\Release sub-folder of the project folder.

- BFScenLib.dll — Provides scenario data routines for .csv files
- SpreadsheetGear.dll — Provides spreadsheet capabilities (third party)
- TeeChart.dll — Provides charting capabilities (third party)
- TSScenarioLib — Provides scenario data routines using EconSML in .xml files.
- XtbmlLib.dll — Manages actuarial tables using XtbML in .xml files.

4.3 Working with scenarios

Every model created with the Workbench includes a multi-risk scenario generator. Included in that generator is the economic scenario generator defined in VM-20 for use with life insurance reserves in the US. The generator is expanded here to include additional risks defined by the user, and to generate scenarios consistent with the Representative Scenarios Method.

4.3.1 Defining risks

Before one can generate multi-risk scenarios, one must define some risks. To do so, use the top menu Scenarios | Define risks... option to create or edit a risk definition file. If you are creating a new file, you will be asked to specify the folder and filename to use. If you are editing an existing file, you will need to specify the file to edit.

After selecting the file, you will see the dialog box below. This dialog box below enables the user to specify the complete set of risk drivers to be included in a set of scenarios, with information about the probability distribution for each risk driver. This information can be stored in a file that is named with the extension .rsdef (for risk definition file).

The list of risk drivers in the file is shown in the top left corner. When you create a new file, the "Equity" and "Interest" risk drivers will already be included, since those are in every set of scenarios used for VM-

20. You can add more risk drivers by typing a name next to the "Add:" button and then clicking the button. You can delete a risk driver by highlighting its name in the list and then clicking the "Delete" button.

Whenever a risk driver other than "Equity" or "Interest" is highlighted, the other parts of the dialog box allow you to specify characteristics of the highlighted risk driver. Characteristics include the following:

Distribution form:

You must select one of the radio buttons to specify the form of the probability distribution for the highlighted risk driver.

Distribution applies to:

The distribution can apply either to experience for a single year (so it changes from year to year in each scenario) or it can apply to the entire lifetime of each scenario (so it is constant for all years of one scenario but varies between scenarios). In most cases the distributions apply to a single year of experience, but the distribution for a risk driver like the rate of mortality improvement might be specified as applying to a scenario lifetime.

Experience study data:

If the selected distribution form is Binomial, Poisson, Beta, or Gamma, then the user must input some experience study data. Three items are required, and they should be based on experience for one year. The items are 1) the number of observed risk events, 2) the total number of contracts exposed to the risk event, and 3) the ratio of the number of observed risk events to the anticipated number based on the table to be used in the projection model. After entering this data, the user should click on "Update selected risk". Then the distribution is calculated based on the entered data, a chart of the probability density function (PDF) is shown on the bottom right, and some percentile points on the distribution are shown on the top right.

The calculation of the distribution based on the experience data is one way to simplify the development of margins for VM-20. The distribution tends to get narrower as the volume of experience data (especially the number of observed risk events) increases, thereby decreasing the margins in reserves based on the generated scenarios.

If the selected distribution is User-defined, then this area is greyed and disabled. Instead, the percentile points on the distribution must be entered manually on the top right of the form. One must then click "Update selected risk" to update the chart of the PDF on the bottom right.

Percentile points in user-defined distribution:

These are percentile points on the distribution of the Actual / Tabular ratio for the highlighted risk driver. If the selected distribution is User-defined, then the user must enter these numbers and then

click "Update selected risk". If the selected distribution is not user-defined, these numbers are updated automatically when the user clicks on the button to "Update selected risk".

When the distribution is updated automatically, the results reflect both statistical risk and parameter risk. Statistical risk is random fluctuation around the mean. Parameter risk is the risk that the mean from the experience data may not be exactly the true mean.

Normally the figure entered for the 50% level (the median) will be near 1.0 because these are percentile points on the distribution of the Actual / Tabular ratio.

<u>Representative scenarios for selected risk:</u>

The checkboxes in this area allow the user to specify which Representative Scenarios are to be generated for this risk driver. This is only relevant when generating Representative Scenarios; the information is ignored when generating stochastic scenarios.

4.3.2 Generating scenarios

To generate scenarios, use the top menu "Scenarios" item and select the option to generate scenarios. When you do so, the dialog box below appears.

GenerateScenariosForm

Risk definition file: C:\Models\Kansas\Term\Input\TermRisks v1.rsdef
Select...

File to generate: C:\Models\Kansas\Term\Input\ScenariosV1.xml
Select...

Start from date (YYYYMM): 201412
Number of months: 480

Scenarios to generate:
- Representative scenarios
- Stoch. exclusion test scenarios
- Stochastic scenarios 100

Generate scenarios

Initial yield curve (U.S. treasuries)

3 mo.	0.0004
6 mo.	0.0012
1 yr.	0.0025
2 yr.	0.0067
3 yr.	0.0110
5 yr.	0.0165
7 yr.	0.0197
10 yr.	0.0217
20 yr.	0.0247
30 yr.	0.0275

Mean reversion point: 0.0400

Before you can generate scenarios you must specify the risk definitions to be used and where to put the generated scenarios. The two "Select" buttons at the top of the dialog allow you to specify a risk definition file and the name of the file to generate.

Once the files are specified, you must specify the starting conditions. These include the scenario starting date, number of months, starting yield curve, mean reversion point. You must type these values into the boxes that are provided. The values shown are just defaults, and you should review and probably change each one.

You must also select the scenarios to be generated. A scenario file can contain either Representative scenarios, Stochastic exclusion test scenarios, or Stochastic scenarios. Use the radio buttons on the left to specify which kind of scenarios you wish to generate. When generating stochastic scenarios, you must specify the number of scenarios to generate. Note that for interest rates and equity returns, the generated scenarios will correspond to the first N scenarios out of the 10,000 scenarios from the official VM-20 generator.

Finally, you can click the "Generate scenarios" button to generate the scenarios. The bar just under that button will be filled from left to right to indicate progress, and a message box will appear when scenario generation is complete.

Once you have generated a scenario file, you can view it using the top menu option Scenarios | View scenarios… to select the file to view.

4.4 Running the asset / liability simulations

When the user selects either the menu option or the toolbar button to run the model, a dialog like the one below appears.

Initially the name of the runtime parameters file to use will be blank. The user must click the "Select…" button to select a file of runtime parameters.

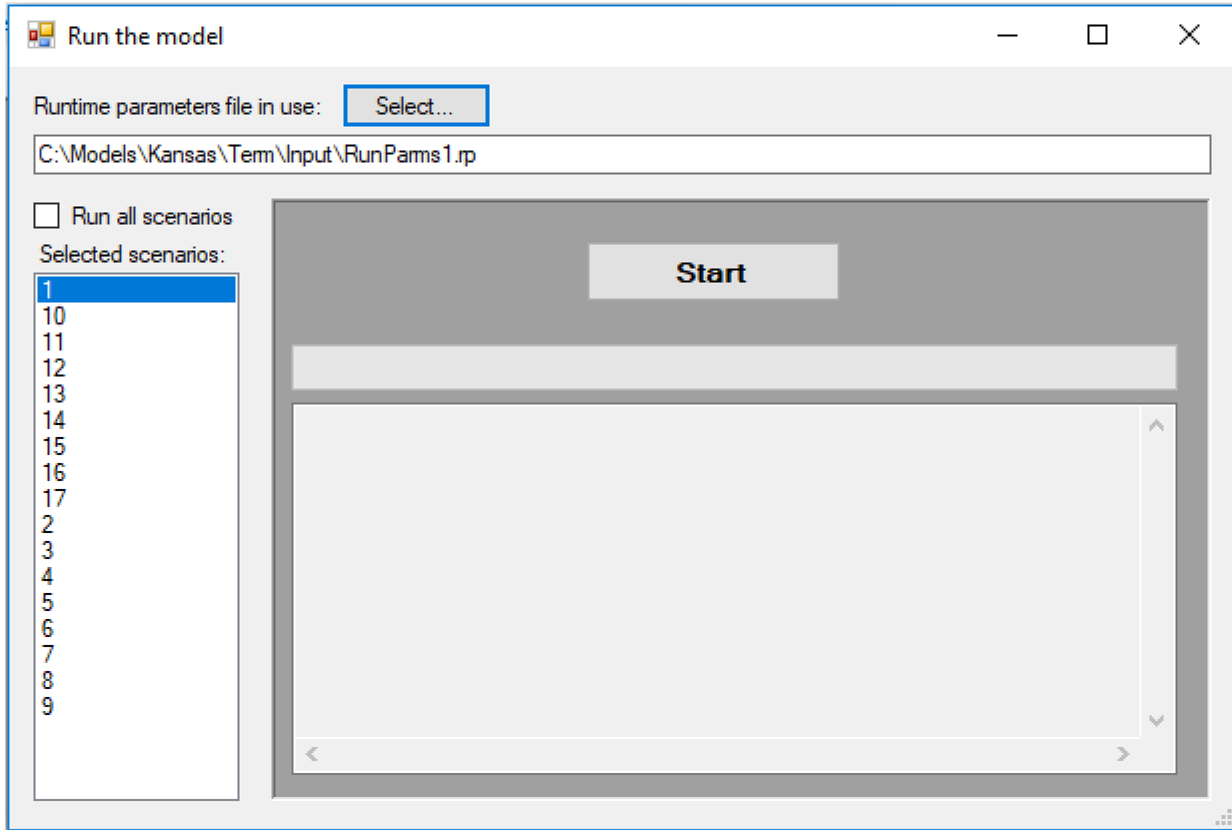
Once the parameters file is loaded, the list of scenario IDs on the left side will be filled in. This list comes from the scenario file specified in the runtime parameters.

To run one or more scenarios, select the desired scenario IDs on the left and then click the "Start" button. The model will start running, and messages will occasionally appear in the large box under the start button. The last message to appear will indicate completion of the model run.

The results of each scenario are saved in the main ledger file named in the runtime parameters. The scenario ID is appended to the name before the .lgr filename extension.

Note that the model is designed to run each scenario on a different processor, using up to four processors at a time. The limit of four is in place due to the likelihood of a data bottleneck if more processors are used on the same machine. When all processors are on the same chip, the same data

paths are being shared for all processors, and their capacity is limited. Models of this sort use lots of data, and experience has shown that use of more than four processors on one chip does not improve performance in such models.



4.5 Viewing the output

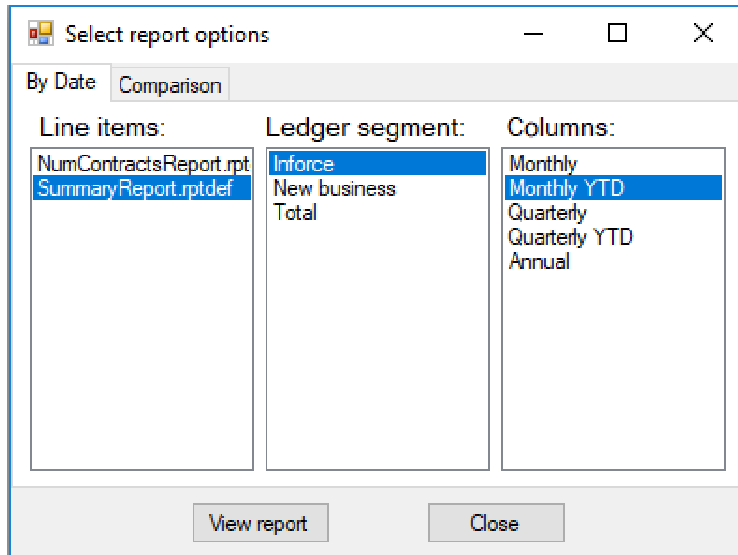
After the model has been run and one or more *.lgr files have been saved, the results can be viewed using the built-in report generator.

One must first select the ledger file to use when generating reports. Use the top menu Reports | Select ledger… menu option or click the corresponding button on the toolbar. After a ledger file is selected, its name will appear in the toolbar box labeled "Reporting from:".

Since the ledger chart of accounts can be user-defined, all reports from the ledger are also user-defined. To generate a report, use the top menu Reports | User defined… menu option. That brings up a dialog like the one below. The user can select:

- **Line items.** Each item in this list is a report definition file specifying the row headings.
- **Ledger segment.** Each item in this list is a subset of the business for which results were kept separately. Normally "Total" is the last item in the list.

- **Columns.** The column headings in the report will be dates, but they can be period-only or year-to-date, and they can be monthly, quarterly, or annual.



When your choices are made, click on "View report". That brings up a window like the one below containing a report with the requested row headings, dates, and segment. The report window looks and acts like a spreadsheet report, and if desired you can enter formulas in it just as you would with a spreadsheet. You can also use the toolbar button to save the report as a spreadsheet file in Microsoft Excel format.

The toolbar has drop-down listboxes that allow you to change the dates and segment of the business being viewed. There are also buttons to insert or delete rows or columns. The rightmost toolbar button is for drill-down. If you click the drill while the focus is on a cell in the report that contains a number that is the sum of other numbers, then rows will be inserted showing the detail items that are included.

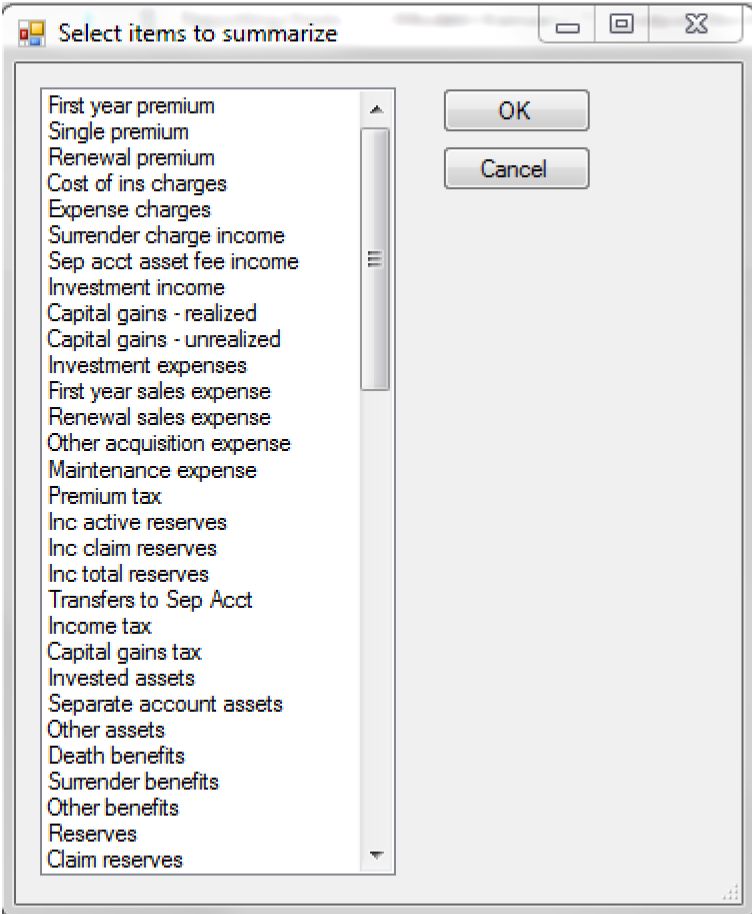
C:\Models\Kansas\Term\Output\Full Inforce\Level4pct_1.lgr: Summary report

Total | Annual

	A	B	C	D	E	F	G	H
1		201412	201512	201612	201712	201812	201912	202012
2	Total premium	0	93,315,727	88,593,703	84,101,886	79,829,407	75,765,206	67,604,822
3	Total investment income	0	8,732,969	11,809,998	14,432,287	16,837,829	19,134,469	20,985,468
4	Total income	0	102,048,695	100,403,701	98,534,173	96,667,235	94,899,674	88,590,290
5								
6	Total expense	0	11,598,189	10,977,112	10,423,918	9,897,992	9,415,378	8,743,614
7	Death benefits	0	34,661,954	38,351,651	41,354,098	44,342,405	47,748,116	48,300,690
8	Surrender benefits	0	0	0	0	0	0	0
9	Total benefits and expense	0	46,260,143	49,328,763	51,778,016	54,240,397	57,163,494	57,044,304
10								
11	Increase in reserves	0	51,946,147	42,343,670	33,669,088	25,301,367	16,562,622	7,861,264
12	Total disbursements	0	98,206,290	91,672,434	85,447,104	79,541,764	73,726,116	64,905,569
13								
14	Operating gain before tax	0	3,842,406	8,731,267	13,087,069	17,125,471	21,173,559	23,684,722
15								

Note that the window title bar names the ledger file from which the report was generated. It is possible to generate several reports from different ledger files and keep all the report windows open at the same time.

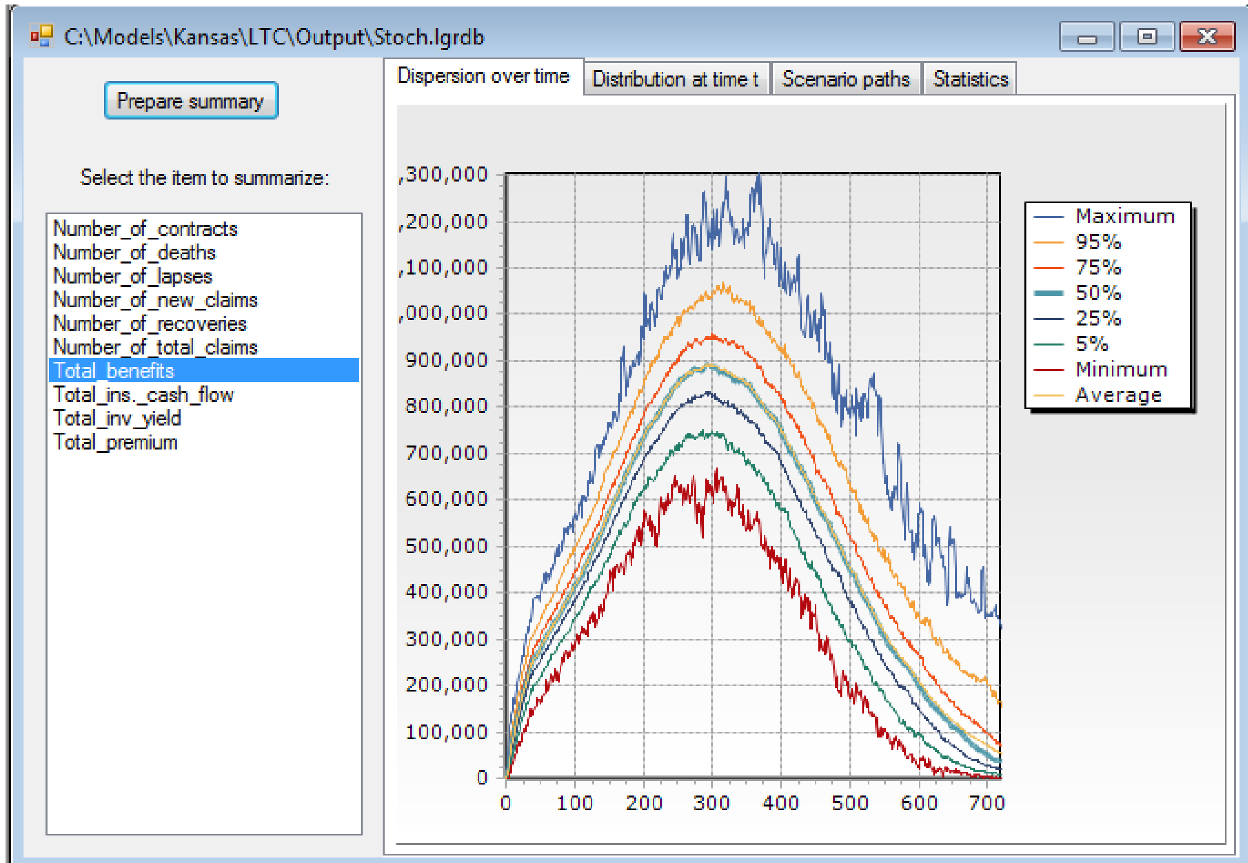
When a set of stochastic scenarios has been run, a large number of ledger files will be produced (one per scenario).   It can be useful to create a summary of key items across all scenarios and view their distribution. This can be done using the Reports | Stochastic summary | Create summary file… menu item. When this is selected, a dialog like the following appears, allowing you to select which items in the currently selected ledger (which must be one ledger from a stochastic run) should be included in the stochastic summary file.



The list of items is the chart of accounts from the ledger, including all virtual (subtotal) accounts. Select one or more items and click OK to prepare a stochastic summary. The created file will have the same name as the main ledger, but without a scenario ID and with the extension .lgrdb.

To view results of a stochastic summary, use the top menu option Reports | Stochastic summary | View results… This brings up a dialog based on the stochastic summary associated with the currently selected ledger file.   A sample is shown below for results from a stochastic model for long-term care insurance.

The listbox on the left shows the items for which stochastic results were collected. Select an item from that list and then click the “Prepare summary” button above the list. All of the tabbed displays on the right will then be updated with information on the distribution of stochastic results.



5. Appendices

Defining and using assumptions

The workbench allows the user to define numerical assumptions to be used. Assumptions are commonly used to set the values of things like the following:

- Contract expense charge rates
- Company expense rates or unit costs
- Tax rates
- Future sales amounts
- The table number of actuarial tables to be used for:
  - Claim rates
  - Contract termination rates

The Workbench does not come with any assumptions pre-defined. This is a major difference from most enterprise modeling platforms that come with very long lists of pre-defined assumptions. In the Workbench you define only the assumptions you need and wish to use, thereby skipping the learning curve associated with understanding a long list of pre-defined assumptions, many of which you may not need or use.

Assumptions are grouped together for purposes of viewing and editing. Each group of assumptions corresponds to a dialog box in which they can be edited together as a list. Most models include several assumption groups.

Below is a sample definition of an assumption group. Each time you ask the Workbench to create or edit an assumption group, a file editing tab like this one is added to the open file editing tabs on the workbench screen.

Assumption group name: Expenses and unit costs
Assumption group number: 1100
☐ Assumptions in this group vary by date

	Number	Text	Mnemonic	Decimals	Description
	1	Per contract issued	IssueExpPerContract	2	Insurer expense per new contract
	2	Per unit issued	IssueExpPerUnit	2	Insurer expense that varies by contract size
	3	Per claim	ExpPerClaim	2	Insurer expense per claim incurred
	4	Per lapse	ExpPerLapse	2	Insurer expense per contract lapse
	5	Per contract inforce	MainExpPerContract	2	Annual insurer expense per contract
	6	Premium tax rate (2% = 0.02)	PremTaxRate	4	Premium tax rate, 2% = 0.0200
▸	7	Percent of premium issued (10% = 0.10)	IssueExpPctPremium	4	Insurer exp as % of premium, 10% = 0.10
*					

Each assumption group must be given a name that describes its contents.

Each assumption group must be given a group number. This can be any positive integer that is divisible by 100, but you must avoid assigning the same number to more than one assumption group. This number is used internally by the Workbench for efficient lookup of assumption values.

The assumptions in each group can optionally vary by date. If this option is selected, all assumptions in the group vary by date, and they are displayed for editing using a column for each assumption and a row for each date. If this option is not selected, the assumptions are displayed for editing as a list with names on the left and values on the right.

Each assumption is defined by five items.

- **Number.** This must be an integer from 1 to N where N is the number of assumptions in the group. Each value 1 through N must be assigned to an assumption in the group. This is used internally by the Workbench for efficient lookup of assumptions during model runs.
- **Name.** This is the name that will be displayed when editing the assumption values.
- **Mnemonic.** This is the variable name that will be used in code to refer to the assumption. When code is generated, all of these mnemonics are put in the namespace A, so that the assumption can be referred to in code by A.MyMnemonic. That way there is no chance that the mnemonic will conflict with other variables that might use that name but are not in the namespace A. The C# code that defines this namespace all of these mnemonics is put into the file AssumpCodes.cs.
- **Number of decimals.** The number of digits to be displayed after the decimal point when editing the assumption value.
- **Description.** A short description of how the assumption is used. This is for your own documentation.

An assumption file contains all of the assumption groups defined within a model. Assumption files are named with the extension *.af.

After defining all assumption groups for your model, you must create an assumption file. This is done using the top menu item Model Assembly | Generate files | Assumptions. This will create the file Assumptions.af in the \Input folder under your model folder. If a file by that name was already there, it will be renamed with the extension .old.af and a new file will be created, carrying over any assumption values for assumptions that were in both the old and new files. Creating an assumption file also creates or overwrites the source code file AssumpCodes.cs that defines the list of assumption groups used by the model.

When using a model, you will sometimes want to save different sets of assumption values in separate files. To create a new file for a new set of assumption values, first use the “Save as…” option to save an existing file under a new name. You will then have two copies of the file with identical contents. Then edit the file with the new name, change the assumption values, and save it. The name of the assumption file that will be used in any run of the model is specified in the runtime parameters.

Actuarial tables and the Table Manager

Models developed using the Workbench include functionality that allows easy use of actuarial tables. The top menu bar includes an item named “Tables” which brings up a sub-menu for viewing, creating, and editing tables.

Every model project is created with a folder named “Tables” where such tables are to be stored. However, if the user has an existing library of tables stored in a different folder, the model can be told to use that folder using the menu option Tables | Select folder for tables…

Inside the Tables folder, each table is stored in a file with the extension .xml. The format used for tables is the same as used by the Society of Actuaries in their online library of tables, and is the same as that used by the Table Manager software that is widely used as an add-in to Microsoft Excel.

When values from actuarial tables are needed in model computer code, the code must specify the table number and age or other parameters needed to retrieve the desired value. One of the following syntaxes is used to retrieve the table value, depending on the type of table:

- `TableManager.GetValue(tableNum, age)`
- `TableManager.GetValueSelect(tableNum, age, duration)`
- `TableManager.GetValueContinuance(tableNum, age, duration, durationAxisName)`

The function syntax after “TableManager.” is exactly the same as is used in worksheet formulas when using the Table Manager add-in for Microsoft Excel.

Chart of accounts and the ledger

The Workbench allows you to define your own chart of accounts for use in the models you build. This gives you flexibility to simulate any set of accounting rules and to store any additional numerical information you wish in the ledger.

The file "MainLgrAccts.lgrdef" defines the chart of accounts in the main ledger. When you open that file for editing, a tab like the following appears in the file edit area:

Every account that you define has all of the following properties:

- **Name.** This is the title used when displaying this item on financial statements. It can include more than one word, as in "Investment income" or "Distribution expenses".
- **Mnemonic.** This is a shortened version of the account name that is used in computer code to refer to an account. The mnemonic must follow the rules for variable names in computer code. That means it cannot include blanks, but can be a combination of words or abbreviations as in "InvIncome" or "DistributionExp". A mnemonic must start with a letter, not a numerical digit.
- **Account Type.** The account type helps distinguish accounts that normally have a credit balance from those that normally have a debit balance, so that the sign of the balance can be displayed appropriately on financial statements. The account type is a single letter equal to one of the following:
  - **'A'** Asset
  - **'L'** Liability
  - **'I'** Income
  - **'E'** Expense
  - **'S'** Capital and Surplus
  - **'O'** Other. Accounts of this type are excluded from the "balancing" of the ledger. They are used to store amounts such as inventory counts, transaction counts, and any other result that would not normally be stored in an accounting ledger. Any account that is defined as a formula sum of other accounts is of this type.
- **Formula.** Financial statements include subtotals and totals. In the Workbench these are made available by defining accounts to hold them. These account balances are not actually stored; they are defined by a formula that can be calculated as needed based on other account balances that are stored. Because these account balances are not stored they are referred to as "virtual" accounts. The accounts that are actually stored have no formula and are referred to as "non-

virtual" accounts. Each account in the formula for a virtual account is indicated by its mnemonic preceded by a + or – sign. Note that the formula for one virtual account can include another virtual account, and you must be careful not to create circular references.

- **Account Number.** Each account you create is assigned a number automatically by the Workbench. It should be noted that the non-virtual accounts are numbered from 1 to N, where N is the number of such accounts. This numbering allows the account number to serve as an array index into an array of account balances. This helps speed the internal processing when account balances must be accessed or updated. Virtual accounts are numbered starting from 1000, leaving room for 999 non-virtual accounts and an unlimited number of virtual accounts.

The ledger account balances for any single time period comprise a two-dimensional array. One dimension corresponds to the chart of accounts and the other dimension corresponds to "segment" of the business. For modeling purposes, the business can be divided into segments in any manner you wish. The list of ledger segments is set up just like the chart of accounts, where each segment has a name, mnemonic, formula and number (but no "account type"). There can be both virtual and non-virtual segments. There should always be a "Total" virtual segment defined as the sum across all the non-virtual segments.

The chart of accounts that you define is contained in two files. The list of accounts is in MainLgrAccounts.lgract, and the list of segments is in MainLgrSegments.lgrseg. These files appear in the "Ledger definition" group in the project file list on the top left of the Workbench screen. You can edit these files to modify the chart of accounts to meet your needs.

When debit and credit entries are made to the ledger, both the account number and segment number must be defined for each entry. In computer code, the mnemonic is used as a variable that refers to the account number. To keep these mnemonics from conflicting with other variable names in the model, they are put in a separate "namespace".   When they appear in code, the mnemonic is preceded by the namespace name and a period. The account names are in namespace "L", so account numbers appear in code as L.AccountMnemonic.   The segment names are in the namespace "S", so segment numbers appear as S.SegmentMnemonic.

Models developed using the Workbench actually maintain two ledgers internally. One is the main ledger we have been discussing, and the other is a separate ledger used just for recording investment portfolio transactions. The accounts and segments in the investment ledger are standardized, so you are not expected to change them.

There is a separate investment ledger for each investment portfolio in the model. At the end of each accounting period, results in the investment ledger(s) are mapped into the main ledger. This is done for the following reasons:

1. The main ledger may keep less detailed investment results. The investment ledger keeps separate results for cash, fixed income, and equity investments while the main ledger may only store the total. In addition, the investment ledger keeps cash flow information that the main ledger may not need.

2. The total investment results may need to be allocated over the segments of the main ledger. For example, the main ledger may define a separate segment for contracts sold in each calendar year. If the investment portfolio supports all contracts, investment results must be allocated across all segments.

The mapping from the investment ledger to the main ledger is defined partly by data and partly in code.

The data used in the mapping is in a ledger mapping file named InvLgrMap.lgmap. Every model you construct has this file. It contains a list of the accounts in the investment ledger, and indicates the account in the main ledger to which each should be mapped. You can edit this file – it falls in the "Ledger definition" group in the project file list on the top left of the Workbench screen. When you edit the file, you see a dialog like the picture below, with the list of investment ledger accounts on the left (these are standardized in all models) and the list of main ledger accounts on the right (this is the chart of accounts you can define as you wish). Clicking the button on the bottom updates the mapping for the investment ledger account selected on the left. After editing the file you must save it for the changes to become permanent.

Inv. ledger account to map from:

Investment income
Investment expense
Realized capital gains
Unrealized capital gains
Invested assets
Income due and accrued
Investment liabilities
Writedowns
Cash income
Cash maturities
Cash purchases
Cash sale proceeds
Market value
IMR
IMR capital gains
IMR amortization

Main ledger account to map to:

First year premium
Single premium
Renewal premium
Cost of ins charges
Expense charges
Surrender charge income
Other ins income
Investment income
Capital gains - realized
Capital gains - unrealized
Investment expenses
First year sales expense
Renewal sales expense
Other acquisition expense
Maintenance expense
Premium tax
Increase in reserves
Transfers to Sep Acct

Update selected mapping

When the mapping requires an allocation, then program code is used to define the allocation. The routine MapInvResults(), inside the Controller, carries out the mapping and allocation. The version provided in the base class Controller (in ModelControllerClass.cs) allocates total investment results in proportion to the total liabilities in each main ledger segment. You can override this method in the Controller class that is set up for your specific model if a different allocation basis is desired.

Report writer

The Workbench allows you to define reports to be generated from information in a ledger. A report definition consists of a list of ledger accounts. That list is used as the set of lines in the report. Blank lines and underlines can be included in the list. Virtual accounts, which are defined as formulas based on other accounts, can be included in the list. This enables quick and easy definition of a financial statement report such as an income statement or balance sheet.

To create a new report definition, you must first select the chart of accounts to be used. The available charts of accounts are those that define the main ledger and those that define the investment ledger. These are contained in MainLgrAccounts.lgract and invLgrAccounts.lgract.

After selecting the chart of accounts to use, the report can be defined using a dialog like the one below.

The box on the left lists all available accounts in the chosen chart of accounts. The buttons in the middle insert lines in the list box on the right, which defines the report. The buttons on the bottom left allow the lines in the report to be re-arranged.

Every model generated by the Workbench includes a report generator that uses these report definitions to create reports in worksheet form. Quite a few different reports can be generated from one report definition, because the user can select the columns for the report at the time it is generated. The columns are normally calendar dates and can be annual, month-only or monthly YTD. The user can select the ledger segment for which the report should be generated. These options can be changed on the fly after the report window is displayed, using controls on the report window.

Reports appear in worksheet form in a report window. The user can manipulate the report in several ways just like a worksheet, inserting and deleting rows and adding simple formulas if desired. Reports can be saved as Microsoft Excel worksheet files for further manipulation, formatting, and analysis.

Risk definition files

After selecting the file, you will see the dialog box below. This dialog box below enables the user to specify the complete set of risk drivers to be included in a set of scenarios, with information about the probability distribution for each risk driver. This information can be stored in a file that is named with the extension .rsdef (for risk definition file).

The list of risk drivers in the file is shown in the top left corner. When you create a new file, the "Equity" and "Interest" risk drivers will already be included, since those are in every set of scenarios used for VM-20. You can add more risk drivers by typing a name next to the "Add:" button and then clicking the button. You can delete a risk driver by highlighting its name in the list and then clicking the "Delete" button.

Whenever a risk driver other than "Equity" or "Interest" is highlighted, the other parts of the dialog box allow you to specify characteristics of the highlighted risk driver. Characteristics include the following:

Distribution form:

You must select one of the radio buttons to specify the form of the probability distribution for the highlighted risk driver.

Distribution applies to:

The distribution can apply either to experience for a single year (so it changes from year to year in each scenario) or it can apply to the entire lifetime of each scenario (so it is constant for all years of one scenario but varies between scenarios). In most cases the distributions apply to a single year of experience, but the distribution for a risk driver like the rate of mortality improvement might be specified as applying to a scenario lifetime.

Experience study data:

If the selected distribution form is Binomial, then the user must input some experience study data. Three items are required, and they should be based on experience for one year. The items are 1) the number of observed risk events, 2) the total number of contracts exposed to the risk event, and 3) the ratio of the number of observed risk events to the anticipated number based on the table to be used in the projection model. After entering this data, the user should click on "Update selected risk". Then the distribution is calculated based on the entered data, a chart of the probability density function (PDF) is shown on the bottom right, and some percentile points on the distribution are shown on the top right.

The calculation of the distribution based on the experience data is one way to simplify the development of margins for VM-20. The distribution tends to get narrower as the volume of experience data (especially the number of observed risk events) increases, thereby decreasing the margins in reserves based on the generated scenarios.

If the selected distribution is User-defined, then this area is greyed and disabled. Instead, the percentile points on the distribution must be entered manually on the top right of the form. One must then click "Update selected risk" to update the chart of the PDF on the bottom right.

Percentile points in user-defined distribution:

These are percentile points on the distribution of the Actual / Tabular ratio for the highlighted risk driver.

If the selected distribution is User-defined, then the user must enter these numbers and then click "Update selected risk". If the selected distribution is not user-defined, these numbers are updated automatically when the user clicks on the button to "Update selected risk".

Normally the figure entered for the 50% level (the median) will be near 1.0 because these are percentile points on the distribution of the Actual / Tabular ratio.

Representative scenarios for selected risk:

The checkboxes in this area allow the user to specify which Representative Scenarios are to be generated for this risk driver. This is only relevant when generating Representative Scenarios; the information is ignored when generating stochastic scenarios. The 16% and 84% scenarios correspond to 1 standard error, while the 1% and 99% scenarios correspond to almost 3 standard errors.

Principle-based valuation and the inner loop

Models developed using the Workbench are used to project financial statements for future dates. Financial statements include a valuation of liabilities, so valuation must be done on future dates. Valuation can be carried out either by using user-specified formulas or using a discounted cash flow approach.

Principle-based valuation is defined here as a valuation equal to the discounted present value of future cash flows. To carry out such a valuation on a date in the future the model must carry out what is called an “inner loop” to project the cash flows starting from the valuation date. The “outer loop” is the loop through time starting from the model start date and running for the projection period. The “inner loop” starts from any valuation date during the projection period and runs for the number of time periods needed for the business on the valuation date to run off or expire. It is common for the “inner loop” to run until a later date than the “outer loop”.

The inner loop differs from the outer loop in other respects as well. Since there is no foreknowledge of the future economic scenario on the valuation date, it is common for the economic scenario used for the inner loop valuation to be different from that in the outer loop for dates after the valuation date. In addition, the experience assumptions used in the inner loop may include margins that are used for valuation but which are not part of anticipated experience in the outer loop.

The Workbench provides a framework for carrying out inner loop valuations. Part of that framework involves generating scenarios for the inner loop, and the Workbench comes already set up to generate and use inner loop scenarios required for US statutory life insurance valuations. This includes the deterministic and stochastic scenario sets defined in the valuation manual (VM-20). In addition, scenario sets based on the proposed Representative Scenarios Method (RSM) can be used.

The subsections below explain how the Workbench handles these aspects of the inner loop:

- How the model controller handles inner loops
- Requesting that the inner loop be included in a model run
- Generating the inner loop scenarios
- Adding margins to assumptions for use in inner loop valuation scenarios
- Capturing and using results of inner loop scenarios

How the model controller handles inner loops

The model simulation performs a valuation at the end of each time step. The valuation is done in two parts. First a formulaic valuation is done using any formulas specified by the user. Then the runtime parameters are checked to determine whether an inner loop should be run. If an inner loop is to be run, the routine MyController.RunInnerLoop() is called. That routine has a section of code for each runtime

parameter that specifies whether an inner loop is to be run. For each such runtime parameter, the section of code does the following if an inner loop is to be run:

1. Clones the model for use in running inner loop scenarios.
2. Adjusts the cloned model so that it knows that it is an inner loop model, and provides it with a reference to the outer loop model where any results should be stored.
3. Adjusts the runtime parameters in the cloned model to start from the valuation date and to run for the number of months specified for inner loops in the runtime parameters.
4. Calls the Scenario Manager to generate the scenario or scenarios to be run in the inner loop
5. Clears the collection of inner loop results (if not previously cleared for this valuation date)
6. Calls the cloned model's RunInnerLoopModel() method to run all the scenarios in this inner loop. At the end of each scenario, the cloned model's CollectResults() method will be called to capture the desired results and store them in a collection of inner loop model results. The collection is keyed by both scenario ID and an item name for each stored result. Items normally stored include the monthly cash flows and discount rates to be used in valuation.
7. Instructs the Scenario Manager to resume the outer loop scenario.
8. Writes a message to the run log indicating the completion of this inner loop for this date.

After all inner loops have been run, the collection of inner loop model results (which is contained in the outer loop model) has been filled but not yet used. The outer loop controller's ApplyInnerLoopResults() method must retrieve information from that collection, perform any calculations required (such as computing the discounted present value of cash flows) and save the key results by making entries in the outer loop's main ledger. The chart of accounts must include accounts where such entries can be made.

For certain inner loops, all of this is handled in the code auto-generated by the Workbench. This includes inner loops for US statutory reserves under VM-20 (the deterministic and stochastic reserves) and inner loops to carry out valuations under the Representative Scenarios Method (RSM).

The user can define other inner loops. Doing so requires several modifications to a model:

- Adding a runtime parameter to indicate whether to run the specialized inner loop
- Adding a section of code to the controller's RunInnerLoopModel() method to check the runtime parameter and set up the inner loop.
- Invent a name to be included in the scenario ID for each scenario to be used. The scenario ID can be that name with a scenario number appended to the end.
- If necessary, refine the Scenario Manager to generate the scenario(s) to be used.
- Add code to the controller's CollectResults() method to store the results for scenarios whose scenario ID includes the name you invented to identify such scenarios. This will be called at the end of each inner loop scenario.
- Add code to the controller's AppyInnerLoopResults() method to retrieve the stored data from the inner loop scenarios, perform any required calculations, and save final results to the main ledger. This will be called after all inner loop scenarios have completed for a valuation date.

The sample code auto-generated for some inner loops can be used as a template or example for the changes needed to introduce specialized inner loops.

Requesting that the inner loop be included in a model run

The list of Runtime Parameters that define a model run must include direction regarding the inner loop.

Four parameters are included in the list by default when any new model is first set up. Each can have a value of Yes or No. The parameters are:

- Calc det reserve — Calculates the VM-20 deterministic reserve
- Calc stoch reserve — Calculates the VM-20 stochastic reserve CTE70
- Calc RSM reserve — Calculates the RSM reserve using representative scenarios
- Calc RSM stoch reserve — Calculates a multi-risk stochastic reserve

You can act on the value of these runtime parameters using code like this:

```
if (runParms.GetParm("Calc stoch reserve") == "Yes"){}
```

An if-statement like that for each such runtime parameter is normally included in the controller's RunInnerLoopModel() method and in its ApplyInnerLoopResults() method.

Two other runtime parameters specify aspects of the inner loop:

- Num stoch scenarios — The number of stochastic scenarios for stochastic reserves
- Num months in valuation — The time period over which future cash flows are projected

The runtime parameters listed above should be kept in each model even if not in use. If they are removed, the code that refers to them should also be removed.

The user can add inner loops of different kinds to the model by adding additional runtime parameters and writing code in the Controller to act upon them. See the previous subsection for a list of changes that must be made to add a specialized inner loop to a model.

Generating the inner loop scenario(s)

The ScenarioManager included in the Workbench includes a generator that can generate several kinds of inner loop scenarios on the fly, starting from economic conditions on the valuation date.

The economic conditions on the valuation date are easily retrieved from the outer loop scenario. The generator also uses a "mean reversion point parameter" that serves as the long term expected mean of

the interest rate on 20-year US Treasury bonds. By default the Workbench will use 4.00%, but this can be changed by the user. To change it, the user must provide the value to be used and set it in the section of code in MyController.RunInnerLoop() when setting up an inner loop. For example, the user could include an assumption in the model's assumption file and retrieve it there. The following line of code could be used to set the mean reversion point using an assumption with the mnemonic "MeanRevPoint":

```
ScenarioManager.meanReversionPoint =
block.GetAsmpValue(aDate, A.MeanRevPoint);
```

Calling one of the ScenarioManager methods in the table below carries out the scenario generation process starting from current conditions and using the mean reversion point. It also puts the scenario manager in a state where the newly generated scenarios are being used.

Command	Scenario ID(s) generated	Description
SetPBRDeterministicScenario(m)	"_PBRDeterministic"	The deterministic investment scenario defined in VM-20 for principle-based reserves.
SetPBRStochasticScenarios(n,m)	"_PBRStochasticX" where X is the scenario number	The stochastic investment scenarios defined in VM-20 for principle-based reserves. Only the first n of the standard 10,000 scenarios are generated.
SetRSMScenarios(r,m)	"_RSMX" where X is the scenario number	The representative scenarios defined by the risk definition file.
SetRSMStochasticScenarios(r,n,m)	"_RSMStochasticX" where X is the scenario number	Stochastic scenarios that include stochastic paths for all the risks in the risk definition file.
SetValuationScenario()	_valuation	Interest rates on the valuation date continue forever.
SetOCIValuationScenario()	_OCIvaluation	Interest rates from 12 months prior to the valuation date resume and continue forever. This scenario has been used in some proposed accounting frameworks to help quantify the effect of interest rate changes in the past 12 months so that it can be recorded as OCI (Other Comprehensive Income).

Several of the commands require arguments in parentheses. The arguments are:

- m Number of months to include in the scenario
- n Number of scenarios to generate

- r Full path name of the risk definition file

The generated scenarios always include future interest rates. The PBR scenarios also include future fund returns. The RSM scenarios include scenario paths for all the risk variables defined in the risk definition file, including interest rates and fund returns.

When an inner loop is complete, the command ScenarioManager.ResumeScenario() must be issued to reset the ScenarioManager to a state where the outer loop scenario is being used.

Adding margins in the inner loop assumptions

Exactly the same code is used for the projections done in the inner loop and outer loop. Therefore, if one wishes to use different assumptions in the projection of contract cash flows in the inner loop, one needs a way to check whether the code is being executed inside an inner loop or an outer loop.

Typically this is done inside the contract processing. The user must write all of the code used to calculate values used in contract processing. Inside the code for MyContractClass, the following statement will execute the code inside the curly brackets only when in an inner loop.

```
if (!Controller.IsOuterLoopModel){}
```

This kind of if-statement can be used to alter or replace values previously calculated. For example, statements can be placed inside the curly brackets to add a margin to a previously retrieved assumption value.

Scenarios developed for RSM (the Representative Scenarios Method) contain paths over time for risk drivers other than investment returns, in the form of actual-to-tabular ratios. The current actual-to-tabular ratio is retrieved by name from the scenario manager using code like this:

```
myRatio = ScenarioManager.currentEnvironment.GetMiscRate("RiskVariableName");
```

When the inner loop is running scenarios other than RSM scenarios, those risk variables will not appear in the scenario so zero will be returned. It is best to check whether risk variables retrieved from a scenario are zero before using them.

You may wish to check whether the scenario is an RSM scenario or a PBR scenario. This can be done with code like the following:

```
if (ScenarioManager.currentScenarioID.Contains("PBR")) {}

if (ScenarioManager.currentScenarioID.Contains("RSM")) {}
```

Note that all generated RSM scenarios are given scenario IDs that contain “RSM”, even when they are generated for the outer loop. The only scenario IDs that contain “PBR” are those generated for an inner loop valuation.

Capturing and using results of inner loop scenarios

The controller for every outer loop model contains a storage facility for results of inner loop models. This was described earlier as an object of the ResultsDictionaryClass, and it is named “modelResults”. It can store any kind of model result keyed for lookup by scenario ID and item name.

Common items to store from an inner loop include the array of projected monthly cash flows and the array of monthly discount rates for use in calculating the present value of future cash flows.

The controller routine CollectResults(scenarioID) is where results are captured at the end of an inner loop scenario. Typically results are extracted from the main ledger, but they can be anything. A line like the following is used to store a variable named myResult.

```
modelResults.AddResult(scenarioID, "Result name", myResult);
```

The variable myResult can be an array or any other type. When it is retrieved later, it will need to be typecast to the kind of item that it is.

Note that the collection of modelResults is never saved as a file. If you wish to save the ledger created during inner loop runs, there is code at the end of CollectResults() that has been commented out but could be used to save the ledger. Saving the ledger from inner loops is not normally recommended because of the large number of files created when stochastic valuations are being done not just once, but on every future valuation date.

Retrieval and use of inner loop model results occurs when the outer loop controller calls its ApplyInnerLoopResults() method. Normally this method has a section for each runtime parameter that specifies whether an inner loop is run. Such a section would begin like this, with extra statements inside the curly braces:

```
if (runParms.GetParm("Calc stoch reserve") == "Yes"){}
```

The statements inside the curly braces normally retrieve items from modelResults, perform calculations, and them make entries to the main ledger of the outer loop model. The auto-generated code does this for the inner loop valuations specified in the default list of runtime parameters. The auto-generated code can serve as a template for the user to write code for other kinds of inner loop valuations.

Principle-based margins and the Representative Scenarios Method (RSM)

The Representative Scenarios Method (RSM) has been developed as a simple and clear process for adding margins to liability valuations. While a full description of RSM is beyond the scope of this

manual, the general idea is that a sensitivity test scenario is developed for each risk driver, that is, each risk for which a margin should be included in the reserve. The amount of extra reserve required for that risk is based on the present value of cash flows in the sensitivity test scenario. The total margin for all risks is based on the sum of the amounts for each risk, adjusted for correlation.

The sensitivity test scenarios are called “Representative Scenarios” because they are designed not only to represent a specific risk driver, but to do so at a stated level of severity on the distribution of results for that risk driver.

The Workbench generates Representative Scenarios using information entered by the user in a Risk Definition File with the extension *.rsdef. The user interface for creating and editing such files was described earlier in this User Guide.

To use RSM for valuations within a model run, the user must set three Runtime Parameters:

- “Calc RSM reserve” must be set to “Yes”
- “Risk definition file” must be set to the full path name of the *.rsdef file to be used
- “Num months in valuation” must be set to the number of months of cash flows to project for valuation purposes.

When this is done, each outer loop scenario will include inner loop scenarios to calculate the RSM valuation at each year-end. An output file will be written for each valuation date to show the results of all the inner loop scenarios. The file name will be the same as the main ledger output file name, but with “_RSMYYYYMM” appended and the filename extension changed to .csv. The YYYYMM part of the filename corresponds to the valuation date, with YYYY being the year number and MM being the month number.

The .csv file that is saved for each date contains information like the excerpt shown below. The columns show:

- The name of each risk driver
- The probability weight for the scenarios (add to 1.00 within each risk driver + anticipated)
- The pattern of experience variation (pop or creep)
- The scenario probability level in the cumulative distribution for the risk driver
- The scenario present value of cash flows

Beneath those columns, four items calculated based upon the scenario present values are shown. These are:

- The central estimate of the liability
- The reserve margin
- The reserve (central estimate plus margin)
- An estimated capital requirement

These items are calculated using source code in the file RSMCalculator.cs. The user can adapt the source code if changes are desired in the way any of these items are calculated based on the scenario present values.

While not shown in the excerpt below, the .csv file also contains the month-by-month cash flows and discount rates from the anticipated experience scenario.

RSM data for 202412				
DefaultCost	0.023	pop	0.999	20,995,825
DefaultCost	0.286	pop	0.841	20,310,980
DefaultCost	0.286	pop	0.159	19,840,307
DefaultCost	0.023	pop	0.001	19,815,209
Expense	0.023	pop	0.999	20,233,858
Expense	0.286	pop	0.841	20,045,390
Expense	0.286	pop	0.159	19,920,581
Expense	0.023	pop	0.001	19,734,162
None	0.382	Anticipated	0.5	19,982,982
Interest	0.023	pop	0.999	19,309,152
Interest	0.286	pop	0.841	19,791,301
Interest	0.286	pop	0.159	20,156,015
Interest	0.023	pop	0.001	20,430,221
Lapse	0.023	pop	0.999	18,524,971
Lapse	0.286	pop	0.841	19,447,067
Lapse	0.286	pop	0.159	20,526,251
Lapse	0.023	pop	0.001	21,444,910
Mortality	0.023	pop	0.999	24,848,919
Mortality	0.286	pop	0.841	21,251,724
Mortality	0.286	pop	0.159	18,716,210
Mortality	0.023	pop	0.001	15,106,580
MortImprovement	0.023	pop	0.999	18,750,666
MortImprovement	0.286	pop	0.841	19,240,699
MortImprovement	0.286	pop	0.159	21,240,861
MortImprovement	0.023	pop	0.001	25,160,549
Central estimate	20,056,592			
Margin	1,890,989			
Reserve	21,947,581			
Capital	7,210,921			

The source code generator

One of the most powerful and valuable parts of the Workbench is the source code generator. This provides the ability to automatically write most of the computer code for a model based upon the product definition provided by the user.

The source code generator is accessed using the main menu option “Model Assembly” and submenu option “Generate code”. This brings up a dialog allowing you to select which part(s) of the model code to generate:

When you click on “Generate selected modules”, the *.cs files for the selected modules will be generated. While doing so, any code that the user has previously entered for some routines will be preserved, while other routines will be completely re-written. If the user has previously modified the code in the routines that get re-written, the modifications will not be carried over to the newly generated code. Therefore it is vital for the user to be aware of any changes made to routines that get re-written and decide whether to avoid using the code generator or to move those changes manually to any re-generated code.

Here is a summary of the files that the code generator can write, including a list of routines in each file. All of those routines are generated each time the code generator is run. In many cases when a previous version of a file exists, the previous version of the routine is carried over because it may contain user modifications. In some cases, however, the routine is always re-written and previous changes are discarded (because generally there should be no reason for the user to make any). The last column in the table below indicates whether a routine is carried over or re-written when a previous version exists.

Program entry point
File name: *ProductName*Model.cs
Description: This is a standard program entry point module.

Routine	**Description**	**Previous code re-written or carried over**
Main()	The main Windows program entry point	Re-written

Model controller
File name: *ProductName*ControllerClass.cs
Description: This code is for a customized model controller for this particular model based on the product definition. A C# class is defined by inheriting from ModelControllerClass. This is done by overriding some of the methods in ModelControllerClass.cs and adding some utility calculation methods.

Routine	**Description**	**Previous code re-written or carried over**
Constructor()	Default version just calls the base class constructor	Carried over

RunModel()	Sets flags to run any inner loops and calls base class RunModel()	Carried over
SetupObjects()	Sets up any objects that are not scenario-specific. Typically this just calls the base class SetupObjects()	Carried over
SetupScenario()	Sets up any objects that need to be re-set at the start of each scenario. This includes the inforce file and sales mix file.	Carried over
PeriodEndProcessing()	If there is an inner loop and this is December, calls ApplyInnerLoopResults(). Allocates investment results from the investment ledger to the main ledger. Calculates the monthly investment yield and saves it to the main ledger. Updates the surplus account.	Carried over
RunScenario()	Calls the base class version of RunScenario()	Carried over
CollectResults()	Called at the end of each inner loop scenario. Captures the vector of cash flows, discount rates, and number of contracts and stores them in *modelResults*.	Carried over
ApplyInnerLoopResults()	Called from PeriodEndProcessing in the outer loop in periods when inner loops are run (e.g. each December). Retrieves results from *modelResults* , performs calculations, and puts final results in the main ledger. Calculations include the present value of cash flows.	Carried over
PVCF()	Calculates the present value of cash flows when given a vector of cash flows and a vector of discount rates.	Carried over
UpdateInforceStatus()	Called before PerformValuation() during processing each month. Can be used to update global things like an interest crediting rate used for an entire block of contracts. By default, this routine is empty.	Carried over

<table>
<tr><td>GetInforceFile()</td><td>Creates an object to represent the inforce block and reads in the file containing the starting list of contracts.</td><td>Carried over</td></tr>
<tr><td>GetSalesMixFile()</td><td>Creates an object to represent the sales mix and reads in the file containing the assumed sales mix.</td><td>Carried over</td></tr>
<tr><td>ModelName()</td><td>Returns the name of this model</td><td>Re-written</td></tr>
<tr><td>CloneForInnerLoop()</td><td>Clones the model</td><td>Re-written</td></tr>
<tr><td colspan="3">Inforce block
File name: ProductNameInforceClass.cs
Description: This code is for a customized inforce block of business for this particular product based on the product definition. A C# class is defined by inheriting from InforcBlockClass.</td></tr>
<tr><td>Routine</td><td>Description</td><td>Previous code re-written or carried over</td></tr>
<tr><td>Initialize()</td><td>Sets the range of main ledger segments used by this inforce block by setting the values of variables named FirstLgrSeg and NumLgrSegs.</td><td>Carried over</td></tr>
<tr><td>AddNewSales()</td><td>Retrieves the amount of sales from the assumptions, creates new contract records and adds them to the list of contracts in force. The user may need to change the name of the sales assumption used to retrieve sales.</td><td>Carried over</td></tr>
<tr><td>PerformValuation()</td><td>Loops through all contracts and retrieves and records the valuation amounts defined by the user in the contract definition file. Takes care of recording both the liability amounts and “change in liability” amounts. Also removes any expired contracts from the inforce block.</td><td>Re-written

Any changes made by the user must be manually carried over when code is re-generated.

This routine does not perform valuations that require an inner loop – that is handled by the controller.</td></tr>
<tr><td>ProcessMonth()</td><td>Loops through all contracts calling each contract’s ProcessMonth() method and returning the total cash flow. This can normally be done just by calling the ProcessMonth()</td><td>Carried over</td></tr>
</table>

	method of the base InforceBlockClass.	
Clone()	Creates a cloned copy of this inforce block. This includes cloning all of the contracts currently included.	Carried over
CreateTable()	Creates a database table containing the list of contracts. A table is used when editing the list of contracts.	Re-written
PutDataInTable()	Puts the list of contracts into a database table	Re-written
GetDataFromTable()	Retrieves the list of contracts from a table.	Re-written

Contract
File name: *ProductName*ContractClass.cs
Description: This code is for a customized financial contract under which amounts will be paid to the contract owner. The contract was defined in a contract definition file. A C# class for this contract is defined by inheriting from ContractClass.

Routine	**Description**	**Previous code re-written or carried over**
Declarations	Declarations of all defined: Master record data fields Calculated variables	Re-written
Constructor		Re-written
SetLgrSeg()	Sets the value of LgrSeg, the ledger segment number where this contract is to be recorded.	Carried over
SalesMeasure()	Returns the per-contract amount of sales. Any assumption for the amount of sales is divided by this to get the number of contracts sold. The default value is 1.0.	Carried over
calcXXX() where XXX is a variable name	Calculation routines for each calculated variable XXX. By default these are blank and code must be entered by the user. When these routines are executed, all master record fields hold their values as of the beginning of the month.	Carried over
processXXX() where XXX is a transaction name	Transaction processing routines for each transaction XXX. The processing involves ledger entries and master record	Re-written

	updates, both of which are defined in the contract definition file.	
processContractChanges()	This routine is a placeholder that is called after all regular transactions are processed for a month. It allows the user to insert special processing for things like annuitization or term conversion. The default does nothing.	Carried over
processMonth()	This is the method that simulates all transactions for the month. It calls all of the calcXXX() methods, writes a record to the audit file if requested, records all transactions in the ledger, updates the master record, and performs any contract changes at the end of the month, all by calling other methods listed here. This method returns the amount of cash flow generated.	Re-written
performValuation()	This method calculates all the formulaic valuation amounts defined in the product definition and makes a credit entry to the ledger account to which each is assigned.	Re-written
WriteAuditRecord()	When audit output is requested in the runtime parameters, this routine creates a line of data for a .csv file containing the date, the contract master record and all calculated variables for the month.	Re-written
WriteToFile()	Writes this contract to an XML file	Re-written
ReadFromFile()	Reads a contract from an XML file	Re-written
Clone()	Makes a cloned copy of this contract	Re-written
Other items generated along with Contract	**Description**	**Previous code re-written or carried over**
Class factory code	A class factory for creating a contract in computer memory.	Re-written
Partial inforce block class	Portions of the	Re-written

	*ProductName*InforceClass where the code depends on internal details of the contracts. These routines enable reading and writing the inforce block using .csv files. The names of master record fields and calculated variables are used for column headings in the .csv file.	
SalesMixClass This code is put in a separate file named *ProductName*SalesClass.cs	A class that handles the sales mix file.	All is re-written except for the routine MakeInforceCell(), which is carried over. That routine is used to create a new inforce record from a sales mix record given a date and an amount of sales.

Statutory reserves and BFReserveLib.dll

The Workbench does not provide complete calculations of statutory reserves for any kind of insurance contract. However, it provides some utilities that you can use or adapt as needed. ***Please understand that these libraries are provided as is and are not warranted to be correct or to satisfy regulatory requirements. They should be considered only a place to start.***

These utilities are contained in BFReserveLib.dll which defines three utility classes:

- CommutationFunctions
- ContinuanceFunctions
- AG38Calculator

The class CommutationFunctions helps to calculate tabular reserve factors for fixed-premium life insurance and annuity contracts with reserves defined by a mortality table and an interest rate. To use it, first create an instance of the class:

```
CommutationFunctions CommFunc = new CommutationFunctions();
```

Then specify the mortality table number and issue age. Note that the issue age only matters when the mortality table is a select and ultimate table.

```
CommFunc.TableNum = 1234;
```

```
CommFunc.IssueAge = 35;
```

Optionally, you can also specify table numbers for lapse rates and X-Factors for use in calculating the statutory reserve factors. If you don't want to use those, set their table numbers to zero (which is the default).

```
CommFunc.LapseTableNum = 1235;

CommFunc.XFactorTableNum = 1236;
```

After you have specified those inputs, the following values are available.

CommFunc.Dx[age]

CommFunc.Cx[age]

CommFunc.Nx[age]

CommFunc.Mx[age]

CommFunc.Ax[age] Life insurance single premium factor

CommFunc.Px[issueAge] Net annual premium factor

CommFunc.CRVM_Px[issueAge] Net annual premium factor for issueAge + 1

CommFunc.NLP_Reserve[issueAge, policyYear] Terminal NLP reserve

CommFunc.CRVM_TerminalReserve[issueAge, policyYear] Terminal CRVM reserve

CommFunc.CRVM_MonthlyReserve[issueAge, policyYear, policyMonth] Monthly CRVM reserve

The policyYear argument is numbered starting with the first policy year as 1. Policy year zero corresponds to the moment before the first premium is paid.

The policyMonth argument is numbered from 1 through 12. Policy anniversaries are assumed to occur at the end of the month, so policy month 12 means the policy anniversary at the end of the policy year, when the monthly reserve is the same as the terminal reserve.

The code provided in the library calculates these factors on a semi-continuous basis. That means that premiums are treated as paid annually and death benefits are treated as payable at the moment of death (rather than the end of the policy year of death). The annual premium assumption is consistent with US life insurance statutory accounting where premiums paid more frequently than annually are treated as being due on an annual basis, with any portion yet to be paid later in the year recorded as a receivable labeled "premiums due and accrued".

The class ContinuanceFunctions provides values defined by a continuance table and an interest rate. This can be useful in computing claim reserves for disability insurance or long term care insurance.

To create an instance of this class you must call the constructor with arguments that specify the table number of the continuance table and the interest rate to use for reserve calculations. For example:

```
ContinuanceFactors ContFac = new ContinuanceFactors(1234, 0.04)
```

The specified table must be a continuance table of course.

Once constructed, the object provides the following methods:

- ContFac.RecoveryRate(incurralAge, durMonths);

Returns the fraction of claims that terminate (recover from disability) at the requested monthly duration after claim.

- ContFac.ClaimRes(incurralAge, waitMonths, durMonths, benMonths);

Returns the month-end claim reserve per dollar of monthly benefits. The claim has a waiting period of “waitMonths” and a benefit period of “benMonths”. The benefit period begins after the waiting period and does not include it.

- double[] ClaimResVector(incurralAge, waitMonths, durMonths, benMonths);

Returns a vector of month-end claim reserve factors with a length equal to waitMonths + benMonths + 1. The last factor in the vector is for the month after the benefit period and is equal to zero.

The class AG38Calcculator carries out calculations according to Actuarial Guideline 38 for certain kinds of universal life insurance contracts. To use it, first create an instance of the class:

```
AG38Calculator AG38 = new AG38Calculator ();
```

Then specify the required inputs. Here is the list, with illustrative values shown:

```
AG38.QxTableNum = 1234;            Mortality table

AG38.MinPremTableNum = 1235;       Minimum premium rate table

AG38.LapseTableNum = 1236;         Lapse rate table

AG38.XFactorTableNum = 1237;       X-factor table

AG38.IssueAge = 35;

AG38.IntRate = 0.04;
```

After the inputs have been specified, call the calculate() method:

```
AG38.calculate();
```

The following vectors of values are then available. The vectors contain one value per policy year. The following are per dollar of face amount (not per $1000).

```
double[] AG38.basicReserve;
double[] AG38.basicNetPremium;
double[] AG38.deficiencyReserve;
double[] AG38.netSinglePremium;
```

The following vectors summarize the segmentation:

`int[] AG38.segNums;`	Each policy year's segment number
`double[] AG38.premRatios;`	Ratio of prem rate to first yr of segment