Plan to Throw one Away: Lessons Learned Developing an Economic Simulator

Michael Battig
mbattig@smcvt.edu
Computer Science & Information Systems
Saint Michael's College
Colchester, VT 05439, USA

Michael Commo
mcommo@us.ibm.com
Information Technology & Computer Science
Essex Junction, VT 05452, USA

Abstract

This paper presents our experiences designing and building an economic simulator for use in a Public Finance course. The project was a joint venture between the Information Systems & Computer Science Department and the Economics Department. Faculty and Students collaborated to create the simulation system. The second version of the software is currently being used in our Economics courses and a third major version is in the early design phases. Plans are underway to roll this software product out to other institutions for use in Economics classes. The current version was created with Visual Basic and contains some performance issues. The next version will resolve some database performance issues and employ an expert system in order to ease the maintenance burden. We also discuss the software hurdles encountered during the development process.

Keywords: information systems, software development, databases, educational software, simulation, expert systems, public finance, economics

1. INTRODUCTION

For many years we in the computing fields have created artificial software environments to mimic the designs of computing systems and their internal architecture. In recent years, however, we have begun to view our systems as having parallel architectures in other domains of science and nature (Denning, 2007). Here we present our experiences of the past several years in building an economic simulation software environment as a joint collaboration between our Economics and Computing departments. In creating this simulated environment, we have sought to evolve our software toward a model that more closely emulates the world of economics than it does the world of computer hardware and software.

For the past several years our Economists have been providing detailed requirements for a custom-built economic simulation system. This software serves several purposes. It provides a platform for student learning and student projects in the Economics Department’s Public Finance class. The software, and its use in class, has been the subject of ongoing research in our Economics Department. In parallel, our

Information Systems faculty members have been searching for ideal projects to not only engage students in the development effort, but to employ novel software solutions, such as integrating expert systems. In this quest for a truly collaborative project, the system is emerging through three distinct versions.

The first version of our simulator was not a custom software solution, but rather a sophisticated spreadsheet that contained the formulas and requirements that would later serve as the foundation and requirements for our software. As a spreadsheet, the system did not work well as a learning platform for students in Public Finance. The current version of our simulator is a Visual Basic program that uses an interface that allows students to focus on parameters of their governmental and financial priorities without getting lost in a sea of economic computations. The current version also uses an Access database to store data related to the user's public finance model. We envision that the future version of our simulator will employ an expert system shell to simplify future maintenance by our experts (in this case our economics professors) and to deal with many performance challenges that our system is currently experiencing.

To give the reader some perspective on the simulator, we will present some data and software metrics. The Visual Basic source code is contained in 17 modules with a total of 5093 non-blank and non-commentary lines of code (8155 lines including blanks and comments). The database consists of five tables: Population, Globals, Education, Options, and Social Security. The most critical table from a performance perspective is the Population table, which contains 729 cohorts, which represents just over 45 million people (see Figure 1).

We should note at this point that our current system has received a lot of interest from economics professors when presented at a relevant conference (Walsh, 2010). Our long term plan is to make this system available to others for use in teaching Public Finance (perhaps even bundled with textbooks). Despite the success of the initial version of the simulation software, this version exhibits major limitations. First, it is quite slow, with several-minute runtimes for many of the steps. Secondly, it is difficult to customize or tweak without major overhauls. Finally, it only simulates the present economy, whereas most of the policy choices made by students have strong implications for the future. A more complete model would be able to simulate the effects of these choices decades into the future. In the following sections we will provide the details of our current system's requirements, look at some of the obstacles faced, and outline our future enhancements for the next version.

2. SYSTEM REQUIREMENTS

The Economics Simulator Application represents the formulaic results of our Economics Department's research in economic modeling programmatically, and attempts to recreate the intricacies found inherent in a complex economy. The ultimate objective of this modeling tool was to have students enrolled in the Public Finance course model their idealistic economy, and then fund it using taxation, among other means. This requirement was, from the developer's perspective, a late change in requirements. The application itself was an early lesson in requirements gathering, making and adhering to design decisions; as well as implementing and testing the design. The application and simulation is essentially an interface from which the user can customize an economy. The economy is represented by a database, and the application processes data in the database based upon user selections and the aforementioned economic modeling formulas from the economics research. The simulation program consists of extraneous features that enable the program to graphically represent results, and also permits assignment submission wirelessly over the internet. Our system, however, is not without defects. We have bottlenecks related to the database interaction, and even logic errors inherent to the functional definition of the underlying mathematical equations that drive the simulation. Looking beyond the defects, there is the potential to improve and expand upon the simulation using (database) stored procedures and triggers, loop unrolling, and even Artificial Intelligence.

This pilot system, however, had a predecessor. The underlying mathematical functions existed in a previous Excel-based system from which they were derived and defined for this system. From this perspective, this project involved quite a bit of software reverse-engineering and re-engineering. The system had previously been implemented on an entirely different platform. Like many projects, we lacked complete requirements, be they functional, performance, security, or otherwise. What we did have was design documentation describing in detail functional definitions and their effects on cohorts in an economy. For brevity, we will refer to a cohort as a group of people having the same age, health, and ability factors. Conveniently, this also translates into one record in the Population table of the economics database (EconDB). For example, one of nine cohorts in the Population table consists of individuals all having an age of 43, *low* health and *medium* ability. These functional designs drive the simulation in that they directly affect the people (cohorts) within the economy, and the results of the economic simulation are generated by querying the database after these functions have been applied to the database. These designs, however, say little about the actual functional requirements of the simulation, and leave much to be inferred by the programmer.

Development History

The software engineering aspect of this project focused primarily on the process of turning these functional definitions into working software. Agile software development methodologies implicitly came into play throughout the project, as the primary focus of the project became producing working software prototypes in order to test the functional interactions with the database, and then determine the resulting economy based on factors derived from the modified database. The process was incremental and iterative in that new functions and groups of functions were added to the system, a prototype was built and tested, and then the process repeated itself. Those working on the project came to value working software, and the ability to respond to change, which just happen to be two primary values from the Agile Manifesto (Fowler 2001).

The prototypes served as a means of testing our implementations of the functional design against the underlying mathematical equations that drive the simulation. When the resulting economy was not as expected, the functional definitions were scrutinized, (as was their implementation in software), and the system was modified or changed in some way to produce the expected results. As with many software engineering projects, requirements had to be cut and many of the initial requirements have yet to be featured in the working simulation model. For instance, people are currently unable to move between cohorts, mainly due to another postponed requirement, that the system will "age" and time will progress. All functions interact on a stagnant database of cohorts, which means that the system is currently incapable of modeling the effects of minute and incremental changes as they propagate with time through generations. An example of this limitation is the inability to model the correlation (if any) between a generous Social Security system and the percentage of low income, or impoverished, cohorts. Despite missing requirements and functionality, the system, in its current state, is described in detail below.

The system's features and its limitations are the result of the platform, database management system, individual modules/components and their interactions, and user interface design. The system was programmed in Visual Basic .NET, and thus, the graphical user interface was designed in Visual Studio 2008 Professional using built-in CASE tools. This greatly expedited user interface design and implementation, but ultimately limited the ability to represent results graphically. A class was designed to remedy this deficiency, and will be discussed later. Microsoft Access was chosen as the DBMS, primarily for the reason that its runtime is easily and freely redistributable, and Access is compatible with 32-bit Windows machines, (which was the target platform). With the platform, language, IDE, and DBMS determined, the interface design, database connections, and the database population table became the primary focus.

Prerequisite to modeling economic scenarios and their interactions, a database had to be built representing the sample population with which the underlying economic functions interact. This in itself necessitated two additional requirements: a user interface, and a means of connecting to the database. The Population table in the database was painstakingly colonized using hand-crafted data from the previous Excel-based system. After soliciting user interface requirements, it became apparent that the system would consist of many sub-sections, each relating to different parts of the economy (including healthcare, education, taxes, etc.), and that the interface should be designed in a similarly modular fashion. Each subsection was encapsulated in its own window, and windows could be switched using menu buttons at the top of the parent frame. As a result of this interface design, each section had to be customized and run independently of all other sections. Therefore the interactions between these disparate sections is only apparent once the "Run Simulation" button is selected. This modular interface design also allows for the addition and deletion of categories should system maintenance be necessary. As stated previously, the interface's primary function is to act on the database, thus a custom class had to be written to manage access to the database(s). See the sample interface in Figure 2 of the Appendix.

The ConnectionAdapters.vb class was written as a means of managing the connection to the database, and abstracting away the details of the requests the program makes to the database. Because many of the functional definitions from the design documentation involved updating attributes for every cohort in the database based on one or more other attributes, thousands of connection objects had to be created and managed by the VB.NET software, thousands of queries (mainly SELECT and INSERT statements) had to be run by the DBMS, and the result sets for those queries had to be returned to the program. This custom class provides the following public methods for use by the programmer: DBupdate(), DBselect(), DBinsert(), and DBdelete(). The class also provides private methods used by the system for managing the database connection, as well as for encryption/decryption. Utilizing these private methods, the class manages the connection string, the state of the database, and connections to the database, and the Object Linking and Embedding (OLEDB) API calls required to execute SQL statements on the database. Unfortunately, this heavy reliance on linking objects to execute SQL statements on the cohorts in the database leads to performance issues that will be discussed later.

With the infrastructure in place to interface with and query a database representing the economy's population, the individual sections of the simulation were constructed to allow users to customize the economy in very specific ways. The earliest working prototypes of the simulation allowed for these individual modules to update the production database as they were customized. This had a significant consequence – that the order in which the modules were run mattered. There was a secondary consequence in that anomalies, (usually resulting from user input and its effect on the simulation), could be greatly exaggerated by the repeated execution of modules that caused anomalous results. For this reason the current economics simulation program includes a "Run Simulation" button on the menu located next to the individual sub-sections/modules. The modules affect a test database, which is restored each time a module is run. This ensures modules can be run only once and any anomalous impact caused by the order in which the individual modules are executed on the database is mitigated by the fact that the "Run Simulation" button executes all the individual modules in the correct order on the production database. The earliest versions of this program included all the modules together as a single simulation run. As alluded to earlier, a change in requirements split the simulation into two phases. The economic sub-sections/modules that were bound to the first phase include pollution, externalities, social insurance (welfare), social security, healthcare, and education – essentially all topics that affect the well-being of the economy. Users working on phase one can tailor their idealistic economy by customizing each of these modules. But once the simulation is run, and the results of phase one have been submitted for grading, users are bound to their economic decisions from phase one when they move onto the second phase.

Economic Requirements

The first phase of the simulation is funded by what our senior economist calls "magic income." It is magic in that the simulation generates enough income via taxation to fund all of the modules customized in phase one (and balance the budget) no matter how expensive. Phase two allows users to access and customize income tax, corporate income tax, and consumption tax systems with the objective of funding their ideal economy and balancing the budget manually using taxation. For example, as part of the income tax system, the user must specify the income tax brackets and their corresponding marginal tax rates, as seen in Figure 3. These marginal rates are then factored into a function which computes and collects income tax from every individual, in every cohort, in the database. Corporate income tax has similar options and objectives. Regarding consumption tax, the user can specify a uniform tax on all consumer goods, or determine an individual tax rate for each item category based on that category of good, and the category's elasticity. As stated previously, a paramount functional requirement of this program was that the user could customize an economy as they see fit, and then generate the income to fund the economy. That requirement is fulfilled by this economic simulation system. However, there are many other interesting classes and methods that were designed and written in order to help fulfill this and other requirements of the system.

Another requirement of the system was that the students be able to submit their assignments to the course instructor electronically from the program. After further requirements gathering, it was determined that the submission needed only to consist of the inputs and outputs of the simulation. Since each sub-category/module (healthcare, education, taxes, etc.) has one or more input forms and one or more output forms, we simply delegated a file name associated with each form/screen, and any time one or more modules are run, (and input/output forms are filled/displayed), their respective files are updated with the most recent input and output data. This was initially implemented using the StreamReader and StreamWriter classes built into VB.NET System.IO. However, any change to the form necessitates changing the code which writes the user input and simulation outputs to the files, therefore, an alternative design was created: the program would make image files of its input forms and output windows. Using the System.Drawing.Imaging classs, along with gdi32.dll (the graphical device interface library), the program essentially takes screenshots of itself by passing its current screen coordinates, and an output file name and location, as parameters into the Imaging classes. The result is a JPEG image containing the currently active screen of the program. Thus, part of the processing for each module of the simulation is to take screenshots of input and output forms using these API calls. At the end of the simulation, the user has a directory containing the input and output screenshots representing their work and this folder of image files is what must be submitted for grading.

Thus we have two new challenges to the project: compressing and combining the image files, and transmitting the resulting zipped folder to the professor. Microsoft provides another API that allows programmers to compress and store directories by utilizing the Shell32.dll library file. After creating an instance of the Shell32.Shell object, the program makes a system call to create the zip file, another add the directory containing the screenshots to the zipped file, and finally a third to save the file to disk. At this point the program has all the information regarding the user's inputs and resulting economy in a single zipped folder (file). The final piece of this requirement necessitates sending the zipped folder to the instructor for review and grading. This would only require one system call (assuming the user had Microsoft Outlook installed). This is a fairly valid assumption for computers on our campus, however, that is not a valid assumption to make in all instances. Therefore the program was designed to send emails standalone. By importing the System.Net.Mail classes, the system creates a new mailMessage() object, and passes all the parameters (including the location of the zipped folder which will be an attachment) needed in order to send the message to the instructor. Before sending the message, other calls allow the program to specify a SMTP mail server (host) name and port number to enable SSL and input user credentials for connecting to the mail server. When a user chooses to submit their results, the screenshot folder is zipped and attached to an email and the program securely connects to Gmail's SMTP server and sends an email with the user's results attached to the instructor's inbox. This fulfills one of our most important functional requirements of any system used in higher education: allow the instructor to easily view and grade student results.

What is the value of image screenshots if the resulting data was merely textual? We alluded earlier that the Visual Studio CASE toolkit for GUI development is somewhat restrictive. For instance, it provides no means for creating graphs or charts – two great methods of succinctly conveying large amounts of information. In the economics simulation, there are results that are numerically very difficult to convey. For instance, the change in welfare benefit with respect to increasing income is eloquently presented as a graph (see Figure 4) that is difficult to present through other means. The solution: build support for making graphs and charts into the economics simulation. In order to achieve this, a graphics (paint) object was instantiated from the System.Drawing classes and literally used to draw the graph within a defined output area on the form. The software had to keep track of the minimum and maximum values on both axes (and print them on the output screen). Because we specified coordinates to the graphing software assuming the origin at the bottom left corner of the graph, the software had to implicitly translate the coordinates to pixel maps which are drawn from the top left corner. This drawing software became useful in depicting results from many of the sub-sections, and this code was later adapted to draw bar graphs as well (see Figure 4).

3. OBSTACLES

As with any pilot system, this project was not without hurdles to overcome. The program exhibited its fair share of programming logic errors. Perhaps even more insidious were logic errors that resulted from unexpected interactions between the underlying mathematical formulas and equations. Recall that the economic simulation functions run independently and they interact by means of shared resources (the cohorts in the database). Also note that in VB.NET, division by zero will cause the Math class to throw an exception and this exception will terminate the program if it is not handled. What we present next is an inherent deficiency of the model that occurs in many of the functional design definitions. As a simple example, define the interest rate as follows:

$$Interest\ Rate = 0.03 + \frac{300}{Average\ Savings}$$

Assume that average savings is a result that can be derived from the cohorts in the database and note what occurs when the average savings is \$1000, \$500, \$250, \$0… It is obvious that this model is not adept to handle Average Savings amounts less than 1000 dollars, as it will drastically inflate the resulting interest rate (or division by \$0 will cause the program to terminate). When the Average Savings is greater than \$0 and less than \$1000, error propagation is a significant concern. The interest rate resulting from this function is used as input into the following functions: EconomyWideProductivity, AfterTaxRateOfReturn, SocialSecurityTrustFundBalance, and TaxableIncome, just to name a few.

The above example shows that an anomalous interest rate (which is derived data) could essentially invalidate the resulting economy due to calculation propagation into other simulation functions. The conclusion of this example is that the functional design was not sufficiently detailed and the model could be broken. Poorly solicited requirements and bad functional design meant that every method which uses derived or user-inputted data had to retrospectively be evaluated for problems similar to the interest rate deficiency described above. The end result of this evaluation was restriction. For example, derived average savings amount was given a minimum value of 1000 to mitigate the problem described above. Even user input had to be restricted by use of combo boxes, sliders, maximum and minimum values applied to numeric text boxes, and by other means. The end result was a more robust simulation, at the expense of some challenge for the user. The severe restrictions on input data intrinsically act as a guide for the user making it easier to model their economy and complete the simulation assignment satisfactorily. Because bad input and derived data can effectively invalidate simulation results, restricting inputs and results was the only alternative. The functional design necessitated these restrictions, but this was not the only area in which the simulation suffered.

Database Issues

The database became a bottleneck within the simulation, but not for the reasons one might initially suspect. Not one of the thousands of queries run by the simulation requires a single table join, since the entire population deliberately resides in a single table. Nevertheless, functions that rely heavily on retrieving statistics about the population or updating the population statistics take a long time to execute. To describe the reasoning for this latency, we return to the previous example regarding interest rate calculation. Interest rate is a function written in VB.NET that is called after any function that updates the savings amounts in the database. The interest rate must retrieve and sum every savings amount from the Population table, average the sum, and then store the result in a separate table. This interest rate function only requires two database queries, one to return the subset of the Population table containing the savings amounts, and another to write the resulting average to a different table (which was designed to track of certain derived data, such as the interest rate). This could be more efficiently implemented as a stored procedure that returns the program the interest rate; or as a trigger that updates the interest rate attribute when any savings amount changes. But since the function only makes two database accesses, it is not terribly inefficient. So why then does the program take so long to run certain calculations? And why was the interest rate function a good example, if it's not that inefficient? The answer is scale. Most of the functions are not computing an average that is stored globally, they are updating attributes on a per-cohort basis, which means updating every row in the population table. This still only takes one query to retrieve (SELECT) a subset of the population table that includes all elements/attributes required for the calculations. But it now takes N update queries, where N is the number of rows in the population table for the function to update an attribute for every cohort in the database.

Some of the many functions/calculations that behave in this way are: calculateWorkHoursPerYear, calculateYearlyIncome, and calculateIncomeTax, just to name a few of the functions used in the Taxes section of the simulation. Now consider a Population table with N=1000 cohorts (which is fairly representative of the simulation's actual Population table size). These functions must execute one DBselect() call and 1000 DBupdate() calls from the connectionAdapters.vb class described earlier. *Note that executing 1001 SQL statements from a VB.NET program is not the same as executing 1001 SQL statements directly from within a DBMS*. As stated previously, the database functions allow the programmer to query the database, with the extra work being handled implicitly. This means 1001 SQL statements in VB equates to creating 1001 OleDbDataAdapter objects, each of which must create a connection to the database, and run the query that was passed into them as a parameter.

What this example depicts is the vast amount of work required to update the database for each function that acts on an attribute for every cohort in the database. Now scale this even further; consider the fact that the taxes module simulation contains 10+ functions, many of which act on one or more attributes for every cohort. Now it's apparent that the taxes simulation is allocating memory for tens of thousands of objects, and each of these objects will connect to and run a query on the database, and some of these queries will return result sets that the objects must handle. The defect we are depicting with these examples is the vast amount of overhead required to accomplish some of the functions that act on the database. As stated previously, many of these functions don't need to be in software, and would be more efficiently implemented as stored procedures or triggers. However, many of the functions do rely on the VB.NET software for calculations, and they cannot be implemented as part of the database. Functions that meet this definition are the true bottleneck routines of the simulation.

4. ENHANCEMENTS

One of the most obvious ways to make our functions more efficient is by loop unrolling. If the program is acting on an attribute for each cohort in the database, loop unrolling by a factor of four (to act on four cohorts at a time) would dramatically improve performance by turning 1000 DBupdate() calls into 250 calls. The drawback to this is that it makes code less readable (which is important in the agile software development methodology), and also makes the SQL update statements more complex as they must update attributes of four cohorts at the same time. There is also extra processing that must be written to account for the situation where the number of rows (cohorts) is not a multiple of four. Loop unrolling could drastically improve the simulation performance by reducing the number of OLEDB objects created, the number of connections established to the database, and the number of SQL statements run. It does, however, require substantial modification to every function and method that falls into this category. This solution does promise to improve performance in future iterations of the economics simulation program.

Performance can be improved by making significant changes to the database, and to the software with which it interacts; but there are other means that also promise great performance improvements for our economics simulator. The field of Artificial Intelligence can improve the performance and robustness of the economics simulation. By implementing bottleneck routines using an Expert System we could not only drastically reduce the amount of processing that needs to be completed, but we could also increase the robustness of the software by introducing fuzzy logic into the system. An expert system implementation of a bottleneck routine would take the population table entirely out of play, and the result would be inferred via rules firing, rather than by scrutinizing the database post simulation. Remember we just determined that these routines could also be improved using loop unrolling, so this is an alternative implementation of a routine, and not a supplement. This is a very appealing alternate approach in that it introduces the idea of fuzzy logic. Fuzzy logic has the potential to fix many of the defects inherent to the mathematical definitions of the functions, (like in the interest rate example above). Rather than defining the function formulaically as it is now, we could describe it using heuristics and confidence factors. For example, the Interest Rate calculation could be re-written using expert system rules (the results are shown in Table 1, Appendix).

This simple rule-based system requires no expert system shell, proprietary inference engine, or special processing to handle fuzzy logic (since it contains no fuzzy logic). This system could be programmed very efficiently as a series of conditional statements in nearly any modern programming language. So why wasn't this used in place of the flawed Interest Rate calculation described above? The simple explanation is that even though these rules appear to imitate the intended purpose of the interest rate calculation, they were not developed by experts. While the interest rate example is simple enough that a translation to a rule based system is trivial, this is not the case for the vast majority of the functions utilized by the economics simulation. Therefore, to make a rule-based expert system implementation of certain functions a feasible approach, a knowledge engineer would have to solicit heuristics from domain experts to try to determine what overarching rules govern the functions being re-engineered.

As mentioned earlier, a major benefit to using expert systems is the ability to introduce fuzzy logic into the simulation. Recall that the result of the interest rate calculation is utilized by a number of other functions. Many of these functions don't require the specific interest rate, and just need to know if the rate is relatively "low", "normal", or "high". Therefore, fuzzy set theory could significantly reduce the complexity of the functional definitions, and in doing so, reduce the chance for the logic errors described earlier. Reducing the chance for logic errors due to derived or input data will in turn reduce the heavy restrictions currently placed on the user and on the simulation, making the simulation more challenging and realistic. All of these benefits could be realized with help from fuzzy logic and the use of rule-based expert systems.

5. FUTURE PLANS & LESSONS LEARNED

Despite the promising benefits Artificial Intelligence promises to bring into this project, there is currently no part of the simulation taking advantage of rule-based expert systems or fuzzy logic. In describing our pilot system, we have unveiled many improvements planned for future iterations of the project. The economics simulator serves as a working economics modeling tool for an economics course pertaining to public finance at our institution. The pilot features custom tools for graph and chart design, as well as the file and networking functionality required for the program to submit student results to the instructor. Currently, the model severely restricts derived data and user inputs in order to ensure results are not skewed by anomalous values. The implementation of the sample population economy as a database table has caused program latency due to repeated object instantiations to handle connections and accesses to the database. This can be remedied in many ways. First by reducing the number of object instances and connections to the database by converting applicable functions into database stored procedures or triggers, and loop unrolling the functions that cannot be implemented at the DBMS level. An expert system could provide the means of improving performance without the loop unrolling.

With the completion of a new version of the project, we will have a new generation of economics simulation software. The user will operate the software in two phases. First, the user will be presented with a detailed menu of policy choices in five broad areas. In *Externalities*, students will have to grapple with traffic congestion, pollution, and CO_2–induced climate change. In *Social Security*, students will have to adjust the Social Security system to remain solvent in the face of longer life spans and a retiring Baby Boom. In *Social Insurance*, students will have to create a system that provides income support to low-income families without dis-incentivizing paid work. In *Health Care*, students will design their own healthcare system, maximizing coverage and quality while containing costs. In *Education*, students have a range of options for raising educational achievement while lowering costs. The health, educational, and budgetary implications of student policy choices will be forecasted out for the next 50 years.

In the second phase of the new system, the policy choices made in the first phase will be locked in place, and students must raise tax revenue to pay for all the programs they created in the first phase. Again, a wide range of options are available to the students, including traditional income taxes, payroll taxes, consumption taxes, and value-added taxes. Once again, the budgetary and economic implications of the students' policy choices will be forecasted out for the next 50 years.

Overall, the current system serves its requirement of allowing users to first customize their economy, and then fund it by means of setting up a taxation system. As we evaluate our work and experience with this system, Fred Brooks' (1975) often-quoted preposition from *The Mythical Man Month* comes to mind. The management decision is not whether to build a pilot system. Every system has a pilot, hence "plan to throw one away; you will, anyhow" (p.116). Is this system merely a pilot that will inevitably be thrown away? Could our program foreshadow new and exciting progress in economics modeling and simulation? Despite the vast amount of progress made in the simulation, there are apparent defects in the current version and obvious improvements that can be made to the model. Through fruition, the pilot software successfully fulfilled its purpose in the classroom; however the real value will come in the next iteration of the system, which promises better design, more efficiency, and inclusion of features dropped from the pilot system due to time and other constraints. In short, the value of this pilot system will not be fully realized until the system's next version.

We stated in the beginning that we are motivated to create a system that encapsulates a design that looks more like our problem domain (an economic model envisioned by the user) than computing architecture. Volumes have been written about the economics of software development (Royce, Bittner, & Perrow, 2009). Yet it remains a difficult challenge that requires enormous time and effort. And as we have shown, the developer must be prepared to throw it all away and start anew. However, as we start anew we are reassured that we possess significantly more experience and knowledge of the problem domain than we did before we began. So, in addition to re-learning one of Brook's theses, we have also rediscovered in working with our economists and students, that software development is indeed a social learning process (Pressman, 2001). We still have much to learn.

5. REFERENCES

Brooks, Frederick P. (1975). The Mythical Man-Month. Addison-Wesley, Reading, MA.

Denning, Peter J. (2007). Computing is a Natural Science. *Communications of the ACM*, 50(7), 13-18.

Fowler, Martin (2001). The Agile Manifesto. *Software Development*, August, 28-32.

Pressman, Roger S. (2001). Software Engineering: A Practitioner's Approach. 5th Ed. McGraw-Hill, New York, NY.

Royce, W., Bittner K., & Perrow, M. (2009). The Economics of Iterative Software Development. Addison-Wesley, Reading, MA.

Walsh, Patrick (2010) "Your Turn: Simulation Software for Teaching Public Economics." Proceedings of 21st Annual Teaching Economics: Instruction and Classroom Based Research. Robert Morris University, Pittsburgh, PA.

Appendix

Cohorts are indexed by the year of their birth, b.
Health is indexed by h (low, medium, high)
Ability is indexed by a (low, medium, high)
Time is indexed by t. The present period is t=T

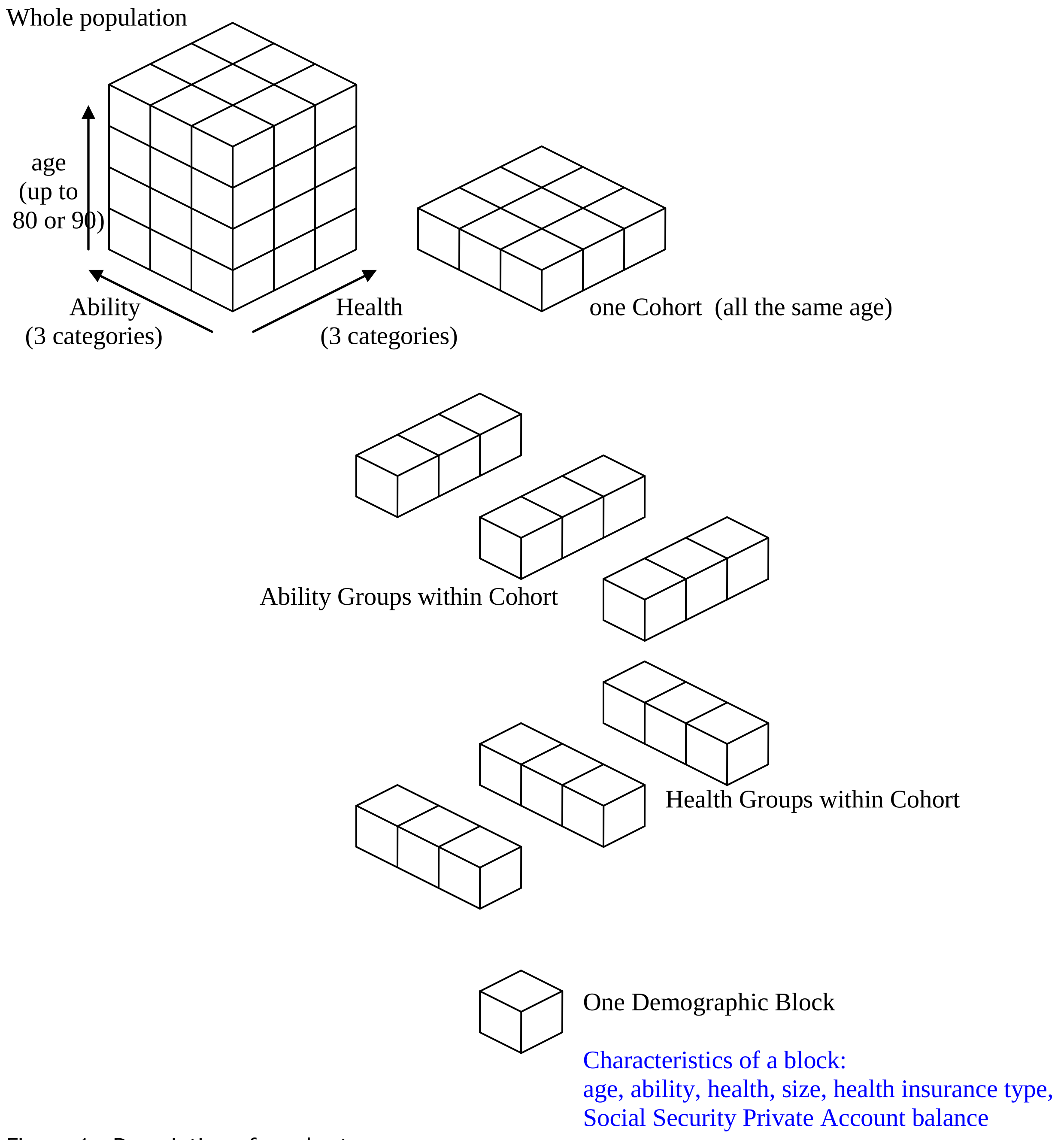
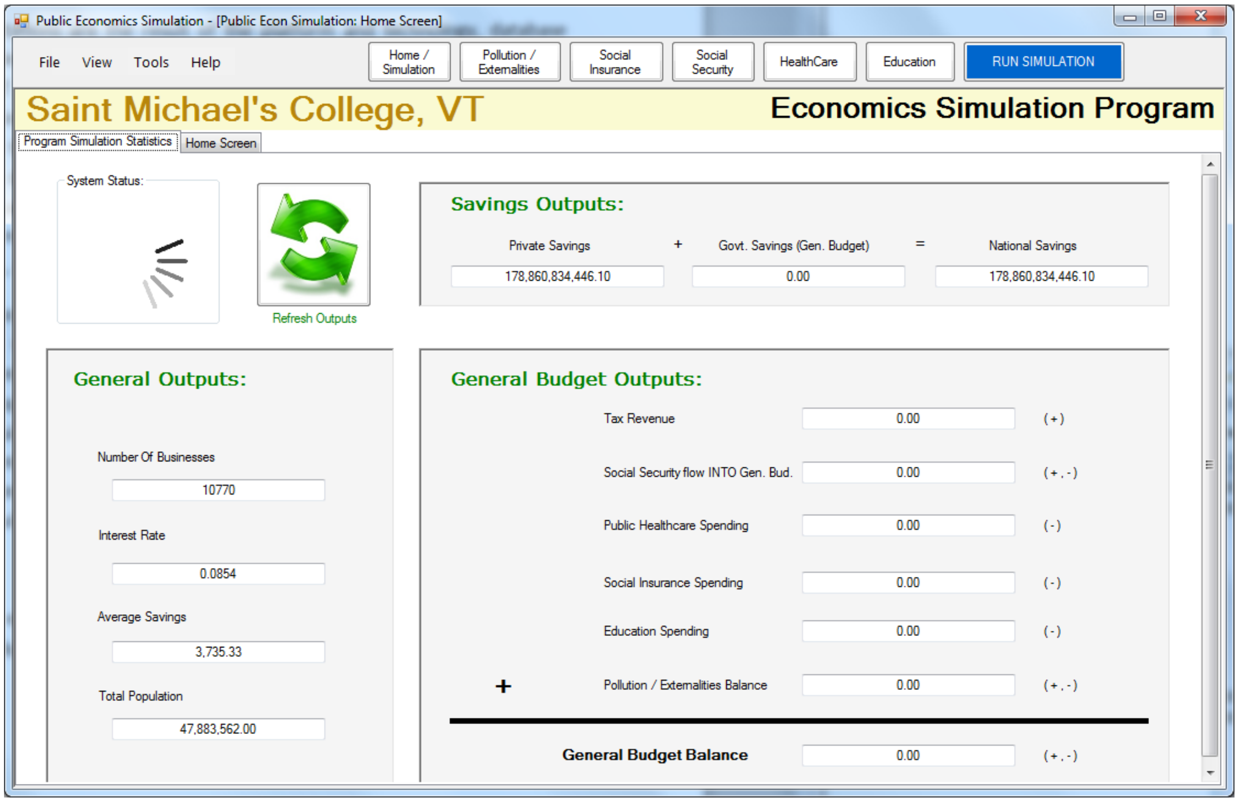

Whole population
age (up to 80 or 90)
Ability (3 categories)
Health (3 categories)
one Cohort (all the same age)
Ability Groups within Cohort
Health Groups within Cohort
One Demographic Block
Characteristics of a block:
age, ability, health, size, health insurance type,
Social Security Private Account balance

Figure 1. Description of a cohort.


Public Economics Simulation - [Public Econ Simulation: Home Screen]
File
View
Tools
Help
Home / Simulation
Pollution / Externalities
Social Insurance
Social Security
HealthCare
Education
RUN SIMULATION
Saint Michael's College, VT
Economics Simulation Program
Program Simulation Statistics
Home Screen
System Status:
Refresh Outputs

Savings Outputs:

Private Savings	+	Govt. Savings (Gen. Budget)	=	National Savings
178,860,834,446.10		0.00		178,860,834,446.10

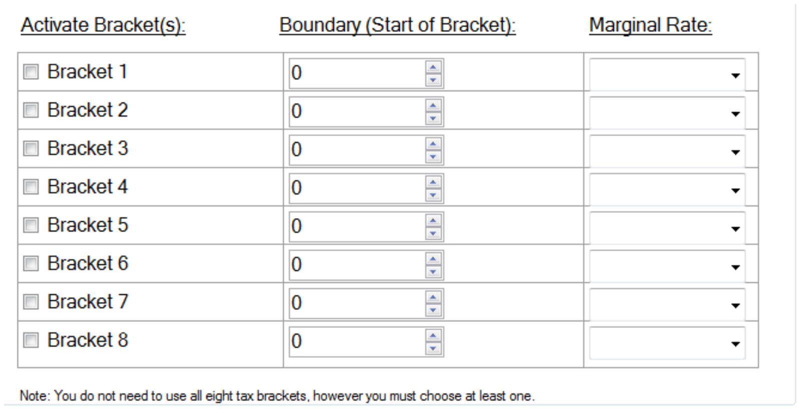
General Outputs:

Number Of Businesses	10770
Interest Rate	0.0854
Average Savings	3,735.33
Total Population	47,883,562.00

General Budget Outputs:

Tax Revenue	0.00	(+)
Social Security flow INTO Gen. Bud.	0.00	(+ , -)
Public Healthcare Spending	0.00	(-)
Social Insurance Spending	0.00	(-)
Education Spending	0.00	(-)
+ Pollution / Externalities Balance	0.00	(+ , -)
General Budget Balance	0.00	(+ , -)

Figure 2. VB Interface for current simulator.



Activate Bracket(s):	Boundary (Start of Bracket):	Marginal Rate:
☐ Bracket 1	0	
☐ Bracket 2	0	
☐ Bracket 3	0	
☐ Bracket 4	0	
☐ Bracket 5	0	
☐ Bracket 6	0	
☐ Bracket 7	0	
☐ Bracket 8	0	

Note: You do not need to use all eight tax brackets, however you must choose at least one.

Figure 3. Income Tax Brackets Interface.

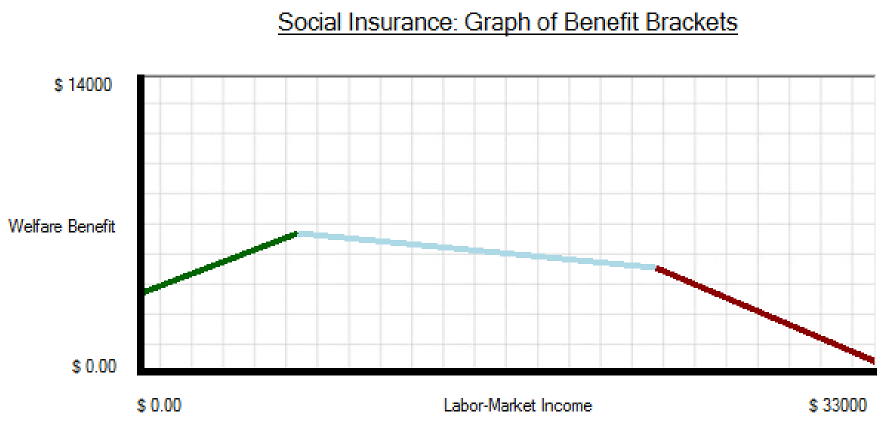


Figure 4. Sample Graph from the Economic Simulator.

Rule 1:	IF AverageSavings <= 1000	THEN InterestRate = 0.35	(Confidence += 100%)
Rule 2:	IF 1000 <= AverageSavings < 1500	THEN InterestRate = 0.25	(Confidence += 100%)
Rule 3:	IF 1500 <= AverageSavings < 2000	THEN InterestRate = 0.20	(Confidence += 100%)
Rule 4:	IF 2000 <= AverageSavings < 2500	THEN InterestRate = 0.15	(Confidence += 100%)
Rule 5:	IF AverageSavings >= 3000	THEN InterestRate = 0.10	(Confidence += 100%)

Table 1. Potential Rules for Future Expert System.