Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.

Running head: INVESTMENT TECHNOLOGY FOR TRADING BUSINESSES

Investment Technology for Trading Businesses: Delineating Requirements, Processes, and Design Decisions for Order-Management Systems

Daniel L. Mark

Regis University

School for Professional Studies

Master of Science in Computer Information Technology

Regis University

School for Professional Studies Graduate Programs

MSc in Computer Information Technology Program Graduate Programs Final Project/Thesis

Certification of Authorship of Professional Project Work

Print Student's Name **Daniel Mark**

Telephone **631-335-0226**

Email **dmarkbusinessonly@gmail.com**

Date of Submission **02/25/2008**

Degree Program **MSc in Computer Information Technology**

Title of Submission **Investment Technology for Trading Businesses: Delineating Requirements, Processes, and Design Decisions for Order-Management Systems**

Advisor/Faculty Name **Doug Hart, Don Archer**

Certification of Authorship:

I hereby certify that I am the author of this document and that any assistance I received in its preparation is fully acknowledged and disclosed in the document. I have also cited all sources from which I obtained data, ideas or words that are copied directly or paraphrased in the document. Sources are properly credited according to accepted standards for professional publications. I also certify that this paper was prepared by me for the purpose of partial fulfillment of requirements for the Master of Science in Computer Information Technology Degree Program.

Student Signature	*Date*
Daniel Mark	2/25/2008

Regis University

School for Professional Studies Graduate Programs MS in Computer Information Technology Program Graduate Programs Final Project/Thesis

Authorization to Publish Student Work

I, Daniel Mark, the undersigned student, in the Master of Science in Computer Information Technology Degree Program hereby authorize Regis University to publish through a Regis University owned and maintained web server, the document described below ("Work"). I acknowledge and understand that the Work will be freely available to all users of the World Wide Web under the condition that it can only be used for legitimate, non-commercial academic research and study. I understand that this restriction on use will be contained in a header note on the Regis University web site but will not be otherwise policed or enforced. I understand and acknowledge that under the Family Educational Rights and Privacy Act I have no obligation to release the Work to any party for any purpose. I am authorizing the release of the Work as a voluntary act without any coercion or restraint. On behalf of myself, my heirs, personal representatives and beneficiaries, I do hereby release Regis University, its officers, employees and agents from any claims, causes, causes of action, law suits, claims for injury, defamation, or other damage to me or my family arising out of or resulting from good faith compliance with the provisions of this authorization. This authorization shall be valid and in force until rescinded in writing.

Print Title of Document(s) to be published: **Investment Technology for Trading Businesses: Delineating Requirements, Processes, and Design Decisions for Order-Management Systems**

Student Signature	*Date*
Daniel Mark	2/25/08

Check appropriate statement:

X The Work does not contain private or proprietary information.

_____ The Work contains private or proprietary information of the following parties and their attached permission is required as well: _____

School for Professional Studies Graduate Programs

MSc in Computer Information Technology ProgramGraduate Programs Final Project/Thesis

Advisor/Professional Project Faculty Approval Form

Student's Name: Daniel Mark Program MSc in Computer Information Technology

PLEASE PRINT

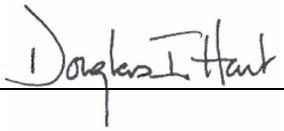
Professional Project Title: Investment Technology for Trading Businesses: Delineating Requirements, Processes, and Design Decisions for Order-Management Systems

Content Advisor Name Doug Hart

Project Faculty Name Don Archer

Advisor/Faculty Declaration:

I have advised this student through the Professional Project Process and approve of the final document as acceptable to be submitted as fulfillment of partial completion of requirements for the MSc in Computer Information Technology Degree Program.

Original Content Advisor Signature  2/27/2008

*Original Module/Class Facilitator Signature*_____
Date

Degree Chair Approval if:

The student has received project approval from Faculty and has followed due process in the completion of the project and subsequent documentation.

Original MScCIT Degree Chair/Designee Signature
Date

Document Revision History

Additions/Modification	**Date**
Abstract	7/1/2007
Outline	8/25/2007
First Draft	12/30/2007
Final Research Documented	1/15/2008
Final Feedback Revisions	2/25/2008

Acknowledgements

I would like to acknowledge the generous contributions of extensive and thought-provoking feedback and guidance provided primarily by Doug Hart and Don Archer. This work would not have been possible without their consistent and timely input regarding academic concerns, content, analytical and writing style, and overall approach to the topic. Throughout more than 5 years of working with Regis University's MSCIT program, staff's universal willingness to assist and accommodate a hurried working-adult student was nothing short of inspirational. I would also like to thank family and close friends for their support and encouragement as I progressed toward this personal milestone. Finally, heartfelt thanks go out to Eric Smith, Rob Howard, and their team at CodeSmith Tools; dedicated practitioners of software development owe a debt of gratitude to imaginations capable of such ideas. I thank you all.

Abstract

The requirements and processes for building a robust order-management system (OMS) for trading of investments within financial services firms are investigated and enumerated. Requirements and process documentation are not readily available to members of the general public because they are considered a source of competitive advantage in a highly profitable industry. This paper provides single-source documentation of those requirements and processes in the context of the Vested OMS application, which was constructed specifically to meet industry needs in this area. This paper describes in detail the core functionality investment businesses currently demand and the software development techniques used to construct a core system to meet those demands.

List of Figures

Figure 1: The Enterprise Library Configuration Application……..38

Figure 2: CodeSmith and NetTiers Template……41

Figure 3: Vested Web Site External Dependencies……45

Figure 4: Vested Namespaces……46

Figure 5: Orders List Sorted by Quantity……...49

Figure 6: Securities – Add/Edit……..51

Table of Contents

Chapter 1: Introduction 10
1.1 Order-Management Systems 11
1.2 Complexity, Scope, and Demand 12
Chapter 2: Definitions and Literature Review 16
2.1 Equity Instruments 17
2.1.1 Common Stock 17
2.1.2 Preferred stock 18
2.1.3 Restricted Stock 18
2.2 Debt Instruments 19
2.2.1 Bonds 19
2.2.2 Commercial Paper 20
2.3 Derivative Instruments 20
2.3.1 Forwards 20
2.3.2 Futures 22
2.3.3 Swaps 23
2.4 Financial Technology Research 24
2.6 Software Design Best Practices 27
2.7 Supporting Trading Business Needs 31
Chapter 3: Methodology 34
Chapter 4: Vested Architecture and Project History 36
4.1 Vested Architecture 36
4.1.1 Caching 37
4.1.2 Security 41
4.1.3 Data Access 42
4.1.4 Business Layer 43
4.1.5 Class Definitions 44
4.2 Using Vested 48
4.3 History and Reflection 52
Chapter 5: Lessons Learned and Next Evolution 54
5.1 Lessons Learned and Learning Curves 55

Investment Technology for Trading Businesses: Delineating Requirements, Processes, and Design Decisions for Order-Management Systems

Chapter 1: Introduction

There exist ample resources for those wishing to study finance in its many forms. From basic how-to manuals for profitable stock trading to advanced texts covering variance in derivative valuations and balance sheet analysis, information is available to the curious and the committed alike. Similarly, software construction documentation is plentiful irrespective of development platform, choice of programming language, or architectural requirements and preferences.

Unfortunately, companies with considerable financial support, such as Charles River Development, Napa Group, Advent, Ameritrade, Fidelity, and every bank that employs algorithmic trading model experts, tightly control the knowledge and guidance they use in developing systems that bring institutional investors electronic management over their trading by combining knowledge of finance with software prowess. The only pools of combined investment and software documentation comes either from bringing knowledgeable people together, intensive long-term study, or reverse engineering of systems such as Microsoft's .NET StockTrader Sample Application.

Knowledge at the intersection of the financial and information technology domains is kept under lock and key to the extent that BBC reporter Ines Bowen was told by one fund in reference to some of its computer modeling mathematicians that "many of them…are autistic". (Bowen, 2007) While anecdotal evidence is very limited in terms of generalizability, if true, it would be

unsurprising both because they are highly skilled and may be perceived as less likely to speak of their work with outsiders. Later in the same article, a head of a different hedge fund spoke to Bowen of his hiring concerns, saying "In today's world there's a good market for social skills. We do not necessarily require that." Automated trading algorithms are now so powerful that there are legitimate concerns about the role of computer programming in causing or deepening market crises.

1.1 Order-Management Systems

Within the finance domain, authors such as Andrew Chisholm, Jim Cramer, Michael Durbin, and Michael Covel proffer guidance on topics from stock picking and timing to calculating the present value of a futures position given formulaic inputs. However, aside from simplified software intended only for concept demonstration, there are no such definitive sources regarding the gathering of requirements or the designing of architectures to specifically address order-management systems (OMS), which allow for the input, storage, and analysis of investment purchase and sale decisions that drive finance-industry profits. Extensive research has documented sets of best practices and design patterns for software development that vary somewhat across platforms but that have most generalities in common; again, no specific guidance is available regarding the tailoring of these ideas to the needs of businesses whose profits are derived from trading investments. Examples of such intensive research include Microsoft's Enterprise Library, Microsoft's Patterns and Practices Group, and the independent "Gang of Four" design patterns legacy. The research presented in this

paper brings together information from both domains to concisely and specifically address the intersection of trading needs, money management requirements, and information technology practices for an industry whose changes generate near-daily headlines, with successes minting new billionaires and failures costing the general public billions, threatening global economic stability, and resulting in expensive government bailouts as in the case of Long Term Capital Management.

The OMS currently offered to institutional investors are not readily available or open for research purposes. Neither the technical and business requirements necessary for development nor the specifications to which these systems must be constructed are obtainable for any purpose outside the companies that market them or their customers, who may pay hundreds of thousands of dollars per year for the use of what amounts to industry-customized create, retrieve, update, and delete applications. Clarifying these requirements, documenting the architecture and design that enables the build-out of a core system that minimizes the resources required to extend its functionality, and constructing said system will permit the dissemination of knowledge for one concrete example of both the "what" and the "how" of these systems – the written business requirements and the mechanical implementation details.

1.2 Complexity, Scope, and Demand

This domain is interesting because of the complexity, scope, market demand, and potential uses of investment automation tools in the financial sector. Further, recent market conditions included credit and debt turmoil, valuation difficulties, and substantial percentage losses attributed to unexpectedly

heterogeneous automated trading systems that resulted in massive amounts of highly leveraged capital committed to unexpectedly duplicated strategies across very large pools of invested capital. Similarity between positions across different firms' computer-driven models forced many money managers to take substantial losses by exiting innumerable investments earlier than planned, before their losses could grow larger.

The complexity involved in developing the data architecture and basic functionality to manage investment systems requires the coordination of hundreds of data points and incorporation of numerous sets of business rules replete with numerical formulas and specific, logic-defining business knowledge. To offer an attractive feature set, the scope of the software must encompass most common asset classes and offer a fine-grained level of control over the system's underlying data. This breadth and depth of functionality must be available in a highly usable, nearly real-time, and high-availability application because it must be operated in a fast-paced and information-intensive environment with a user base easily numbering in the hundreds.

The scope of deployed OMS is delineated by their use within the largest and smallest of money managers, including retail, thrift, and investment banks, hedge funds, trust funds, pension managers, non-profits, government agencies, university endowments, and more. They are typically licensed and sold on a percentage-of-portfolio-managed basis, such that the annual fee for ongoing maintenance and support of the software is composed of a percentage of the funds the software is used to manage. The funds, or assets, under management are

determined by summing the value of the securities for which orders are placed in the OMS. This licensing model is often complemented by a per-transaction fee. Finally, maintenance in the form of customizations, performance, and usability are commonly paid for with fees added to the licensing costs of the customer or customers that most vocally requested them.

The future market for OMS shows significant potential as a subset of the market for information technology in the financial services sector, especially given the anticipated tripling of the total dollar amount managed by hedge funds around the world within the next three years. While money flows into hedge funds, the number of trades per investment-industry dollar should increase because hedge funds trade more actively than private investment banks and other asset aggregators. According to industry data, assets under management in the secretive hedge fund industry totaled approximately $1.3 trillion as of December, 2006. This within firms that often use leverage to double, triple, and even quadruple the compounding impact of each invested dollar.

I first present requirements summaries that illuminate the business challenges and environments within which OMS are used. Mandatory features are explored at length because current systems in the marketplace, with unique individual differences between features, include many capabilities that are not central to order management. Next I explore logical differentiation and categorization that must be constructed to process orders for most common types of securities. Business-oriented requirements, primarily regarding position-related calculations are discussed in detail. I summarize current considerations regarding

software, including state of the art rapid application development (RAD), design patterns, and best practices theory. Finally, business requirements and technological decisions are brought together, concluding in a discussion of Vested, the web-based application built upon business requirements and technical foundations revealed throughout. The scope of this document is limited by resources; documentation encompassing all possible rationale and mechanical specifics of a full-scale OMS requires the cooperation of at least a small team over a long period of time.

Chapter 2: Definitions and Literature Review

The business concepts and definitions that must be understood as a prerequisite for building an OMS revolve around the items for which orders will most commonly be entered: common stock, restricted stock, preferred stock, futures, options, and swaps. It can further be helpful to understand one basic classification level that separates investment types; some basic investment vehicles, such as common stock, give their holders an ownership percentage of the public company while others, such as corporate bonds, are analogous to a loan the investor gives the offering corporation in return for periodic interest payments and repayment of principal in the future.

Unlike stocks and bonds, futures, options, and swaps are all derivative instruments. Their value is not intrinsic as in the case of standard equity and debt instruments; rather their value changes according to the value of another asset, known as the underlyer. Using an option as an example, which gives its buyer the option, but not the obligation, to buy or sell at a predetermined price on or before (an American-style option) or only on (an European-style option) a specific date in the future, the value of the option is based on the value of what it is an option to buy. If it is an option to buy 100 shares of IBM in one month for $100 each and IBM is currently trading at $150 per share, it is easy to understand that the options value will change as the value of its underlyer (IBM common stock) changes. Equity, debt, and derivative investment vehicles are the basic building blocks of most money managers' actively traded portfolios.

2.1 Equity Instruments

The person or legal entity that owns 1 share of a public company's common stock, whose total number of outstanding shares (called the "float") is 100, owns 1% of the company. Were the company sold for $100, this person would be due $1. This is termed equity because the shareholder literally owns a piece of the company.

2.1.1 Common Stock

Common stock derives its name from the fact that it is the most frequently utilized equity-related investment vehicle. It is available for purchase to nearly any individual with money to invest and has generated a higher long-term percentage return on investment than traditional debt-related instruments. As mentioned earlier, its owners would be entitled to proceeds upon the sale of the publicly-held company. In addition, owners of common stock are typically given a say in high-level management decisions through shareholder votes and are often paid periodic dividends. Ownership grants "a claim to a part of the corporation's assets and earnings." (Investopedia, 2007)

Common stock is most often traded on exchanges, such as the New York Stock Exchange, the NASDAQ, the FTSE in England, and the Shenzhen Stock Exchange in China. These exchanges bring buyers and sellers together in a controlled and regulated environment where individual and institutional investors trade using their own money or funds they invest on behalf of others. Common stock can also be traded without an exchange in a process known as over-the-counter (OTC) trading, but typical stockholders limit their activity to exchanges, or even to exchanges located in their home countries. Exchanges in one country

frequently offer the ability to buy pseudo-stock in publicly held companies whose formal stock trades only in other countries. An example of this type of situation would be the listing of Baidu.com's common stock on the NASDAQ exchange, but only as a depository receipt. A depository receipt is fundamentally different from a share of common stock, but both trade at essentially the same underlying values.

2.1.2 Preferred stock

"Owners of preferred stock", according to Investopedia, "receive dividends before common shareholders and have priority in the event that a company goes bankrupt and is liquidated"
(Investopedia, 2007). Additionally, preferred stock does not necessarily grant the holder voting rights regarding the high-level management decisions whose outcomes are the result of common shareholder's voting. Preferred stock may be callable, also known as redeemable, meaning that the issuing firm has the right to buy the stock back at a certain price, taking it out of circulation by returning cash, common stock, or a combination of both to the preferred shareholder's owner. Preferred stock can be exchange-traded or OTC and it is typically not found in individual investors' accounts.

2.1.3 Restricted Stock

Restricted stock is under a sales restriction and must be registered with the SEC or fall under the regulations exempting it from registering before it can be sold. "Insiders are given restricted stock after merger and acquisition activity,

underwriting activity, and affiliate ownership in order to prevent premature selling that might adversely affect the company. Restricted stock cannot be sold without registration with the SEC." (Investopedia, 2007)

2.2 Debt Instruments

Other instruments, most prominently bonds, are considered debt-related. Through these investments, public companies are loaned money for a fixed period of time. A simple example would be a common corporate bond in the amount of $1,000,000 with a maturity date one year from its inception date. The investor would receive periodic interest payments over the course of the year, almost always at regular monthly intervals, and would always receive a repayment of the principal $1,000,000 at year's end, except in the notable and risk-defining case of default.

2.2.1 Bonds

Corporate and municipal bonds are duration-limited loans used primarily by businesses and governments to borrow money used for the organization's expenses. Investors buying bonds frequently deal in increments of at least twenty-five thousand dollars, making them too expensive for the majority of investors. The corporation or government entity makes periodic interest payments to the investor and repays the entire principal amount of the bond at a future maturity date. Issuers assist institutions looking to raise money this way by packaging, rating, and offering bonds to investors, whose concerns include risk, interest rate, duration, principal amount, rating, and funding purpose. An investment in a United States Treasury Bill is considered essentially risk-

free because the chances of default are remotely small. Municipal bonds, offered to investors by local government agencies, are somewhat riskier, with corporate bonds being the riskiest of the three. In order to compensate investors for taking more risk, corporations raising money through bond issuance offer to pay investors higher rates of interest over the life of the bond.

2.2.2 Commercial Paper

Commercial paper operates in a market very similar to bonds, but these investor-bought loans are rarely originated with a term more than one year. Further, investing in them typically requires an even larger amount of money than in traditional bonds, a greater tolerance for risk, and increased due diligence on behalf of the investing party. Commercial paper operates in a manner similar to long-term financing but with rates of interest closer to those of short-term loans. This makes it an attractive source of capital for businesses that have ongoing operational finance needs.

2.3 Derivative Instruments

Derivatives are the third primary type of investment vehicle. A derivative's valuation depends on the presence of another investment instrument, which is referred to as the derivative's underlyer. There are four basic types of derivatives that form the foundation for numerous permutations within each category. The four basic classifications are forwards, futures, options, and swaps.

2.3.1 Forwards

Stock forwards involve OTC agreements between a buyer and a seller to exchange goods in the future at a price determined at the time of the agreement.

They involve counterparty risk to the extent that a buyer may be unable to buy at the agreed-upon date or a seller may be unable to sell. A primary consideration when trading forwards is the determination of the price at which goods will be exchanged in the future.

A theoretical fair price is calculated by considering the amount of cash the selling, or short, party would need to borrow in order to buy the amount of the asset that the long party, which is buying in the future, agrees to purchase. The interest rate at which this money can be borrowed and the duration of the forward contract are combined to add a cost of carrying to the short (selling) parties cost of acquiring the contracted quantity. Finally, any dividends that are payable to the short party over the duration of the life of the contract are subtracted from the cost of carrying. If a potential long party wants 100 shares of IBM in 12 months and those shares are purchasable today for $100 each without any transaction costs, the short party will need to borrow $10,000 to acquire the shares today. Assuming an interest rate of 4 1/2 percent per year, the cost of carry would be $450 and the fair price for future trading that can be agreed upon today would be $10,450 ($104.50 per share). If an annual dividend of $1 per share is paid, a cumulative dividend of $100 would be subtracted from the cost of carry, yielding a $10,350 fair forward price.

In order to perform standard investment calculations, the forward security type requires the following information: the date, which must be in the future, that the exchange will occur, the price at which the seller agreed to sell and the buyer agreed to buy on that future date, underlying investment information, and the fair

or theoretical forward price. In practice, the short party must ensure the contract is executed above the fair price in an attempt to guarantee a profit. Because forwards are OTC instruments and therefore engender greater counterparty risk than the nearly identical futures contract, the percentage amount above the fair price that the seller will negotiate toward will be greater in order to compensate the short party for taking the perceived risk that the long party will not have the funds necessary to settle the contract at maturity.

2.3.2 Futures

The future contract shares the same fundamental structure and price determination methodology. The primary differences between futures and forwards are the standardization of the contract and the almost complete removal of counterparty risk through a margin and clearinghouse system. Whereas forward contracts are available OTC and thus can be arranged with any terms the parties agree to, futures contracts are traded over exchanges and are standardized in format with terms that vary according to the dynamics of the underlying asset. Futures contracts require essentially the same information for calculation of fair and agreed-upon prices; the spread between these prices may be smaller because perceived counterparty risk is reduced.

Common futures include natural gas, crude oil, orange juice, gold, bonds, interest rate, and equities. Using natural gas as an example, the standardized elements within the contract include: unit of trading, the type of deliverable, the tick size (minimum change in price) expressed as a fraction of a point where a point is 100 basis points, the tick value that is derived from the tick size and a unit

of trading, the price quotation, and the contract months last trading day in the last delivery day (Chisholm, 2004).

2.3.3 Swaps

Swaps involve the exchange of payments between the involved parties at fixed intervals for a fixed period of time. Each individual payment is referred to as a leg. The basis for determining how much payment must be made to the counterparty differs according to which side of the swap the trader takes. An interest rate swap's payment legs would differ across parties because the interest rate applicable to one party's payments is fixed for the duration of the swap while that applicable to the counterparty's payments is floating, or open to change.

A so-called plain vanilla swap includes a notional principal monetary amount that is fixed at the start and does not vary. This principle deemed notional because it is never exchanged but is only used as the basis when combined with interest rates for determining payment leg amount. One party multiplies the notional principal times a fixed interest rate and makes a payment based on the outcome of this calculation on multiple regular dates in the future. The other party does the same but with a variable rate of interest. When each floating payment is made the interest rate applicable to the next floating payment is determined based on a benchmark reference rate plus or minus any adjustments or conversion ratios applied to that rate as per the swap agreement. (Chisholm, 2004)

If the notional amount is $100,000 and the fixed rate portion has an interest rate of 5 1/2 percent per year, the party paying fixed will pay the floating party $5,500 per month. If the floating party's rate is benchmarked against the

federal funds rate and the federal funds rate is 4 1/2 percent with quarterly rate resets, both parties know that the fixed payment will be $4500 per month for the first three months. The fixed party in this example would be hoping the floating parties benchmark interest rate increases, which may result in future payments being lopsided and the fixed party coming out on top. A fixed party might take this type of position in an effort to offset, or hedge, their exposure to rising interest rates. They may be faced with paying back a loan whose payments are tied to a fluctuating benchmark rate. By entering into an interest rate swap, a firm can effectively lock the interest rate which had otherwise been variable, thereby protecting its profitability and allowing it to more accurately forecast future cash flows.

2.4 Financial Technology Research

Unsurprisingly, plenty of research has focused on how to make money by trading the most common instruments in the financial markets. More often than not this research is implicitly or explicitly focused on the selection of assets into which funds are invested. Most market participants would at least be aware that research is performed and publicly available regarding investment decisions, but the majority of investors are likely unaware of research focused on automating investment decision-making.

Fernández and Gomez (2007) explored one common method for automating portfolio selection by enhancing the traditional mean-variance model with diversity assurance. This methodology involves determining a desired percentage level of return, finding a group of securities whose mean historical

return matches the desired percentage return from step one, finding the variance from that mean within the group of selected securities, and owning, from within the overall group of assets that historically returned that percentage, those with the least variance from the mean return level. While many groups of selected securities could return the desired percentage, following these steps theoretically ensures that the target return is pursued with the least possible amount of risk. Fernandez and Gomez refer to this portfolio as the "efficient frontier".

Yon and Clack (2004) discussed a genetic programming approach to a solving a challenging problem: how to ensure diversity within an automatically selected portfolio when faced with dynamic economic environments. Ultimately, an equation that takes numerous variable values into consideration assists with making buy and sell decisions. Their system attended to a set of 24 equity-related factors and was self-training. When the economic environment changes significantly, their system was capable of altering the decision-making equation in order to address changes in the environmental factors.

While some have focused on achieving investment performance goals automatically (imagine building substantial wealth without having to invest time to get it), other research has questioned whether it is even possible to predict what a given stock market's rate of return will be. The results seen by Olmeda and Moreno (2007) suggest that market returns are "clearly nonpredictable". Their assertion is that what cannot be predicted cannot be profitably exploited.

However, contradictory research (Greenblatt, 2006) indicates that there is an incredibly simple formula that essentially uses price-to-earnings ratio and

earnings yield as the only two criteria for finding excellent stock purchases. Greenblatt's techniques frequently yielded returns double and nearly triple those of the overall market while simultaneously producing barely positive returns even when the market's overall returns were down considerably for substantial time periods. Greenblatt recalled the advice of Benjamin Graham, the original practitioner-advocate of value investing, to buy stock only when there is a margin of safety available; only buy when the stock is available for purchase at a price below its true value. While not a fully automated process, Greenblatt did perform significant back testing of value investing's essential underlying concept against a major database of historical stock information, while mitigating common stock market research weaknesses such as look-ahead bias, survivorship bias, and transaction cost inclusion.

Covel (2007) also suggests that it is possible to reliably generate long-term returns much greater than those of the general market. He summarized the experience of numerous high-profile trend following traders whose profits are derived not through the application of value investing guidelines as used by Greenblatt or defined by Graham, but by following price trends. His comparison of the returns generated by Abraham Trading, for example, with those of the S & P 500 showed that $1,000 invested in Abraham Trading's trend following business would have turned into approximately $34,051 whereas investing in the S & P 500 over the same period would have netted only $4,280.

As value investing and trend following, likely two of the easier to use investment strategies – and certainly less complex than building fully automated

algorithm-based systems –, become better known, new potential customers for a usable order management system that offers a simple way to input and track orders appear.

Whether one is interested in automated trading systems using statistical algorithms that assist with buy and sell decisions, learning systems whose algorithms can be trained at the outset and that automatically change to compensate for macroeconomic factors, or just general investing tips for the average individual investor, there are many sources available that provide widely varying levels of guidance irrespective of whose research an investor believes is more accurate. However, if one is interested in building an OMS for institutional investor use, there are few, if any, sources for ideas regarding business rules or implementation concerns.

2.6 Software Design Best Practices

There are numerous goals of best practices in the software industry. These best practices define specific techniques for constructing software that meets performance, scalability, and maintainability goals. These goals are of particular concern when constructing an OMS for money-focused businesses.

In database-driven systems performance is often a primary concern. This is especially true for OMS because users are often stereotypical traders whose personal incomes are on the line. They have strong preferences about how they spend extra time throughout the day. An OMS that does not perform quickly leaves less time for the analysis required to keep profits and minimize losses.

Best practices that address performance include caching, use of only high-speed programming techniques, and partial-page post back for use with Web-based technologies.

Caching involves temporarily storing data retrieved from the systems database so that when the same data is needed again another trip to the database, which usually resides on another computer some distance from the application using it, is not necessary. Avoiding unnecessary trips to the database can result in substantial performance improvement.

High-speed programming techniques should be used in place of other available alternatives in order to mitigate the risk of poor performance. For example, when using Microsoft technology, performance improvement will result from avoiding cursors on the database side and favoring forward-only data reader objects on the client's side.

Web considerations often include application response time because Web servers are often located great distances from the client application. In part because web applications use the stateless HTTP protocol, entire web pages are often sent long distances to and from the Web server each time the user interacts with the page, even if the user's interaction really affects content from only one specific section of the page. The recent rise of AJAX, which was originally designed to send XML data via HTTP, has made it easy to send the server only what it needs to refresh affected content, speeding response times by lowering the amount of data that must travel from client to server and back.

According to the Free On-line Dictionary of Computing, scalability is defined as “How well a solution to some problem will work when the size of the problem increases” (Free On-Line Dictionary Of Computing, 2007). Multi-user applications may perform quickly enough when there are five users but may slow down substantially as more users are added. Applications that slow down to substantially would be considered to have scalability problems. Another scalability concern is growth in the amount of data an application manages over time. Within an OMS the number of orders increases with each passing trading day, resulting in a larger amount of data due to the increased number of rows in order-related tables.

Solutions to scalability problems typically involve scaling up, in which more power is added to a single computer by adding memory and or CPUs, or scaling out, where entire extra computers are added. However, Malcolm Davis presented results from a BEA study that showed software design — the techniques used to write the application's code – was the most frequent culprit in production scalability problems (Davis, 2006). Davis’s results suggest that scalability can and should be addressed by best practices. Applications must be designed with scalability in mind in order to avoid performance decreases and maintenance demands as the number of users and amount of managed data grow. Techniques for addressing application-related scalability concerns in a web-based environment include minimal use of shared server resources for session management, flexible paging of returned data, proper indexing of table data, and the use of high speed programming techniques.

Software maintenance accounts for a large percentage of the time and money spent on a software project overall; reducing the amount of time required to make maintenance changes can significantly impact project cost. According to Pfleeger (2001) and Pigoski (1996), 40 to 60% of maintenance time is spent merely reaching an understanding of the current version of the software in the context of what changes must be made. There are three types of maintenance: corrective, adaptive, preventative, and perfective (Swanson, 1976).

All types of maintenance can be eased through adoption of naming conventions, thorough and standardized documentation, and consistent standardization of typical solution techniques. One way to minimize the amount of time spent on maintenance is to adopt naming conventions within application code. According to Microsoft, "A consistent naming pattern is one of the most important elements of predictability and discoverability…" (Microsoft, 2007). A second way to ease maintenance is to methodically document code, which helps developers capture and pass along the general knowledge and any extraordinary techniques required to originally build and maintain an application. As developers that are less familiar with an application's code are called upon to maintain it, it will take less time for them to begin work if they can study the application in precisely the same manner as they have studied other applications previously. Finally, the reuse of solution techniques, such as simple drop-down lists for lookup tables or checkbox sets for joining tables can shorten the amount of time developers spend learning how individual applications solve common problems.

2.7 Supporting Trading Business Needs

Meeting the requirements for a even a basic OMS mandates a system with features that support the tracking of buy, sell, buy-to-cover, and sell short orders for securities within a traditional equity and debt investment portfolio, as well as handling derivatives processing. The user interface must support the entry, storage, display, deletion, and editing of data related to funds, managers, accounts, securities, orders, and allocations. Its data storage must enable performance reporting across funds, managers, and securities.

It must use a data model centered on accounts and transactions related to specific users and funds. It should be easily extensible and allow for future customization. New investment types with unique variations on business rules and technical requirements must plug in to the system's framework.

The central feature required of an OMS is support of order entry, meaning it must enable the insertion and storage of order-related data for the purchase or sale of specified quantities of a given security. The order-entry process must facilitate associating each order any allocations that may contain partial amounts of the total trade. It must record transaction-related information including: trade, settlement, and placement dates, an executing broker, an account, a price, commission, a security identifier, trade direction (long, short, buy to cover, sell long), order status. Allocations must be self-identifying while also relating to an order and orders must be supported across accounts and funds.

Once an order has been entered into an OMS, it must be possible through the user interface to search for, locate, and retrieve all of the order's related details.

Profit and loss calculations are essential to any investment system. It must be possible to calculate the value of the securities held in a given fund in order to determine the total dollar value of the fund.

Some value calculations will, at their simplest, be a matter of multiplying price times quantity, whereas other calculations may require variables, formulas, and potential payoff tables. At their most complex, value calculations will involve the changing values of asset-backed securities, such as mortgage-backed securities, which require periodic re-computation because anticipated future repayment rates cannot exactly match what borrowers with flexible payment terms will pay.

Requirements for the storage of security information that must be related to orders include identifying security number, ticker, and CUSIP. Security name, type, description, issuer, and currency must all be stored.

Funds are one way in which many money managers and institutional investors organize their holdings and offer investment products to their clients under the terms of legally binding agreements. Funds hold positions across various asset classes, so an OMS must offer the ability to manage positions across funds.

The available information for funds must include a fund identifier, name, type, description, currency, market value, and manager. Table 3 lists fund-related data the system should make available.

Money managers often use accounts at numerous financial institutions such as prime brokers, broker/dealers, and custodians while establishing their own internal account numbering systems. Account numbers at any given financial institution are guaranteed, at some level, to be unique; using accounts from more than one financial institution requires establishing account mappings to avoid the possibility of two institutions using the same account number and the resulting inability on the money manager's side to track the two accounts separately. Financial institution information must relate to the system's accounts and it must include an institution identifier, name, and type. The requirements for account data tracking minimally include account identifiers, financial institution identifier, account number, and type.

The order-management and security information requirements will vary according to instrument type, which will include common stock, preferred stock, restricted stock, corporate bonds, municipal bonds, commodity futures, interest-rate futures, and index futures.

Chapter 3: Methodology

Research methods used relied most heavily on review of academic literature over a five-year period. Other research and preparation involved reading of numerous entire investment books, computer science textbooks, professional employment experience at both a software company that made software for the investment industry and a hedge fund that used two industry-leading OMS, and passing three Microsoft technical certification exams resulting in bestowment of a Microsoft Certified Application Developer credential. I also made myself substantially more familiar with advanced and somewhat unusual – relative to my professional life – development systems and techniques, such as Ruby on Rails, object relational mapping (ORM), and automated code generation. More traditional concerns, such as maintenance and scalability, were also researched intensively.

A traditional iterative development process was used, beginning with the definition of the problem, requirements, and consideration of use cases. I next developed a data model that was ultimately refined over more than 10 iterations and that defines the relationships within the model. Repetitive updates to stored procedures within the database after each iteration's data model changes made the repetitive process extremely repetitive, cumbersome, time-consuming, and error-prone. This prompted a review of the state-of-the-art in relational database modeling for software development in general. Specific attention was focused on ORM, which addresses the need to represent tables of data as objects in code. This led to the discovery of code-generation tools and ORM frameworks.

The processes followed through each subsequent development iteration, listed without a fine level of detail, included: creating a backup of the current version prior to making any changes, adding columns, removing columns, assigning default values to columns, changing column data types, changing names of tables or columns to ensure naming consistency, defining new relationships between tables, assigning indexes to tables, normalization resulting in a greater number of tables, updates to non-generated code, and regeneration of generated code.

This project's deliverables include this written document, ancillary bureaucratic documentation, all code for the software that fulfills its requirements, technical documentation of that code, samples of data housed within the system, and demonstration of the working copy of that software.

The software requires the availability of one Microsoft SQL Server 2000 or higher, one Microsoft IIS Web server 5.1 or higher, and a web browser on the client side with Microsoft's Internet Explorer being the only officially supported client.

The outcome of this work is a highly scalable and easily maintainable system suitable for near-immediate use across even large real-world businesses. I view the end result as a resounding success because it achieves its primary goals and does as much as is possible to minimize the amount of time required for standard maintenance demands.

Chapter 4: Vested Architecture and Project History

The project began nearly 6 years before it was completed. Initially, I knew that I wanted to be a highly capable programmer and solution architect and I knew that I would prefer earning more money to earning less. While working full time as a programmer in support of New York City's finance industry I was exposed to automated trading systems. I learned that there are numerous individuals and companies of all sizes that use software to automate large parts, or even the entirety, of their decisions regarding buying and selling securities. In New York City the orders placed by the systems frequently run into the millions of dollars, making trading software an essential part of businesses with a lot of cash.

4.1 Vested Architecture

Vested has been designed in accordance with best practices as defined by Microsoft's Patterns and Practices group and implemented within their Enterprise Library product. They produced a set of reusable components that address common software development concerns including logging, exception handling, data access, caching, security, cryptography, and validation. Vested relies on the second version of these Microsoft components to enable loosely coupled data access and object caching.

When appropriately leveraged, these components provide immeasurable benefit to the developer by reducing the amount of code the developer must author when code accesses data stores. In addition, Enterprise Library code is, perhaps arguably, less error-prone and more scalable than code any individual developer might write, in part because of rigorous testing internally at Microsoft and in part because the patterns were culled

from the minds of many highly regarded developers following repeated successful implementations. I believe these components, because they have been under development within Microsoft for years, represent a bridge to the future of Microsoft-related software development.

4.1.1 Caching

Vested implemented a custom caching mechanism that served as a wrapper around the Enterprise Library's Caching Application Block. This caching system allowed Vested to store business-layer objects, such as orders, securities, and users, in memory so that they can be accessed faster. This wrapper functionality was implemented in Vested's EntityCache class and was configurable through Microsoft's Enterprise Library Configuration application, which is shown in. The importance of faster access was critical in this web-based application because data usually must travel farther before becoming available to the user.

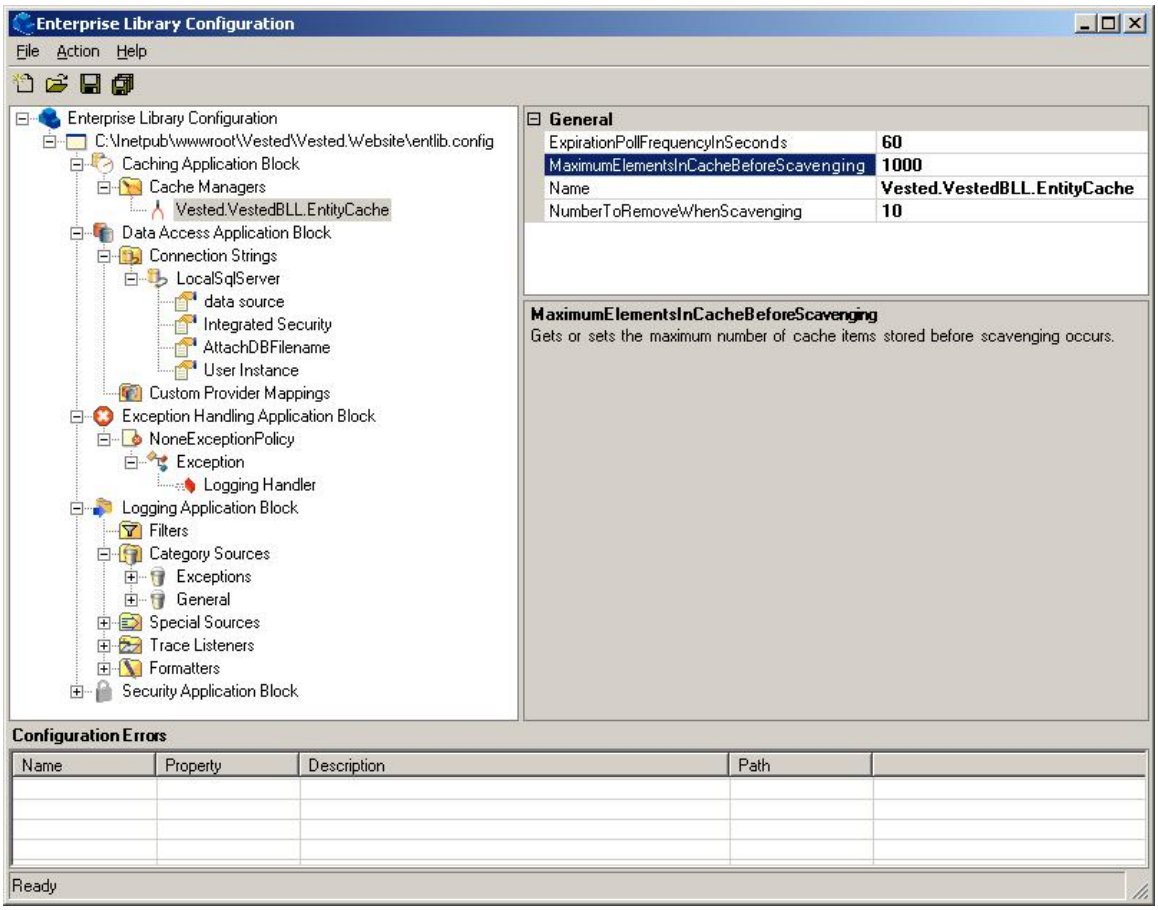


Figure 1: The Enterprise Library Configuration Application

Vested's caching system, similar in this regard to its data access layer, does have dependencies on other Enterprise Library components and could be considered a heavyweight object inasmuch as it provides some features that are not heavily utilized but whose code is still present. It must use the Library's Common and ObjectBuilder blocks to manage configuration and construction tasks. These heavyweight dependencies, however, enable the caching system to be solely focused on faster object access for the application. The use of factory and provider design patterns permits Vested's EntityCache class, which provides caching on an entity-by-entity basis, to use the Enterprise Library's CacheManager object for implementation of the most-common caching functionality needs, such as adding items, removing items individually, and

emptying entire caches. Because the CacheManager is designed to completely encapsulate the responsibility of handling cached objects, Vested can use it and is therefore insulated. Further, changes to cache implementation logic require modifications in only one place while client systems can continue to use the new functionality without changing themselves.

Most importantly, this mean that the developer does not have to write the code required to cache entities, but can instead implement interfaces and call pre-written methods that will perform caching for them while automatically remaining mindful of best practices and significant patterns. For example, the EntityCache class, when adding an entity to the cache, requires only one line of code, which is used to call the Enterprise Library's caching functionality. The EntityCache class does not have to consider all possible situations because the Enterprise Library handles those situations for it, including what if the entity is already in the cache. Microsoft's Enterprise Library code manages this situation for Vested by removing the original item from the cache and then inserting the item that Vested was trying to cache. It also guarantees a simple way of ensuring that the adding process completed by checking that the item Vested attempts to add exists within the cache after it was added. If it does not exist, there was a problem within the Enterprise Library-managed addition process.

Of nearly equal importance is that this framework code does not need its functionality tested before it can be deployed. It must be understood, but for the vast majority of uses at least, it has already undergone significant testing at Microsoft and is currently functioning in systems around the world.

Vested's framework of generated code, as output by the CodeSmith and NetTiers template seen in Figure 2, includes a set of default parameters that customize the implementation of the EntityCache class. Configurable parameters include the maximum number of objects that the cache will hold (1000 by default), the maximum amount of time any object will remain in cache (60 minutes by default), how frequently the cache should be polled to determine which objects are expired (once per minute by default), and the minimum number of items that should be removed from the cache at a time during scavenging (10 by default). The EntityCache class is declared static so that it is shared. It makes available functionality to add an entity into the cache, remove an entity from the cache, or retrieve an entity from within the cache. Callers can pass in a string identifier that uniquely identifies an entity in order to manipulate the cache in regard to that individual entity.

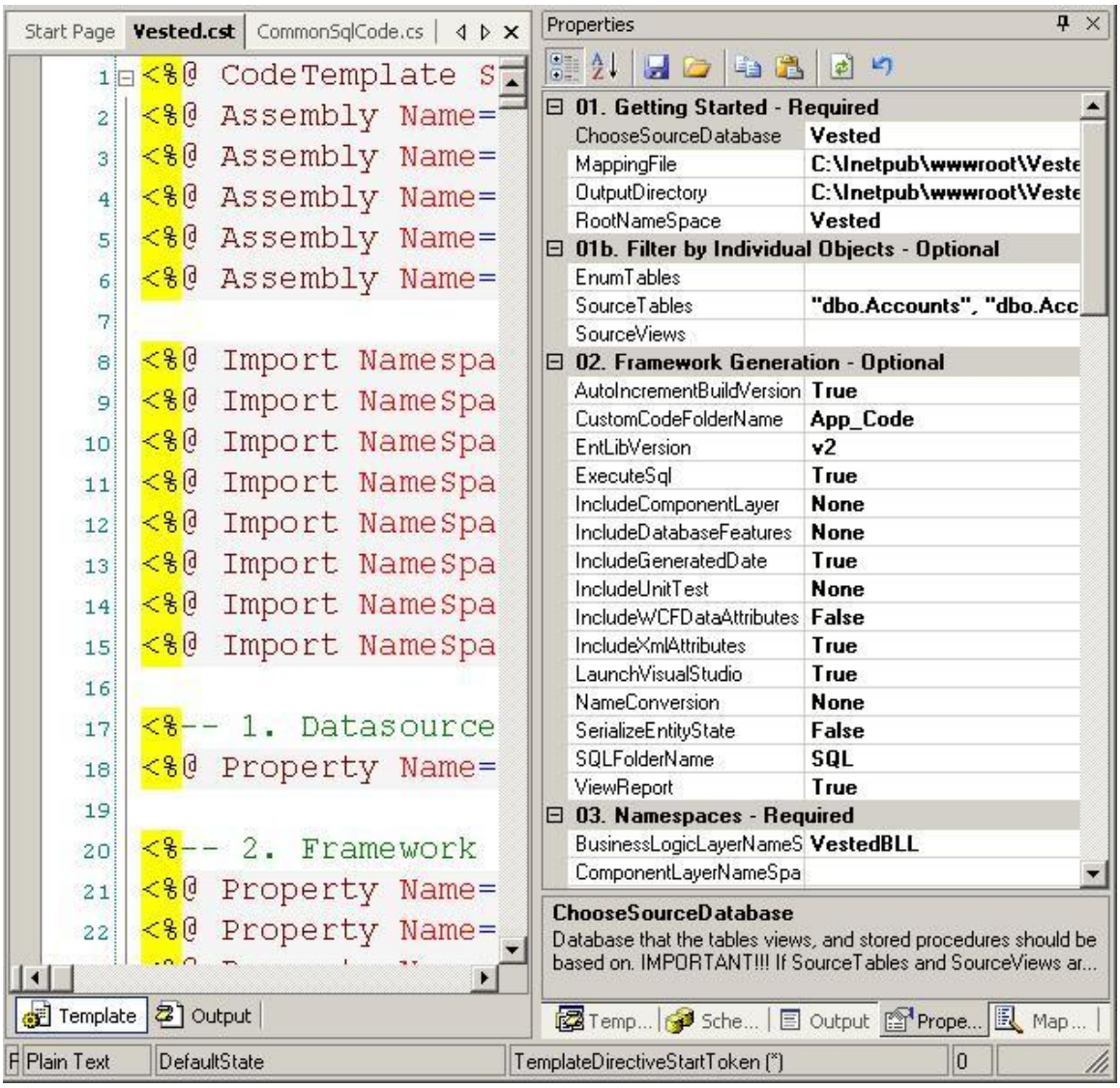


Figure 2: CodeSmith and NetTiers Template

4.1.2 Security

Vested provides a security feature intended to minimize the likelihood of a successful SQL injection attack. This feature was implemented through the construction of a regular expression that checks non-parameterized user input for security-sensitive SQL strings, such as “grant”, “exec”, “sysuser”, and “--“.

Vested maintains awareness of end users through the implementation of Users, Groups, and Permissions object while recording a lightweight audit trail via standardized

create and modify user and date columns. Each table in its data model includes CreateDate, CreateUser, ModifyDate, and ModifyUser columns. By not allowing these columns to contain null values at the database level, Vested enforces an audit policy that records who created and changed orders, securities, and other database objects.

This is a simplistic audit system in that it does not keep an ongoing history once an item is created. Where a full-featured audit system would retain information about each individual update to an item in case items are updated more than once, Vested retains only the latest update information. For example, if a security is changed twice, Vested does not retain information about who performed the first change and when; only the most recent update's audit information is available.

Mindful of the clichéd mantra of knowledge being power, Vested was developed with the guiding principle that giving users access to their OMS information is to give them power. Because complexity in software can often obscure intentions, every effort was made to keep the user interface simple to use, even if the code base of the system behind that interface was extensive and complex. While the system is ripe for customization in terms of roles and authorization, its immediate implementation gives all users access to all data.

4.1.3 Data Access

Data access is enabled through calls into the Enterprise Library's Database object. The most common of these calls are ExecuteReader, ExecuteNonQuery, ExecuteDataSet, and ExecuteScalar. When requirements dictate that data be returned, as when the user is viewing a page of orders or securities, Vested uses ExecuteReader almost exclusively because it performs significantly faster than ExecuteDataSet. ExecuteReader returns an

object that implements the IDataReader interface while another utility class provides a function generic and capable enough to take any object that implements IDataReader and convert it into a DataSet.

Vested's data-related capabilities include utilities for generating SQL statements that are used to interact with the database. These features include the ability to dynamically query the system using like, not like, contains, starts with, ends with, and null values to mention a few. Applications often offer dynamic where clause building so that their databases can be searched for an item with an id number of 5 using the same code that could alternatively search for an item with an id number of 10, 50, or any other integer. Vested's framework takes this concept substantially farther in that it offers the developer the opportunity to dynamically query any table in the database based on values in almost any column of that table (Vested uses neither).

Access to the application's data begins with a request for a web page in the system. Convention is relied upon heavily, with pages relying on naming conventions to establish associations with database tables, such as the orders and securities tables. To request the orders page is to request a list of orders, which is essentially the same as requesting multiple rows of data from the orders table simultaneously. Beginning with the web page, the request is forwarded to the appropriate domain entity (the orders entity in this example) in the application's business layer. The domain object passes the request along to a provider class, and the request is tranformed into a data-friendly representation via stored procedures that actually get the data from the database.

4.1.4 Business Layer

Factory patterns that define how objects are created are central to Vested's object-oriented design. The EntityFactoryBase class provides entity creation functionality that is generically used to manage the creation of each distinct type of object in the business layer. Every table in the template data source's database is generated into a business layer class and is considered an entity. Each of these entities is created through the EntityFactoryBase class's Create methods, which manage performance optimization by maintaining a list of string parameters that are used to determine which type of entity is created. As discussed in Design Patterns Explained, abstract factory patterns address the problems of combinatorial explosion, unclear meaning, and creation logic while avoiding both tight coupling and low cohesion (Shalloway, 2007).

The business layer also includes management, location, and comparison functionality. Management features are made available through the EntityManager class, which provides overall management of entities, including creation and entity state tracking so that Vested is aware when an entity's current state is unchanged, added, changed, or deleted. Location functionality includes storage of objects in a weakly-referenced dictionary collection while comparison features include a Compare function that implements the Icomparable interface's requirements, returning 0 if two objects of the same type are equal.

4.1.5 Class Definitions

Though the framework used to create Vested included web service and Windows Forms generation capability, these were excluded from the solution's implementation. While those features may be included in future revisions, the current architecture follows a traditional 3-tier structure in which the user interface, business-specific logic, and data

manipulation logic are separated into physically distinct components that each encapsulate necessary code and provide generic functionality. The business layer is organized with the Vested.Entities namespace and the data layer resides within the Vested.Data namespace. A list of the external and internal components Vested reuses is shown in Figure 3.

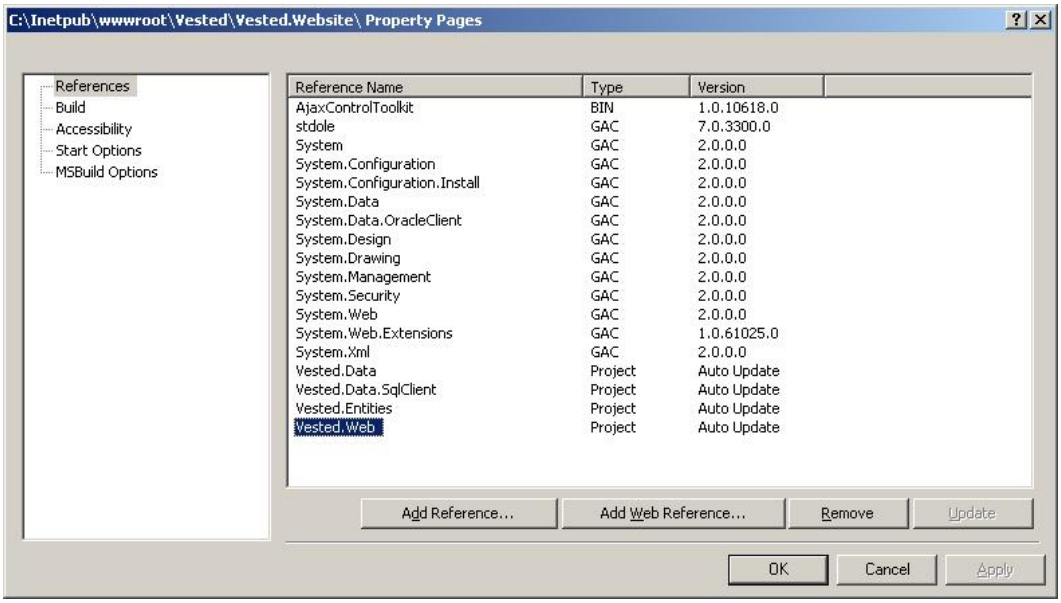


Figure 3: Vested Web Site External Dependencies

Vested uses an extensive class and interface hierarchy to enable data flows, 3-tiered separation of concerns, and loose coupling in the context of an OMS system. The top-level class in this hierarchy is referred to as an entity. In fact, it is technically named EntityBaseCore, but for simplicity's sake, it is referred to as an entity. All other business-layer objects, such as securities, orders, or user objects, are also entities because they automatically extend the entity class. This hierarchy includes generated and non-generated abstract, partial classes. It also includes definition and implementation of at least one interface for each business class. The order object, for example, implements the Iorder interface, whereas the security object implements the Isecurity interface. The

Orders class inherits from the partial and abstract OrdersBase class, which in turn inherits from the partial and abstract EntityBase class, which in turn inherits from the partial and abstract EntityBaseCore class. Figure 4 illustrates Vested's organization into functionally and physically distinct layers separated into namespaces.

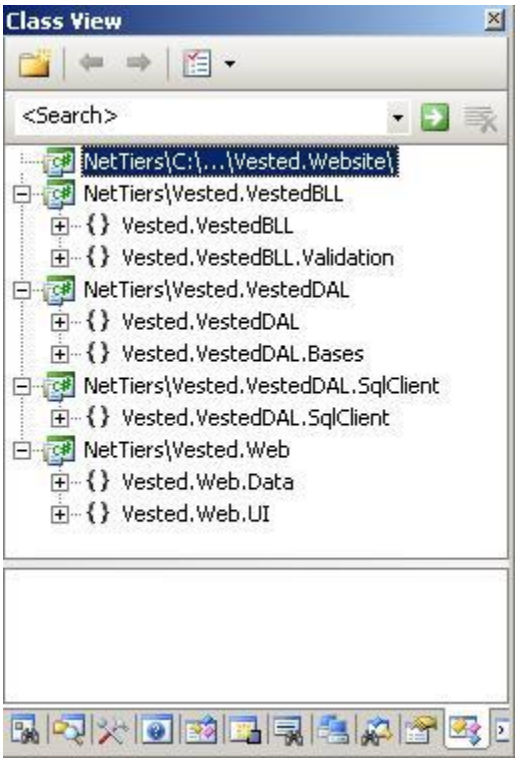


Figure 4: Vested Namespaces

The EntityBaseCore class in turn implements the IEntity, INotifyPropertyChanged, IDataErrorInfo, and IDeserializationCallback interfaces. The OrdersBaseClass implements seven interfaces: Vested.Entities.IOrders, IEntityId<OrdersKey>, System.IComparable, System.ICloneable, IEditableObject, IComponent, and INotifyPropertyChanged. The files composing this highly standardized architecture across the entire application are largely generated, but the architectural

implementation intentionally leaves the partial Orders class untouched whenever the main code base is regenerated to accommodate database changes.

Also of note is the fact that the architecture is customizable at numerous levels. Vested customized the architecture to include implementation of the IEntityCacheItem interface at the business class level so that business objects such as securities and orders could be cached. When an instance of a business-layer class is located or created by the EntityManager class, the EntityManager will first check if the object being sought implements the IEntityCacheItem interface. If so, it will look for that item in the cache before attempting its retrieval and construction.

The user interface employs master pages to standardize the look, feel, navigation, and functional aspects of the system's appearance. Composite controls, which are those including both visual and logical features, were used to build filtering and searching capabilities into each page's data grids. FormView controls were used within user controls and included type-appropriate input validation and automated checking for required fields. These user controls were then embedded in their related pages. Minor customizations of these controls allowed for implementing various filters for some pages, but not for others. An example of this can be seen in the orders page's (shown in Figure 5) from and to date fields. This page's instance of the composite filtering and searching control was customized to allow users to search orders over specific date ranges.

The code generation system's features for retaining customizations across multiple regenerations of a new version of the system were referred to as its "merging strategy". Using a simple string-based naming convention within the application's code files allows the developer another method to preserve customizations. The code

generation system “merges” freshly generated code with the customized named regions the developer provides. Two different customization options, the aforementioned partial classes and named regions, allow the developer to customize both the user interface and the system logic.

4.2 Using Vested

When entering orders, the process most central to normal system use, the user must input all required data, and will be prompted with bold red “Required” messages next to all required fields that are still unfilled when the insert button is clicked. Required fields for order entry are defined by business requirements. For example, it is not possible to execute a trade without both price and quantity, irrespective of the trade’s other details. That real-world business rule is modeled within Vested by requiring the user to provide data for both fields, which are each set to disallow the null value in Vested’s data model. This synchronization between business realities, data modeling, and interface is essential to the system. Additionally, users will automatically be prompted about data type incompatibilities. If the user tries to insert or save changes that set a date field, such as order date, to a plain text string like “tomorrow”, the system will disallow the insert or save and inform the user of the problem.

The orders screen also enables users to quickly see all orders for a given security or all orders of a given type by clicking on the security’s name within the order’s row or the order row’s order type id column (See Figure 5).

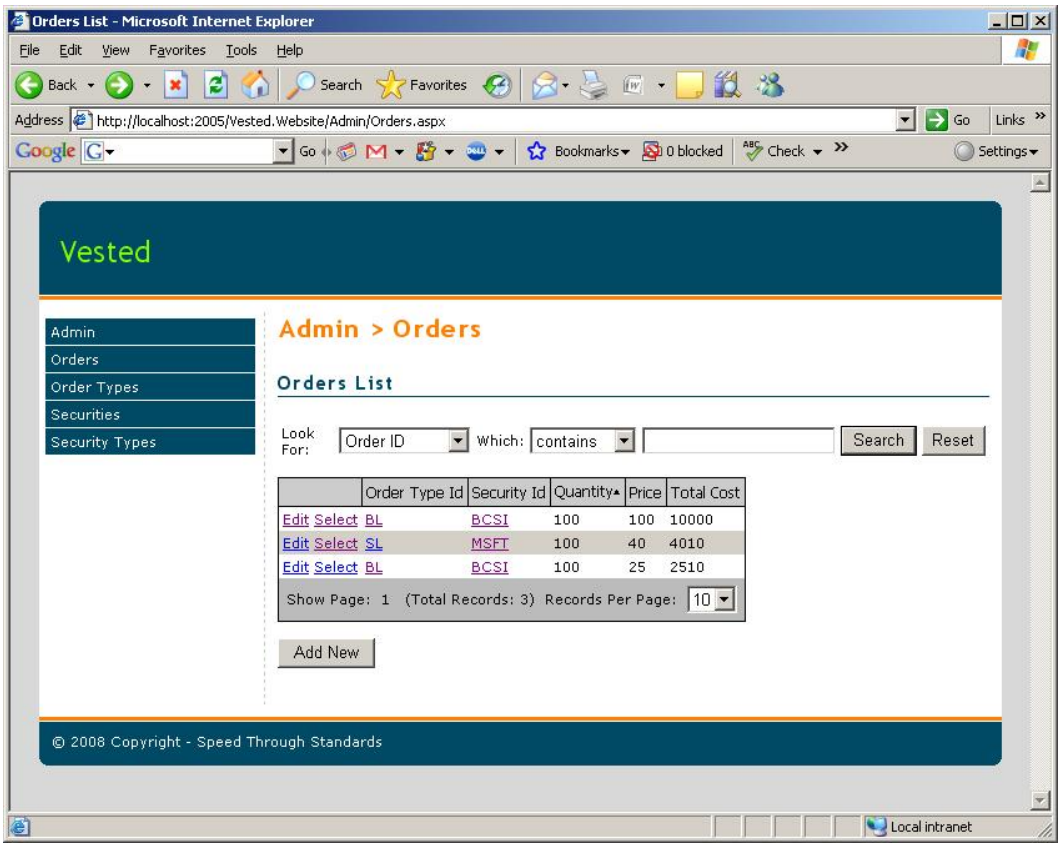


Figure 5: Orders List Sorted by Quantity

Orders list, as with all list pages within the application, provides built-in sorting functionally for all columns that are stored in the list's related database table; therefore, the orders table is related to the orders list page. All list pages support searching Vested's database. Searching for an order is enabled across every order table column (OrderTypeID, SecurityID, Quantity, Price, TotalCost, etc). Users may search for securities, orders, other users, groups, or any other piece of data stored by the system. Further, search functionality is parameterized so that every table column can be combined with each of the four search types: contains, starts with, ends with, or equals. This gives users the freedom to search in numerous ways. Because some types of data within SQL Server are less searchable than others, only columns that have the following

data types can be searched: ANSI strings, Boolean, byte, currencies, dates, decimals, doubles, integer types, and xml.

Foreign key columns have navigational features that provide uniform functionality across the entire system. When a user clicks on either the Security or Order Type Id columns, they are sent to edit pages that then retrieve primary key values from the query string. Edit pages use those primary key values to query the system for the details of one security or order type identified by the passed key value. The interface presents those details to the user for viewing and editing.

Figure 6 illustrates the result of clicking the "BCSI" link in the orders list page. Navigational features bring the user to the securities - add/edit page where details regarding the BCSI security can be viewed or edited. The page, using the database's foreign key from the orders table into the security table, is capable of simultaneously displaying orders for the security being viewed or edited within a collapsible frame (the frame labeled "Orders Details" in Figure 6). Edit pages system-wide, as in the securities add/edit page pictured, permit the editing of one record at a time.

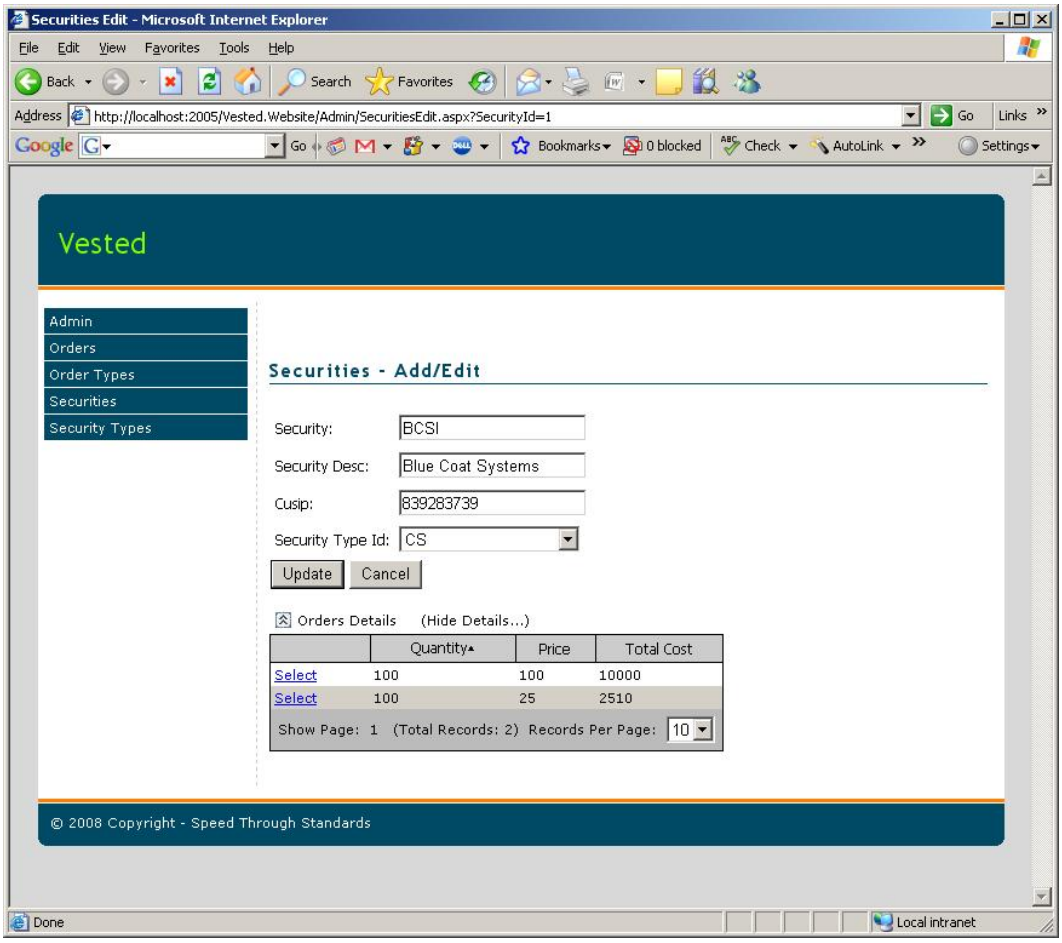


Figure 6: Securities – Add/Edit

Standardized grids are used throughout Vested to provide users with a common look and feel that they are likely to be familiar with already, given the wealth of spreadsheets found in the financial services sector. These grids allow for sorting and navigation and have customizable records-per-page settings. Every user can have his or her preferred number of results shown onscreen by manipulating the records per page value in each grid's lower left corner.

4.3 History and Reflection

At the project level the initial ambitiousness of my interests was the first thing that went wrong. I am still debating whether it was a good event or not that I chose to use .NET 2.0, with which I was almost completely unfamiliar, as the platform upon which to build the application. I had been using .NET 1.1 only for years. However, that one small choice opened up a door of possibilities that would not have been available in such a powerful form had I not been working within Microsoft's Visual Studio 2005 with .NET 2.0.

The most significant of these positive occurrences related to platform choice was my discovery of code generation. This capability of modern programming frameworks allowed me to simultaneously discover, experience, and learn new material related to automated unit testing, object modeling, aspect-oriented programming, presentation, RAD, and many more fast-changing but incredibly powerful software design concepts. This project was managed primarily through a state-of-the-art remote interface into Regis University, partly through iterative discovery, feedback, and development between myself and Dr. Doug Hart, and partly by sheer force of my own will to continue managing and motivating myself to devote time in pursuit of only the highest-quality productive effort. It was only possible because I was able to take the lessons I learned at work each day and put them to use in this work immediately upon arriving home.

The first milestone in the project was literally the final determination of what system to build. Originally, I was very interested in systems that could not only make buy and sell decisions in place trades but could also automatically adapt, or learn, when

marketplace dynamics changed. Realizing the ambitiousness of that undertaking, I considered implementing a system that would allow me to track my own investment purchases and sales online, from anywhere at any time. Thinking that such a system might be small in scope and that having something more substantial might actually help my career considerably, I ultimately decided to apply the lessons I've learned while working with programming code on four separate OMS over recent years.

Subsequent milestones included the finalization of the data model and the realization of the potential impact of code generation, not just in this project but in the sense that, if mastered, it could be a phenomenally productive skill. Significant project changes resulted from almost every piece of feedback that I received from academic peers. As I learned more about the industry while at work and through several nonfiction and textbooks, smaller changes to the existing project plan occurred.

The project undoubtedly meets its primary goal which is to manage orders. Orders can be entered, viewed, changed, deleted, and searched. The system accurately models the appropriate relationship between orders in the securities that those orders purchase or sell. It also links orders to accounts, accounts to funds, and people to all of the above. Orders are handled in accordance with generic and generally accepted methodologies dictated by the type of security related to the order.

Chapter 5: Lessons Learned and Next Evolution

I learned so much over the course of this work that it is actually quite difficult to recall everything, begging the question "If one cannot remember it, can one really say one has learned it?" In the programming technicalities domain I covered everything from aspect-oriented programming, unit testing via assert statements, generic types, null able types, master pages, interface usage, numerous design patterns including their rationale and application, serialization, object data sources, object relational databases, high-speed .NET programming, scaffolding, weak references and garbage collection, configuration and convention, source control, agile project management, service-oriented concepts, and many more.

In the finance business domain I covered margin accounts, simple and complex derivatives, securities lending, leverage, accounting, legal structures, currency conversions, hedge funds and fund-of-funds, master-feeder structures, and partnership interests. I became comfortable with most so-called alternative investment strategies: distressed debt, equity market neutral, merger arbitrage, and statistical arbitrage to name a few.

Meanwhile, the daily workplace taught me people skills, prioritization needs, truly *rapid* development skills, and forced me to realize that people are the bridges between the external businesses and internal departments that technology systems support. My experience and my coursework combined to give me project management skills while an unimaginably demanding work environment managed by a stereotypically superhuman multitasked resisted my inclinations toward formality, standardized development, and process documentation.

I found myself working with people whose real lives were dramatized in books I was reading and movies I was seeing. The deals they made literally were the stuff of legends. It was my fascination with the paychecks they had been pocketing that led most directly to all of this learning; there is much to be said for inspiration just as much knowledge and material comfort can be earned from hard work. I believed in myself, in my ability to complete this, and in so doing I matched inspiration with potential.

5.1 Lessons Learned and Learning Curves

If I had it to do over again, I would not take a two-year hiatus from the Master's degree program as I did from about 2004 through 2006. I unexpectedly moved across the country in pursuit of love a year or so after starting the program. I would move for the same reason again, but I would not let it break me from my studies. I also would have adopted the cutting edge much sooner. If I had known the capabilities of an upgraded development environment when paired with judiciously selected plug-ins like CodeSmith, NetTiers, and ReSharper, I would have begun exploring programming's state-of-the-art sooner. In the final analysis, my only regret is that I did not have more time – though I will soon – to become familiar with the specific code generation tool I chose to use for this project. When I discovered it, I had literally just finished building out a large system for my employer over the course of nine months. Had I known of and used code generation, that new system could have been done in two months. The power of productivity and efficiency gains made possible by intelligently leveraging other people's work are amazing.

Another lesson I learned early on was to strip out the details of a system during initial development. Focusing on the core components, needs, standardizations, customizations, and use cases only really helps discover the needs a system must meet. Initially, I was faced with a system I generated at home with hundreds of files in it and multiple layers. The scope of its features and the sheer amount of generated code were daunting. I discovered how helpful it could be to create another version of the Vested database, giving the new copy only two or three tables. This would reduce the amount of generated code substantially and allow my time, analysis, and learning curve to focus on the architecture, the what, and the how, of all the new technologies I had discovered.

Learning how to customize the system, which was every bit as time-consuming as just figuring out what it basically did, how, and why, I found it helpful to generate a system from only a few tables so that I could analyze the preservation of one-to-many and many-to-many relationships in the data access layer and debug the generated system's lifecycle. It was this hands-on approach, as much as referring to documentation that made the project possible. There are no books on how to use most of these tools and you cannot take a course in developing with them. The only way to learn is through intensive long-term study and actual use. I have spent literally months studying what these sorts of frameworks make possible; I could have shortened that timeframe substantially by focusing on small core pieces initially instead of attempting to build an entire system all at once.

The project did not meet my own initial expectations, because I had conflicting ideas about what the project should be. I thought that as long as I was already spending the time doing something, I might as well try to make it something I would use. That

logic originally had me considering a system that would record just my own trading information so that I could more accurately track the performance in my own investment accounts, which had been doing quite well for a number of years. I took the initial thought process further, rationalizing that I may as well make it something that other people could use. You never know, maybe somebody would buy OMS software from me!

I have put so much time into this effort in total. Endless hours of typing followed endless hours of programming followed by bed and then repeated ad infinitum. And it hasn't been just the programming and typing; I must have spent $300 on extra-curricular textbooks from the finance arena and easily a couple hundred hours reading all of them. This project is the apex of something that has not been just about my reading books or following daily financial news from Bloomberg and the Wall Street Journal, it is the culmination of what my life has become over the past few years of living in New York City. The number of hours seems countless, making the number of years the only meaningful metric. This project will not end here because I have invested my life into it and it is my product.

The next steps will be to enhance reporting capabilities and add more security types. Fundamentally, these are the only steps necessary before this software could be marketed as a commercial OMS. I have considered changing some of the .NET framework's controls within the software into more robust controls from third party vendors, such as the data display grids available from DevExpress, Infragistics, and Xceed. I have not decided yet to implement third-party grids, but I would like to for their

features – the drawbacks are the learning curve and the cost but the payoff of sleek and highly customizable interfaces may decide it.

In conclusion, I strongly recommend that individuals in general make a habit of investing their own money. I have come to believe that it is possible, without really a whole lot of work, for people to affect their own return on investment to the extent that they retire considerably sooner than they otherwise may have. I believe it is possible to invest in a manner that yields returns reliably higher than standard benchmark indexes, higher than mutual funds, and as high as the best hedge funds, and I believe these things are possible for individual and institutional investors alike, whether they use an OMS, Ameritrade, Schwab, or any other mechanism that grants access to markets. I also strongly advocate that individual software developers and development teams make adherence to proven best practices and pattern usage standard throughout their systems. I have become a vocal advocate of object-relational mapping systems and code generation. For the individual developer interested in furthering his or her development skills and career, these two technologies seem an excellent place to start.

References

Anquetil, N., de Oliveira, K.M.., de Sousa, K.D., and Dias, M.G.B., Software maintenance seen as a knowledge management issue, Information and Software Technology Volume 49, Issue 5, May 2007, Pages 515-529.

Bowen, I. (2007, November). Mathematicians' role in market mayhem. Retrieved December 12, 2007 from http://news.bbc.co.uk/2/hi/business/7109805.stm

Davis, M. (2006, July 19). Scale Up vs. Scale Out [Msg 1]. Message posted to http://weblogs.java.net/blog/malcolmdavis/archive/2006/07/scale_up_vs_sca.html

Free On-line Dictionary of Computing (2007, December). Scalability. Retrieved December 4, 2007 from http://foldoc.org/index.cgi?query=scalability&action=Search

Greenblatt, J. (2006) The Little Book that Beats the Market. Hoboken: John Wiley and Sons.

Investopedia Stock (n.d.), Retrieved August 20, 2007 from http://www.investopedia.com/terms/s/stock.asp

Microsoft's .NET Framework General Reference Naming Guidelines. (n.d.). Retrieved January 9, 2008, from http://msdn2.microsoft.com/en-us/library/xzf533w0(VS.71).aspx

Pfleeger, S.L. (2001). *Software Engineering: Theory and Practice* (2nd ed.). Upper Saddle River: Prentice Hall.

Pigoski, T.M. (1996). *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Hoboken: John Wiley & Sons.

Swanson, E. B. (1976). *The Dimensions of Maintenance*. In Proceedings of the Second International Conference on Software Engineering, pages 492-497, San Francisco, October 1976.

Yan, W., & Clack, C.D. (2006). Behavioural GP Diversity for Dynamic Environments: an application in hedge fund investment. *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (pp. 1817-1824). New York: ACM.

InVested:
Unbreakable Software for Unpredictable Markets

InVested is an order management system (OMS) for trading common securities.

Daniel Mark
Regis University
May, 2008

7 Habits of Highly Successful People

Habit number one: Put first things first.

If it wasn't for people like you,
people like myself and my class mates
might not have the choice available to us to go out and further ourselves, to exercise and develop our minds.

It took more patience and dedication over time
than almost anything I've ever done, and I'm sure teaching
takes the same from you every day.

Thank you many times over.

Table of Contents

- Introduction
- Users Perspective
- Methodology
- Architecture and History
- Lessons Learned
- Main Features
- Workflow: Security Setup
- Use Case 1: Order Entry
- Use Case 2: Reporting
- Filtering
- Calendar
- Remaining
- Conclusion

Introduction

- Recent market turmoil
- Background with OMS
- Market size

Users Perspective

- Security type: comprehensive
- Profit and loss
- Performance
- Low maintenance

Methodology

- Iterative SDLC
- Code generation
- Maintenance focus

Architecture and History

- Architecture:
  - Standard 3-tier
  - Inheritance: type hierarchy
  - Interfaces: composition
  - Composite controls
- History: Program trading to OMS
  - Learning automation – a program whose automated buy and sell recommendations become more effective over time.

Lessons Learned

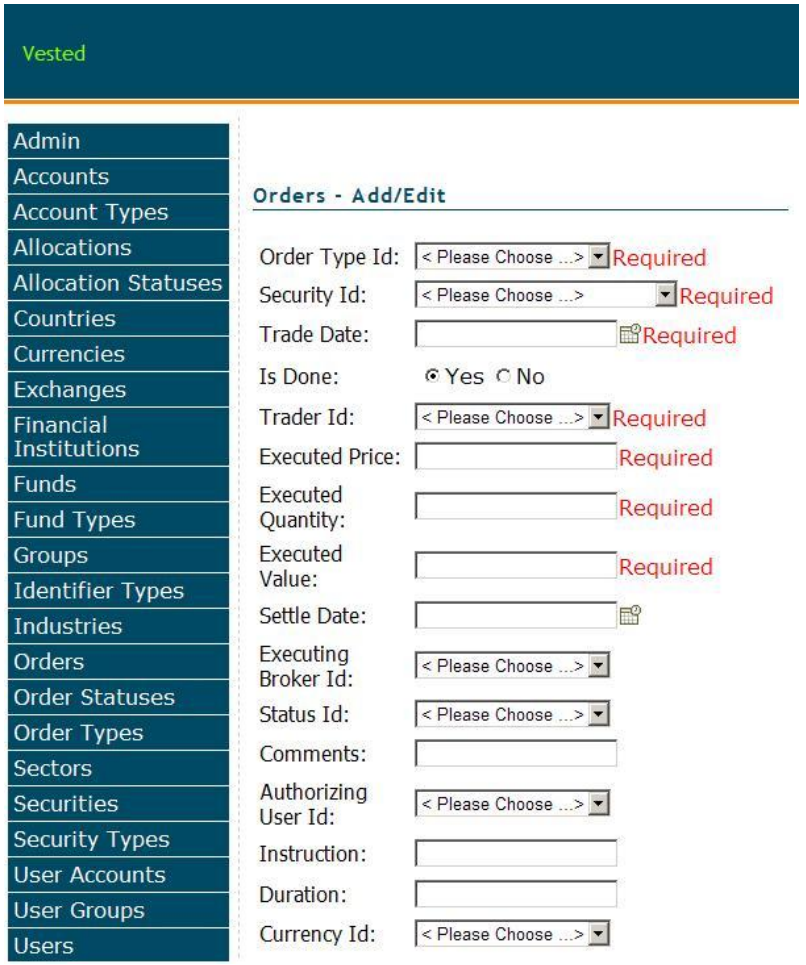
- Initial focus
- Requirements gathering
- Scope
- Feature creep
- Cutting edges

Main Features

- Order entry with data validation and protection
- Filtering
- Usable and responsive – as with Ajax calendar
- Reporting
- Use of Enterprise Library
- Available as a web service
- Unit testing automation

Order Entry

- Add or edit
- 7 Required fields
- Tab indexing
- 10 supported security types

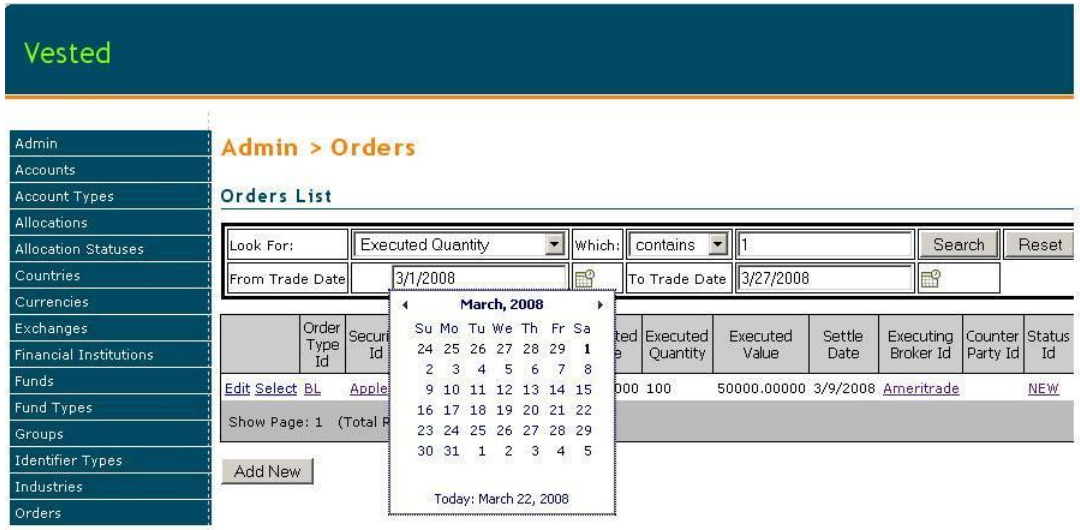


Filtering



- Filters: by trade date
- Search: across all columns
- Criteria: contains, starts with, ends with, equals
- Enhancement: from and to date for every date column

Calendar



- Responsiveness and validation
- Leap years correctly
- Significance within finance – business rules and dollar amount calculations rely heavily on dates

Reporting

- Export to excel
- Filtering ++
- Aggregation
- Better grid

Workflow: Security Setup

- Sequential process (security must exist before trades can be placed with it)
  - Set up new security (enter its data) with minimum required fields
  - Use the security in an order
- Security Types
  - Stock: common, restricted, preferred
  - Bond: corporate, government, municipal
  - Call Option
  - Put Option
  - Future and Forward
  - Credit Default Swap
  - Equity Swap
  - Interest Rate Swap
  - American Depository Receipt
  - Exchange-Traded Fund

Use Case 1: Order Entry

- Order entry
  - Required fields dictated by type of security being bought/sold
  - Automatic calculation of values, e.g. price x quantity = value
  - Permanent record associated with user, date, and security

Use Case 2: Reporting

- Orders
  - Given security, date range, security within a date range
  - Orders by user/trader
- Profit and loss
  - By fund/account
  - Daily, monthly, year-to-date
- Exposure and risk
  - Exposure by investment, sector, industry
  - Leverage, investment versus hedge

Remaining

- Refactor
- Navigation
- Reporting
- Calculated columns
- Security types
- Bits
- Process flows
- Sign up customers!

Conclusions

- Difficulty level
- Invest
- Churchill: Don’t stop