Spring 2007

Model-Driven Software Development

Susan Minton
Regis University

ACKNOWLEDGEMENTS

This thesis is dedicated to my husband and children who, in many ways, had to persevere with me through all of my educational endeavors. They truly are my inspiration. I would like to thank my advisor Dan Likarish for all his support and guidance throughout my Masters program and the thesis process. Two people deserving special mention are my parents. They always encouraged me during school, and taught me to strive for excellence in everything I do in life.

ABSTRACT

“Model-Driven Software Development – Techniques and Case Study”

Model-Driven Software Development (MDSD) is an emerging technology approach that has potential to revolutionize the software industry. MDSD has the ability to both increase software delivery velocity, while at the same time reduce complexity and reuse software assets. Experts in the field believe that the MDSD approach helps to abstract away the growing interdependencies of enterprise software development by use of sophisticated tools, models, and automatic code generation. Through the use of Unified Modeling Language (UML/UML2) and other related technologies, the models are intricate enough to fully represent a system domain and then generate system code to represent that system. The case study evaluates the key factors of velocity, modeling complexity, code generation, and code completeness. Using both Model-Driven Software Development and so-called traditional methods of development, both techniques were applied against a real-world system for First United Methodist Church Children’s Ministry. The two techniques were measured and critiqued for their effect on the software development. Future direction of MDSD and potential impacts are presented.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES / EXHIBITS

INTRODUCTION

The software industry represents a significant cross-section of our global economy. As software delivery demands increase, so do development costs. Complex software architectures and remote development teams can lead to delivery challenges such as code duplication, productivity losses, and implementation uncertainty. The key to improving software delivery in the future is to significantly increase system clarity and delivery velocity while at the same time driving down both development complexity and costs. Model-Driven Software Development (MDSD) can accelerate software delivery and reduce development costs by supplying concise domain models, reducing complexity, increasing system component reuse, and automatically generating code.

In the next few years, many challenges lie ahead in the software industry. The largest challenge is growing software complexity. Highly complex architectures require extremely skilled engineers. The high cost of these skilled software engineers is compelling companies to cut corners, like understaffing or off-shoring which contributes to project complexity. Additionally, complex architectures require prolonged learning curves and ramp-up time. All of these factors lead to lower productivity and longer software cycles. Contrary to that, business expectations are now focused on quicker software deliveries with higher quality and lower costs. All of these trends are paralyzing traditional software deliveries.

Model-Driven Software Development offers a potential solution. It is defined as "a new software development paradigm for distributed project teams involving 20+ people. […] MDSD aims at developing software from domain-specific models. Domain analysis, meta modeling, model-driven generation, template languages, domain-driven framework design, and the principles for Agile software development form the backbone of this approach." (Bettin, 2004a). In a nutshell, MDSD is a new way to develop software by using a model centric approach, not code centric. Generated code becomes an artifact from the modeling and design stages. An engineer *programs* using models, not software languages.

MDSD has two primary goals. The first is to "improved manageability of complexity through abstraction. The modeling languages enable 'programming' or configuration on a more abstract level." (Stahl, Voelter, 2006). The second is to "increase the development speed through automation: runnable code can be generated from formal models using one or more transformation steps." (Stahl, Voelter, 2006). The basic concept is called forward engineering: producing code from abstract, human-elaborated specifications (Wikipedia, 2006).

Model-Driven Software Development has the potential to revolutionize software engineering. MDSD separates the business and application knowledge from the underlying solution technologies - no more backward compatibility or new technology churn issues. From CIO's, to system architects/designers, to software engineers, all can begin envisioning business systems through architectural domain models instead of segregated application systems. Domain modeling leads to unified and standardized system implementations, as well as increasing software modularity and reuse.

The significance of MDSD is simplicity: addressing software complexity through abstraction using models. If you reduce the complexity, you reduce costs and increase delivery velocity.

MDSD faces industry inertia. The transition to Model-Driven Software Development is similar to the transition assembler programmers made when moving to third-generation languages such as COBOL. The 3GL's insulated developers from the instruction sets of specific processors and memory. (Bettin, 2004b). It was a leap of faith for programmers to no longer control machine level code. But in contrast, "we are all model-driven developers. When we write programs in Smalltalk, Java, or C#, we don't expect it to execute directly. We expect it to be transformed into the language of some machine that can cause our model [program] to do its job." (Mellor, Clark, Futagami, 2003).

In order to demonstrate the MDSD paradigm, the applied study utilized the Model-Driven Software Development approach. The study encompassed the modeling, design, and construction of a church ministry tracking system to demonstrate MDSD techniques. Effort measurements and observations were logged throughout the development of the system.

The thesis is structured into three major sections: framework, execution, and analysis. The first section provides context for the topic of Model-Driven Software Development techniques, impacts, and expert opinions. The next section addresses the actual MDSD case study. It describes the system background, the approach, and outcome using MDSD tools and techniques. The last section examines the case study, focusing on key issues: complexity, delivery velocity, code generation, and reuse. It provides a discussion on issues, an evaluation of Model-Driven Software Development, and the potential impact on software deliveries.

MODEL-DRIVEN SOFTWARE DEVELOPMENT

MDSD Distilled

Model-Driven Software Development is a new approach to software development that aspires to develop and generate software from domain-specific models. What does that mean? From a simplistic perspective, it means that software engineers will program using detailed models to represent the system rather than coding with computer languages. But MDSD goes much deeper than that. It consists of highly focused model development involving domains, transformations, domain template languages, and eventually code generation. MDSD origins began in domain engineering or "software product line engineering, which is the discipline of designing and building families of applications for a specific purpose or market segment" (Bettin, 2004b). Domain engineering advocates that productivity gains can be achieved during system development when developers leverage the same domain-specific knowledge.

Because MDSD is a relatively new discipline, there are few recognized experts in the MDSD field. Jorn Bettin and Markus Voelter are the primary published experts. Together, they are published or are quoted in almost every MDSD textbook and article, including this paper. Most of the recent advancements in the field are also authored by them.

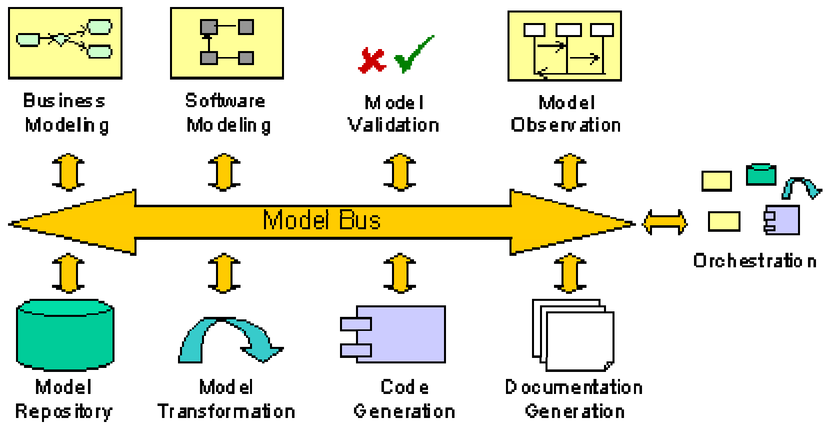
So, what is the promise of MDSD? According to Voelter, the goals and value proposition of Model-Driven Software Development are listed in table 1.

Table 1: Model-Driven Software Development value proposition (Stahl, Voelter, 2006)

MDSD Goals
1) increase your development speed
2) enhanced software quality through the use of automated transformations and formally-defined modeling languages
3) implementation aspects can be changed in one place
4) higher level of reusability and makes expert knowledge widely available in software form
5) improved manageability of complexity through abstraction [models]
6) building a productive environment through the use of process building blocks and best practices

Model-Driven Software Development is rooted in very recent technologies. MDSD stems from object oriented analysis and design. Most of the modeling is based on UML/UML2, MOF, and OCL technologies. The methodologies are centered on Agile or RUP SE, and the transformations and interoperability use XMI. See figure 1 for a simplified overview of some of the components involved with MDSD.

Figure 1: MDSD components (Eclipse, 2006)



Models specifications were standardized by the Object Management Group (OMG) for Model Driven Architecture (MDA). Both MDA and MDSD support the use of models for system development. The primary distinction between MDA and MDSD is that MDA is a set of guidelines for models; MDSD represents the entire software lifecycle from model to code generation. "[Model Driven Architecture] MDA is not to be confused with Model-Driven Development (MDD), also known as Model-Driven Software Development (MDSD). MDD is an approach to software development where extensive models are created before source code is written or generated. MDA is the OMG implementation of MDD. The MDA concept is implemented by a set of tools and standards that can be used within an MDD approach to software development." (SEI, 2006). So, MDA is a specialized subset of MDSD. The original MDA standard does not directly address the transformation and generation phases. OMG is addressing them in a separate initiative called QVT. For MDSD, domain-specific transformations and generation are vital stages.

Because MDSD is based on new technologies and new approaches, it also introduces new terminology. *Domain Engineering* models the overall business context and *Application Engineering* models the actual platform and system context. The latter is much closer to traditional UML modeling. *Domain-Specific Language* (DSL) is the template modeling language, semantics, and syntax used for expressing key aspects of the domain. See figure 2 for a sample of DSL. *Platform Independent Model* (PIM) is the model describing the business logic, undiluted by technical concerns. *Platform Specific Model* (PSM) is the model used to describe the actual platform implementation.

Figure 2: Sample DSL Transformation Rule (SEI, 2006)

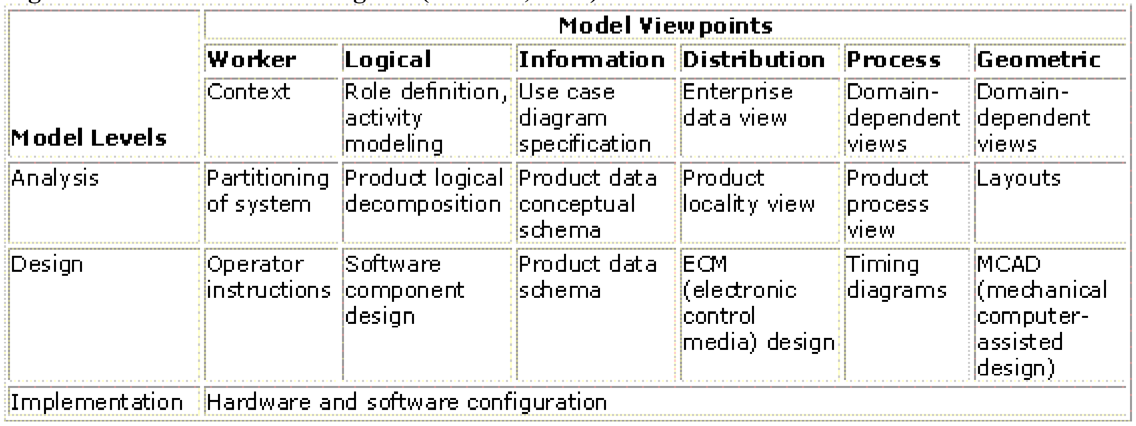
```
if (UMLClass) {


create Java class named <UMLClass.className>Bean.java
create methods in <UMLClass.className>Bean.java for each operation in UMLClass
create attributes in <UMLClass.className>Bean.java for each attribute in
UMLClass
create Java class named <UMLClass.className>.java for remote component
interface
create Java class named <UMLClass.className>Home.java for remote home interface
create Java class named <UMLClass.className>Local.java for local component
interface
create Java class named <UMLClass.className>LocalHome.java for local home
interface
…


}
```

Model-Driven Software Development tools and techniques are the key to the entire MDSD approach. The tools drive the iterative development processes and act as a repository for the domain knowledge. MDSD also has an established set of guidelines and processes for producing model to code transformation. There are basically two flavors, RUP SE and Agile.

Tools are the foundation for MDSD. In order to fully perform Model-Driven Software Development, there are three tool types you need – meta-modeling tool, specification tool generator, and model-based template interpreter and generator. There are many commercial tools available, but many of them are very immature and quite expensive for the average company or entrepreneur. Most tools range from $9,000 to over $250,000. Most open source tools are listed on the Generative Modeling Technologies (GMT) website (http://www.eclipse.org/gmt). The most notable tool suite is from Jorn Bettin called openArchitectureWare (oAW). Other vendors offering MDSD tools are Rational Software (Rational Software Architect – RSA), Gentleware (Poseidon), and Borland (Together Control Central). Some tool offerings represent a suite of integrated technologies, whereas others are suites of pieced together components to fulfill the entire MDSD modeling need. XMI acts as the interoperability backbone between tools.

The MDSD RUP SE approaches modeling from a system perspective. This approach models the system from a top-down, incremental perspective with constant refinement of model granularity at each subsequent system level. See figure 3 for the RUP SE architecture framework and table 5 for the RUP SE process flow.

Figure 3: MDSD - RUP SE diagram (Balmelli, 2006)

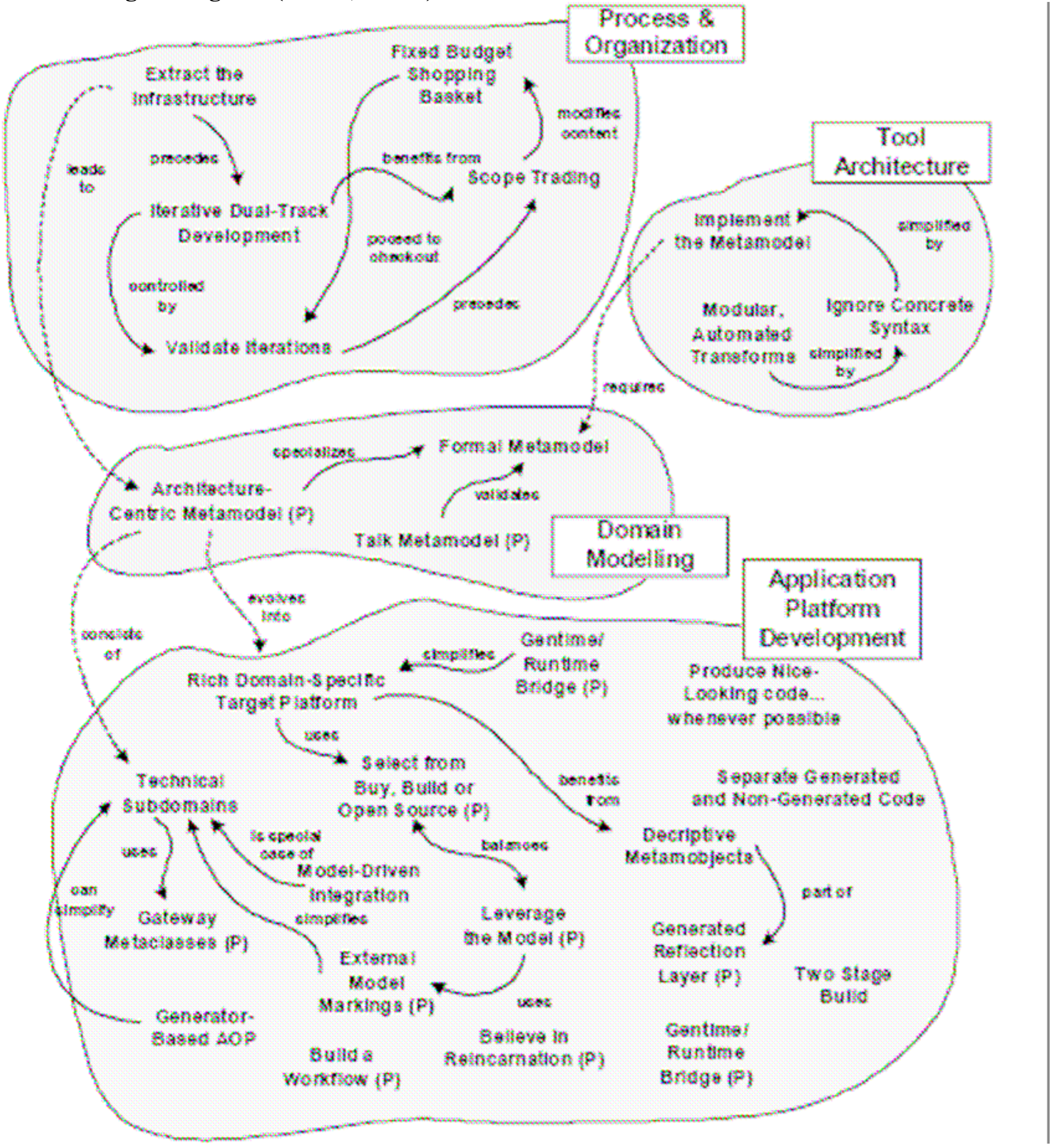


	Model Viewpoints					
	Worker	**Logical**	**Information**	**Distribution**	**Process**	**Geometric**
Model Levels	Context	Role definition, activity modeling	Use case diagram specification	Enterprise data view	Domain-dependent views	Domain-dependent views
Analysis	Partitioning of system	Product logical decomposition	Product data conceptual schema	Product locality view	Product process view	Layouts
Design	Operator instructions	Software component design	Product data schema	ECM (electronic control media) design	Timing diagrams	MCAD (mechanical computer-assisted design)
Implementation	Hardware and software configuration					

The Agile approach is more iterative in nature, and the steps are intentionally loosely coupled to allow for parallel and cyclical development. See figure 4 for the Agile MDSD flow and table 6 for a summary of the process flow steps.

So what kind of impact might one expect from MDSD? According to studies conducted, MDSD techniques have the most significant impact on development velocity. The Middleware Company performed a study in 2003 and claimed that a 35% reduction in development time was realized when employing MDA techniques (Middleware, 2003).

Figure 4: MDSD - Agile Diagram (Bettin, 2004c)



They stated that slightly higher effort was required during the first modeled application due to ramp-up, but significant velocity gains were experienced during subsequent applications on the same platform. Another study conducted by Jorn Bettin (Bettin, 2002) developed a small ecommerce application using three methodologies: 1) manual coding/no modeling, 2) UML-based coding, and 3) MDSD and domain-specific modeling. Using the manual coding technique as the baseline, the results indicated that the UML-based coding approach actually increased overall effort by 5%, and the MDSD saved an astonishing 52% over manually coding the application.

MDSD is revolutionary, but critics have raised many concerns surrounding MDSD limitations. See table 2 for a brief summary.

Table 2: MDSD Limitations

1) **Idealistic**: Full MDSD approach may be too idealistic for some real world artifacts that are seen as necessary. It supports implementation of models to executable code, but does not support database schema, code tuning, etc. (Wikipedia, 2006).
2) **Large Investment**: Model driven tools and technology involve a large initial investment in configuration and potential transformation modification. It is not well suited for one application being deployed on multiple platforms, or just a single application. (SEI, 2006).
3) **Specialized Skill set**: Engineers are required to have a high level of expertise as modelers and architects. They are scarce market commodities (Wikipedia, 2006).
4) **Lack of Standards**: MDSD lacks mature and practical standards for model transformations (Bettin, 2004b).
5) **Tool Maturity and Interoperability**: End to end MDSD requires a suite of tools. Many of the tools are too immature to offer complete transformations. Most of the tools do not have standardized interoperability (Cook, 2004), do not support exchange markings and transformations (SEI, 2006), and are restricted to one level of transformation (SEI, 2006). The lack of mature and adequate tooling support could become an adoption barrier to model driven techniques. (SEI, 2006).
6) **Code Manageability**: A modeler builds a model without the code management in mind. So the generated code is often not readable or maintainable. (Mellor, Clark, Futagami, 2003)
7) **Resistance**: There is a lot of industry inertia to overcome in order for MDSD to be adopted.

MDSD Interpreted

On the surface, Model-Driven Software Development seems like a brand new paradigm. But under the covers, it is shifting the iterative development process from UML/prototypes/coding over to abstraction/modeling/generation. In essence, it pushes the development effort further up in the lifecycle. This shift alone could save money due to the early detection of software issues.

There's no doubt that MDSD is a leap of faith for software engineers. To release control of code, and exist in only a model capacity, is unnerving to most developers. Models have been viewed strictly as initial brainstorming and documentation up to this point. This mindset is also the largest acceptance barrier for MDSD.

MDSD advocates abstraction - the ultimate solution to reuse, standardization, and cost reductions. Continuous abstraction leads the ability to represent complexity in a much more concise manor. Abstraction supports design patterns, templates and other generally accepted software practices. Domain Specific Language (DSL) takes abstraction one step further by capturing the unique needs of a domain and encapsulating into domain semantics.

The models provide much needed context for software engineers. Today, models are viewed as simple documentation. In the MDSD paradigm, models are at the heart of software engineering. The models represent how the system functions, integrates, and operates. In addition, the models drill down to low level implementation details to clarify implementation details and data attribution.

The maturity of MDSD is its own impediment. The prerequisite knowledge required to process MDSD represents a barrier for most developers. Deep and intricate

UML/UML2 knowledge is required to perform MDSD proficiently. The MDSD techniques are not agreed upon by experts in the field, which leaves developers thrashing and searching for guidelines. The tools are complex and require large learning curves. Many of the tools are in developmental stages waiting for standardized techniques.

The biggest benefit of Model-Driven Software Development is the integration of the analysis and design processes directly into the end-to-end development process. Analysis, design and code flow seamlessly. "Model-driven development is still not widespread, but the potential is large. A software development environment with off-the-shelf models and mapping functions […] will change the way in which we build systems. Instead of building and rebuilding systems as the application or the technology infrastructure changes – an expensive proposal to be sure – we'll select models, subset or extend them, then weave them together to build the system." (Mellor, Clark, Futagami, 2003).

MDSD APPLIED

Does Model-Driven Software Development truly increase delivery velocity and reduce complexity? A case study was conducted to assess the engineering impacts associated with a real system using both traditional UML and OO coding, and MDSD system methodology. The applied study compared and contrasted the methodologies from three primary viewpoints: delivery speed, system modeling complexity/abstraction, and code generation.

Background

The case study focused on a tracking system for First United Methodist Church (FUMC). The church needed a system to manage organizational information for the

Children's Ministry. (See appendix B for more details.) Most of their informational needs centered around three areas: recording child demographic data, tracking attendance to church sponsored activities, and developing educational curriculums for Sunday school and confirmation.

The FUMC technology requirements were straight-forward. Most of the user interactions were simple data entry screens and event management. The customer's deployment requirements included a stand-alone application, multiple data entry stations, and a database backend. The church primarily operates desktop applications using Microsoft technologies and MS Office desktop suite. In-house applications were developed in C# or Visual Basic to leverage Microsoft licensing. MS Access was the primary database, and common data was stored on a network LAN for backup purposes. The case study was developed on Microsoft technologies and C# language.

The actual case study focused on a specific piece of the entire system. The rollout plan was intentionally planned in phases in order to prove the initial technology, the MDSD approach, and then the incremental deliveries to follow. The initial phase, and the focus of the thesis work, was narrowed to the subsystem which focused on the recording of child (called Constituents) demographic data. Due to the nature of this application, the two most important architectural layers were the presentation layer (user experience) and the domain layer (persistence).

Framework

The case study was conducted like an experiment. See table 3 for the experiment framework. The subsystem for child demographic data was fully created using both traditional UML with C# manual coding and MDSD techniques with generated code.

The case study "controls" were the UML models and the methodology approach.

Table 3: Case Study Setup

Experimental Design	
Title	Model-Driven Software Development Techniques
Problem Statement	1) Enterprise software is growing in complexity 2) Architecture intricacies create longer software cycles and drive up costs 3) Traditional development methodologies lead to silo development efforts
Hypothesis	MDSD can speed up software delivery while reducing costs and increasing software reuse
Experiment Procedures	
- Materials	1) Software system for FUMC Children's Ministry to track Constituent demographic data 2) MDSD technique 3) Poseidon 5.0 4) UML/UML2 5) Visio 2003 6) Visual Studio 2005 / C# code
- Controls	1) Business models and Data models 2) RUP SE approach 3) UML model granularity 4) N-tier architecture layers
- Variables	1) Development Tools - Poseidon for MDSD development - Visio 2003 and VS2005/C# for traditional development 2) Code Generation
Data Collection	
- Metrics	Track development effort by phase
- Observations	Code quality/completeness Modeling complexity
MDSD Conclusion	Software Velocity Software Complexity Software Reuse Overall

The models were developed with roughly the same granularity. RUP SE was the methodology, with some MDSD Agile techniques used as augmentation. The study "variables" were the tool suites and generation techniques. The tools were Visio and

Visual Studio C# for the traditional approach, and Poseidon 5.0 for the MDSD tool. The "results" were measured in terms of code and time effort. Generated code comparison was done using the model's domain objects. Code completeness was compared using both C# and Java since Poseidon was more Java compatible. The estimated percentage code complete was based upon previous C# coding experience. The tracking metrics measured the execution of each development phase within RUP SE. The expectations from the applied study were to observe how MDSD techniques impact software delivery and code deliverables.

Constraints

During the setup for the applied study, tools proved to be a significant roadblock. Many MDSD tools were evaluated: System Architect (Rational), eGen (Gentastic), openArchitectureWare (GMT), Visio Enterprise Architect, and Poseidon (Gentleware). Most of the MDSD tools on the market produce Java code; none of the evaluated tools could produce C#. Poseidon was the ultimate choice because it had the ability to produce template based C# code. It can produce full Java code, which was used for side-by-side code comparison. UML2.0 and OCL could not be fully exploited due to the in-depth knowledge required for those technologies within the tools

Results

The study's results were surprising. The modeling effort was very inconsistent. In the traditional coding mode, the new models and code base were pulled forward from an existing infrastructure and application. The traditional UML modeling was relatively quick, basically refactoring of objects. Domain Specific Language (DSL) was not available in traditional UML models. Platform Independent Model (PIM) and Platform

Specific Model (PSM) were modeled using Visio based notation, which is not .NET compatible.

With the MDSD approach, the study attempted to export XMI (from the traditional models) to use for infrastructure baselining, but that capability was not available using Visio. The models started from scratch. DSL was not utilized because the functionality could not be located within the Poseidon tool. PIM context and analysis diagrams were slow, but the design level modeling went very fast. Much of the Poseidon tool is tailored around class diagramming. Due to the fact that Poseidon is not .NET platform compatible, the PSM model could not be developed or transformed. Some J2EE functionality was exploited due to better compatibility within Poseidon. Overall, the MDSD modeling effort was made much more difficult due to the tool deficiencies. The UML complexity involved to transform granular MDSD models to code was also more difficult.

The tracking metrics demonstrated increased velocity. For the entire system vertical slice, the traditional approach took 24.25 development hours of effort for models and manually created code. The MDSD approach was a total of 11 development hours, representing about a third of the original effort. Most of the MDSD gains were realized in the coding phase. Traditional coding took 12.25 hours, whereas MDSD code generation was 1.5 hours. The traditional coding approach would have been more than triple in effort had it not been for the pulling forward of an existing infrastructure and code base. So, the net gains would have been even greater.

Code completeness was contrary to the generation results. The generated C# code was simply stub code; Java code was a bit more robust, but would require much more

UML modeling to generate more robust code. Since UML diagrams were one of the study's constants, the models were left relatively the same, even when generating Java. So, some of the code completeness deficiencies were due to the UML modeling depth. The C# code was about 25% complete for the domain classes, and only 10% complete for the remaining system classes. The Java code was better. It was near 30% complete for the domain classes and 15% complete for the remaining system classes. Performing a simple extrapolation given the code effort and the code completeness, the total coding effort would have been somewhere near 6 hours for more robust domain classes. This represents half the traditional coding effort. Thus, the case study approximated the same results as Jorn Bettin's code study (Bettin, 2002).

MDSD ANALYZED

Critical Assessment

Does Model-Driven Software Development live up to the hype? A brief SWOT analysis (strength-weakness-opportunity-threat) critiques the MDSD approach.

MDSD has two primary strengths. Complexity is reduced through better model abstraction and encapsulation. This leads to higher system component visibility and reuse. The case study displayed trends that indicated that MDSD would identify reuse opportunities much more quickly than traditional coding methods. Automatic code generation increases delivery and creates code uniformity. However, the case study also demonstrated that much of the code still has to be handcrafted due to the tool deficiencies.

MDSD weaknesses include a highly specialized modeling skillset, tool immaturity and difficulty of use, and code refinement challenges. The UML, UML2.0

and OCL depth of knowledge required to manipulate the models properly to produce the desired code behavior is significant. It could potentially require years of experience to get code to generate precisely from well-formed MDSD models. Until colleges and universities change their focus away from traditional methodologies and coding, people will not be properly tooled for MDSD modeling.

The tool maturity was the most difficult part of the case study. The tools are large, and somewhat difficult to understand and operate. Some of the core concepts like DSL, transformations, tool standards, and XMI export are not yet fully supported. Lastly, the level of code granularity is not mature. As an example, none of the tools had a way to specify screen layouts and other human factors requirements. In the author's opinion, these all represent significant roadblocks and financial impacts for software companies.

There are many opportunities for MDSD. First, MDSD modeling gives visibility to the enterprise software implementations. Engineers can leverage all previously modeled systems and designs by simply reviewing the model abstractions. The net is increased software velocity and design consistency. Second, the code produced from MDSD generation is highly standardized and very uniform. That translates to lower maintenance costs because developers can more quickly identify with the code base. Last, the conversion of developers from procedural languages to object oriented technologies can be bridged by using MDSD.

Threats represent a major area for MDSD. First, the inertia that stands in front of this approach is gigantic. The resistance to model centric engineering both inside and outside the software industry is almost insurmountable. Today's mainstream workplace mindset equates coding to productivity, not models. Without the tools generating robust code, MDSD will never be fully adopted. Developers will fallback to traditional coding to get their deliverables, and leave MDSD modeling stranded. Reverse engineering techniques could help, but again the tools need to be mature enough to properly represent the original source code. The second major threat to MDSD is the "all or nothing" aspect to the approach. Model-driven software development doesn't lend itself to hybrid enterprise solutions – some systems traditional, some model driven. This negates some the MDSD reusability strength. System assets that are not visible to the models cannot be leveraged.

Conclusions

Can Model-Driven Software Development provide reduced costs while at the same time speeding up software delivery? In the author's opinion, Model-Driven Software Development is the correct vision for the software industry, and a natural next step in software progression. It's not idealistic, it's real. Engineers and customers will ultimately see benefit from simplified systems and consistent system behaviors. It will speed up software deliveries. But, it is also the author's opinion that MDSD will not receive mainstream adoption. The transition will be too slow due to the developer resistance and immaturity of tools. Software engineering should be enabling for corporations, not inhibiting. Speed to market and development responsiveness are too large of expectations in most corporations to overcome the MDSD perceptions.

The case study was the key contributor in reaching this conclusion. See table 4 for a summary of the case study summary. It confirmed that MDSD is good, but still has a lot of room for advancement. The case study demonstrated the code delivery velocity, but the tradeoff was an increased effort surrounding models and code management. More fully defined UML models would have certainly altered the results of the case study for Java code generation, but not in C# code. The ramp-up time required for full MDSD modeling is significant. Cost savings would only be realized in the long run. Overall, the case study did not fully meet the original thesis expectations. However, it did clarify the current conditions and future direction of MDSD.

Table 4: Case Study Result Summary

MDSD Summary	Software Velocity	
	Pros	Cons
	1) faster code delivery	1) longer modeling cycles
	Software Complexity	
	1) less complex to create code 2) representation of domain specific need through DSL 3) coding abstracted away through models/code generation 4) nice visualization capabilities	1) less code complete 2) higher modeling complexity 3) higher prerequisite modeling knowledge required
	Software Reuse	
	1) able to leverage modeled classes multiple time	1) must model everything in order to have visibility
	Overall	
	1) Conceptually much cleaner 2) Clear abstract and reuse capabilities 3) Extremely quick code generation	1) Tools are very immature, causing deficiencies in MDSD processes

Future Research

The road to success is always under construction. Model-Driven Software Development reflects the possibilities that lie ahead for the software industry. MDSD shows potential, but will encounter huge resistance. It will be interesting to witness the direct correlation between MDSD tools maturity versus industry adoption. If the tools can develop more fully and the knowledge can become more conventional to IT, then MDSD will alter the face of the software industry.

ANNOTATED BIBLIOGRAPHY

Balmelli, L. (2006) *Model-Driven Systems Development*. Retrieved Dec 27, 2006, from http://www.research.ibm.com/journal/sj/453/balmelli.html

Discusses the IBM approach to MDSD, which seems to make a lot of sense to me. Called Rational Unified Process for Systems Engineering (RUP SE). Probably used for Applied Case Study. This article seemed to "speak" to me more than others on process. (5*)

Bettin, Jorn. (2002) *Measuring the Potential of Domain-Specific Modelling Techniques.* Retrieved Dec 2, 2006, from http://www.dsmforum.org/events/DSVL02/bettin.pdf

The article addresses the metrics of MDSD directly compared to a fully "manual" software development and with "traditional" UML-based software development. These metrics will be highly useful for the Critical Analysis section of the Thesis. The example metrics seem valid for the small example, but would be concerned how they translate to other applications. (4*)

Bettin, Jorn. (2004). *Model Driven Transformation?*. MDSD Introduction. Retrieved Dec 30, 2006. http://www.modeldriventransformation.com/

The first half of this article is good with terminology, but the second half is not relevant. Terminology and approaches in this domain space are used very interchangeably. Due to the theoretical nature, some of the articles are confusing in respect to what arena they are trying to address. I think a terminology or glossary will be necessary in order to clarify concepts more precisely. At minimum, a section is needed to discuss the various approaches, distinctions, and their differentiations. (MDA,MDD, MDSD, Software Factories). Many of Jorn's articles are repeats (3*)

Bettin, Jorn. (2004) *What is MDSD? MDSD Introduction*. Retrieved Dec 4, 2006, from http://www.mdsd.info/mdsd_cm/page.php?page=intro

Might be good reference to cite for a glossary in the paper if one is needed. This series of websites from Jorn Bettin are all excerpts from his textbook Model-Driven Software Development. They provide good basis, but the textbook is more comprehensive. Probably use the text though, better fully compiled source. (2*)

Bettin, Jorn. (2004) *Model-Driven Software Development: An emerging paradigm for Industrialized Software Asset Development.* Retrieved Dec 1, 2006, from http://www.softmetaware.com/mdsd-and-isad.pdf

This article presents some of the reasons that MDSD is becoming an emergent technology. The "why is this important" perspective. Again, it's excerpts from his textbook. It provides a good foundation for the MDSD approach. It also looks at the economics of software development. Very nice reference articles on metrics of software. Has a nice reference diagram for the Traditional-to-MDSD transition (pp. 22) (3*)

Bettin, Jorn. (2004) *Model-Driven Software Development Activities. The Process View of an MDSD Project*. Retrieved Dec 6, 2006, from http://www.softmetaware.com/mdsd-process.pdf
This article provides a high level overview of the essential software design and development activities in sequential order. I will utilize for the applied study. (3*)

Cook, Steve. (2004) *Model-Driven Architecture and Domain Specific Modeling*. Retrieved Nov 11, 2006, from http://www.bptrends.com/publicationfiles/04-04%20COL%20MDSD%20Frankel%20-%20Bettin%20-%20Cook.pdf#search='model%20driven%20software%20development%20isbn'
The article is primarily grounded from an MDA perspective, but branches out to embrace the MDSD approach. It discussed the need for more generators. Discusses some of the economic aspects of future MDSD growth. It also contains an excerpt from Jorn Bettin's textbook and replicated from the Softmetaware.com articles above. The section from Steve Cook's response article provides a good counter-discussion to MDSD and MDA. (3*)

Eclipse. Eclipse MDDi Project. (2006) *Eclipse Model Driven Development Integration*. The Eclipse Foundation. Retrieved Dec 17, 2006, from http://www.eclipse.org/proposals/eclipse-mddi/
This article seemed overwhelming at first. It describes the integration of model driven technologies into a single platform. The project is still developing, so a bit evolving. (3*)

Hailpern, B., Tarr, P. (2006) *Model-driven development: The good, the bad and the ugly*. Retrieved Jan 12, 2006, from http://www.research.ibm.com/journal/sj/453/hailpern.html
This article presents an overview of MDD, and then does an analysis. A very concise article on the warts, stigma, and challenges for MDD. (5*)

Mellor S, Clark A, Futagami, T. (2003) *Model-Driven Development*. Retrieved Dec 2, 2006, from http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231145
Article discusses pros and cons of models/modelers, and model transformations (2*)

OOPSLA Conference 2004. *Model-Driven Software Development: Introduction & Best Practices*. (n.d.). Retrieved Nov 12, 2006, from http://www.oopsla.org/2004/ShowEvent.do?id=101
Throughout all of the research, the two names below continued to be published and referenced by other authors as authority:
"**Jorn Bettin**, SoftMetaWare: Jorn Bettin is a software consultant with a special interest in techniques to optimise the productivity of software development teams

and in designing large-scale component systems. Prior to co-founding SoftMetaWare in 2002 he spent 13 years as a consultant and mentor in the IT industry in Germany, New Zealand, and Australia. He has implemented automated, model-driven development in several software organisations, has worked in methodology leadership roles in an IBM product development lab, and enjoys leading international teams dispersed across several locations."
"Markus Voelter, Independent Consultant: Markus works as an independent consultant on software technology and engineering. He focuses on the architecture of large, distributed systems. Over the last years, Markus has worked on several model-driven software development projects in the enterprise and embedded world. Examples include banking, automotive and radio astronomy. Markus is a regular speaker at the relevant national and international conferences. For example, he has presented at ECOOP, OOP, OOPSLA, ACCU. Markus is the (co-)author of several patterns, many magazine articles, as well as Wiley's "Server Component Patterns" book" (1*)

SEI Software Engineering Institute. (2006). *Model Driven Architecture MDA.* Carnegie Mellon University, 19 January, 2006. Author: G Lewis.http://www.sei.cmu.edu/isis/guide/technologies/mda.htm

Good quote to show the relationship between MDA, MDD, and MDSD. This article is a concise source for MDA approach, definitions and differentiations. But only MDA. It quickly runs through the MDA background and standards. This will be useful for the MDA foundational aspects. The article discusses some of the MDA restrictions, immaturity, and future. It's a good counter discussion. (3*)

Stahl, T., & Volter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ: John Wiley & Sons, Inc..

This reference text addresses the entire practice of MDSD. Since the concepts are relatively new, section 1 discusses the theories behind the MDSD approach. The authors seem to make it a point to differentiate and separate themselves from the CASE tools and Model Driven Architecture from OMG. The second section goes into a deep dive on the specifics of MDSD models and generation. This section is very architecture framework in nature. This should be the most useful area as it relates to the case study. It appears that the authors assume the average IT professional has access to model generators and code generators in order to show their ideas in action. That is simply not reality and could prove to be a roadblock for the applied study. The final two sections won't be utilized much. Section 3 focuses on the software development lifecycle this is needed to support MDSD techniques. The final section discusses management required for MDSD, such as pilot programs, roles, and organizational structure. This text might be too focused on building "the perfect architecture" and theories. It assumes a thorough understanding of MDA approach and standards, as well as UML notation. This is a bit ironic since the authors try so hard to differentiate MDSD from MDA. (5*)

Uhl, Axel. (2003) *Model Driven Architecture is Ready for Prime Time*. IEEE Software.

Retrieved Nov 30, 2006 from http://computer.org/software

This article discusses the advancements and maturity in software engineering. MDA is the next logical step in the maturity cycle, as well as counterpoint anchored more in today's reality rather than theoreticals. (4*)

Wikipedia. (2006) *Model-Driven Architecture*. Wikimedia Foundation, Inc., Retrieved Dec 17, 2006, from http://en.wikipedia.org/wiki/Model_Driven.Architecture

This article gives good context for the MDA approach as a building block toward MDSD. Good definition. (2*)

REFERENCE LIST

Balmelli, L. (2006) *Model-Driven Systems Development*. Retrieved Dec 27, 2006, from http://www.research.ibm.com/journal/sj/453/balmelli.html

Bettin, Jorn. (2002) *Measuring the Potential of Domain-Specific Modelling Techniques.* Retrieved Dec 2, 2006, from http://www.dsmforum.org/events/DSVL02/bettin.pdf

Bettin, Jorn. (2004a) *What is MDSD? MDSD Introduction*. Retrieved Dec 4, 2006, from http://www.mdsd.info/mdsd_cm/page.php?page=intro

Bettin, Jorn. (2004b) *Model-Driven Software Development Activities*. Retrieved Dec 1, 2006, from http://www.softmetaware.com/mdsd-process.pdf

Bettin, Jorn. (2004c) *Model-Driven Software Development: An emerging paradigm for Industrialized Software Asset Development*. Retrieved Dec 1, 2006, from http://www.softmetaware.com/mdsd-and-isad.pdf

Cook, Steve. (2004) *Model-Driven Architecture and Domain Specific Modeling*. Retrieved Nov 11, 2006, from http://www.bptrends.com/publicationfiles/04-04%20COL%20MDSD%20Frankel%20-%20Bettin%20-%20Cook.pdf#search='model%20driven%20software%20development%20isbn'

Children's Ministry. (2004) First United Methodist Church in Colorado Springs Online. Retrieved Nov 4, 2006, from http://fumc-cs.org/education/kcity.html.

Eclipse. Eclipse MDDi Project. (2006) *Eclipse Model Driven Development Integration*. The Eclipse Foundation. Retrieved Dec 17, 2006, from http://www.eclipse.org/proposals/eclipse-mddi/

Hailpern B., Tarr P. (2006) *Model-driven development: The good, the bad and the ugly.* Retrieved Jan 12, 2006, from http://www.research.ibm.com/journal/sj/453/hailpern.html

OOPSLA Conference 2004. *Model-Driven Software Development: Introduction & Best Practices*. (n.d.). Retrieved Nov 12, 2006, from http://www.oopsla.org/2004/ShowEvent.do?id=101

Middleware. The Middleware Company. (2003). *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach – Productivity Analysis.* Retrieved Dec 3, 2006. http://www.compuware.com/dl/MDAComparisonTMCfinal.pdf

Mellor S, Clark A, Futagami, T. (2003) *Model-Driven Development.* Retrieved Dec 2, 2006, from http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231145

SEI Software Engineering Institute. (2006). *Model Driven Architecture MDA.* Carnegie Mellon University, 19 January, 2006. Author: G Lewis.http://www.sei.cmu.edu/isis/guide/technologies/mda.htm

Stahl, T., & Volter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ: John Wiley & Sons, Inc..

Volter, M, Bettin, J. (2004) *Patterns for Model-Driven Software Development*. Retrieved Nov 17, 2006, from http://www.voelter.de/data/pub/MDDPatterns.pdf

Wikipedia. (2006) *Model-Driven Architecture*. Wikimedia Foundation, Inc., Retrieved Dec 17, 2006, from http://en.wikipedia.org/wiki/Model_Driven.Architecture

GLOSSARY

Application Engineering. MDSD models for the actual platform and system context including PSM and deployment specifications

Constituent. A term used for a child who is not yet confirmed and not a church member.

Domain. A bounded area of knowledge. Domains can relate to knowledge about vertical industries (business domains and also to software implementation technologies (technical domains).

Domain Engineering. MDSD models for the overall business context including analysis, design, and domain specific concepts.

Domain-Specific Language (DSL). MDSD template modeling language, semantics, and syntax used for expressing key aspects of the domain.

Meta Object Facility (MOF). An approach adopted by OMG that is a UML metamodel, or the model that describes the UML itself

Model Driven Development (MDD). A term used interchangeably with MDSD/MDA.

Model Driven Architecture (MDA). Adopted by the Object Management Group (OMG) in 2001. MDA is an approach to software development that provides a set of guidelines for structuring specifications expressed as models. It is model driven because it provides a means for using models to direct the course of understanding, design, construction, and modification. (Wikipedia, 2006).

Object Constraint Language (OCL). A language used for declaring and describing UML model rules.

Query/Views/Transformations (QVT).A standard developed by OMG, compatible with MDA, to process model transformations from source to target.

Platform Independent Model (PIM). The model describing the business logic, undiluted by technical concerns.

Platform Specific Model (PSM). The model used to describe the actual platform implementation details.

Unified Modeling Language (UML). An object modeling language and specification often utilized in software engineering.

XML Metadata interchange (XMI). A standard for exchanging metadata models and information using XML as the basis. Allows for interoperability between models.

APPENDIX A – MDSD APPROACH

Table 5: MDSD - RUP SE approach (Balmelli, 2006)

RUP SE Steps

Overall Process

1) **Models**: RUP SE has four types of architectural models: Context, Analysis, Design, and Implementation. Each represents a more granular drill down of the system development. (IBM, 2006).
2) **Viewpoints**: A viewpoint is a subset of the architecture model that addresses a certain set of engineering concepts (IBM, 2006).
3) **Views**: Views are the intersections between Models and Viewpoints. Views contain the artifacts to fulfill the viewpoint. (IBM, 2006).
4) Each model and viewpoint is a separate diagram that is comprised of view artifacts.
5) Each subsequent granularity level had direct built-in relationships maintained from the parent diagram.

Domain Engineering Process

1) The context level model is the top level model which shows the entire system as a single entity and its external entities.
2) The analysis model level represents the architectural perspective of the system, the internal elements, and subsystems.
3) The design level shows the actual system's software specifications. This is most closely related to traditional UML modeling. The analysis and design levels represent the Platform Independent Model (PIM).

Application Engineering Process

1) The implementation model level which refines the PIM into the Platform Specific Model (PSM) with the modeling of chosen domain technologies.

Code Generation Process

1) Generate the code from PSM - Rational SA will perform the model-to-text code generation.
2) Manual code – Create any hand crafted code that cannot be represented in models.
3) Iterate over models to refine granularity and refactor code changes.

Table 6: MDSD - Agile approach (Bettin, 2004b) (Stahl, Voelter, 2006)

Agile Steps

Overall Process

1) If possible, extract the existing infrastructure from a running application or a prototype as a baseline.
2) Develop the initial infrastructure with one initial application simultaneously (Iterative Dual-Track Development). The development will switch focus between infrastructure (normally one step ahead) and domain-specific application development.
3) Validate each iteration, refinement, and application against the metamodel and models.
4) With every step, elaborate on the domain-specific knowledge and modeling and generate code to validate results. When necessary, re-extract the platform infrastructure.

Domain Engineering Process

1) Domain analysis and design - Develop the product platform model and implement it into metamodel tool. Implement all domain specific notation and constraints into the metamodel.
2) DSL - Build the domain-specific language (DSL). The DSL is a template, and resembles pseudo-code by nature. It acts as the reference model standard for all classes within the domain. This piece is probably the most critical for Model-Driven Software Development. It will later be validated and iterated with an actual reference implementation.
3) PIM - Refine the domain-specific metamodel into an architecture-centric model. Normally this is specified in UML and OO terms. This model encompasses some of the core architectural concepts and stereotypes that are required within the system. This model also represents the Platform Independent Model (PIM) and is a technology-independent representation of layered architecture.
4) Transformers – Create generator templates that will convert the UML/XMI to skeleton implementation model.

Application Engineering Process

1) PSM - Transform the PIM into a Platform Specific Model (PSM) through the use of DSL, Transformers and rules. This model will include the some specific technology entities like J2EE or .NET. In addition to target platform UML, it includes configuration files, deployment information, and other domain specific artifacts.
2) Technical Subdomains - Partition the system into distinct technical subdomains to keep the models simple. Make existing or legacy system integrations separate technical subdomains to achieve model-driven integration.
3) Code Separation - Segregate generated code from hand-crafted code in models
4) Interfaces - Model fully externalized interface definitions. This support component based architecture.

Code Generation Process

1) Generate the code from PSM – This step is purely mechanical. No additional information is included.
2) Manual code – Create any hand crafted code that cannot be represented in MDSD or DSL.
3) Reincarnate as many code changes back into the model as soon as possible.

APPENDIX B – CASE STUDY NARRATIVE

Business Problem

First United Methodist Church was founded as the first church in Colorado Springs in 1871. The Children's Ministry is primarily focused on religious development of children who range in grades from kindergarten through sixth.

The following excerpt from FUMC website provides information on the mission of the Children's Ministry.

> **FIRST UNITED METHODIST CHURCH – CHILDREN'S MINISTRY**
>
> First United Methodist Church has a long-term vision of transformed lives through children's Sunday school. We want our children to experience the love of Christ, to embrace Christ personally, and to develop into young adults with a mature and solid faith. To do this, we are providing a fun and purposeful learning environment that will help them remember their Sunday school experience all the days of their lives. Studies have shown that children retain better what they learn when they experience it in multiple ways.
>
> First United Methodist Church is investing in a multi-dimensional learning environment for our kids. Your child will learn the principles of the Christian faith through art, music & movement, cooking, drama, video, puppets, stories, science, computers, and yes - even games! (Children's Ministry, 2004)

FUMC Children's Ministry has experienced turnover in leadership over the past five years. During that turnover, much of the information concerning the constituency has been lost. In addition, many of the processes and procedures have been lost, revamped, or pieced together. The ministry has managed most of their data needs through the use of manual processes, spreadsheets, and word documents. This disconnected approach is cumbersome and difficult to process data efficiently. It often takes hours to pull together basic information and reporting. They need a solution for managing administrative information.

User Goals/Purpose

The new Director, Cheryl Ledford, has realized the need to standardize and automate portions of the ministry. She is looking to revitalize the Children's Ministry through automation. Her vision is to reinstate the constituency information, improve the attendance, and begin a curriculum program leading to confirmation. As part of that vision, she is in need of three main tracking systems: children, attendance, and courses. Automation would result in providing new opportunities to provide superior services to children, parents, volunteers, and teachers. The primary goals would be higher data visibility and accessibility, management of information, and data integration with other church software.

Business Impact

By providing a single, integrated tracking system, the ministry can streamline processes and provide more accurate information. Ultimately they will begin to grow the children's department within the church. Additionally, by having the information more accessible, FUMC should have the ability for timely and concise reporting.

Vision – Approaches/Solutions

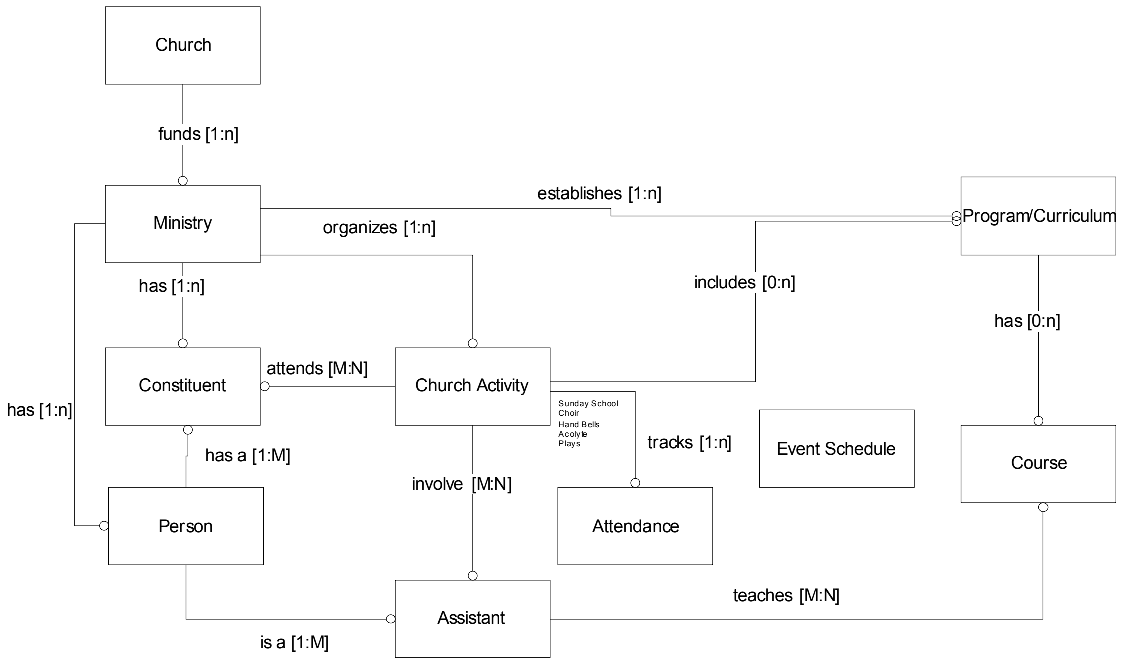
Once the basic system requirements and design work are complete, FUMC will be faced with the opportunity to evaluate alternate paths to complete this project. The Buy versus Build options should need to be considered and compared against delivery speed.
There are commercial software packages on the market that could track this information. Preliminary software investigations have determined that *FellowshipOne* (http://www.fellowshipone.com) or *Church Windows* (http://www.churchwindows.com) may represent good buy options.
The system rollout will be implemented in multiple phases.

- Phase 1 will be a basic proof of concept to demonstrate the basic system needs and software layers. Additionally this phase acts as the proof of concept for Model-Driven Software Development using the constituent data.
- Phase 2 will continue build out the architectural layers from Phase 1, to embody and more robust system.
- Phase 3 will include additional layers for security processing and possible barcode scanning for check in.
- Phase 4 will provide a layer for processing with interface with other church software packages.

APPENDIX C – CASE STUDY EXHIBITS

Business Models

Figure 5: Entity Relationship Diagram



Data Models

Figure 6: Logical Data Model

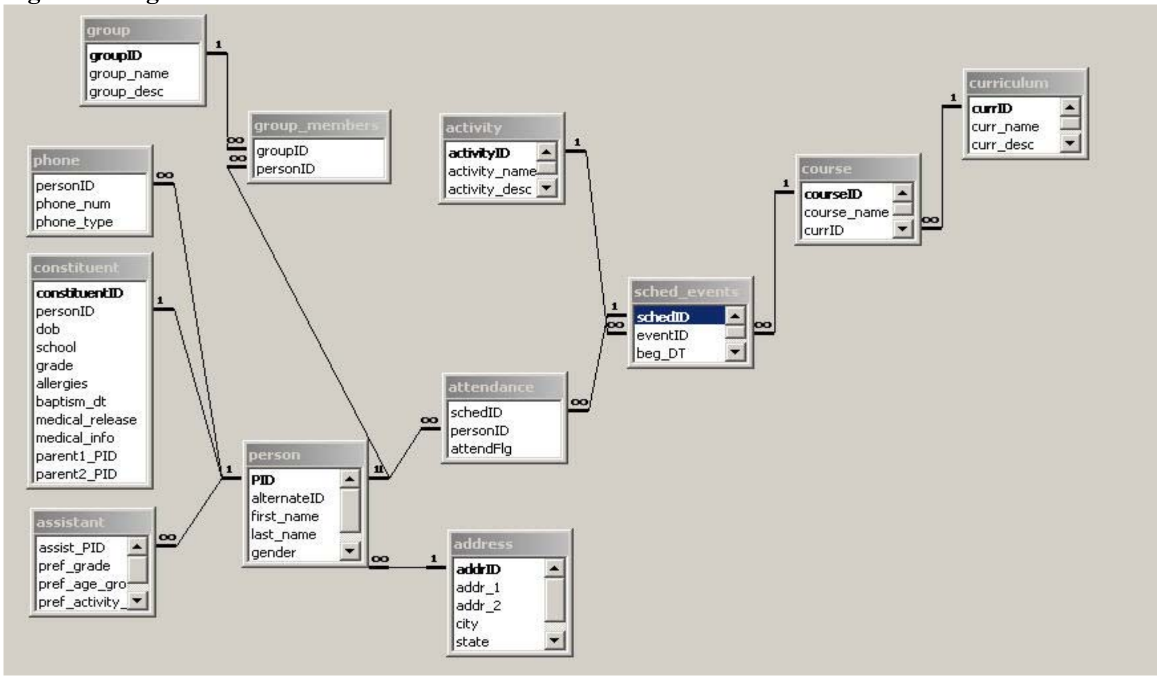
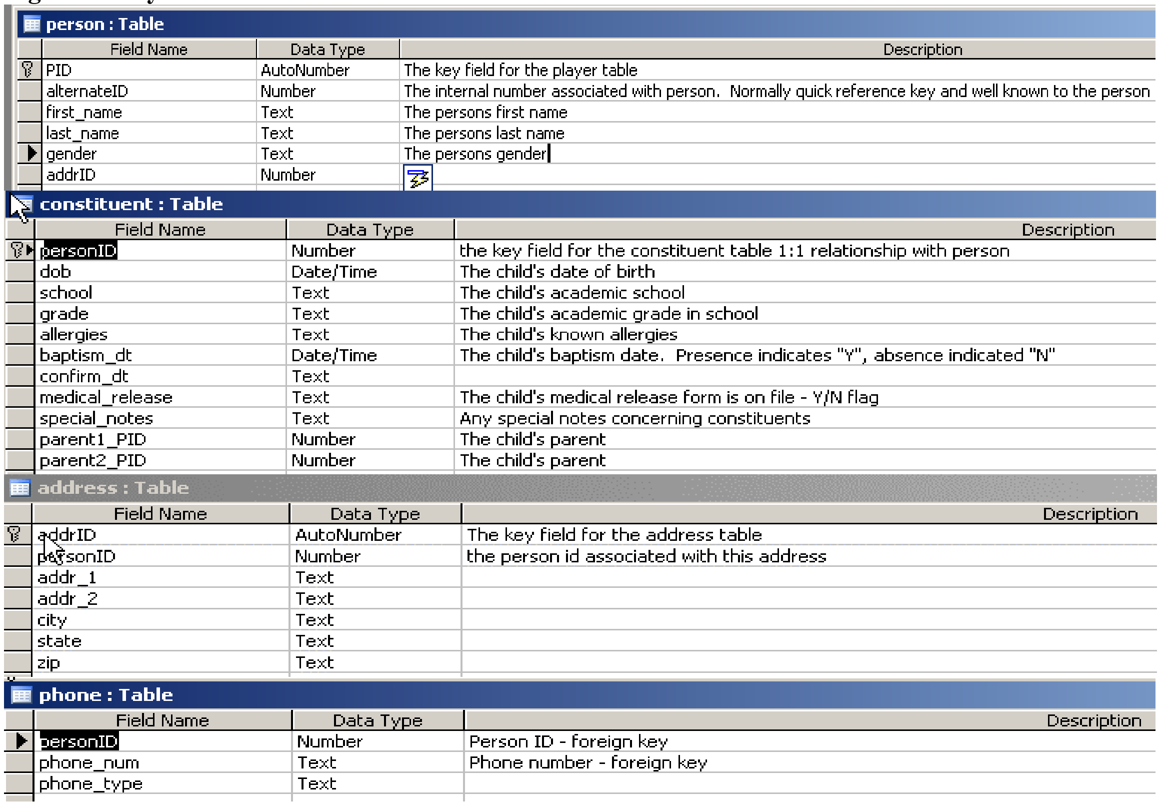


Figure 7: Physical Data Model - Constituent related tables



person : Table

Field Name	Data Type	Description
PID	AutoNumber	The key field for the player table
alternateID	Number	The internal number associated with person. Normally quick reference key and well known to the person
first_name	Text	The persons first name
last_name	Text	The persons last name
gender	Text	The persons gender
addrID	Number	

constituent : Table

Field Name	Data Type	Description
personID	Number	the key field for the constituent table 1:1 relationship with person
dob	Date/Time	The child's date of birth
school	Text	The child's academic school
grade	Text	The child's academic grade in school
allergies	Text	The child's known allergies
baptism_dt	Date/Time	The child's baptism date. Presence indicates "Y", absence indicated "N"
confirm_dt	Text	
medical_release	Text	The child's medical release form is on file - Y/N flag
special_notes	Text	Any special notes concerning constituents
parent1_PID	Number	The child's parent
parent2_PID	Number	The child's parent

address : Table

Field Name	Data Type	Description
addrID	AutoNumber	The key field for the address table
personID	Number	the person id associated with this address
addr_1	Text	
addr_2	Text	
city	Text	
state	Text	
zip	Text	

phone : Table

Field Name	Data Type	Description
personID	Number	Person ID - foreign key
phone_num	Text	Phone number - foreign key
phone_type	Text	

User Experience Mockups

Figure 8: UI Mockup - Main Menu

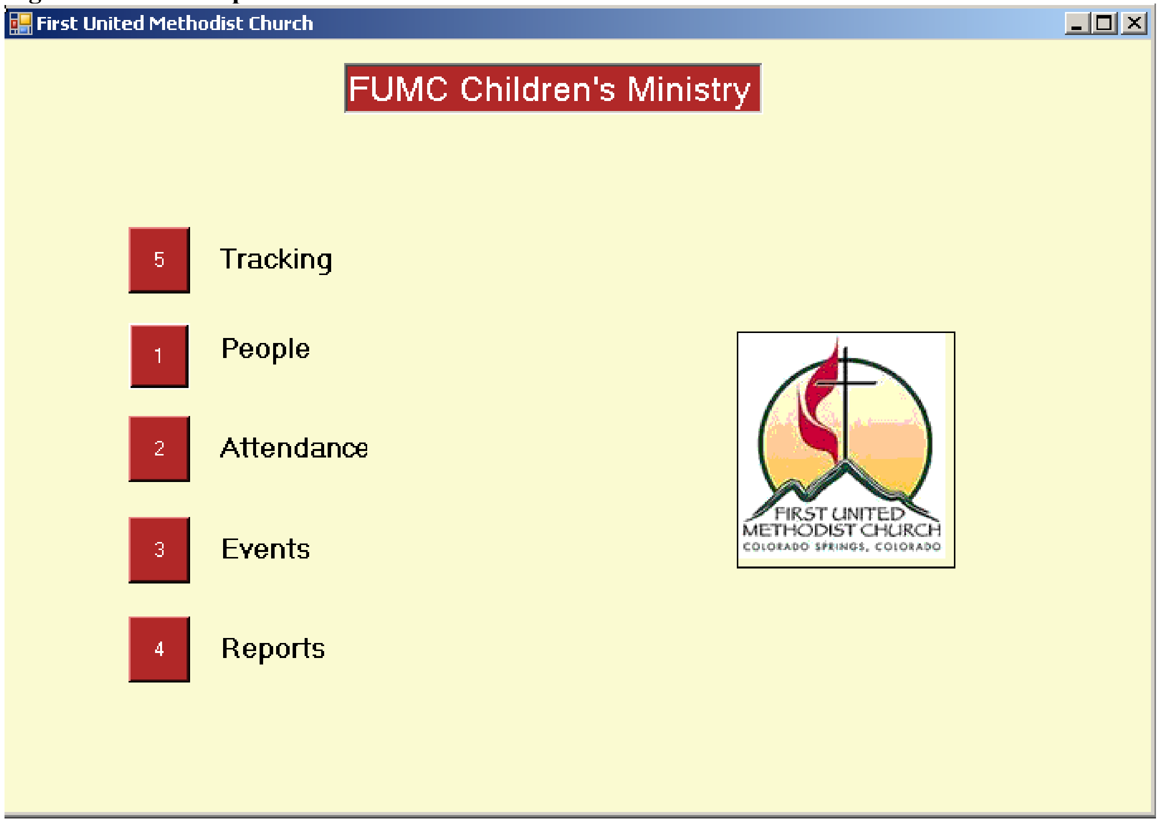
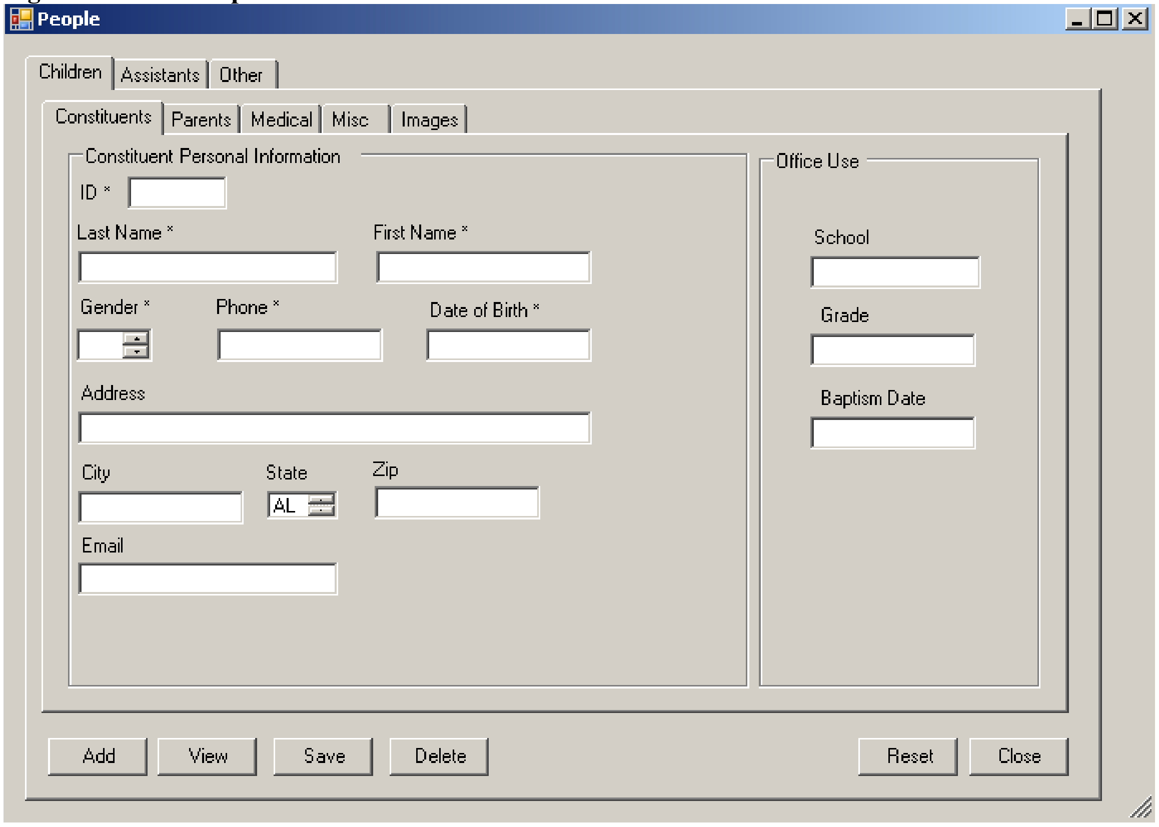

First United Methodist Church
FUMC Children's Ministry
5
Tracking
1
People
2
Attendance
3
Events
4
Reports
FIRST UNITED METHODIST CHURCH
COLORADO SPRINGS, COLORADO

Figure 9: UI Mockup - Constituent Screen



APPENDIX D – MDSD MODEL EXHIBITS

RUP SE Context Models

Figure 10: Use Case - Overall Context

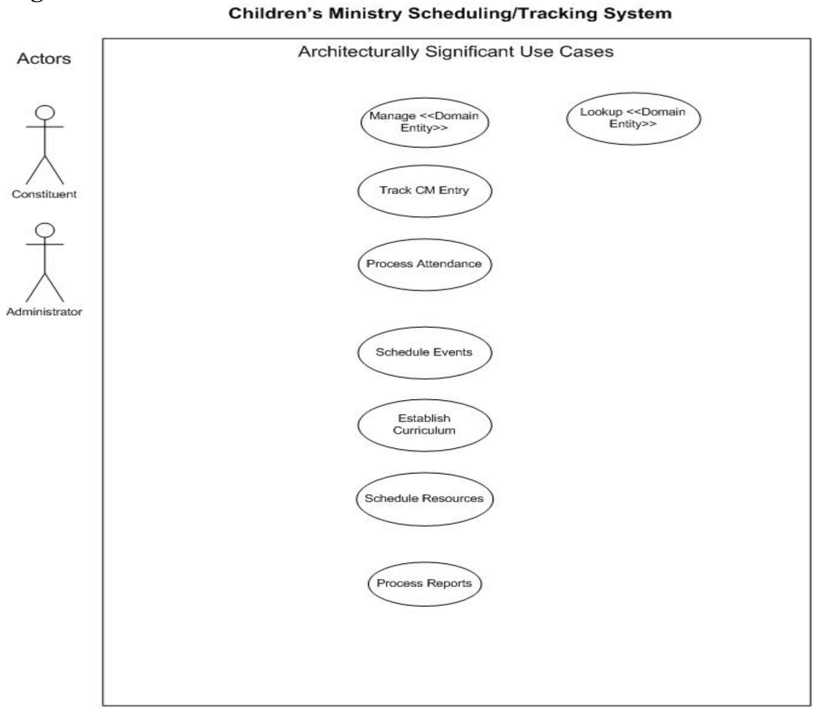


Figure 11: Use Case - Manage <<Constituent>>

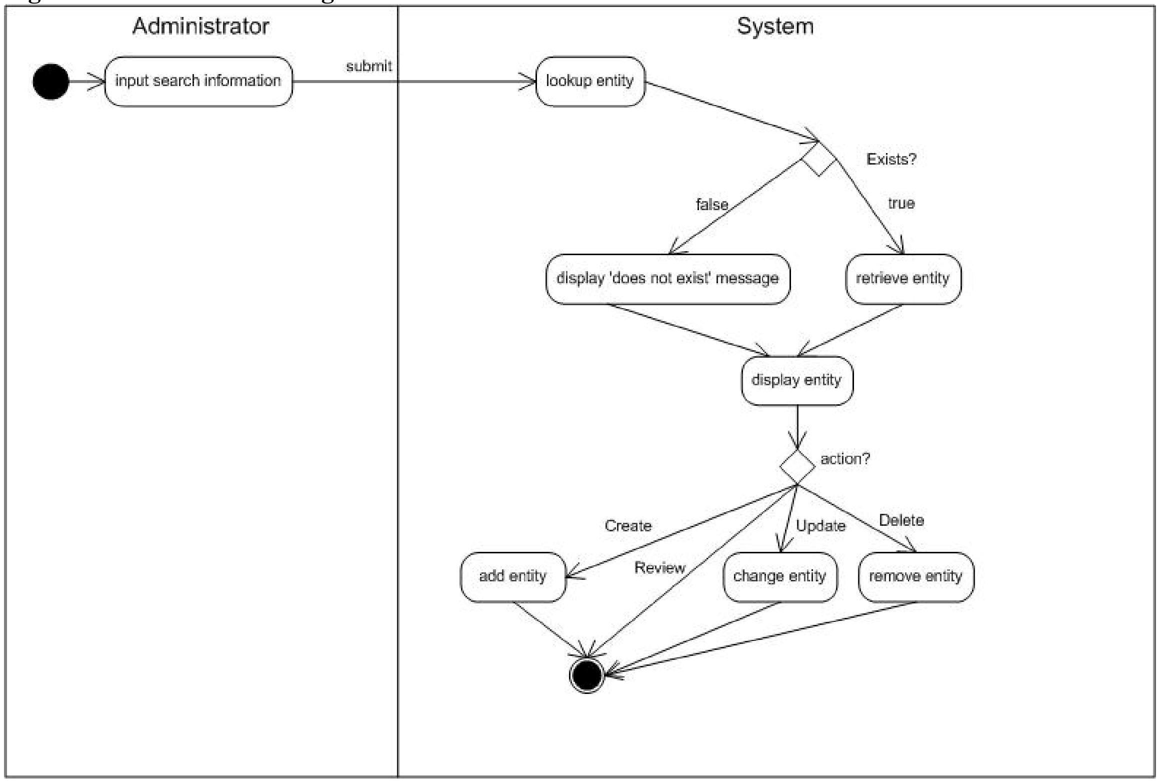


Figure 12: Analysis Diagram: Manage Constituent – Class Participants

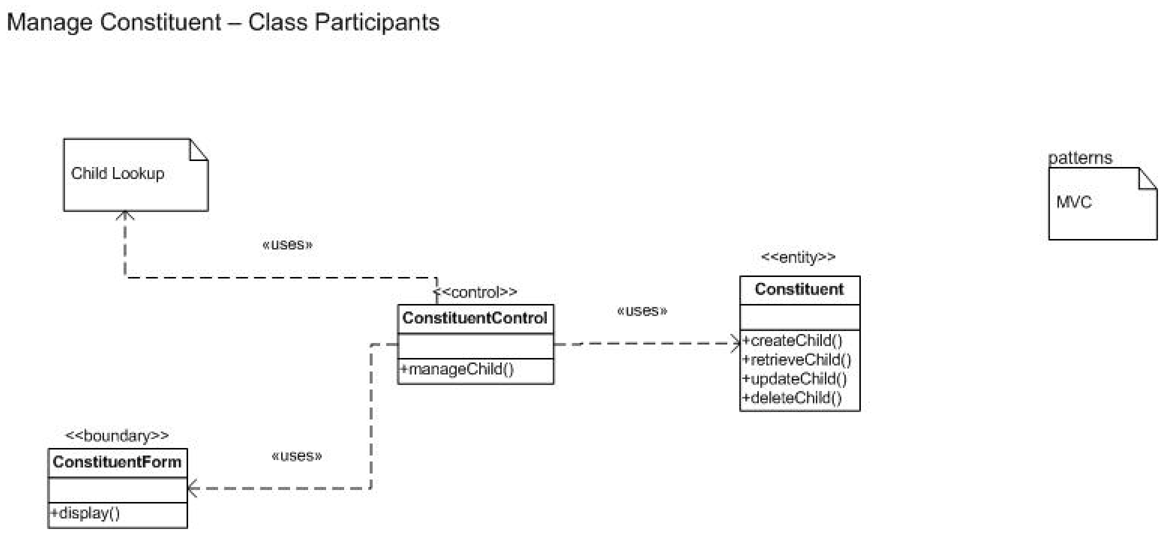


Figure 13: Analysis Diagram: Manage Constituent – Sequence Flow

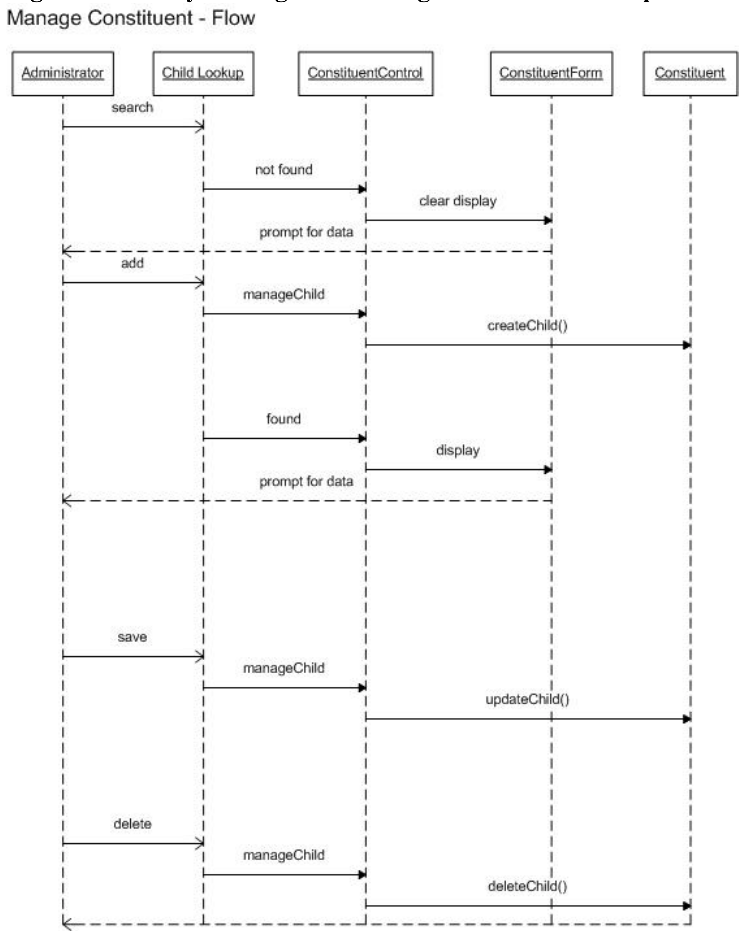


Figure 14: PIM - Presentation Layer

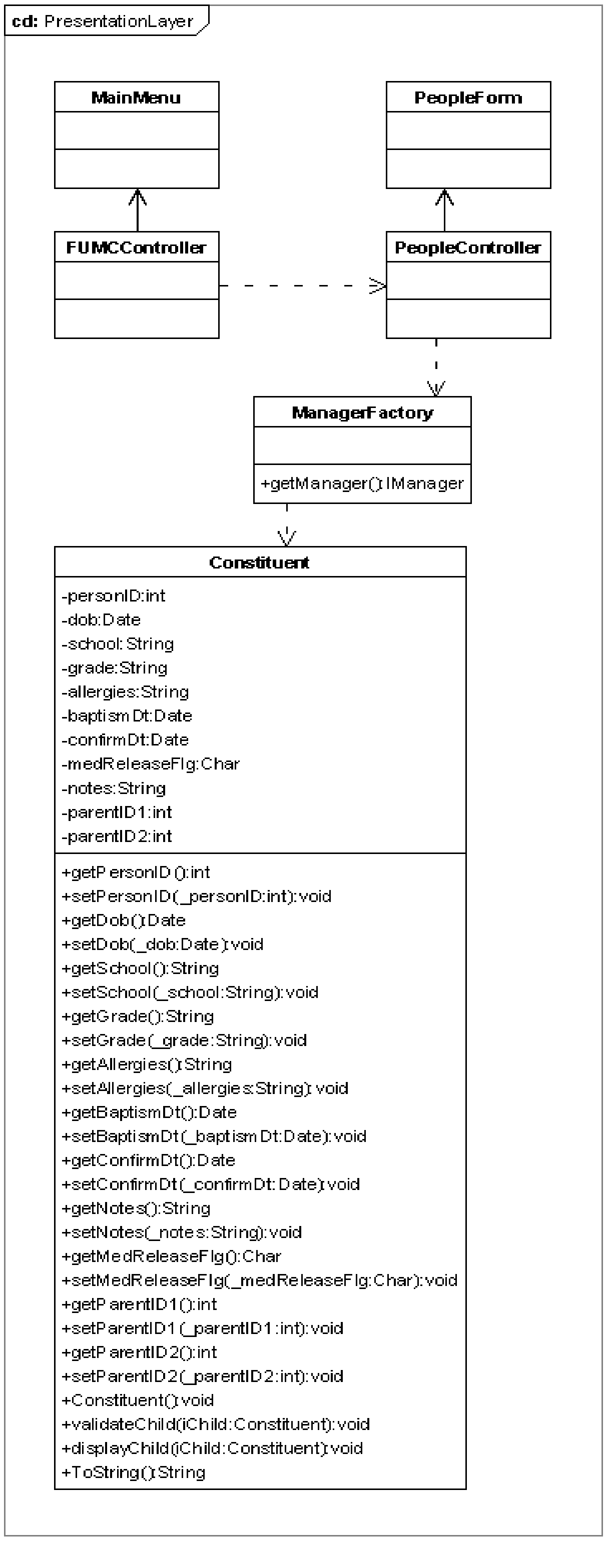


Figure 15: PIM - Business Layer

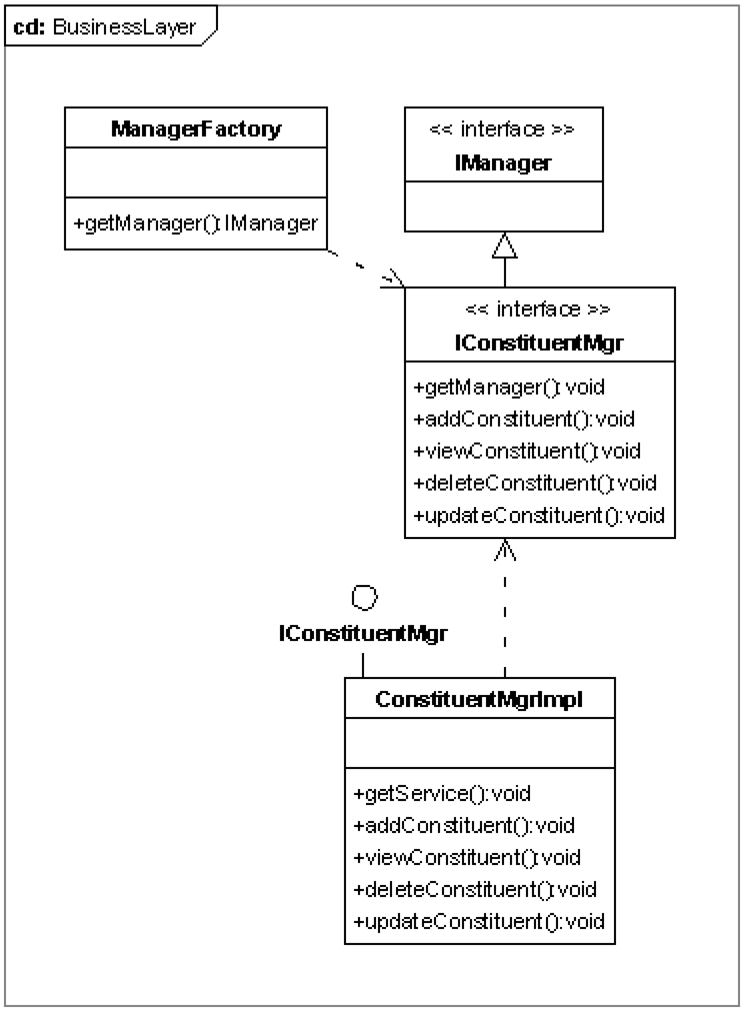


Figure 16: PIM - Data Access Layer

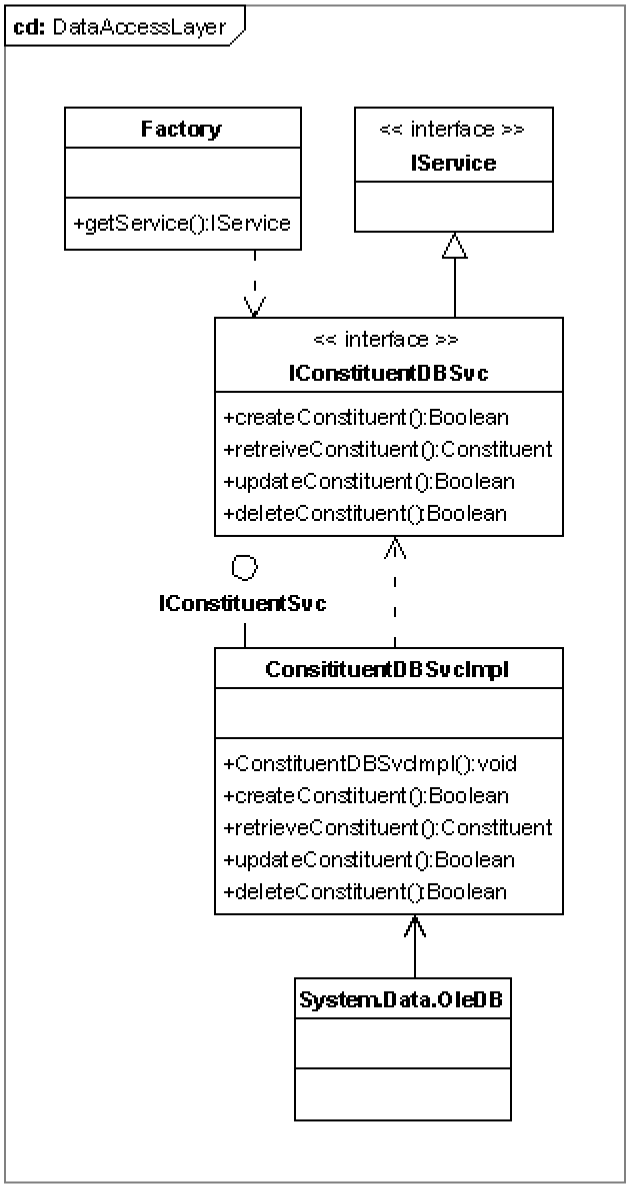
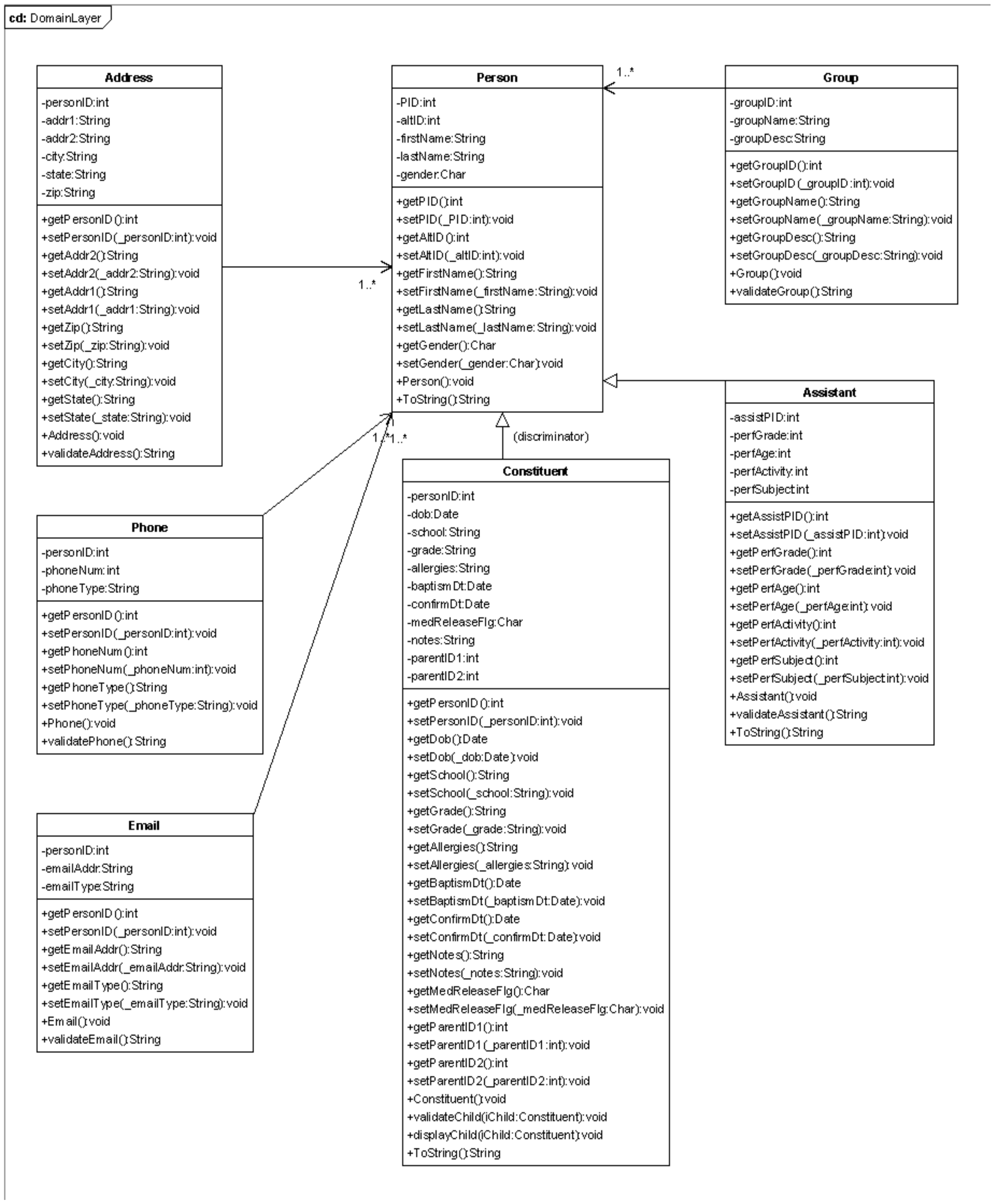

cd: DataAccessLayer
Factory
+getService():IService
<< interface >>
IService
<< interface >>
IConstituentDBSvc
+createConstituent():Boolean
+retreiveConstituent():Constituent
+updateConstituent():Boolean
+deleteConstituent():Boolean
IConstituentSvc
ConsitituentDBSvcImpl
+ConstituentDBSvcImpl():void
+createConstituent():Boolean
+retrieveConstituent():Constituent
+updateConstituent():Boolean
+deleteConstituent():Boolean
System.Data.OleDB

Figure 17: PIM - Domain Layer



APPENDIX E – CASE STUDY METRICS

Effort Tracking

Tasks	Subtask	Date	Actual	
Analysis				
	Interview	11/1/2006	0.75	
	Context	11/1/2006	1	
	Requirements	11/3/2006	1	
Traditional UML and C# coding approach				
Modeling				
	Use Cases	11/3/2006	1	
	Use Cases	12/18/2006	1	
	Use Cases	1/18/2007	1	
	Analysis diagrams	1/21/2007	1	
	Analysis diagrams	1/29/2007	2	
	Analysis diagrams	1/30/2007	2	
	Analysis diagrams	1/31/2007	1.5	
	Design diagrams	2/1/2007	1.5	
	Design diagrams	2/2/2007	1	
Code	Coding	2/3/2007	3	
	Coding	2/3/2007	3	
	Coding	2/3/2007	2.5	
	Coding	2/3/2007	1.25	Total code = 12.25
	Coding	2/3/2007	2.5	complete with initial draft code for constituent
Total			24.25	
MDSD approach				
Modeling	Use Cases	2/5/2007	1	
	Use Cases	2/6/2007	1	
	Use Cases	2/7/2007	0.5	
	Analysis diagrams	2/7/2007	2	
	Analysis diagrams	2/8/2007	1	
	Design diagrams	2/12/2007	2	
	Design diagrams	2/13/2007	1	
	Design diagrams	2/15/2007	1	
Code	Code Generation	2/16/2007	1.5	shell code stubs only for all methods, minimal documentation with documentation blocks
Total			11	estimated 25% code complete with domain object generation