FORMAL METHODS IN PRACTICE: ANALYSIS AND APPLICATION OF FORMAL MODELING TO INFORMATION SYSTEMS

by

Peter Andrew Geer

A thesis
submitted to the Faculty of the Graduate School of the
State University of New York Institute of Technology at Utica/Rome
in partial fulfillment of the requirements
for the degree of Master of Science

December 2011

Accepted and recommended for partial fulfillment of the requirements for the degree of Master of Science at the State University of New York Institute of Technology at Utica/Rome by:

Jorge Novillo (adviser)

Roger Cavallo

Michael Pittarelli

Abstract

Formal methods have been in development and use for 40 years. Proponents have long claimed that the application of formal methods would be a boon to the software development industry, leading to fewer defects, greater levels of quality, and lower overall cost. Furthermore, they promise to lend a formal, mathematical basis to the design and coding of software systems, allowing practitioners to apply more powerful methods of analysis and validation.

Despite these claims, uptake of formal methods in industry has been sluggish at best. While there have been a number of high-profile case studies published, formal methods are not commonly used in the development of "typical" business software, e.g. small to medium sized information systems. This thesis will explore the evolution of formal methods, their current uses and trends, and how they compare to current non-formal and semi-formal methods. It will describe some areas in which formal methods could be productively applied to the development of non-critical information systems and provide an example of how such an application can be specified and implemented using formal modeling with a modern, dynamic programming

language.

Contents

List of Tables

1 Introduction

Formal methods have been a controversial topic since they emerged forty years ago. They have been held up by some as the answer to the woes of software development, as a way to put the "science" back into "Computer Science." They offer a mathematical precision in program analysis that gives us the possibility of having certainly with regard to program correctness, something that simple testing and informal review can never provide. On the other side of the debate, detractors have dismissed formal methods as an exercise for academics. They have been described as too complicated and costly, requiring special training to apply and time and effort disproportionate to the gain in software quality.

As with most such disputes, the truth lies somewhere in between. Formal methods are by no means a staple of the main-stream software engineer's repertoire. However, they are not unheard of in industry, as there are a number of case studies in the literature that report the use of formal methods having a positive affect on the outcome of software engineering projects. Such success stories seem to be predominately with regard to so-called "critical system," although some cases of non-critical business systems using formal methods have been reported.

This thesis will explore the feasibility of using "light-weight" formal methods in non-critical information systems. It will examine some of the case studies of formal method use from the literature, examine some of the methods that are used in industry, and analyze the results of these studies and draw general conclusions as to the efficacy of formal methods in different areas of application. These will then be applied to a "light-weight" formal treatment of a web-based information system. This example is intended to demonstrate the application of light-weight formal modeling to modern web application development with dynamic programming languages and to provide a template approach to applying formal modeling to similar systems.

2 History

2.1 Formal program meaning

Broadly speaking, formal methods refers to the effort to assign formal mathematical meaning to the artifacts of the software development process. This formalism can be applied to high-level system or requirements specifications, low-level source code, and any level in between. The broad goal of these methods is to increase the quality of software development by making the requirements, design, or implementation of a system more explicit, thus making defects more readily apparent.

There are a large number of formal methods and notations described in the literature. They vary widely in their focus, how they are applied, and how widely they have been used, making them difficult to categorize. Furthermore, there are so-called semi-formal methods, which can serve a similar purpose, but do not offer a fully formalized semantics.

Traditionally, one method of categorizing formal methods has been to divide them into three groups based on how their semantics are defined:

axiomatic, denotational, or operational. Methods using an axiomatic semantics, such as described by Dijkstra and Gries, define operations by reference to their effects on program state, e.g. via pre- and post-conditions. Methods using a denotational semantics, such as VDM, fix the meaning of program operations by defining them in terms of mathematical structures, such as functions. Finally, notations using an operational semantics, such as Z, define the meaning of statements in terms of how they modify the state of some abstract machine.

Another way to roughly categorize formal methods is based on how they are applied. Two of the most obvious categories are modeling methods and code verification methods. Modeling methods, such as Z, describe a system abstractly, via a model constructed in the notation. Code-verification methods, such as Dijkstra's, describe the formal meaning of an implementation (usually in programming language code) directly, rather than indirectly via higher level models.

Many formal methods cut across these categories. For instance, VDM-SL is a modeling language, but sufficiently low-level models can be constructed as to define the behavior of individual programming constructs. Likewise, while VDM's semantics is denotational, its refinement method allows for an axiomatic interpretation of operations.

While these categorizations may have been more meaningful when formal methods were first gaining traction, they are less useful today. Formal methods have developed in many directions and there has been a great deal of cross-pollination between methods. The following sections will sketch the evolution of formal methods and describe some of the variations and directions it has taken.

2.1.1 Beginnings

Work in formal methods started in the 1960's, with R. W. Floyd and C. A. R. Hoare. Floyd's *Assigning Meaning to Programs* [Flo67] and Hoare's *An axiomatic basis for computer programming* [Hoa69] put forth a method of describing the meaning of a computer program mathematically.

Floyd's work was one of the earliest examples of the type of formal semantics that later evolved. Using example programs in a flowchart based language and in ALGOL, he described a system based on pre- and post-conditions for program execution. The meaning of a programming construct was fixed based on axioms and rules of inference. The method was essentially to start from an initial condition and, at each statement in the program, apply the inference rules for that statement to deduce what is true after it is executed. This is essentially the same type of method which Hoare later described.

2.1.2 Hoare Logics

Hoare's paper [Hoa69] introduced the basis of axiomatic semantics for programming languages upon which others later expanded. The paper introduces Hoare Logic and the so-called Hoare triple. This became the basis of further work by Hoare, Gries, and Dijkstra, among others.

Hoare Logic defines the meaning of a program statement using axioms that relate the precondition and postcondition of program execution. This is described by the triple P{Q}R, where P is the precondition of the program, Q is the program text itself, and R is the postcondition. This formulation has an operational semantics analog, where P can be seen as an initial system state and R as a final system state, with Q being a program that moves the system from the initial state to the final state.

Using Hoare logic, it is possible to deductively prove that a program is "correct." That is, one can use predicate logic to demonstrate that, for a given program, the postcondition of the Hoare triple is implied by the precondition. For each statement in the programming language, Hoare Logic defines an axiom which describes the meaning of that statement. By starting from the precondition and applying the appropriate axiom to each statement in the program, it is possible to determine the postcondition implied by the program. If this calculated postcondition matches or implies the original stated postcondition, then the program is "correct" with respect to the stated pre- and post-conditions.

In addition to this notion of simple correctness, Hoare Logics also have a notion of partial versus total correctness. This distinction refers to the termination of looping statements. Informally, a "partially correct" program guarantees that, for program specification P{Q}R,*if the program terminates*, then if P is true initially, R will be true upon completion. It does *not* guarantee that the program *will* terminate (in which case there is no postcondition to check – the program loops indefinitely). Total correctness goes the extra step and establishes not only the correctness of the Hoare triple, but also proves that the program will, in fact, terminate. This is accomplished by extending the inference rules for looping, adding a bound condition to establish that the program cannot loop forever.

2.1.3 Dijkstra and Gries

Dijkstra and Gries expanded on Hoare's premise. This is exemplified in [Gri81], which summarizes the axiomatic semantic definitions for most of the basic imperative programming constructs. This includes the basic procedural programming statements such as conditionals and iterative statements, as well as more advanced constructs, such as the simultaneous assignment and procedure calls [GL80].

These definitions use the $wp()$ predicate transformer introduced by Dijkstra. This transforms a program statement and postcondition predicate into the "weakest precondition" for that statement – informally, the weakest condition that must hold prior to statement execution to make the postcondition true afterward. For example, the predicate transformer for an assignment statement is defined by the axiom $wp(``x := y'', P) = P_x^y$, where P_x^y denotes the postcondition P with all instances of x textually substituted for y. Thus if the postcondition is $i = x + 3 \wedge j < 7$, then the weakest precondition is

$i = y + 3 \wedge j < 7$. This is analogous to the Hoare triple $P^y_x\{Q\}P$.

Other, more novel work has been done with Dijkstra's wp() semantics. One example is the modification of the axiomatic semantics discussed by Nelson and Broy [BN94]. They expand on the nondeterminism in Dijkstra's original semantics, seen primarily in the definition of the `if` statement, which does not specify a method for resolving cases where more than one of the set of guard conditions is true. By removing the *Law of the Excluded Miracle*, they expand this nondeterminism and introduce a "fair choice" operator to execute two statements and nondeterministically return a result.

2.1.4 Object Oriented program semantics

The early work of Hoare, Dijkstra, and Gries focused on the semantics of procedural programming languages similar to ALGOL. Since then, software development has moved to a largely object-oriented paradigm. This gives rise to the need for further work to define the formal meaning of common features of object orientation. A number of researchers have turned their efforts in this direction.

One example of this is the work of Naumann and Cavalcanti [CN00]. They developed a formal semantics of a Java-like object-oriented language called ROOL. They use a semantics based on predicate transformers and take into account common object-oriented features such as visibility control and dynamic binding, as well as recursive classes and methods.

Another direction in object-oriented formalism is represented by JML, the Java Modeling Language.[LBR99] JML is an behavioral interface specification language for Java. It allows developers to specify the behavior of an interface or class in Java source code using special JML statements embedded in Java comments. These statements use a logic and model-based semantics similar to that of VDM.

2.2 Formal modeling

Formal modeling, as the name suggests, is the use of a formal language to create a model of a system. Such models can be created at various levels of abstraction. While some, such as JML, are designed to work at the source code level, this thesis is concerned primarily with more general notations, such as Z and VDM. These can operate at the source code level, but are not specifically designed to do so and are in no way tied to a particular implementation language.

Modeling languages are one of the most common formal methods used in published case studies. While they do allow for the rigor required to prove properties of the system, in many studies, there is no attempt to undertake such proof. The modeling language is employed simply as a design tool and an aid to understanding and specifying the system. Some case studies suggest that the use of formal modeling can be beneficial even when no hard proof is attempted. Such uses of formal methods without the use formal proof, or with only targeted, small-scale application of proof, are sometimes termed "light-weight formal methods".

2.2.1 VDM and Z

Two of the oldest and most established formal modeling languages are the Vienna Development Method Specification Language, or VDM-SL, and Z. VDM developed from work begun in the late 1960's and early 1970's at IBM's Vienna laboratory. Z was developed in the 1970's by the Oxford Programming Research Group.

In contrast to notations like Dijkstra's language or JML, VDM-SL and Z are not tied to any particular programming language and so are normally not used at the code level. They are, rather, used to model the specification or design of a system. Both VDM-SL and Z describe systems in terms of the data types involved, some variables representing the state of the system, and some operations on that state. The behavior of these data and operation is expressed through pre- and post-conditions as well as state invariants. Since these conditions and invariants are written in terms of a formal logic using set theory and predicate calculus, it is possible to prove various properties about them.

2.2.2 Semi-formal Modeling

In contrast to *formal* modeling languages, there are also semi-formal modeling languages. Undoubtedly the most well known such language is the

Unified Modeling Language, or UML. The UML is a graphical notation for modeling the structure and behavior of a system. While UML has a standard syntax and semantics, specified by the Object Management Group (OMG), it is semi-formal in the sense that it does not have a notation for describing the the semantics of the various components of a system. For example, in a UML class diagram, there is no formal notation for describing the meaning of class methods. (There is the Object Constraint Language which can be combined with traditional UML, but it is relatively new and not widely used by comparison.)

2.3 Purpose and Need for Formal Methods

There are many possible reasons for adopting formal methods. They address a number of needs in computing science and software development and so can offer different benefits depending on the reason they are applied. This section outlines a few of the reasons that have been put forth for adopting formal methods. Later chapters will explore these and other reasons in more depth – the following is simply meant to set the tone of this thesis.

2.3.1 Complexity

One of the reasons for the development of formal methods was the explosion in software complexity that started in the 1960s. At that time, software systems were quickly becoming more complex, but advances in tools and methods for development did not keep pace. Thus, there was an apparent need for new methods that would allow developers to get a handle on this complexity. Formal methods made this possible by providing a mathematical framework for analyzing programs.

2.3.2 "Scientific" programming

The term "software engineering" is, in some ways, nothing more than a cruel joke. Engineering is generally considered a practice founded on sound, disciplined scientific reasoning and empirical data. Yet software development, to a large extent, remains an extremely *ad hoc* and haphazard field of endeavor, often appearing to be based as much on fads as on facts.

Formal methods of analysis would provide a mathematical basis for software development, putting it in line with proper scientific or engineering disciplines. Without such a mathematical grounding, it can be argued that programming is no more than a black art, or at best a "craft" akin to wood working.

2.3.3 Improve reliability

Perhaps the most obvious and compelling reason to adopt formal methods is that mathematical analysis offers a method by which programs can be shown to be free of bugs on some level. As E. W. Dijkstra famously quipped, "Program testing can be used to show the presence of bugs, but never to show their absence." Of course, even program proof is not enough to guarantee that a system will have zero defects. However, it does allow for stronger assertions of correctness than testing does. It can also be applied to a system even when real-world testing is not feasible for cost or safety reasons.

3 Overview of Formal Methods

Formal methods are not a unitary entity, but rather an umbrella term for various levels and methods of formal analysis.

As discussed in the previous chapter, the earliest efforts in formal methods focused on code-level analysis. This is exemplified by the axiomatic system described in Gries's *The Science of Programming* [Gri81], which focuses on proof of code correctness and the derivation of programs. Such methods focus on the lower levels of implementation, i.e. justifying specific code.

While such approaches are useful, they are not well equipped to address problems in system specification or design. It has long been known that errors in these higher levels of system construction can be the costliest to fix if not discovered until late in implementation. To this end, researchers have developed a variety of higher level formal modeling notations. These are designed to describe the behavior of systems independent of particular implementation. This allows for formal analysis to be performed earlier in the development cycle and find design defects.

This chapter will discuss some of the more commonly used and cited formal methods currently in use, as well as how they have been applied.

3.1 Methods

There are a considerable number of formal methods that have been discussed in the literature and used in industry, varying considerably in scope and approach. This section will provide an overview of a few of the more common or noteworthy methods, with attention to the methods that were used in the case studies presented later.

3.1.1 SPARK

Perhaps the most noteworthy code-level formal method is SPARK Ada, the cornerstone of the "correctness by construction" approach of Altran Praxis. Researchers and engineers at Praxis have used SPARK successfully in many commercial projects and have published a number of journal articles detailing their results.

SPARK is simultaneously a programming language and a tool set. SPARK-Ada is a subset of the standard Ada programming language designed to be susceptible to static proof of correctness. SPARK programs are valid Ada programs, but contain comments, called contracts or annotations, describing the semantics of the code, including pre- and post-conditions, data constraints, etc. The SPARK tool set contains utilities to validate these annotations, generate and discharge proof obligations, etc.

The SPARK tool set is designed to insure high levels of system correctness. Thus most of the published case studies involving SPARK have involved critical or high-integrity systems. Several of these systems are discussed in the next chapter.

3.1.2 VDM and VDM-SL

The Vienna Development Method, commonly referred to as VDM, is a true formal development *method*, including a specification language, refinement rules, and proof theory. It was developed in the 1970's at IBM's Vienna laboratory. It arose from work done in the late 1960's using VDL, the Vienna Definition Language, to describe abstract interpreters for programming languages such as PL/I. VDM was the result of the Vienna group's attempts to systematically develop a compiler from a language definition. [BJ78] VDM was standardized by the ISO in 1996.

The general principle of VDM as a method is that of refinement. This is the process of converting abstract specifications into more concrete representations through a mathematically justified series of steps. To this end, at each step in the process, VDM provides rules for satisfying proof obligations and retrieving more abstract representations. Because this process results in a representation that is closer to a real "thing" and is, in that sense, less "refined," it is sometimes referred to as *reification* rather than *refinement*.

Perhaps the most commonly used portion of VDM is the specification language, VDM-SL (sometimes imprecisely referred to simply as "VDM"). It is a state-based modeling language with a logic based on first-order predicate calculus and set theory. VDM-SL uses a denotational semantics, with operations defined in terms of functions mapping inputs to outputs.

While VDM-SL is often used as a requirements specification language, it is equally well suited to system design. It allows for a great deal of flexibility in data definition and has a rich set of primitive types. VDM-SL also has two modes of specification: implicit and explicit. An implicitly specified operation is described using a precondition and postcondition, not unlike a specification in Hoare Logic. An explicitly defined operation, by contrast, is defined in much the same way as a procedure in a programming language – as a series of explicit applications of functions and primitive statements. Operations specified in this mode can sometimes be translated almost directly into program code, if desired.

3.1.3 Z

The Z (pronounced as the British "zed") formal modeling language was developed in the late 1970's by the Programming Research Group at Oxford University. It is based on first-order predicate logic and the Zermelo-Fraenkel set theory. The Z notation was formalized by the ISO in 2002 as ISO/IEC 13568:2002.

Z has been in use for many years and is one of the more widely discussed formal methods in the literature. It has spawned a number of variations to support more advanced and specialized features, such as object orientation and concurrency. It also has a relatively large number of resources and supporting tools available, both commercial and freely available.

The most distinguishing feature of Z, and also its primary structuring construct, is the schema. Z schemata are a boxed notation used to describe both system state and operations. They encapsulate data and state invariants as well as providing for pre- and post-conditions. Z also allows for the composition of schemata, making it possible to build up specifications in an incremental manner.

3.1.4 Object-Z

As the name suggests, Object-Z is an object-oriented extension of Z. It was developed at the University of Queensland in the late 1980's and early 1990's. [Smi00] Object-Z is a conservative extension to standard Z, maintaining the same syntax and semantics such that valid Z specifications are also valid Object-Z. The extended notation adds classes and support for object-oriented concepts such as inheritance and polymorphism.

3.1.5 VDM++

VDM++ is an object-oriented extension to VDM-SL. It was developed in the 1990's by Larsen and Fitzgerald as an attempt to bridge the gap between traditional formal modeling and object-oriented design.

Being an extension to VDM-SL, VDM++ inherits many of its characteristics. The main difference is in the introduction of classes. This is visible in the structure of VDM++ specifications, which are written using classes and inheritance for structuring rather than the modules used in VDM-SL. Interestingly, there is also a visible difference in the syntax of VDM++, which has shifted from the traditional mathematical notation to an ASCII-based notation similar to a programming language.

VDM++ will be discussed in greater detail later in this thesis.

3.1.6 TCOZ

Timed Communicating Object Z, or TCOZ, is an example of a hybrid formal method, composed of Object-Z and Timed CSP. Timed CSP is an extension of CSP, which stands for Communicating Sequential Processes. CSP is a notation developed by Hoare in the late 1970's for modeling process concurrency and interaction. Timed CSP extends this notation with primitives for representing time. TCOZ integrates these two notations by identifying Z schemata with CSP processes – for example, representing non-terminating processes via Object Z classes and processes for state update events as operation schemata.

3.2 Ways to apply FM

In the past, it has been common for formal methods to be represented as an all-or-nothing approach to development. In fact, formal methods have sometimes been equated with proving code correctness. However, this is far from the case.

There are many different ways to approach formalization. A number of case studies have shown that formal methods can be profitably applied to systems without going so far as to prove every line of code. This section will describe some of the ways in which formal methods have been applied. Some examples of various levels of formalization will be given in the examination of case studies in the next chapter.

3.2.1 Full formalism with proof

The most "heavy-duty" application of formal methods is the use of full formalism and proof, which is often thought of as the classic formal methods scenario.[Hal90] In this approach, a system uses formal methods in the design, such as by writing specifications in a formal modeling language like Z, as well as in the implementation. This is the stereotypical "every line is proven" approach, in which all steps are formally justified.

In practice, it is uncommon for development teams to apply formal methods in this way – it is considered to be impractical for most projects. Few of the published case studies involve this level of formalism. Without high-quality tool support, the demands of full proof could easily become unmanageable. Furthermore, case studies suggest that significant improvements to reliability and correctness can be had without such extensive verification, so lighter-weight methods are often judged to be more economical.

3.2.2 Full requirements or design

A more common application of formal methods is to use them in the requirements and design of the system. By using a formal notation to write the design or requirements of a system with an eye toward proof, system architects can formally analyze these documents for correctness, consistency, and completeness. For highly detailed formal models, this might include proof and model animation, i.e. execution of the formal model for validation purposes. This is, of course, not possible with natural language design documents.

Such application of formalism can be thought of as the high-level equivalent of code proof. It allows system designers to construct proofs of system properties such as security, which can be critical to the success of a project. However, as with full formalization, the literature suggests that this level of proof is not necessary to realize benefits.

3.2.3 Semi-formal design or requirements

Another, less intensive way to use formal modeling languages for the design or requirements of a system is for purposes of increased tractability. In this scenario, designers use a formal notation to specify parts of the system, but do so for the purpose of explicitness and clarity. In this scenario, large-scale proof of correctness or consistency is generally not a goal.

The idea behind such application is to discover defects in the early stages of the project. As Brooks has pointed out [Bro95], the most costly errors in software systems are design errors detected during the coding or testing phases of the project. In this approach, formal methods are used to subject requirements and designs to greater scrutiny, with the intent of discovering errors while the cost of fixing them is still low. The result of this is manifested as fewer defects discovered during the coding, testing, and deployment phases of the system life-cycle.

3.2.4 Targeted application

Formal methods can also be used in a very light-weight, targeted way. They can be applied not to an entire system, but merely to selected subsystems, with or without performing proof activities. For example, a development team might apply formal specification to prove the security properties of an access control system. Another example would be applying formalism to add clarity and explicitness to a particularly devious or vaguely defined subsystem.

Such targeted application allows practitioners to avail themselves of the advantages of formal methods only for those areas where it will be most helpful. This can include areas of requirements, design, or even code verification. While this approach does not offer guarantees of correctness, it can help to alleviate concerns that formal methods are too expensive or difficult to use while still increasing system quality.

3.2.5 System review

Formal methods can also be useful in efforts to review, evaluate, or verify an existing system. Though most practitioners recommend that formal methods be incorporated into a project from the ground up, they can still be useful when questions are raised about the reliability of an already developed system. For example, the use of formal methods was deemed the route of least expense in achieving the certification of the Darlington nuclear generator station.

3.2.6 Test case generation

As with natural language specifications, formal specifications can be used for test development. Black-box testing, the development of test cases from system specification documents rather than implemented software, is a common technique in software testing and quality assurance. Formal specifications aid in this type of testing, as they allow for more precise and explicit specifications, making the test case design more straight forward.

In addition, the mathematical nature of formal methods makes them susceptible to automated analysis. Thus, in some cases, it is possible to automate the generation of test cases from the specification. The clearest example of this is unit testing, where the test cases are generally small and straight-forward. The pre- and post- conditions of formal specifications provide easy targets for test cases, as they provide natural boundary conditions to verify.

3.3 Places to apply formalism

Just as different formal methods can be applied in different ways, so too can they be applied to different parts of a system. This section will describe some of the ways that formalism can be applied to the various portions of a system and phases of system construction.

3.3.1 Code level

As previously discussed, one of the most obvious and well known ways to apply formal methods is to program source code. A formal, mathematical meaning can be defined for the various programming language statements, independent of how they are implemented on a computer. This makes the text of a program itself susceptible to mathematical analysis, allowing the program to be proven to be “correct,” i.e. to satisfy a given formal specification. This is exemplified by the axiomatic semantics of Hoare [Hoa69], Dijkstra, Gries [Gri81], *et al.*

Typically, the purpose of code-level formalization is to ensure that the code is free of semantic errors. For instance, with the semantics described in [Gri81], programs can be annotated with pre- and post-conditions at various points. Through the semantic rules of the language, it is possible to prove whether, upon termination of the program (if, in fact, it does terminate), the precondition entails the truth of the postcondition. Thus the correctness of the program “specification” can be established.

3.3.2 System design

As discussed above, formal methods can also be used at the system design level. Formal specification languages like Z or VDM-SL can be used to model the components of a system. This allows designers to reason formally about the system prior to implementation and makes it easier to spot contradictions or ambiguity in the natural language specification documents.

Note that formal modeling languages can be applied to both high-level design and detailed design. That is, they can be used to represent the components of a system as well as the abstract data types and operations used in implementation. System designers may even choose to use formal methods at both levels, allowing them to prove that a detailed design is consistent with a high-level design.

3.3.3 Database design

As demonstrated by Barros [dB94a], it is possible to apply formal modeling languages to the design of relational databases. Relational database constructs, such as tables, relations, and constraints, can be mapped to features of notations based on logic and set theory, such as Z, allowing a database schema to be subjected to formal analysis.

Such analysis is particularly of interest in the development of information systems, which are often driven by data. In such applications, formal modeling of the database schema can be used to capture complex data constraints and evaluate potential data models. Such an exercise can be useful not only for formal system analysis, but also for design and documentation purposes. Formal models can be built at a higher level of abstraction than is available in standard SQL, allowing for simplification of complex entities and non-relational constraints (e.g. triggers and stored procedures).

3.3.4 Requirements analysis

In addition to system design, it is also possible to use formal modeling methods to provide a formal description of certain system requirements. This allows system designers to subject the requirements to consistency analysis and the system design to coverage analysis, allowing them to detect errors at a very early stage. It may also facilitate iterative requirements development by providing a structured representation that can be systematically refined.

Of course, this is not practical or possible with all requirements. For instance, a system may have user interface requirements which are difficult to formalize and which would not offer much return on the investment. However, as with other areas of formal analysis, it is not always necessary to formalize all requirements to get beneficial results.

3.4 The “methods” in formal methods

Some researchers have quipped that “the problem with formal methods is that they are just formal, not methods.” [CMCP$^+$99] This section will examine some of the issues of methodology and areas of application for formal methods.

3.4.1 System review

Formal methods of various stripes can be used for the methodical review of an existing system. Analysts can translate an existing implementation or design into a formal notation and subject it to consistency or correctness analysis.

One example of this is [Pol01], in which the SAZ method, a combination of the Z modeling language and the Structured Systems Analysis and Design Methodology(SSADM), was used to review a GIS information system. The use of formal modeling notations for this task enables the reviewer to demonstrate inconsistencies in the system, determine if all requirements have been met, and so forth.

While such application may be thought of as a classic scenario for formal methods, it should be noted that such scenarios are not dominant in the published case studies. Many of the most noteworthy case studies have involved integration of formal methods into the development process, rather than using them for *post hoc* validation. Such early integration tends to uncover errors earlier in the process, saving time and money, which is not possible with after-the-fact analysis.

3.4.2 Automated tool support

Because formal methods are mathematical, they are at least theoretically susceptible to automated analysis. That is, software can be used to do the “heavy lifting” of consistency checking, discharging proof obligations, and so forth. This leaves the engineers involved with more time to concentrate on the development of the system, rather than getting bogged down in the mathematics of verification. In this scenario, the application of formal analysis is worked into the development and becomes another step in the process, similar to the writing and execution of unit tests in test-driven development.

One example of this type of development method is the SPARK language and tools used by Altran Praxis. In the methodology employed by Praxis, the supporting tool set can be viewed as the other part of the pair in the pair-programming prescribed by Extreme Programming[AC03]. The programmer develops his software with SPARK annotations, which the tool set can then use for semantic analysis, theorem proving, information and data flow analysis, etc. With a machine handling the “grunt work” of the analysis, the engineer is free to spend more time analyzing the results of that analysis.

3.4.3 Limited formalization

A common theme in modern formal methods use is that formalism is best when it is targeted. There is often no need to formalize every part of a system, as productivity and reliability gains can still be realized, probably at lower cost, by selective application. In particular, formal methods use is most often focused on the design and specification stage of a project, with less emphasis being placed on code-level proof. In a limited application of formalism, developers can identify the highest-risk or most critical sections of a system and apply formalism to the design or implementation to increase confidence in the reliability of the system.

One example of this application is Praxis' development of the Multos smart-card system[HC02]. In this project, Praxis developed a secure certification authority for the system using formal methods. However, rather than developing the entire system using formalism, they used Z and SPARK only in targeted areas, i.e. the modules where reliability and security were critical. For other parts, such as the graphical user interface (GUI) and the underlying operating system, they used commercial, off-the-shelf (COTS) software.

3.4.4 Micro-methods

It has been suggested [Jac98] that the proper role for formal methods is not so-called "constructive methods", but rather "micro-methods" – small-scale, highly focused applications. The argument is by analogy to traditional (e.g. civil and industrial) engineering, where large-scale design innovation is rare. Rather, many of the design details of an item to be produced are predetermined – a bridge has to have a deck, and there are only a few acceptable ways to build it. Likewise, in software development, starting completely from scratch and coming up with a completely new way of doing things is, or should be, relatively uncommon.

The idea is that formal methods should be the *basis* of software development practice, not the practice itself, in the same way that mathematics and physics are the basis of civil engineering, but do not make up the every-day practice of it. Thus formal analysis should underly the methods and "rules of thumb" used by software practitioners, giving them a scientific backing.

This approach could be viewed as somewhat similar to the application of design patterns in object-oriented analysis and design. Systems may be composed of various patterns or components which have been subjected to formal analysis and can be combined in formally defined ways. Thus systems would have a formal underpinning without requiring developers to spend a great deal of time doing the formalization. However, while such an approach has been suggested, it has not seen significant adoption in practice yet.

4 Applications of Formal Methods

Traditionally, formal methods have been used primarily in the development of security- and safety-critical systems. Such systems are typically defined as ones in which failure can lead to catastrophic loss, such as loss of life or compromising national security. The common thread in such systems is that the cost of failure is extremely high and therefore the correctness of the system is paramount. In such cases, a reasonable amount of extra cost or effort to guarantee system correctness is considered acceptable, if not essential. Formal analysis is a natural fit for such situations, as it allows for demonstrating the correctness in a rigorous, systematic manner.

However, despite common misconceptions on this subject [Hal90], formal methods are not used or useful only in the area of critical systems. They are useful for any area in which the reliability of the system is important, especially if the cost of fixing post-release bugs is high. Embedded systems are an excellent example of this. Upgrading the software on a ROM chip in a non-networked device is relatively difficult, requiring either a recall notice or non-trivial action on the part of the device owner. Such updates can be costly both financially and in their impact on a company's reputation.

There have been many case studies on the use of formal methods in industry published over the years. These cut across many industries and include a variety of methods applied to systems of varying types and sizes. This chapter will provide an overview of some of these case studies and discuss their results.

4.1 CICS

CICS is the Consumer Information Control System, an on-line transaction processing system, developed by IBM, with thousands of users world-wide. CICS was originally released in 1968 and was developed using conventional software engineering methods. For CICS 3.1, released in 1990, IBM redeveloped several parts of the system. They worked in conjunction with the Oxford University Programming Research Group (PRG) to apply formal methods to the effort.

As their method of choice, the project team used primarily the Z notation. Z was used for higher-level specification and design work, but was not mapped down to the detailed design or code in a systematic way. Rather, some of the code was formalized using a variant on Dijkstra's guarded command language. In general, there was no formal relationship between the high-level Z specification and this guarded command code and there were no large scale proofs attempted. [FF96]

The CICS effort included the development of new subsystems and the redevelopment of existing subsystems. This included 268,000 lines of new and modified CDL (IBM's Common Design Language) code. Of this, 37,000 lines were based on Z specifications, while another 11,000 were partially specified in Z. The Z specifications themselves occupied approximately 2,000 pages.

The results of this project were hailed as a success for formal methods, winning IBM and Oxford the UK Queen's Award for Technological Achievement [BH95b]. IBM reported that the "Z code" had 2.5 times fewer defects than the conventionally developed code. They also reported a 9% savings in the overall project cost, which was attributed to the use of formal methods.

However, it should be noted that [FF96] has raised questions about the legitimacy of these claims. In particular, the paper points out that the publicly available data does not provide a basis for the precise claims of success. For example, it is unclear exactly what comparison leads to the 2.5 times figure and it appears that the 9% savings does not include post-release defects. So while CICS may well have been a success, it is not clear how large a success.

4.2 FM vs. CMM experiment

In a 2001 article, Smidts et al. described an experiment pitting two development teams against each other. [SHW02] Both teams worked independently to develop a system to the same specification, with one team employing formal analysis methods and the other using the more conventional Capability Maturity Model (CMM) level 4 development methodology. Both teams were given what they agreed was a "generous" schedule and funding as well as unlimited access to the customer for clarification of the specification and the ability to negotiate for a reduction in scope if required. The deliverables were to include C++ source and object code along with documentation on process, design, and testing.

The system to be designed was the Personnel Access Control System (PACS), which controls access to buildings using a swipe-card system. Included with the specification for the system were reliability guidelines. The system was to have reliability of 0.99/transaction for level 1 failures and 0.90 for level 2 failures. A level 1 failure was defined as a state in which the software is unresponsive or incorrectly processes cards or PINs, either allowing or disallowing entry when it should not. A level 2 failure was defined as one for which there was an operational work-around, e.g. someone going through the entrance too slowly and requiring a guard to override the system to keep the gate opened. The reliability of the delivered systems was tested by an outside lab.

The formal methods team in this study chose to employ Haskell and Specware as their primary tools. Haskell is a purely functional programming language with a number of advanced features. Specware is a specification and proof environment which allows users to write specifications in the Metaslang formal specification language, refine them, and then automatically generate C, Java, or Lisp source code for the application. In this case, the developers actually chose to build a prototype system in Haskell, automatically convert that to a Specware specification, and then use the code generation features of Specware to create the C++ source code.

In the final analysis, neither team was able to fully meet the specification. The measured reliability for the product delivered by the CMM4 team was 0.56 for level 1 flaws and 0.97 for level 2 flaws, while the formal methods team achieved rates of 0.77 for level 1 and 0.65 for level 2 flaws. So while the CMM team did meet the reliability requirement for level 2 defects, the formal methods team was significantly more reliable with respect to level 1 flaws – though they still came nowhere near the required reliability. However, given the unorthodox approach of the formal methods team, it is unclear if their performance is indicative of any general trend.

4.3 Light-weight review

One example of the use of formal methods in a non-critical system is that of [Pol01], which describes the application of a light-weight formal review method to a GIS information system. The system in question was an Oracle database used to store topographical and tourist information.

This case study involved creating a formal Z specification from a partial structured system specification. The analysis used a technique known as SAZ, a combination of the Z specification language and SSADM, the Structured System Analysis and Design Method. This resulted in a Z specification that was more structured than typical Z.

The formal review resulted in the addition of a number of constraints to the system specification. However, the problems discovered were not especially serious, i.e. the cost of fixing them would not have been outrageous. It was noted that these issues may have been serious for a more public, higher-profile system.

Overall, this case was considered a successful application of formal methods, but did not provide evidence of a clear and compelling benefit. The study was somewhat subjective and hard to generalize, as well as lacking a base-line for comparison with other methods, making it a sub-optimal test of formal methods. However, it did at least establish that such review methods are feasible and useful for non-critical systems.

4.4 Pondage power plants

In [CMCP$^+$99], an application of semi-formal methods to the development of a Pondage power plant is described.

This case study used a formal specification language known as TRIO and a “double-spiral” evolutionary development method. This involves starting from a natural language specification which is revised into a partial TRIO specification and then converted to the final specification through a series of refinements and verifications. The methods are referred to as semi-formal because they do not require that all parts of the system be formalized.

The authors of this case study conclude that the use of formal methods consistently results in discovering more defects in the specification stage, which ultimately leads to lower costs. This is based on 10 years of experience, including projects such as a digital energy meter, dam static safety elaboration unit, Pondage power plant control system, the Open DREAMS project, a traffic light control system, and a flight control system.

In the case of the Pondage power plant, the study concludes that the system developed with TRIO cost 15% less than a system developed with more conventional methods. It is noted that, in the price breakdown, the TRIO development spent twice as much on the design and validation stage, but less on all the other development stages, resulting in the net 15% savings.

4.5 Radiotherapy machine

A 2004 study by Dennis et al. [DSRJ04] dealt with the use commutativity analysis to find errors in radiation therapy machine. The study involved translating a design specification originally written in OCL (the UML Object

Constraint Language) into the Alloy formal specification language. The use of the Alloy tool set allowed researchers to automate this analysis. Such automation which was not possible with the current OCL tools.

This study examined commutativity of operations, i.e. the effect of changing the order in which operations were. The system was was designed to be single-user and single-threaded, and so had no notion of concurrency – all commands were executed in series. However, the system had multiple control panels located in different rooms. Thus it was possible for two users to issue different commands at the same time without each other's knowledge. In some cases, executing these commands in the wrong order could leave the system in an undesirable state.

The analysis successfully uncovered several operation pairs that did not commute. While these are not necessarily errors, having non-commutative operations could put the system in an ambiguous state. Users might not be able to predict the outcome of an operation based on the observed system state, which could result in a negative impact on patient care.

4.6 International Survey

In 1993, Craigen et al. released a two volume report on applications of formal methods. [CGR93] The report, which details the results of studies on several large, high-profile formal methods-based projects, is one of the most comprehensive studies on formal methods.

Step	Description	Classes	KLOC
1	Management tools	153	25
2	Tool-writers class library	99	12
3	Tools (5 total)	81	8

Table 4.1: SSADM code break-down

Requirements	7%
Specification	29%
Development & unit testing	50%
Testing and delivery	14%

Table 4.2: Total effort on SSADM by phase

4.6.1 SSADM

In 1987, Praxis High Integrity Systems began working on a Computer Aided Software Engineering (CASE) tool set to support the Structured System Analysis and Design Method (SSADM). The project involved the creation of an infrastructure, i.e. a framework, to support the method as well as a toolkit to support actual development. The infrastructure was formally specified in Z and implemented in Objective C.

Of the 45KLOC in system (where KLOC denotes "thousand lines of non-empty, non-comment code), 37KLOC were specified in the 350 pages of Z. This included 550 schemata and 280 top-level operations. The code break-down is shown in table 4.1. Steps 1 and 2 collectively took 2235 work days to complete, while step 3 took 483. The estimated effort was 6400 work-days, based on an assumption of 1120 function points. The percent of effort expended in each phase of the project life-cycle is given in table 4.2.

The project used only a single tool to support the use of formal methods:

a prototype parser and type-checker obtained from the FORSITE project. This was the only tool readily available at the time and was judged to be just barely adequate.

Several factors make it difficult to draw conclusions about formal methods based on this project. The first factor is the fact that, for reasons outside the scope of Praxis's work, the customer decided not to use the final product. This makes it impossible to assess the long-term effects of the use of formal methods on maintenance, as no maintenance was done. Second, while Praxis claimed that the project was delivered within budget, no data was collected on the price breakdown within the project. Thus it is not possible to evaluate the cost effectiveness of formal methods.

Praxis did collect data on productivity, as per the client's requirement. For this, they used the COCOMO model, which predicted productivity of 11 lines of code per day. The measured productivity under this model was 17 lines per day. While this supports Praxis' claim of increased productivity due to the use of formal methods, Craigen et al. regard the COCOMO data as not that meaningful, owing to the difference between the development model assumed by the method and that actually used in the project.

4.6.2 Inmos transputer

In 1985, Inmos Ltd. began a project to use formal methods for the design of microprocessors, in particular their Transputer family of 32-bit Very Large Scale Integration (VLSI) circuits.

There were three interrelated projects at Inmos. The first used Z to create a specification for the IEEE floating point standard, the second used Z and Occam to design scheduler for T800 Transputer, and the third used the Communicating Sequential Processes (CSP) and Calculus of Communicating Systems (CCS) to design the Virtual Channel Processor of the T9000 Transputer.

The results were generally positive. Inmos met their quality and time-to-market goals. They also saved an estimated 3 months of development time on the T800, saving $4.5 million. The project also won the Queen's award.

4.6.3 Darlington nuclear generator

The Darlington Nuclear Generating Station (DNGS) represents a retrospective formalization, i.e. reverse engineering. In this case, an existing software system was subjected to formal analysis in order to gain confidence in its reliability.

The system in question was the DNGS shutdown control system. It was developed by AECL for Ontario Hydro, the operator of the DNGS. Formal methods were not used at all in the development of the control system, but were introduced after development had been completed. This course of action was taken because Ontario Hydro was having difficulty obtaining a license from the Atomic Energy Control Board (AECB) of Canada due to questions about the reliability of the system.

This project did not use a specific formal method. The requirements specifications were based on the Software Cost Reduction (SCR) style and the actual formalization used tabular representations. The Software Design Specification (SWDS) was formulated mathematically in a tabular format. The function tables were also developed for the various routines in the system, with linkage tables describing dependencies. Verification consisted simply of showing that the SWDS and function tables were consistent. No tools or automation were used in the formalization effort - all proofs were constructed by hand.

As a result of the formalization effort, the AECB found that the software shutdown control system was no longer a barrier to licensing. However, they stated that the system was not suitable for long-term use and that it would have to be redeveloped. The cost of the verification was $4 million, 25% of the $16 million cost of the total cost of the shutdown system.

4.7 Various Praxis projects

Altran Praxis (formerly Praxis Critical Systems and Praxis High Integrity Systems) has published a large number of papers on projects they have done using formal methods. Praxis has successfully used a relatively wide range of formal methods, including Z, VDM, CSP, and SPARK. This section will examine some of the case studies published by Praxis and the results of those projects.

4.7.1 Multos CA

In [HC02], Hall and Chapman describe the development of the Multos-CA, a certification authority for the Multos smart card scheme, by Praxis Critical Systems. This project was noteworthy in large part due to the stringent security requirements mixed with the requirement for the use of commercial, off-the-shelf (COTS) software.

The system was developed using a "correctness by construction" approach. This entails collecting requirements and refining them down through a series of high and low level designs. The Reveal method was used for requirements engineering.

Praxis used a number of formal methods in the development process. Z was used to develop both a top-level formal specification and a formal security policy model. A Z specification was also produced for the module used to manage cryptographic keys. A process model was developed using CSP (Communicating Sequential Processes) to ensure the correctness of threading and inter-process communication in sensitive code. For the coding phase, these models were translated into Ada95 tasks, rendezvous, and protected objects.

The system was coded in multiple programming languages, using a "right tool for the job" approach. Security sensitive portions of the system, accounting for about 30% of the code, were written using SPARK, Praxis's statically verifiable subset of Ada95. Another 30% of the system consisted of custom inter-process communication (IPC) mechanisms and API bindings implemented in full Ada95. Other portions, such as the user interface, were implemented in C++ and were kept as separated as possible from the security-sensitive code. This dichotomy was due in part to the customer requirement for the use of a commodity operating system, Windows NT 4.

Overall, the certification authority was a success, meeting requirements and budget constraints. The average overall developer productivity for the project was 28 lines of code per day, which compares well with industry averages. The defect rate was remarkable as well. In the first year after system acceptance, only 4 faults were discovered. That makes for a defect rate of 0.04 defects per thousand lines of code.

4.7.2 SHOLIS

SHOLIS is the Ship Helicopter Operating Limits System. It is a system, developed by Praxis Critical Systems, to aid in the safe operation of naval helicopters. The system contains a database of SHOLs (Ship Helicopter Operating Limits) and compares these to sensor readings. When a limit is violated, the system raises visual and/or audible alarms. It was developed to SIL4 (Safety Integrity Level 4) as per UK Defense Standard 00-56. [KHCP00]

The SHOLIS was built using Z for the design phase and SPARK Ada for coding. It used a design cycle consisting of:

1. English requirements (about 4,000 statements).

2. Software Requirements Specification (SRS) in English and Z (about

   300 pages).

3. Software Design Specification (SDS) in English, Z, and SPARK.
4. Code in SPARK.
5. Testing.

The final system comprised about 133,000 lines of code total, which included 13,000 lines of Ada declarations, 14,000 lines of Ada statements, 54,000 lines of SPARK flow annotation, 20,000 lines of SPARK proof annotation, and 32,000 blank or comment lines.

This project made extensive use of proof, both at the Z and SPARK level, the use of which was judged to be quite successful. In total, 150 Z proofs were carried out in 500 pages, including 130 at the System Requirements Spec level and 20 at the System Design Spec level. About 9000 SPARK verification conditions were generated, with 3100 for safety and functional properties and 5900 from the RTC generator. Of these 6800 were automatically discharged by the SPARK Simplifier and the rest were discharged manually with the SPARK Proof Checker or with informal reasoning.

4.7.3 Lockheed C130J

Praxis also participated in Lockheed's project to performed a major upgrade to the Mission Computer (MC) system on the Hercules II air-lifter, also known as the Lockheed C130J. The system was developed by multiple contractors to satisfy the civil certification DO-178B and UK Defense Standard 00-55. It used semi-formal specifications written with the Consortium Requirements Engineering (CoRE) method and Parnas tables. The core of the MC, which accounted for approximately 80% of the total system, was written in SPARK Ada. This code was subjected to the static analysis supported by the SPARK tool set, including semantic checking, data- and information-flow analysis, and program proof. [Ame02]

Lockheed claimed a number of improvements due to the use of SPARK. Among them were:

- Code quality increased 10 times over typical code developed to DO-178B Level A.
- Developer productivity increased by 4 times compared to previous, comparable projects.
- Development costs were half that of typical non-safety critical code.
- Re-use of the development process led to "further productivity improvement of four" on the C27J air-lifter project.

Another significant conclusion of the study is that SPARK code was found to have only 10% of the errors of traditional Ada, which itself had 10% of the errors of C code. Lockeed reported that the static analysis supported by SPARK decreased Modified Condition/Decision Coverage (MC/DC) testing costs by 80% from the expected testing budget. In addition, there was no statistically significant difference found between the residual error rates of DO-178B Level A and Level B code, raising questions about the efficacy of MC/DC testing.

4.7.4 SIL4 failure

In contrast to the successful application of SPARK in the Hercules II, Chapman gives a brief example of a less successful application of SPARK[Cha00]. This unnamed project had the goal of developing a SIL4-compliant real-time embedded control system. It used an object-oriented design style driven by a CASE tool. The system was not initially developed with SPARK, but was converted to SPARK *after* testing in order to meet regulatory requirements.

In this case, the attempt to “SPARK-ify” the code after the fact proved fatal. The CASE tool generated a “flattened” code structure, with all state stored at the global level. This goes against good SPARK practice. In addition, some of the generated code violated the semantic rules of SPARK, requiring code changes late in the development cycle. At this point, progress slowed and the system failed to meet requirements.

4.7.5 CDIS

The *Central control function Display Information System* (CDIS) is an Air Traffic Control (ATC) information system developed for UK Civil Aviation Authority to display information to air-traffic controllers. It was developed under contract by Praxis Critical Systems using a number of different formal methods. [PH97]

CDIS was designed using VDM to define the system data and operations on it. The definition proceeded in a top-down fashion, after it became apparent that the initial idea of using VDM to refine the low-level design was not appropriate. The system also used CCS, the Calculus of Communicating Systems, and entity-relationship modeling.

The completed system consisted of approximately 197,000 lines of C code, accompanied by 1200 pages of specification documents and 3000 pages of design documents. The total effort for the project was calculated at approximately 15,500 person-days. Average developer productivity was 13 LOC/-day, which was better than the average predicted by the Cocomo estimation method and at least as good as comparable projects undertaken by Praxis. During development, the fault rate was 11/KLOC, with the final product having 0.75 faults/KLOC in the first 20 months after delivery.

5 Controversy and debate over applicability

Over the years since formal methods research began, there has been much debate over their usefulness and applicability in practice. This chapter will examine details of some of the arguments for and against the use of formal methods.

5.1 Why are formal methods not used?

While formal methods are used in industry, they have not enjoyed wide adoption and remain a relatively niche area of software development. It is not entirely clear why formal methods have failed to achieve any real mainstream success. There have been many case studies published documenting the successful use of formal methods, and many papers written by evangelists addressing myths and misconceptions. Despite this, formal methods have not enjoyed anything approaching wide adoption.

By way of contrast, there has been significant uptake of some so-called semi-formal development methods. The prime example of this is UML, the Unified Modeling Language. UML is a graphical notation, designed and maintained by the Object Management Group (OMG), for describing various aspects of object-oriented systems. It has become the *de facto* standard for object-oriented modeling.

The UML is "semi-formal" in the sense that the UML standard determines valid syntax for UML diagrams, but does not provide for the meaning of those diagrams. For instance, a UML class diagram can describe a class's member variables, methods, and relations to other classes, but it provides no formal way of stating what the class or its methods do. There is no notion of pre-conditions, post-condition, state invariants, and so forth. (Such features are available in the OCL, but that is a more recent development and is not as widely used as traditional UML diagrams.)

Despite the dominance of object-oriented programming in industry, use of UML is system design is still far from ubiquitous. Estimates vary, but surveys have suggested that UML is used in anywhere from 15% [Sli04] to 34% [Zei02] of software projects. While hardly universal adoption, this at least indicates that UML has become a mainstream development tool. By contrast, formal methods are still regarded as a niche technique, with studies indicating no evidence of widespread uptake in the software industry in general.

There are a number of possible factors contributing to the relative lack of acceptance of more formal methods. Three of the more commonly mentioned explanations – tool support, method complexity, and general lack of awareness – are discussed below.

5.1.1 Tool support

There is an apparent lack of tools to support formal methods, at least compared to more conventional software development tools. There are many tools available, but according to [CGR93] and others, there is not a good range of tools available. In particular, [CGR93] cites the lack of mid-range commercial quality tools. There are a few large-scale commercial-quality environments and a quite a few small, research-quality tools, but not much in between.

While the situation has undoubtedly improved since the survey by Craigen et. al., formal methods tool sets are still not widely used in every day software development. The positioning of formal methods tools does seem to be improving, as there are an increasing number of cases where the tools are integrated into popular software development tools and environments. Examples include the Object-Z editor released by the Community Z Tools (CZT) project, which is built as a plug-in to the jEdit text editor; the Overture project tools, built on the Eclipse platform; and the symbolic execution debugger of the KeY project, which is also an Eclipse plug-in. However, this trend appears to be driven primarily by the formal methods community, rather than by the tools vendors directly. The projects appear more oriented to supporting existing users of formal methods than driving new adoption.

5.1.2 Method complexity

Formal methods are often perceived as being too complex. They use esoteric notation, mathematical reasoning, and other techniques that are not staples of traditional software development. These are perceived as a burden to developers, who may have spent years writing perfectly good software without them.

This burden seems especially heavy when applied to non-critical systems. While it is easy to justify the extra time and effort of rigorous mathematical analysis when failure can cost lives or ruin a company, the argument is more difficult when the cost is simply delay or embarrassment. In addition, the development curve of formal methods traditionally shifts more effort into the specification and design phase, which conflicts with recent trends toward "agile" methods that favor early implementation.

However, as Bowen and Hinchey point out in [BH05], the inherent complexity of many formal methods is not really that great. For example, the Z and VDM-SL notations do not require any particularly advanced mathematical ability. They require only an understanding of set theory and predicate logic, both of which are typically required learning in a undergraduate program in computer science. So clearly the mathematics required is not outside what is manageable by the average professional programmer.

Similarly, while the unfamiliar notation employed by many formal methods may be initially daunting, there is no inherent reason why this ought to be a barrier to entry. The field of software development is overflowing with esoteric languages and notations. For instance, the average web developer is currently required to be proficient in at least five languages: HTML, CSS, SQL, JavaScript, and a server-side programming language such as Java or PHP. Of these, only the final two might bear any meaningful resemblance to one another, depending on the server-side language. Another obvious example is XML, which, while relatively simple on its own, has given rise to innumerable customized schema (RSS, ATOM, XSD, WSDL, SOAP, etc.) as well as associated notations such as XSL and XPath. So while unfamiliarity of notation may be a factor, given the proliferation of notations already in widespread use in industry, it does not appear obvious that it should be the deciding factor.

Despite this, complaints about the complexity of formal methods are not entirely unfounded. While the mathematical reasoning required to use modeling languages like Z is certainly not beyond the education of an undergraduate computer science student, it is worth noting that such training is far from universal among professional software developers. There are many professional software developers who are self-taught and did not take university computing classes. According to the Bureau of Labor Statistics [BoLS09], in 2006 only 68% of Computer Programmers and 80% of Computer Software Engineers held a Bachelor's degree, and not necessarily in computer science or mathematics. So, depending on an organization's staffing, a significant percentage of the development staff may have no prior exposure to formal logic and set theory and bringing the team up to speed may require a non-trivial training investment.

5.1.3 Lack of awareness

Many software developers simply are not aware of formal methods. While formal methods classes are taught at the graduate and undergraduate levels, they are not necessarily a core requirement in most computer science curricula. Furthermore, they are not often written about outside academic circles. Popular computer programming and software development books do not typically contain any extended treatment of formal methods – or, indeed, any treatment of them at all.

In addition, what education there is about formal methods is not always sufficiently broad to represent the plethora of options available. In the United States, particularly, classes often tend to focus on a single aspect, such as code verification. In the literature, however, more focus has been placed of formal specification methods in recent years.

5.2 Reasons to use formal methods

Formal methods practitioners and proponents have advocated the adoption of formal methods for a number of reasons. These vary widely and impact software projects in different ways and at different levels.

5.2.1 Increased reliability

The most obvious reason to use formal methods when developing software is to increase reliability. In fact, this was the original promise of formal methods – to prove that systems were free of defects. And while such Utopian visions have fallen out of favor, the literature does support claims of increased reliability. Use of formalism in design and/or implementation has been linked to fewer defects, both at the code and design level. The case studies in the previous chapter, and numerous others in the literature, provide a great deal of support for this, both through anecdotal evidence and quantitative evidence from project data. While it is not generally possible to say how much of the reliability increases seen in these projects was due to formal methods, it seems safe to conclude that there is a relationship.

A number of cases have also demonstrated the usefulness of formal analysis in detecting defects through *post hoc* review of systems, including [DSRJ04] and [Pol01]. While such reviews do not typically carry the other benefits attributed to formalism, they do offer some value as an analysis tool independent of the rest of the development process. They also provide a clear demonstration that formal methods can be effective in detecting system defects.

5.2.2 Increased tractability

Increased system and code tractability is another good reason to use formal methods in development. Since formal notations are mathematical and unambiguous, unlike natural language, they allow for greater precision when specifying or describing system behavior. This allows the system specification to be more easily validated by the development team or by outside regulators.

The use of formal modeling languages like Z and VDM-SL can provide the same type of benefits as less formal techniques such as UML. They provide a more abstract language for the description and design of the system which can be simultaneously more explicit that natural language and easier to modify than code. In short, they provide a convenient language for system design and specification. This makes them useful as a communication tool whether or not their full mathematical precision is exercised.

5.2.3 Decreased cost

A number of practitioners have claimed that the use of formal methods can actually lead to an overall *reduction* in system cost. In such cases, the cost of adding formal methods to the design phase of the project is more than made up for by savings in the development and validation phases. The net effect is a shifting of the cost curve for the project, with more effort being spent in the requirements and design phase and less in the implementation and testing.

There are several examples of this in the literature. The most prominent is the CICS redevelopment by IBM, which claimed a 9% cost savings due to the use of Z. Another example is the use of the TRIO method in the development of pondage hydroelectric power plants, which [CMCP$^+$99] reports resulted in a 15% savings over the traditional approach.

The SPARK case studies from Praxis also serve as good examples of savings. In a number of case studies, Praxis has been able to attribute the use of SPARK and other formal methods to a decrease in overall project cost from projected budget. The use of formalism allowed them to find errors faster and so take a significant amount of effort out of the usually costly debugging and testing phase of the project.

5.2.4 Proof of quality

Formal notations, being mathematical in nature, are susceptible to proof. Proof provides more confidence in correctness than testing alone and therefore is helpful in providing evidence of quality. This is particularly useful for satisfying quality concerns of third parties, such as regulatory agencies.

Perhaps the most pronounced case of this is the Darlington Nuclear Generator Station. In this case, the use of formal methods to describe the software shutdown system provided greater assurance of the correctness of the system, convincing the regulatory committee to certify the station. This saved a great deal of time and money over redeveloping the system to the committee's satisfaction.

However, proof of correctness, security, or other properties can be useful in many contexts. For instance, [SW03] suggests that one such context might be e-commerce systems. By using formal methods to model the security properties of the system, e-commerce providers would be able to provide verifiable demonstrations of security. In light of the high-profile data leaks that have surfaced in recent years [DS09], such proof could be a useful business tool in addition to improving overall system quality. It might also serve a useful role in regulation, both by government and industry, by establishing more meaningful metrics for the demonstration of system security.

5.2.5 Advancement of software engineering

One novel argument for formal methods is that they are required for the development of a genuine discipline of software engineering. Formal methods can be considered the mathematics of computer programming. And since "real" engineering is based on mathematics, so must be software engineering.

Holloway makes this argument in [Hol97]. He presents it as a persuasive technique to encourage the adoption of formal methods among software engineering practitioners. It is described as a remedy to the apathy or antipathy of many developers toward formal methods, given that experiments and case studies have adequately demonstrated the efficacy of such methods.

It should be noted that increasing the quality of software engineering as a discipline does not necessarily entail the adoption of full fledged formal methods for all or even most software projects. Nor does it entail the majority of developers becoming experts on any given formalism. The discipline could benefit simply by providing building-blocks based on formal methods. The idea would be to give a formal basis for software engineering without forcing unnecessary formalism on developers, much as engineers use reference material based on hard physics, but seldom do the experiments or calculations on which those references are based.[Jac98]

5.2.6 Generation of test cases

Use of a formal model of a system can reveal possible areas for problems, but can also be exploited for other practical uses. One such example is the generation of test cases. This can, theoretically, be carried out automatically, owing to the fact that formal specifications are susceptible to machine analysis. However, a formal model or specification can also be useful to programmers or quality assurance professionals generating test cases manually, as it is the authoritative source for how the system is supposed to work. Formal specifications also make more explicit areas such as initial and final states, boundary conditions, other common starting points for test case design.

5.3 Arguments against formal methods

Despite the above, there are a number of arguments that are commonly raised against the efficacy or applicability of formal methods. Several of these are described below.

5.3.1 Formal specifications are not generally meaningful

One objection to formal specifications is that a specification should express the customer's understanding of the desired system. In other words, specifications and requirements documents are, by their very nature, an expression of the informal goals for the system to be constructed, and so must be immediately understandable in order to be useful.[CF98] This rules out formal specifications *a priori*.

This objection, by its very nature, hinges on the specification reader's understanding of the purpose of the document. It is obvious that, at the highest level, a specification, whether it be a description of the proposed system or of customer requirements, must be a natural-language document. For example, it is difficult to imagine how a business requirements document could be written using a formal notation. Such requirements define the reason for creating the system, which in all likelihood are not amenable to formal analysis.

However, as noted by Bowen and Hinchey [BH95a], adopting formal development methods does not preclude the use of other, more traditional methods. A formal specification is not the same kind of document as a natural language specification, and so one should not be used as a replacement for another. The purpose of a formal specification is to refine a natural language specification, adding a greater degree of precision which can expose subtle design problems. It serves as a compliment to a natural language description, not a replacement.

5.3.2 Inapplicability of FM to certain areas

In [EB04], the authors argue that formal design methods are simply not useful or appropriate for some problem domains, with the particular example being large Multi-Agent Systems (MAS). The argument is essentially that, in many, if not most cases, IT environments are a complex and rapidly changing mish-mash of interconnected systems, i.e. a "messy" environment. This makes it difficult, and in some cases impossible, to consider a system in isolation and perform a proper formal analysis and specification. And even if it is possible, the goal of the system may very well change several times before the specification process can even be finished.

Similar arguments may be applied when contrasting formal methods with agile development methods. The purpose of agile methods are to respond to change, whereas, at first blush, formal specification methods appear antithetical to change. The purpose of a *specification*, after all, is to describe the workings of a system. When the system requirements change rapidly and often, the value of detailed specifications, formal or otherwise, becomes questionable. When the system design is a work in progress and goals can change on a weekly basis, the perceived value of specification in general is decreased.

The same can be said for small, simple projects which are not necessarily subject to rapid change, but are, rather, fairly self-explanatory. In such cases, it might be argued that a formal model or specification is overkill. That is, an informal, or even implicit, specification may be "good enough" for such a project. If the requirements are modest and already well understood, it may not be clear that a formal treatment of the system would be worth the effort involved. The developer might have it implemented by the time the specification was finished!

In its general form, this argument is not especially controversial. Though there are many documented cases successful formal methods use in industry, most advocates admit that that formal methods are not universally appropriate. The real question is not whether they are always useful, but in what circumstances they are useful and to what extent.

5.3.3 Expensive

Perhaps the most obvious objection to formal methods, particularly from a management standpoint, is that they are expensive. Successful implementation of a formal methods project in an organization can require the purchase of supporting tools, training for engineers and designers, and time and effort to integrate formal methods into the existing development process, among other expenses. This is on top of the time spent using formal methods and the ramp-up time to become productive. Once the project gets going, there is still the concern that adding formalism into the development will slow down the overall process.

There is a great deal of truth in this objection. Bringing an organization up to speed on formal methods, particularly if they will be used extensively, is a very expensive process [BH95b]. There is a great deal of training required in order to use them successfully, and between developer time, consultant fees, and supporting software, the costs can quickly add up.

However, it should be noted that this is not fundamentally different from the ramp-up expenses associated with adopting any other new tool or technique. Whether the organization is adopting .NET or Extreme Programming, there is nearly always some expense in terms of training and support while the development team becomes comfortable with the new way of doing things. The main difference with formal methods appears to be one of degree rather than kind, i.e. formal methods are typically more alien to developers than a new programming language or development framework, and so may require more training than other methodologies and tools.

While ramp-up costs may be significant for formal methods use, it is a one-time cost. The vast majority of software development tools and methodologies place their focus on long-term cost savings. There is considerable support in the literature that formal methods provide real benefits in this area, increasing system reliability, and thereby decreasing long-term support and maintenance costs, while simultaneously maintaining or even decreasing initial development costs. So while formal methods may not be appropriate or cost-effective for one-time use on a particular project, the evidence suggests that the initial investment can pay off over many projects.

6 Applying Formal Methods in Non-Critical and Information Systems

6.1 Lessons

After several decades of use in industry, a number of lessons about the application of formal methods have emerged. While many of these were gleaned largely from experience with critical systems, they appear to be applicable to software projects in general. Some of the major lessons are outlined below.

6.1.1 Long-term investment

While it is often perceived that formal methods will increase the cost of a project, this need not be the case. A number of case studies by Praxis High Integrity System and others show that formalism can be applied to a system without increasing the cost and, in some cases, actually decreasing it.

However, it is clear from the literature that such benefits are not accrued simply by the adoption of formal methods. Praxis in particular has published a large number of case studies involving formal methods use over the course of many years and has shown that a degree of expertise in the judicious use of formal methods and their integration into the development process is necessary to derive the full benefit of their adoption. Furthermore, it is not at all clear that short term adoption, e.g. for a single project, will necessarily confer any benefit at all in terms of cost.

This suggests that the most fruitful approach to formal methods for an organization is to integrate them into the standard development process. This encourages re-use of tools, methods, and development artifacts. It also allows the initial ramp-up costs to be amortized over a larger number of projects. Formal methods are, therefore, most cost effective when they are treated as a long-term investment.

6.1.2 Expensive to start

Formal methods involves a somewhat different skill set than is used in traditional or *ad hoc* development methods. Specification is a different skill than coding and formal specification is different than natural language specification. The higher degree of precision and mathematical notation require training and experience to become accustomed to.

This can lead to a relatively large up-front investment. This may come in the form of capital outlay for training and tools, or simply in the cost of reduced productivity as the development staff comes up to speed. It is therefore advisable for organizations wishing to pursue formal methods to view it as a long-term investment. Again, this will defray the costs over several projects. This means, among other things, fostering in-house expertise in the organization's methods of choice. This will reduce reliance on outside contractors or researchers while providing a knowledge base through which new team members can more easily come up to speed.

In the wider community of academics and formal methods practitioners, this cost can be eased by continued improvements in tools and educational materials. Improvements in the quality and quantity of free tools, as well as less academic, more example-style documentation, would will serve to lower the barrier to entry for individual developers as well as smaller organizations.

6.1.3 Need resident experts

As Bowen and Hinchey pointed out in [BH95b], most projects that have used formal methods successfully have had local expertise in their use. In some cases, this meant working with researchers, such as the CICS redevelopment, or building on in-house expertise from previous projects, as with Praxis.

Bootstrapping formal methods experience from nothing appears to be relatively difficult. Again, this is probably due to the differences between developing with formal methods and developing without them. While new programming languages and development frameworks typically build on other skills a developer might have, this is less true of formal methods. Formal methods typically use unfamiliar notations and skills such as set-theoretic logic which, while common in academic computer science programs, are not as widely used in traditional software engineering.

Opportunities for mitigating the costs of formal methods start-up presents a challenge to the formal methods community. High quality tutorial and self-educational materials could help development organizations to build formal methods knowledge on their own. This is important because organized training is expensive, and also because it may be difficult or impossible to acquire in some markets. Self-training material would also allow independent developers to more easily become acquainted with formal methods on their own, as part of self-led professional development.

6.1.4 Tools helpful but not necessary

In contrast to what one might expect, success with formal methods is not necessarily tied to the use of advanced tools. As reported in [CGR93], many of the case studies used few if any tools, but still managed to achieve some degree of success.

That said, there can be little doubt that some tool support is important in the use of formal methods. For example, tools for preparing editable, electronic artifacts are very important. However, while they can undoubtedly be useful, tools for proof and other mathematical activities are not a necessity. This is clearly the case for efforts that are not focused on proof, such as light-weight formal specification for the purpose of elucidation rather than demonstrable correctness.

One argument for the necessity of tool support is that it will relieve the mathematical burden that formal methods places on non-experts. Since not everyone is a mathematician or an expert in the formal method in question, should not the purpose of tools be to fill in these deficiencies?

Of course, a tool cannot truly take the place of understanding a method, so a certain amount of training will always be required. However, depending on the method in question and how it is applied, the mathematics is not necessarily beyond the reach of the programmers building a system – in fact, if no formal proof is attempted, then there is practically no math at all. It could also be argued that the real benefit of proof tools is not to the non-expert, but to the experts and practitioners themselves. This is exemplified by Praxis' SPARK programming environment, which provides tools to automatically discharge proof conditions of a program. These allow a machine to do the "grunt work" of cranking out the "easy" proofs, freeing up the engineers to work on the "hard" problems.

6.1.5 Right method for the right job

Just as with programming languages, there is no "one size fits all" formal method. There are a number of different methods, and variations on these methods, that specialize on different areas. For example, there is Z for general modeling, Object Z for object-oriented systems, TCOZ for real-time systems, and so forth, each with it's own particular focus and approach.

To a certain extent, this is understandable and inevitable. The same proliferation occurs in programming languages, with many different types of languages, variations on languages, and hybridizations of different types of languages evolving to fit the varied needs of projects and preferences of teams. Indeed, as [Jac98] points out, this can actually be considered a strength. A method that is universal in applicability cannot, by definition, take full advantage of the details of a particular problem domain.

However, this also presents a problem to wide-spread adoption of formalism. If no single notation or method is suitable to all problems, how are organizations to choose methods in which to invest? As previously noted, formal methods can have a relatively high start-up cost, so it may not be feasible to invest in several methods to start. It is also more difficult to build in-house expertise in several methods.

As with programming environments, in such cases it is probably safest to choose the more established option. Thus a strategy of starting with general purpose modeling languages such as Z and using experience built with those to branch out into specialized methods is probably best. The more established notations will be likely to have more support in the form of tools, books and tutorials, and community participation.

6.2 Formal Methods in Information Systems

Different problem domains require different types of analysis. This section will discuss some of the particular questions and issues that arise when applying formal methods to information systems.

6.2.1 Role of formalism: specification vs. design

The first question to arise in applying formal methods to information systems is *how* it should be done. For non-critical systems, full code-level verification is not likely to be required. This leaves selected verification of subsystems, formal design, and formal specification as the primary modes of application. Of these, formal design and formal specification are the most universally applicable.

The precise role of formal methods in a project will depend, in part, on the existing development process used by the organization in question. For traditional waterfall-style, it is easy to see how formal modeling could be applied at the specification level and then refined downward to the design level. For a team using an agile development model with minimal up-front specification, this might not be appropriate.

The "specification" approach involves translating the system requirements into a model in a formal notation. The model will most likely define the high-level entities used in the system, their properties, and interactions. The model will not attempt to describe any technical aspects of implementation. Rather, the goal is to capture the key features the system should have. This allows for more explicit documentation of the system requirements as well as systematic analysis of the consistency and completeness of the requirements.

In the "design" approach, formal methods are used to describe the planned or actual implementation of the system. The constructed models would include information on the objects to be included in the system, their properties and operations, and some details of how they work. Unlike specification, the intent here is that there be at least a loose descriptive relationship between the implementation and the formal model.

It is possible to use either or both of these approaches in a project, depending on the nature of the process in use. A development effort can start with a formal specification of a system and refine this down into a design. Alternatively, the formalization effort could cease after the specification and use an informal design process. It is also conceivable to simply skip the formal specification and do the system design in a formal notation, though this is generally not the approach recommended by practitioners.

6.2.2 Business rules

One of the major pitfalls of developing information systems is the proliferation of business rules. The actions and analysis performed by a system often have to contend with a large number of requirements and restrictions imposed by business needs, organizational guidelines, or laws governing the business.

Such complexities offer an excellent opportunity to apply formal analysis, as formal modeling languages are typically well suited to documenting constraints and relationships between parts of a system. Also, tools to support model checking can be used to establish the correctness and consistency of the business rules prior to implementation.

The main challenge to applying formalism to business rules involves requirements drift. Whether it is due to changing business needs, new legislation, or some other reason, it is common for these rules to change over time. Changes to requirements necessitates changes to the formal models, which contrasts with the common prejudice associated with the "waterfall" method of development, that specification and design are up-front activities which, when finished, are *over*. With formal designs and specifications, this is certainly not the right approach.

The proper use of formal modeling methods, according to most experts, requires that specification documents become more "live" than they are traditionally considered to be. That is, rather than being completed prior to the start of development and serving as a guideline, a formal specifications must be a central part of the development process, undergoing revisions and "patching" just as the code does. After all, a formal specification that is not updated with the code is of no more use than a similarly out of date English specification.

This is an area where tool support could be particularly useful. Amey [Ame02] describes the use of the SPARK examiner with a semi-formal CORE requirements document. The tool helped to keep the document "live" by haranguing programmers into clarifying vague requirements. Of course, this particular method suffers from the drawback that it relies on tools designed for full code-level correctness checking, which is not always necessary. However, one could most likely devise a convention-based method for automatically tying portions of code to portions of specification, allowing for more automated checking of consistency with code. For instance, if file naming conventions were used to tie code to particular formal models, it would be possible to add hooks into a version control system to check that the specification is updated when the code is updated.

6.2.3 User interface specification

Most case studies do not bother to formalize the software user interface (UI). Indeed, it can be argued that, with proper design and architecture, there is little need to do so. If business logic is properly separated from the user interface layer, then little of real importance happens in the UI layer. One might wish to verify that the actions initiated by various interactions with the UI are correct, but this does not require the rigor of formal methods.

Furthermore, the main quality concerns in user interfaces have to do with design and user experience, not correctness. For example, a good UI must be clear and concise, and should consider the expectations of the user and operate in a way that is consistent with them. Such things are not readily addressed by formal specification methods, as they are relative to the target user and tend to be subjective.

However, user interface details can play an important role in the requirements and specification phase of a project. For non-technical project managers and stake holders, the user interface is the the most visible part of the system and the portion to which they can most easily relate. It is, therefore, not uncommon for them to specify system requirements in terms of screens or controls in the UI. This suggests that some projects could benefit from light-weight UI formalization.

One example of such would be the explicit mapping to the system of UI-based requirements. This could be especially useful when the project requirements are low quality or are incomplete. For instance, an informal requirements document might contain screen mock-ups and natural language requirements which are not consistent with each other, or the specifications for a second version of a product might contain requirements that assume some functionality of the system which differs from reality.

Some level of formalization could also be helpful in capturing the application control flow. This is especially important in web applications, where the application flow is not always obvious due to the stateless model of HTTP. At the most basic level, this can be captured with relatively little effort using, for example, simple indicators in the postcondition.

A certain degree of formalization could also be useful in the specification of complex input forms. Formal modeling languages lend themselves to specifying input constraints, and can easily capture such common input control constraints as size limit, data type, selection from a specific set of choices, default values, etc. However, the usefulness of such formalization, especially if done on a large scale, may not out-weight the effort. In many situations, it is likely that, rather than adding new information, modeling UI constraints would simply be repeating other, previously specified system constraints. Such a technique would be most useful in cases where the system requires large, complicated data input forms which do not map directly to system objects.

6.2.4 Databases

Relational databases present one of the most obvious opportunities for formalization. They are extremely common in business applications and typically hold much of an organization's most important information. Furthermore, the integrity of the data they contain is of critical importance and can be enforced by a complex set of constraints which lend themselves to mathematical modeling.

As has been shown by Barros [dB94b] and Simpson [SM03], relational databases are well suited to being modeled via formal methods. For example, the Z language uses notation that maps quite directly to the concepts of relational theory. The Z schema is roughly equivalent to a relational table, with the variables in the declaration portion being analogous to columns and the predicate portion of the schema mapping to relational constraints.

Although the mapping is not quite as direct, the same principles can be applied to other notations that use set theory and support record types and constraints, such as VDM-SL. The set theoretic concepts used in such languages still correspond reasonably well with that used by standard SQL, so the same information can be expressed. A more detailed example using VDM++ is presented in the next chapter.

Relational schema to Z schema

Continuing with the Z example, the names and types of Relational Database (RDB) columns can map almost directly to the variables of Z schema. The main difference is in the predispositions of how each system handles data types. Most relational databases deal with concrete primitive types, such as integers, strings, and dates. Z, on the other hand, is more focused on handling abstract data types.

This does not represent a problem, so much as a difference in focus. While Z may have a bias towards abstract types, it need not be restricted to them. One could simply define Z types that correspond to the constraints on primitive types supported by a particular Relational Database Management System (RDBMS). One might also choose to define more abstract types to correspond to the complex types available in some RDBMS systems, such as coordinate types for Geographic Information Systems (GIS), Binary Large Object (BLOB) types, and so forth. It is simply a matter of choosing the appropriate level of detail in the specification as part of the refinement process.

Relational constraints to Z predicates

Translating a relational database schema without constraints into Z is nearly trivial. Adding constraints is not much harder, as the constraint notation for Z can be used to capture the same information. In fact, being more general, formal modeling languages are able to capture *more* information, including constraints that are difficult to express in SQL or which, for whatever reason, need to be enforced at the application level.

The main difficulty in moving to a formal modeling language is the more complex notation. For many common types of constraints, the formal logic is much more verbose than the standard SQL notation. For example, compare a simple `PRIMARY KEY` declaration in SQL to the equivalent Z constraint:

$$\exists\, k : Country \rightarrowtail CountryDetails \bullet (\forall\, c : Country \bullet (k\ \ c).name = c)$$

Unsurprisingly, the situation only becomes worse with cascading foreign key constraints or complex `CHECK` constraints. Mapping formal models to the database at this level of detail could easily become cumbersome for complex data models. Whether it is a useful exercise will most likely depend on the system in question.

There is an "impedance mismatch" when mixing relational and imperative programming in business systems. That is, the concepts and techniques used to model the object-oriented application code and the relation database schema for a system are different and mixing the two can present difficulties.

Use of a formal notation such as Z or VDM-SL with relational databases allows a measure of consistency at the specification level, as the same language can be used when describing relational and other parts of the system. During the refinement process, it may also ease the mapping of one type of model to the other.

6.3 Goals

This section discusses some of the proposed goals toward which formal methods advocates should work. These items should make formal methods somewhat more palatable to mainstream software developers and thus increase their adoption. This, in turn, would lead to a more formal and scientific practice of software development.

6.3.1 Black-box formal methods

Probably the most effective way to increase uptake of formal methods would be to remove the complexity involved in applying them. Ideally, this would mean allowing software practitioners to use formal methods without having to know anything about formal methods. At they very least, it means allowing them to use formal methods productively without having engage an on-site expert to consult.

Tool support can go a long way towards this goal. Of course, the extent to which it is feasible depends in large part on the nature of the formal methods in question. It may be feasible to completely hide the formal basis of some specialized tools from users. However, for general purpose formal modeling languages, it seems unlikely that the need for an understanding of the mathematics can be completely eliminated. However, even relatively simple tools such as parsers and type checkers are useful in taking some of the analytical burden off of developers. Such checks are especially helpful because they require no effort on the part of the user.

Integration with existing development tools provides an opportunity in this area. It reduces the barriers to adoption and allows for greater integration into the development process. The Overture project shows great promise in this regard. It is already well integrated into Eclipse and allows users to read and write VDM specifications just as they would code files for a typical programming language. Another very useful integration is with UML-based tools, as UML converters allow a relatively transparent conversion from the common graphical modeling language to a formal modeling language. This also presents opportunities for automatic interconversion and updates between models.

From a non-tool perspective, methods, patterns, and guidelines for formal specification development are areas in need of improvement. Modern software development takes place in a wide variety of environments and the design techniques and patterns which are well suited to one do not always map directly to others. This applies not only to the move from non-formal to formal methods, but from one problem domain to another. Developers new to formal methods in general or to their application to a particular problem space would benefit from examples that are directly applicable to their needs.

6.3.2 Design patterns for FM

In software development, “design patterns” are specific design idioms that recur in a variety of projects. They are established methods of accomplishing some particular design goal and are generally independent of a particular implementation language or problem domain. They are normally described in a standard form and using a standard notation (UML).

Formal methods could benefit from some similar concept. While formal specifications are necessarily system- and domain-specific, it is still possible to write template specifications for specific common parts of systems to serve as a guide to developers.

There are several ways for practitioners to address this. The simplest and most direct method is simply to apply formal methods to the description of existing design patterns. This can, to some extent, be accomplished mechanically using tools to map UML to a formal modeling language such as VDM-SL or Z. However, this only addresses the application of formal modeling at a low level and does not demonstrate approaches to modeling.

Another potential application is the formal design of reusable system components, such as authentication and authorization modules, rule-based data validation frameworks, and so forth. This approach aims to build reusable design specifications which can be implemented in any number of programming languages and technologies. Such a project might look almost like a design for an application development framework

A third possibility involves the development of patterns for application of formal methods. This scenario addresses the use of formal methods as a mid-level design tool. It would involve the publication of template models or examples of the application of formal methods. In this way, it would be similar to the way patterns are presented in the literature on object-oriented design, as small, targeted approaches to solving particular common problems. Likewise, the formal methods community could supply small, easily used examples of how to gainfully apply formalism in common situations.

6.3.3 Standard notation or general method

In order for formal methods to enjoy widespread use, it would be helpful to adopt a certain degree of notational consistency. This has two benefits. First, it enables the promulgation of the techniques described in the previous section. Using a standard notation would allow researchers and practitioners to focus their efforts and reach a larger audience. Second, it would provide a clearer and more unified front to mainstream developers. Presenting a smaller set of viable methods provides a clearer path to the use of formal methods and a smaller and more well defined area of learning.

One of the lessons learned from industrial case studies is to use the right method for the right job. In other words, there is no single “correct” method or notation - each method has it’s own distinct area of specialization and applicability. While this is an important lesson, it is still compatible with a common notation. While there are many situations where specialized methods are most appropriate, there are still fairly general purpose methods that can be widely applied. These can serve as a common base or starting point, much the same as general purpose programming languages such as Java and C++ are suitable, if not optimal, for most common tasks.

The question of which methods are most appropriate for general use is open to debate, and must, of course, ultimately be decided by community consensus. However, the most obvious choices are the VDM and Z families of modeling language. Both of these notations are over 20 years old, have ISO standards, and have relatively large user communities and bodies of published books and articles on their use and application. Furthermore, both notations are relatively general purpose and can be readily applied to "every day" business systems and other common problem domains. Both also have a number of variations and offshoots that can be applied in more specialized cases. Thus they are already positioned for wide-spread use in a way that more niche or specialized notations are not.

7 Modeling a Web Application Using VDM++ and PHP

This chapter presents an example of formal development methods applied to a portion of a “typical” web-based application. This includes modeling the high-level design of the application as well as the low-level design and translation into executable PHP code.

The purpose of this effort is to demonstrate the application of formal methods to the type of non-critical system which is often developed by small organizations. The goals of this project are threefold:

1. To provide an example of how small development teams can apply a formal modeling language to the development of a web-based system.

2. To demonstrate the application of formalism to a modern dynamic language.

3. To develop a reusable framework of formal modeling techniques for use in future web applications.

7.1 Choice of technology and notation

The target platform for this web application is the LAMP stack. LAMP is an acronym for Linux, Apache, MySQL, and PHP (though other "P" languages, such as Python or Perl, are sometimes substituted), which are the target operating system, web server, database, and programming language, respectively. The LAMP technology stack is one of the most popular web development platforms in current use. All the major components are available at no cost under open-source or free software licenses and can be easily used with a variety of free development and administration tools. This makes the technology stack relatively inexpensive to deploy and so has made it very popular among smaller groups operating on a limited budget – hobbyists, not-for-profit groups (e.g. free software projects), and start-up companies.

Such groups are prime candidates for evangelism of formal methods. They typically do not have large budgets for tools or training and do not have ready access to academics and experienced practitioners. They may also be less likely to have formal education requirements, as they focus more on work experience and the demonstrated ability to do the job at hand. They are thus less likely to have prior exposure to formal methods.

This thesis will not present any explicit treatment of the operating system and web server, as the application will be largely independent of them – they will merely serve as an execution platform. It will also not deal in any depth with the particular database management system, as the application will target a generic superset of ANSI SQL-92 rather than a particular vendor's SQL implementation. Instead, the focus will be on the implementation language – PHP.

7.1.1 Formal notation

The formal notation used for this project is VDM++. It is an object-oriented formal modeling language based on the VDM, which began development in the 1990's. VDM was originally developed in the 1970's by IBM's Vienna lab and accepted as an ISO standard in 1996. [FLM$^+$05] It is therefore one of the older, more established formal methods in current use. It is also a relatively general-purpose method, suitable for the type of procedural/object-oriented web applications that are typically developed in PHP. The choice of VDM++ over straight VDM-SL is a consequence of the heavy use of object-oriented programming in current software development, which is more easily accommodated in a notation with built-in support.

Other factors which informed this decision were the syntax of VDM++ and the methods for data refinement. Standard VDM-SL, and by extension VDM++, provides an ASCII text-based syntax, reminiscent of a programming language, which is easily typed into any text editor. The main competing method considered for use, Object-Z, uses the same box schema notation as the Z language upon which it is based. Such notation requires special tools, and so Object-Z is usually written either with a special editor or using LaTeX markup. The VDM++ syntax is therefore somewhat easier to deal with and should be more familiar to programmers, thus easing adoption.

Furthermore, VDM has guidelines for the refinement or *reification* (to use Jones's term [Jon90]) of data. This is the method by which the "pure" data representation used in a specification are translated into less abstract equivalents that can be more easily implemented in a conventional programming language, most of which do not have native support for abstract types such as sets. VDM includes a standard method of reification that allows the specifier to demonstrate that one representation is an acceptable refinement of another. While such arguments can be constructed in the Z notation, there is not a standard method for doing so. While this thesis will not make extensive use of VDM's refinement method, such a method clearly has the potential to be useful to specifiers, especially when dealing with the complex and sometimes ill-defined data types that arise in languages such as PHP.

7.1.2 Tooling

While pencil and paper design skills are still very important, modern software developers rely heavily on software tools. For this project, two sets of tools were used. The formal design and specification were written primarily using the Overture tool set. The VDM++ Toolbox was also used for the initial modeling and for some particular tasks, such as pretty printing.

The VDM++ Toolbox was used initially because it is a considerably older and more established tool that Overture. It has a full feature set, providing parsing, type checking, integrity checking, pretty printing, code generation, an interpreter, and communication with the Rational Rose UML modeler. The vendor also provides extensive documentation. However, the toolbox does not provide an editing environment – users are required to configure an external editor. For this project, that external tool was the GVim editor with a VDM++ syntax highlighting plug-in installed. Another drawback of the VDM++ toolbox is that it is free only for non-commercial use. While this may not affect it's desirability for use on open-source project, it could certainly give pause to small commercial development teams.

By way of contrast, the Overture Project was formed with the explicit goal of building free and open-source tools. Furthermore, the tool's design is quite modern – it is built as a series of plug-ins to Eclipse, a popular and widely used open-source integrated development environment (IDE). This provides the user with a rich, interactive editing experience. It also enables immediate feedback, as parsing and type checking are performed in the background while the user works and presented on the screen with visual cues. This also allows for convenient integration with conventional development tools, as there are PHP development profiles available for use with Eclipse as well.

7.1.3 Overview of the PHP language

PHP is a popular scripting language used primarily for the development of web applications. “PHP” originally stood for “Personal Home Page” and is now a recursive abbreviation for “PHP Hypertext Preprocessor”. As these names suggest, over five major versions PHP has undergone a great deal of evolution. It started out as a simple template system and HTML preprocessor, later developed into a full-fledged procedural programming language, and now supports most common object-oriented programming features. Today, PHP code is used in a wide range of large and sophisticated applications, including some of the largest sites on the Internet, such as Facebook and Digg.

As a language, PHP is somewhat disorganized. There is no formal specification – the language is simply defined by the “official” implementation. It is a mixed paradigm language, somewhat similar to C++ in that it has support for object-oriented programming, but also supports completely procedural development. PHP can even function as its own template engine, mingling data with sequential code. Source files are dual mode, with PHP code being contained in special angle-bracket delimited blocks, such as this `<?php echo 'Hello world!'; ?>`, and all characters outside such blocks being treated as data to write directly to the program's output stream.

The data type system in PHP is loose, implicit, and dynamic. Variables are created at run-time and need not be declared in advance – this includes the public instance variables of classes. The type of a variable is determined by the value assigned to it, with types not being fixed, i.e. a string can be assigned to a variable containing an integer. While there is support for explicit type casting, most simple types can be cast to each other implicitly. This includes implicit casting of strings to integer and floating point types, strings to numeric types, and casting of any type to boolean.

In addition to dynamic data typing, PHP has many other highly dynamic features. One of the more unusual is so-called “variable variables”. This is a feature whereby PHP allows variables to be referenced based on the contents of other variables. This allows for code such as the following:

```
$a = "test";
$b = "a";
print $$b; # prints ‘‘test’’
```

Here the use of the double dollar sign indicates that the named variable contains the name of the target variable, somewhat analogous to the use of pointers to pointers in C, except that variable variables operate based on symbol-table look-ups for variable names. The same technique can be used to call functions, using a variable that simply contains the function name.

This concept extends to object-oriented programming as well. Object fields and methods can both be referenced through variables containing their corresponding name. In addition, objects can be created based on variables containing a class name, so that rather than explicitly naming the class to instantiate, objects can be created based simply on the contents of string variables. For example, assuming that a “User” class is defined, an instance of it can be created by applying the `new` keyword to a string variable containing the value "User". The following illustrates these capabilities.

```
//The traditional version
$user_obj = new User();
$user_obj->last_name = "Smith";
$user_obj->save();

// The equivalent using variable variables for fields ,
// methods, and class instantiation.
$class_name = "User";
$field_name = "last_name";
$method_name = "save";

$user_obj = new $class_name();
$user_obj->$field_name = "Smith";
$user_obj->$method_name();
```

One other interesting aspect of object-oriented PHP is the "magic methods". These are methods which, if defined by a class, will be called automatically by the runtime in certain circumstances. Three of the more common and noteworthy magic methods are __get(), __set(), and __call(). As the names imply, __set() and __get() are called when there is an attempt to assign or retrieve an undefined field on an object. These are sometimes used to implement encapsulated properties on classes, mimicking the field-like syntax of C# rather using the getter and setter methods common in Java. Similarly, the __call () magic method is invoked when an undefined method is called on an object. This is sometimes used for dynamic, convention-based APIs, such as querying a database table by multiple fields without explicitly writing separate methods for each, e.g. to call a common query method using the names getById(), getByName(), etc. rather than supply parameters.

Challenges to formalism

Features such as those described above can enable developers to write powerful programs using very little code. However, if used carelessly, they can also be sources of confusion and subtle errors. Not only do these features make it difficult to assess the meaning of code prior to run-time, they can be used to add implicit behaviors to the system or to override typical behavior.

Another complicating factor is the nature of PHP's development. PHP is as much a product as a language. It does not have a formal specification of the type that would be submitted to a standards body, such as exists for languages like C++ and Ada. Rather, the language is defined informally through natural language documentation (the language reference manual) and operationally by reference to the behavior of the "official" PHP interpreter. The language is, therefore, subject to change with new releases of the interpreter. Furthermore, PHP has a significant amount of run-time configuration. In some cases, the behavior of certain key aspects of the language can change significantly based on the current configuration. The chief examples of this are features such as “safe mode” and “magic quotes”, which alter the way in which certain functions and standard variables work. Thus the correctness of a PHP program can vary greatly with the environment in which it is executed.

These factors can make it difficult to directly model some behaviors of PHP applications using standard modeling notations. This leaves developers in the position of either choosing not to use such features or constructing their formal models in a more static manner – expressing what is meant to be accomplished by a module rather than what it actually does.

7.2 Development Approach

The development of this application uses a light-weight approach to formal modeling. It will follow a fairly typical development process model, representative of real-world processes used by small development teams.

The starting point for the system design will be a (fictitious) list of informal system requirements. This will be followed by an elaboration and clarification of the requirements, from which the system design will be derived. The system design will be refined and elaborated, and then converted into the final code.

Formal models will be constructed at both low and high levels of detail. Analysis will be performed on the models solely for purposes of elucidating the system design. No attempt at formal proof of correctness or consistency will be attempted.

This project will attempt to keep the formal models "live" documents. As modifications are made to the system design during later phases, the models will be updated to reflect this. This will be particularly the case for the most detailed model model, which most closely describes the system as implemented.

7.3 System requirements

The sample application to be implemented is a simple document management system. It needs to support multiple users, allow users to create, retrieve, update, and delete documents, and manage security constraints on such operations. As this is a sample application, it will focus on the underlying aspects of the system, i.e. the underlying framework and authentication and authorization subsystems, rather than additional high-level functionality.

The following are the requirements for the system. The initial requirements statement is highly informal and broken down into a simple list of needed features. Such lists are not uncommon artifacts of the initial requirements gathering for a system and nearly always need some level of clarification.

I1 The system must support multiple users.

I2 The system must allow users to read, write, edit, and delete documents.

I3 It should be possible for documents to be made secure, so that only selected users can view them.

I4 Only the person who created a document should be able to edit or delete it.

I5 The owner of a document may grant other users permission to edit, delete, or read it.

I6 System administrators should be able to change the permissions and ownership of a document.

This list may be made more explicit by separating the above requirements into their respective components. This is done to more clearly illustrate the translation from the informal requirements to the formal model.

R1 The system must have user accounts. (Implied by I1 above)

R2 The system must store documents. (Implied by I2 above)

R3 The system must support multiple concurrent users. (From I1 above)

R4 The system must allow for documents to be created, read, edited, or deleted. (From I2 above)

R5 The system must restrict read/edit/delete operations on documents to only specified users. (Generalization of I3 – I6 above).

R6 Documents must have an "owner". (Implied by I4 – I6 above)

R7 Any user may create new documents. (Implied by I3 – I6 above)

R8 When a document is created, its owner is set to the user who created it. (Implied by I4 above)

R9 The system must maintain a list of "permissions", i.e. users who are allowed to read/edit/delete each document. (Implied by I5 and I6 above)

R10 The system should maintain a list of users who are administrators. (Implied by I6 above)

R11 The system should allow the owner of each document to modify permissions on that document. (From I5 above)

R12 The system should allow users who are administrators to modify ownership and permissions on any document. (From I6 above)

7.3.1 Requirements specification

Having collected a satisfactory, if somewhat basic, list of requirements, the first step in system development is to translate the requirements into a high-level formal specification. This begins by simply identifying the entities involved in the system, the constraints upon them, and their relations to each other.

Required entities

Initially, it is clear from requirements R1 and R2 that the system must have at least two types of objects: documents and users, which will be represented by the VDM++ `Document` and `User` classes, respectively. The exact shape of these objects is not made explicit in the requirements. However, it is safe to assume that users will have at least the username and password members that are typical in multi-user environments. Likewise, it can be assumed that documents will have at least some kind of content, which, for the sake of simplicity, will be a simple character string. From requirement R7, it can also be concluded that a document will be associated with a user object representing its owner.

From requirement R4, there are at least four operations on document objects - creation, editing, deletion, and reading. For user objects, requirement R3 and the security requirements imply that there must be login and logout operations, as access control makes little sense without some form of authentication.

The next step is to consider the security constraint requirements. Because these requirements do not pertain entirely to either documents or users, they can be modeled these using a separate `Security` class. The members of this class will include the administrator list and permissions list from requirements R9 and R10. The administrator list can be modeled as a simple set of `User` objects. For permissions, more precision is required.

Access control objects

At this point, the rules regarding how permissions function must be clarified. From requirement R9, there are four components to the elements of the permission list: users, documents, the type of access (read/edit/delete), and the authorization status (granted/denied). Logically, these can be broken into two parts of a mapping. That is, the user and document will determine an authorization mapping from access type to authorization status.

In VDM++, this can be modeled by a somewhat complex mapping. First, define the union type `AccessType = <read> | <edit> | <delete>`, which defines the domain of the authorization mapping. The authorization state can be modeled by a simple boolean. The set of possible permissions can then be defined by the mapping: `Permission = map (User * Document) to (map AccessType to bool)` The list of permissions stored by the system can then be expressed simply as a variable of this type.

While this definition of `Permission` appears to capture the intent of requirement R9, it immediately raises implementation issues. If permissions are to be stored as a mapping of the Cartesian product of users and documents to an authorization mapping, the most obvious implementation is that the system must store a record for all possible combinations of users and documents. The advantage of this approach is that checking of permissions can be trivially modeled as a map application. Thus a lookup of read permissions for a document d1 and user u1 can be written as `permissions(mk_(u1, d1))(<read>)`.

However, there are a number of problems with this approach. First, it raises the issue of storage size. The storage size equation would be `card Permission = card User * card Document`, which seems excessive. However, it also raises issues of performance and management. Whenever a new user or document is added to the system, a number of permission records equal to the cardinality of the other set must be added to the permission store. In addition, it is inconvenient for users and administrators who want to specify document access to individually consider each and every user in the system. For instance, in a 500-person organization, this design does not allow a quick and easy way to specify that read access to a document is denied to everyone *except* a particular 20-person department. Such scenarios are not uncommon in business, and so it is important to take this into account.

Another possible design is to work from a "default" set of permissions and only store explicit changes. So, for example, we might specify that read, edit, and delete permissions are granted for the document owner and administrators and that for all other users, read permission is granted and edit and delete permission denied. This implementation can use the same `Permission` type for the permission list as the previous version. However, permission checking cannot be done with a simple map application. Rather, an operation must be added to account for the default permissions. This could be implicitly specified in VDM++ as:

```
PermissionCheck(u: User, d: Document, t: AccessType) r:
    bool
```

```
ext rd permissions, administrators
post
    if mk_(u, d) in set dom permissions then
        r = (permissions(mk_(u, d))(t))
    else
        r = (u = d.owner or u in set administrators or
             t = <read>);
```

This design would radically reduce the storage requirements over the previous version. In the worst-case scenario, the storage requirements would be the same, while in the best-case, where no permissions were ever changed from the default, no storage at all would be required. Furthermore, adding new documents and users would not require any additions to the permission store at all, improving performance.

The management issue, however, remains. Continuing the previous example of granting read access on a document only to a certain department within an organization, it would still be necessary to store a record for each user outside that department. Changing the default permission set would not solve this problem, but simply shift it to be more or less favorable to different usage scenarios.

Adding groups

This leads us to reconsider the original requirements. It is clear that it would be desirable for users to be able to specify access permissions based on membership in one or more groups. We will therefore add requirements to account for this:

R13 The system should allow administrators to create groups of users.

R14 When setting permissions, users should be able to grant or deny access too all members of a group.

Groups are easily modeled as a set of users, `Group = set of User` in VDM++. However, for ease of management, groups will have to contain some metadata, minimally a group name, and are therefore better modeled as a class containing a set of users.

With the addition of groups, it is now necessary to modify the way permissions are modeled. Since requirement R9 is still in effect, it is necessary to support permissions for both users and groups. The most straight-forward way to handle this is to model permissions as a single mapping, from access objects, i.e. users and groups, to authorization mappings. This could be modeled with a union type:

```
AccessObject = User | Group;
Permission = map (AccessObject * Document) to (map
    AccessType to bool);
```

To check permissions with this model will require defining an operation as above. One way to do this is to attempt to adapt the above user-only `PermissionCheck` operation.

```
PermissionCheck(u: User, d: Document, t: AccessType) r:
    bool
  ext rd permissions, groups
  post
    r =
      if mk_(u, d) in set dom permissions then
        permissions(mk_(u, d))(t)
      else
          exists g in set groups &
             mk_(g, d) in set dom permissions and
             u in set g.members and
             permissions(mk_(g, d))(t)
```

Note that the postcondition for this operation has some interesting implications for the application security policy. First, like the last version, it specifies the order of precedence for permissions. That is, individual user permissions take precedence over group permissions. Second, the "else" portion of the outer conditional specifies a "default deny" security policy. Specifically, the predicate specifies that the requested access to the document is only granted if the user is a member of a group with the specified access. If the user is not a member of a group with access, or if there are no permissions specified for the any of the user's groups, the predicate will be false and access will be denied.

7.3.2 System model

To model the application of security checks, it is necessary to represent the system state, in particular the currently logged in user. This is accomplished by adding a `System` class with members for an instance of the security class and sets of users and documents, as well as a member for the current user. This class will also contain the operations that represent the various tasks performed through the user interface.

Beginning with requirement R13, the functionality is filled out by adding the `CreateGroup`, `AddUsersToGroup`, and `DeleteUsersFromGroup` operations. These use preconditions to enforce the requirement that these actions can only be performed by administrators. The same precondition suffices for the `ChangeOwner` operation to satisfy the ownership change portion of requirement R12. For the `SetPermission` operation, adding a comparison of the document owner and current user to this satisfies the security checks for requirements R11 and R12.

To round out the functionality of the application, the system class also requires operations for users to log in and out, as well as operations to access documents – `ListDocuments`, `CreateDocument`, `EditDocument`, and `ReadDocument`. These operations correspond roughly to the different pages of the web application. Since access to some of these pages is controlled, it is necessary to model the results of the page access. At the level of detail used in this specification, it is sufficient to model binary success/failure or granted/denied as simple boolean results. Thus the `Login` operation returns `true` on a successful login and false on failure, while the `ReadDocument` and `EditDocument` operations return true if the user is able to read/edit the document, and false otherwise. For the `ListDocuments` operation, the return value is a set of documents reflecting the documents in the system to which the user has access.

At this point, the high-level specification covers the bulk of the system requirements. The next phase in the development process is refinement and detailing of the model. For this project, this will consist of two levels of refinement. The high-level model will first be translated into a component-level model, which will then be converted into a detailed design.

7.4 Component-level specification

From a review of the system requirements specification and common design patterns, it is relatively straight-forward to decompose the system into its various components. This will serve as the starting point for a more detailed system specification.

The high-level model begins by declaring a top-level `System` class to model the overall system state. This serves as the entry point to the application and the repository of any global state.

The next step is the separation of entities. The initial requirements specification deals only with the entities present in the system – documents, users, etc. In designing a system, it is useful to separate these domain objects from display logic and other orthogonal concerns. To this end, the system entities are modularized, creating controller classes and domain object classes for each.

In this design, the controller classes encapsulate the user-facing web pages which make up the application. Each class represents a logical portion of the system with each class method representing an individual web page. This provides a straight-forward convention for organizing the system and meshes well with the Model-View-Controller implementations commonly found in PHP-based application development frameworks.

The domain objects corresponding to each controller represent the system data, storing document content, user information, etc. At this level of detail, these classes simply define the key data contained in each object, with any needed methods being left for later. The notable exception is the introduction of the `ActiveRecord` class, from which the domain objects inherit. This class specifies the design pattern used as the object-relational mapping and persistence mechanism for the domain objects.

The final portion of the Model-View-Controller pattern mentioned above is a simple `View` class. This class simply represents the HTML and other client-side display code emitted by the application. It is largely a place-holder at this level of detail.

The final component in the system is the persistence layer – the database. At a high level, the database can be viewed as simply a set of tables – there is little immediate need to model connections or complex constraints. For the sake of speed and simplicity, rather than specify the database schema up front, the component-level specification simply defines the database tables as sets of domain objects. This serves to document the entities in need of storage. It also implies a fairly simple data storage structure, wherein domain objects map more or less directly to tables.

7.5 System Design

Having developed a list of requirements and formalized them at a fairly high level, the next task is to proceed to the system design. Because this is a relatively small and simple project, the design model is written at a moderate to low level of abstraction. That is, it describes the architecture and design of the system, but does not go to great lengths to be translatable directly into executable code. The purpose of this design is to create a model of the system behavior as a reference for implementation and to aid in analysis of the system.

7.5.1 Architecture

The system will have a fairly standard three-tier design, building on a number of published design patterns [Fow02]. It will use the common web-based variant of the Model-View-Controller pattern, wherein communication between the model and view can be considered “batched” in page updates. It will also use a Front Controller for unified request handling and an Active

Record pattern for object persistence.

The use of these common patterns offers some advantages with respect to formal modeling. As they are widely used in PHP-based applications and application development frameworks, they also offer an opportunity for specification re-use – the formalism done for one project can be repurposed and adapted to another. In addition, because these patterns are in common use, they are well understood and so provide good examples for pedagogical purposes, as they lower the learning curve for formal modeling. These patterns will be discussed below.

Front Controller

The Front Controller is a single point of entry into an application. It receives and handles requests, performs any initialization or common processing, and then dispatches them to other classes for primary processing. In "traditional" PHP, a page-based model is used, in which each page of the application is an entry point, and so is responsible for calling any common code on it's own. The front controller consolidates this logic into one location.

In PHP, it is common to use a modified version of the front controller. The PHP runtime does a great deal of the handling for GET and POST requests, including parsing out the data fields and populating global variables with the data, as well as providing access to cookies and server information such as the request URI. Therefore, the handler portion of the front controller is reduced to simply dispatching of requests to the appropriate classes. That is the approach taken in this application.

Model-View-Controller

The Model-View-Controller or MVC pattern is a technique for separating business logic and presentation. It is a three-tiered design, with the "model" being the problem domain object, the "view" being the presentation layer, and the "controller" being the mediator between the two.

In PHP and web applications in general, it is traditional to use a modified version of the MVC pattern. MVC was originally used in graphical thick-client applications and implied a design in which updates to either the model or the view would immediately be propagated to the other. In web applications, updates are generally batched and applied on a per-request basis, i.e. when a page is posted, updates on all fields are applied at that time.

Note that the pattern does not include a concept of a "business" layer as distinct from the others. For simple applications, this layer is often skipped, with the business logic being split between the controllers and domain objects. Thus, for purposes of his application, the controller models consist of methods that display and manipulate domain objects based on user input.

Active Record

Active Record is a data access pattern. An active record is essentially a domain object that encapsulates its own database representation, in contrast to designs where database rows and domain objects are treated as distinct entities. This pattern has the advantage of a relatively simple design and works well when the object model and database schema are isomorphic.

In PHP, the dynamic nature of objects is often exploited when implementing an Active Record pattern. Specifically, developers are able to take advantage of the fact that PHP can create member variables dynamically, and simply create the object fields on the fly rather than declaring them in advance. A simple implementation of this might look like the following:

```
public function loadUser($id) {
  $find_user_query = 'SELECT * FROM users WHERE user_id
      = ?';
  $find_statement = $this->db->prepare($find_user_query
     );
  $find_statement->execute(array($id));
  $result_row = $find_statement->fetch();
  foreach ($result_row as $name=>$value) {
    $this->$name = $value;
  }
}
```

The class method in this example loads member variables from a database row. The database query returns an associative array, with field values indexed by field name. Since PHP allows object variables to be referenced indirectly, using string variables that contain the field name, it is possible to iterate over the columns returned from the query and directly set a variable of the corresponding name. As PHP also allows for dynamic variable creation, it is not even necessary to declare these variables beforehand.

Such an approach has both advantages and drawbacks. The main advantages are compactness and self-maintenance. For large database rows, this makes the code for loading objects much shorter than explicitly referencing each field. It also eliminates the need to make changes to this portion of the data access code when the schema is updated. Similar tricks can be performed with UPDATE and INSERT queries, where the data access code queries the DBMS system tables to retrieve the table schema and build the SQL statements, and then injects the values of fields with corresponding names into the query. This approach provides an extremely easy to use data access layer with little to no maintenance for simple changes such as adding a column to a table.

The main disadvantage of this approach is that it is not immediately obvious what the object contains. The mapping from class members to database fields is implicit, and so changes to the database schema have the potential to affect the code in unforeseen ways. For purposes of this project, another significant issue is that it is very difficult to model this dynamism using a formal modeling language. Not only does the VDM++ notation lack a way to describe dynamic variable creation, but it also lacks a similar concept of indirect reference. However, as VDM++ is a modeling language, it is antithetical to the purpose of the notation to directly represent particular implementation patterns. It is more important to capture the features and functionality of the system. For the detailed model, an explicit data access model will be assumed.

7.5.2 Database Design

As with all web applications of any complexity, this system will require a data store in the form of a relational database. While VDM++ does not offer any special notation for relational database description, it is still possible to model a database at various levels of detail.

For purposes of this application, it is not necessary to model the database or data access at a low level. Rather, it will suffice to represent the structure and constraints on the database rows. This can be accomplished using invariants on VDM++ types and instance variables.

A `Database` class can be defined to represent the database schema. The structure of each table is defined using a VDM++ record type. These define the names and data types of each column. For each row type, a corresponding table type is defined, which is simply a set of the row type. Instance variables of each of these table types represent the physical database tables.

Constraint Modeling

Constraints on the database tables can be represented by invariants on either the table type or on the `Database` class instance variables. Which location is appropriate depends on the constraint. For instance, the following invariant on the `UserTable` type is used to capture a uniqueness constraint on usernames:

```
public UserRow ::
    username: seq of char
    password: seq of char;


public UserTable = set of UserRow
inv usrtbl == forall r, s in set usrtbl &
              r.username = s.username => r = s;
```

Here a simple `UserRow` record type is defined, consisting of a username and password. A `UserTable` type is then defined as a set of `UserRow`. Note that the invariant here can be on the `UserTable` type because no other tables need be involved.

The invariant asserts that for any two rows in a `UserTable`, if the usernames are identical, then the rows are identical. Here the equality comparison is interpreted as identity and not simply equivalence. This is implied by `UserTable` being a *set*, which by definition does not contain duplicate members. This technique can easily be extended to express more complex uniqueness constraints, such as compound primary keys, simply by adding ANDed equality comparisons to the antecedent of the logical implication.

Other types of constraints can also be modeled by table or row type invariants. One example is simple CHECK constraints. For instance, the system designer could express that the password field on a `UserRow` is meant to contain an MD5 hash by including a clause in the type invariant such as the following:

```
inv row == len row.password = 32 and
                forall c in elems row.password & isLetter(c)
                    or isDigit(c);
```

Here `isLetter()` and `isDigit()` return true if the character is a letter or digit. Such functions do not map directly to anything in standard SQL, but they are useful for capturing requirements. In many cases, it may be possible to do checks of this kind with vendor-specific character handling functions in a particular RDBMS. For instance, both MySQL and PostgreSQL offer regular expression matching function that can be used to check for alphanumeric characters. Alternatively, the requirement could be enforced in code rather than in the database.

More complex constraints involving other tables must be expressed as invariants on the instance variables of the `Database` class. This is due to the fact that, in this specification, database tables are represented by instance variables, which cannot be referenced in type definitions. Perhaps the most common such type of constraint is the foreign key. These can be expressed by clauses in the class state invariant such as the following:

```
public Users: UserTable;
public Documents: DocumentTable;


inv
```

```
(forall d in set Documents &
 exists u in set Users & d.owner = u.username);
```

In this listing, the invariant asserts that for every record in the `Documents` table, there exists a row in the `Users` table that shares the same username as the document owner. This captures the definition of a foreign key constraint of the form:

```
FOREIGN KEY Documents(owner) REFERENCES Users(username)
  ON UPDATE RESTRICT ON DELETE RESTRICT
```

The RESTRICT on updates and deletes is a consequence of the natural meaning of the database state invariant, that is, the state cannot be changed in a way that violates the invariant. The RESTRICT clause is the closest match to this in SQL, and is also the default action for enforcing foreign key constraints in most relational database systems.

Foreign key actions

For constraints that modify data, such as cascading deletes and updates, an invariant will not suffice. It is necessary to use an operation to capture the constraint. The following `DeleteUser` operation provides an example.

```
public
DeleteUser(key: String)
ext wr Users, Documents, GroupMembers
```

```
post Users = {u | u in set Users~ & u.username <> key}
  and
    Documents = {d | d in set Documents~ & d.owner <>
      key} union
              {mk_DocumentRow(d.id, d.title, d.
                content, nil) |
               d in set Documents~ & d.owner = key}
                 and
    GroupMembers = {gm | gm in set GroupMembers~ & gm.
      user <> key};
```

The postcondition for this operation has three conjuncts. The first subtracts the deleted user from the Users set, while the third enforces a cascading delete on the group members table. The second conjunct describes an `ON DELETE SET NULL` condition, reconstructing the Documents table from two set comprehensions: the rows not owned by the deleted user and the set of record constructs that duplicate the rows owned by that user, but with the username field set to null.

Data Access Modeling

Data access and storage can be modeled in a similar fashion. Modifications to database tables can be described simply by modification of the table variables, while data retrieval can be modeled by creating subsets of the table variables.

The `iota` expression is used to select a distinct value from a set. This maps naturally to selection of a single row based on a primary key. The following operation demonstrates this, returning the single UserRow tuple that contains the appropriate username:

```
SelectUser(key: String) r: UserRow
ext rd Users
post r = iota u in set Users & u.username = key;
```

Similarly, multi-record retrievals, which map to select queries returning multiple rows, can be modeled with set comprehensions. For a single table, the pattern is the same as for individual rows, except substituting set comprehension for the iota expression. For example, the following operation returns the set of documents for a particular user:

```
SelectDocumentByOwner(owner: String) r: set of
    DocumentRow
ext rd Documents
post r = {d | d in set Documents & d.owner = owner};
```

Similar techniques can be used to model more complicated database access. For instance, nested set comprehensions and membership tests can be used to select rows based on other tables, much like joins and subquery expression in standard SQL.

7.5.3 Control Flow Modeling

Formal methods, including VDM++, are not commonly used to model the user interface (UI) of an application. As UIs tends to be visually oriented, mathematical methods are not well suited to the type of design they employ. This application is no exception – as the user interface is not particularly complex, there is little benefit to be had from attempting to model it in detail.

There are, however, some aspects of the user interface which can be useful to model. The first is control flow. The flow from one page to another has implications for usability as well as security and access control. For legitimate users, it is desirable that requests to disallowed resources do not simply fail, but leave the application in a usable state, preferably with some indication as to why the request failed. For malicious users, it is also important that requests not simply fail because, if not properly handled, an uncontrolled error could reveal information on potential attack vectors.

The second aspect is input specification. In some cases, it may also be useful for a formal specification to indicate the required inputs and outputs for each page, as well as any constraints on them. For lengthy or complex data input requirements, this can make the required input validation more explicit.

Explicit interface specification

There are any number of ways to model user inputs and control flow. The most appropriate method will depend on the purpose of the formalization and the complexity of the interface specification.

One method for detailed description of a page or screen of a user interface is to model it as a VDM++ class. In this case, important input elements, such as text boxes, can be modeled by instance variables. Simple data validation rules can then be modeled as constraints on the instance variables, while more complex validation and in-page modifications to controls can be handled as operations.

It should be noted that this method is useful primarily in more complicated scenarios. In simple cases where the user input mirrors a domain object, i.e. there is a one-to-one relationship between the class members and input controls, this is probably unnecessary. In such cases it may be clearer to simply omit explicit UI consideration and rely on the validation rules for the domain object.

Simple view modeling

The current application is simple enough that there is little to be gained by a detailed description of the user interface. In such a case, input and control flow information can be captured as part of the page design.

This can be best illustrated by an example. Consider the following excerpt from the document editing page:

```
if acl.HasPermission(current_user, doc, <edit>) then (
  if {"title", "body"} subset dom POST then (
    doc.title:= POST("title");
    doc.content:= POST("body");
    doc.Update();
    return self.Redirect("/document/view/", doc)
  ) else (
    view.Load("document_edit");
    return view.Render(doc)
  )
) else
  return self.Redirect("/error/denied/");
```

In this portion of the model, the current page performs a permission check on the selected user and document. If the check fails, control is redirected to an error page. In the case where the check succeeds, the inner "if" block represents the self-posting behavior of the form. The required input fields are indicated by the domain of the POST data. For the case where the POST keys are present, the page updates the database and redirects to the document viewing page. For the case where one or more of the required fields is *not* present, the page loads and renders an input form.

Control flow requirements

Before proceeding, it is necessary to define requirements for the application flow. This includes the enumeration of the pages in the application and how control is transferred between them.

R15 Any request by a user who is not logged in will be redirected to the login page.

R16 The system will have pages to create, edit, and delete documents; to create and edit groups; to add and remove group members; and to create new users and to log users in.

R17 On successful completion of a request, the pages in R16 will redirect to the appropriate listing page. On failure, the input form will be redisplayed.

R18 Input forms on the pages mentioned in R16 will submit to themselves.

R19 If a user does not have permission to edit or delete a document, requests for those pages will be redirected to an error page.

R20 If a user is not an administrator, requests to the group and user management pages will be redirected to an error page.

For this system, the flow of control can be captured by a simple model of the HTTP response returned for each page request. It is not necessary to describe the exact content of the response, but rather its nature. In this case, the critical data are whether the response is a display or a redirect, and the type of display or target URL respectively. The response is thus modeled as a simple two-field record type, with a response code to indicate a redirect (HTTP 302 Redirect) or display (HTTP 400 OK) response and a body field for the response text or redirect URL.

To condense and abstract the interface model, the simple response type is combined with the `View` class, which represents the view in the Model-View-Controller design. The `View` class has a `Load` method, which represents the reading of a template file for a page. The `Render` method then converts the loaded templates into a response to be sent to the client. To complement this, the abstract `ActionContorller` class has a `Redirect` method, which models redirection of the request to another page. These model the loading and output of a page to the user. The name of the view loaded provides information as to the nature of the page without detailing the precise content.

These methods are called in the concrete subclasses of `ActionController`. They are used to indicate the display of input forms, listings, and redirection on successful actions and security denials. This is most easily demonstrated by the body of the `Read` method on the `DocumentController` class:

```
if acl.HasPermission(current_user, doc, <read>) then (
  view.Load("document_view", doc);
  return view.Render()
) else
  return self.Redirect("/error/denied/")
```

This method performs a simple permission check and then either renders the document to be read or redirects to an permission denied page. Another common example of this is checking the POST variable to see if it contains form data and then processing the data or, if no data is present, displaying the input form. Since the purpose is simply to signify the type of action to be performed, this simple modeling will suffice.

8 Implementation of the Formal Specification

Having constructed a detailed model of the system, the next phase of the project is to implement the final system. The model will guide the implementation and serve as design documentation. This chapter will discuss some of the considerations involved in this phase, including variations from the model and issues not discovered until the implementation phase.

8.1 Approach

This project uses a light-weight, iterative approach to formal modeling, where the modeling language is a specification and design tool. Thus, the implementation and design are not completely separate parts of system construction. The general principle is that the model serves as a design tool and guide to coding, but implementation issues can override the original design, requiring it to be updated.

The general approach to the implementation portion of this project is to treat the code as the final refinement of the formal design. The code is therefore very similar to the detailed VDM++ model. In fact, in some places, the PHP code is essentially a direct translation of the VDM++. This similarity is particularly visible in the controller classes. The main purpose of the controller methods is to glue the domain objects and user interface together. Since the interfaces to the domain objects and inputs from the user interface are the same in both the model and code, there is, in many cases, no need to change the logic described in the model controllers.

There are several advantages to this approach. One benefit is that it provides a very straight-forward development path. The bulk of the design work is done in the modeling stage, so there is less to do and less chance of introducing design errors in the implementation stage. The model could be passed off to a junior or outsourced developer to be translated into code without introducing undue risk of adding design defects late in the process.

This close correspondence to the implementation also allows the model to serve as a type of low-level design documentation. Such documentation can be useful to developers during implementation and maintenance, as well as assisting with on-boarding of new staff. They can also be useful to quality assurance staff as source material for the design of test cases, as the documents can provide information about edge cases and limitations that need to be accounted for in testing.

However, there are some disadvantages to keeping the implementation close to the formal model. The primary drawback is that this approach limits the implementation options and may make it difficult to take advantage of certain features of the implementation language. This issue is magnified in the case of highly dynamic languages such as PHP. It can be somewhat difficult to model highly dynamic features in a static modeling language, so in areas where these are useful, the correspondence to the model is not as close.

8.2 Architecture and Design

The overall architecture of the system is essentially as described in the component-level and detailed VDM++ models. It consists of three layers: domain objects, user interface, and a controller layer connecting the other two. These are supported by a front controller handling dispatching of requests and a data access layer which encapsulates database queries. This is a fairly standard architecture for PHP applications and is used in many popular application development frameworks.

Due to the close correspondence between code and model, the classes defined in the implementation are the same as those in the model. The main differences are in the `PHP` and `DB` classes. The `PHP` class was omitted from the implementation entirely. In the model, this class served mainly as a representation of the run-time environment, and so was not necessary. Likewise, the `DB` class in the model encapsulated both data access and the storage format for the data. In the final system implementation, this serves only as a data access layer. The storage and organization of the data is left to the database management system, so replicating this in code would be redundant.

8.3 Design Variations

Despite the correspondence previously mentioned, there are several areas in which the implementation differs from the model. In some cases, these differences are due to simple convenience. Other differences are due to considerations beyond the scope of the formal model. Some of these are described below.

8.3.1 Addition of view logic

The modeling of the view layer of the application was quite simple, expressing only if a view should be loaded or if the request should be redirected. However, in the actual implementation, there is a significant amount of programming logic involved in the view templates. For instance, the correct display of URLs in the view markup requires some processing based on the configuration of the front controller. While it is possible to avoid this runtime processing of URLs by simply writing the views with a static URL, this requires more specific configuration of the web server. This is unnecessary and undesirable, as it makes setup for development and test environments more complicated.

To this end, several methods and fields have been added to the `FrontController` and `ActionController` classes as well as the `View` class, in particular the `route()` and `link()` methods. These methods on the `ActionController` class calculate the correct URLs for dynamic requests and static resources respectively based on configuration settings. The configuration needed for these methods is done on the `FrontController`, which passes this data on to the concrete `ActionController` to which requests are dispatched. The `View` class simply contains convenience wrapper methods that delegate back to the controller.

8.3.2 Usability revisions

As user interface design and implementation progressed, it became clear that some adjustments were necessary for the sake of usability and ease of management. These include adding listing pages for groups and listings of document permissions. These pages were added to the formal model and then implemented.

The pages in question were not themselves central to the functioning of the system, but were useful primarily in terms of facilitating user interaction. In other words, they served as a gateway or supporting interface to other features. For instance, without some listing or search page, it is difficult to see how a user might access the page to administer groups and view their members. Since these pages do not offer conceptually important functionality, they did not stand out as areas in need of formal analysis.

This is in part a result of the approach to formalism – the goal of the model was to clarify the system design, not to precisely specify all functionality. It is, therefore, not surprising that some aspects were overlooked in the original model. There is, associated with this, a risk of overlooking unexpectedly complicated portions of the system. However, this can be addressed by continuing to treat the formal model as a "live" document. Continuing to update the model as part of the design process for changes raised in implementation helps to maintain the other benefits of modeling.

8.3.3 Dynamic instantiation

While the VDM++ model includes a traditional static mapping for creating and calling controller actions, the actual implementation uses a dynamic method. This pattern is not uncommon in PHP applications and there are several advantages to this approach. The most obvious is that it reduces the maintenance requirements for the code – there is no mapping to update when a new controller class or page request is added.

For purposes of mapping the specification to code, there is no easy way to model this. As VDM++ has no notation for reflection or the ability to create classes or call methods dynamically by name, modeling must be indirect. One method is simply to explicitly define the mapping from URL strings to classes and methods in the model. This captures the key pieces of data that need to be conveyed, i.e. the mapping from URL to class methods, but not the actual functioning of the front controller. The alternative method is to simply declare reflection operations for checking the existence of classes and methods and define the dispatcher in terms of those. Since these operations would be either defined implicitly, or simply left undefined, this method would impede using model animation to validate the system. However, it would also give a more accurate picture of how the dispatcher actually functions.

8.3.4 Permission list population

Upon implementation of the formal specification, a key deficiency was noticed in the handling of access control lists – they were not being populated. That is, the routines to create and persist new documents did not account for the initial set-up of a document's ACL. So much focus was placed on the authorization rules for retrieving documents, that setting the rules for each new document was overlooked.

This issue was addressed by returning to the low-level formal model and adding appropriate methods. In this case, a SetDefaultPermissions method was added to the ACL class, while the Create and Edit methods of the DocumentController class were updated to call it.

8.4 Results

The goal of this project was to demonstrate the useful application of light-weight formal modeling to a typical information system. The VDM++ models were built as part of the design process, as tools for understanding the system and requirements, not for actually validating system correctness. To this end, the project was a modest success.

The high-level models were, in some ways, analogous to designing with UML diagrams. In many places, they simply declared the classes to be used and their methods and properties. While this is certainly useful information, it is nothing that could not be more concisely displayed in UML. The only real advantage to using VDM++ is for notational consistency with sections of that system that benefit from more formalism, or simply for the sake of completeness.

However, in the areas of the high-level specifications where detailed modeling was done, it was found to be useful to the development process. VDM++ provided a sufficiently powerful, yet abstract, notation with which to describe the essential characteristics of the system. The notation was flexible enough to capture design decisions where desired while deferring decision on less important aspects of the system.

At the code level, the detailed VDM++ specification was found to be very useful. For the core server-side processing handled by the system, specifying the individual operations in VDM++ during the design phase was found to significantly reduce the effort required in the implementation phase. In many areas, the implementation was simply a straight-forward translation of the specification. This is particularly visible in areas such as the concrete controller classes and the data access class. In the first case, the bulk of the algorithm design was done in the VDM++ models, so the implementation in PHP code was largely a translation of the model into code. In the second case, the post-conditions could be easily mapped to SQL, making the conversion to code relatively simple.

One circumstance in which this level of detail was useful was during a delay in the implementation phase. At one point during the project, the implementation was placed on hold for several weeks. When development recommenced after this delay, the level of detail captured by the formal model allowed work to continue with only minor interruption. Since the formal model captured the design and intent of the code, there was no need to re-evaluate the code or revisit design notes. This characteristic could be very useful in organizations with with limited resources and rapidly shifting priorities.

In this project, formal models were found to be useful as a design tool and guide to implementation for a single-developer effort. The up-front effort of building the formal models was found to be helpful and, qualitatively, to reduce the amount of effort expended in the implementation phase.

9 Conclusion

Over the years, formal methods have proven their usefulness in industry. This has been demonstrated by a large number of industrial case studies, involving a variety of methods and notations, which have reported measurable savings in development effort. Such cases are most common in large or critical systems, but not smaller information systems developed by small teams. Small scale business systems, where there is little need for intensive formal analysis, have typically not been considered the realm of formal methods.

Light-weight formal methods seem to hold the most promise for use in typical small-scale projects. The literature suggests that these applications of formalism can achieve meaningful results without requiring as large an up-front investments of time and money in training as traditional "heavy-weight" applications. This makes them more approachable and applicable to individual programmers and small groups seeking to improve their process quality.

The sample project described in the preceding chapters demonstrates the applicability of light-weight formal modeling to a typical non-critical system using a modern dynamic language. It illustrates the use of formal models as a software design tool, similar to UML and similar notations. In addition, the project demonstrates the application of several common design patterns to the formal model and translates them into the implementation. While the designs decisions made here are not the only valid choices, these methods and patterns are relatively common and idiomatic in the target implementation language and can easily serve as a template to be adapted for use in other, similar projects.

Further work in this area may include refinement and abstraction of the modeling approach presented for this project. Expansion of the use of models into proof and model animation would be a useful area of inquiry. There are also opportunities for investigating alternative design approaches. For instance, the project leaves room for other approaches to modeling data access and representation of domain objects. There is an opportunity to abstract the models and code from the base system to build a more reusable framework. This project takes a step in that direction, but remained tied to a particular system for purposes of demonstration. A fully generalized framework for developing PHP applications, built on top of formal models, would provide an excellent resource for evangelism of formal methods to main-stream developers.

A Listing of system requirements

A.1 Informal high-level requirements

I1 The system must support multiple users.

I2 The system must allow users to read, write, edit, and delete documents.

I3 It should be possible for documents to be made secure, so that only selected users can view them.

I4 Only the person who created a document should be able to edit or delete it.

I5 The owner of a document may grant other users permission to edit, delete, or read it.

I6 System administrators should be able to change the permissions and ownership of a document.

A.2 Elaborated requirements

R1 The system must have user accounts. (Implied by I1 above)

R2 The system must store documents. (Implied by I2 above)

R3 The system must support multiple concurrent users. (From I1 above)

R4 The system must allow for documents to be created, read, edited, or deleted. (From I2 above)

R5 The system must restrict read/edit/delete operations on documents to only specified users. (Generalization of I3 – I6 above).

R6 Documents must have an "owner". (Implied by I4 – I6 above)

R7 Any user may create new documents. (Implied by I3 – I6 above)

R8 When a document is created, its owner is set to the user who created it. (Implied by I4 above)

R9 The system must maintain a list of "permissions", i.e. users who are allowed to read/edit/delete each document. (Implied by I5 and I6 above)

R10 The system should maintain a list of users who are administrators. (Implied by I6 above)

R11 The system should allow the owner of each document to modify permissions on that document. (From I5 above)

R12 The system should allow users who are administrators to modify ownership and permissions on any document. (From I6 above)

R13 The system should allow administrators to create groups of users.

R14 When setting permissions, users should be able to grant or deny access too all members of a group.

R15 Any request by a user who is not logged in will be redirected to the login page.

R16 The system will have pages to create, edit, and delete documents; to create and edit groups; to add and remove group members; and to create new users and to log users in.

R17 On successful completion of a request, the pages in R16 will redirect to the appropriate listing page. On failure, the input form will be redisplayed.

R18 Input forms on the pages mentioned in R16 will submit to themselves.

R19 If a user does not have permission to edit or delete a document, requests for those pages will be redirected to an error page.

R20 If a user is not an administrator, requests to the group and user management pages will be redirected to an error page.

B Formal Models

The following sections contain code listings for the VDM++ models created for this project. The code listings are word-wrapped to fit the page.

B.1 High-Level Specification

Listing B.1: spec.vdmpp

```
-- High-level specification.
-- This traces out the key entities and operations that
    make up the system.


-- The system will have user accounts and allow users
   to log in and out.
class User
  instance variables
    public username: seq of char:= "";
    public password: seq of char:= "";
  operations
    public
    Login: () ==> ()
    Login() == is not yet specified;

    public
    Logout: () ==> ()
    Logout() == is not yet specified;

end User

-- The system will have documents.
class Document
  instance variables
    public title: seq of char:= "";
    public content: seq of char:= "";
    -- Documents will be owned by a user account.
    public owner: User;

  operations

    public
```

```
Document: seq of char * seq of char * User ==>
    Document
Document(name, body, current_user) == (
   title:= name;
   content:= body;
   owner:= current_user;
);

-- Create a new document
public
Create: () ==>()
Create() == is not yet specified;

-- Delete an existing document
public
Delete: () ==> ()
Delete() == is not yet specified;

-- Modify an existing document
public
Edit: () ==> ()
Edit() == is not yet specified;
```

```
    -- Open an existing document for reading.
    public
    Read: () ==> ()
    Read() == is not yet specified;

end Document

class Security
  types
    public AccessType = <read> | <edit> | <delete>;
    public AccessObject = User | Group;
    public Permission = map (AccessObject * Document)
       to (map AccessType to bool);

  instance variables
    -- The administrative users
    public administrators: set of User:= {};
    public groups: set of Group:= {};
    public permissions: Permission:= {|->};

  operations
```

```
-- Check if a user has permission to access a
   document.
-- Checks both user and group permissions, giving
   precedence to
-- individual user permissions.
public
PermissionCheck(u: User, d: Document, t: AccessType
   ) r: bool
ext rd permissions, groups
post
  r =
    if mk_(u, d) in set dom permissions then
      permissions(mk_(u, d))(t)
    else
      exists g in set groups &
        mk_(g, d) in set dom permissions and
        u in set g.members and
        permissions(mk_(g, d))(t) = true;

-- Set a new permission
public
SetPermission(obj: AccessObject, d: Document, perm:
    AccessType, status: bool)
    ext wr permissions
    post permissions(mk_(obj, d))(perm) = status;

--      public
--      RemovePermission(obj: AccessObject, doc: Document
   , perm: AccessType)
--      ext wr permissions
--      post not exists x in set rng permissions & dom x
   = perm;

end Security

-- Users can be aggregated into groups for access
   control.
class Group
  instance variables
    public name: seq of char;
    public members: set of User;

  operations
    public
    Group: seq of char ==> Group
    Group(nm) == (
      name:= nm;
      members:= {};
    );

    -- Add users to a group
    public
    AddUsers(us: set of User)
    ext wr members
    post members = members~ union us;

    -- Remove users from a group
    public
    DeleteUsers(us: set of User)
    ext wr members
    post members  = members~ \ us;

end Group

-- Represents the system as a whole.  This class
   contains the top-level actions
-- for each type of entity.
class System
```

```
types
  PageSpecifier = <list_documents> | <read_document>
     | <list_groups> | <show_group> | <login>;


instance variables
  security  : Security:= new Security();
  users     : set of User:= {};
  documents: set of Document:= {};
  current_user: [User]:= nil;
  next_page : PageSpecifier:= <list_documents>;


operations


 -- Group operations --


 -- Create a new group
  public
  CreateGroup(name: seq of char)
  ext wr security, next_page
  pre  current_user in set security.administrators
  post next_page = <show_group> and exists g in set
     security.groups & g.name = name;
```

```
-- Add users to a group
public
AddUsersToGroup(g: Group, us: set of User)
ext rd current_user
    wr security, next_page
pre  current_user in set security.administrators
post next_page = <show_group> and us subset g.
   members;

-- Remove users from a group
public
DeleteUsersFromGroup(g: Group, us: set of User)
ext rd current_user
    wr security, next_page
pre  current_user in set security.administrators
post next_page = <show_group> and g.members inter
   us = {};

-- Document security operations --

-- Change the owner of the document.
public
ChangeOwner(d: Document, new_owner: User)
```

```
ext rd current_user
    wr next_page
pre  current_user in set security.administrators
post next_page = <read_document> and d.owner =
   new_owner;

-- Set the permissions on a document.
public
SetPermission(d: Document, ug: Security`
   AccessObject, type: Security`AccessType, status:
    bool)
ext rd current_user
    wr security, next_page
pre  current_user in set security.administrators or
    current_user = d.owner
post next_page = <read_document> and security.
   permissions(mk_(ug, d))(type) = status;

-- Authentication operations

public
Login(username: seq of char, password: seq of char)
    r: bool
```

```
ext rd users
    wr current_user, next_page
pre  current_user = nil
post next_page = <list_documents> and
   if exists u in set users & u.username = username
       and u.password = password then
      current_user in set users and current_user.
         username = username and r = true
   else
      current_user = nil and r = false;

public
Logout()
ext wr current_user, next_page
post next_page = <login> and current_user = nil;

-- Document operations

public
ListDocuments() r: set of Document
ext rd security, documents, current_user
pre  current_user <> nil
```

```
post r = {d | d in set documents & security.
   PermissionCheck(current_user, d, <read>)};

public
ReadDocument(d: Document) r: bool
ext rd security, current_user
pre  current_user <> nil
post (security.PermissionCheck(current_user, d, <
   read>) and r = true) or r = false;

public
CreateDocument(title: seq of char, body: seq of
   char)
ext rd current_user
    wr documents, next_page
pre  current_user <> nil
post next_page = <read_document> and
     documents = documents~ union {new Document(
        title, body, current_user)};

public
EditDocument(d: Document, title: seq of char, body:
    seq of char) r: bool
    ext rd security, current_user
        wr next_page
    pre  current_user <> nil
    post next_page = <read_document> and
         (security.PermissionCheck(current_user, d, <
            edit>) and d.title = title and
         d.content = body and r = true) or (r = false);

end System
```

B.2 Component-Level Specification

Listing B.2: spec-comp.vdmpp

```
-- Component-level specification
-- This breaks down the system entities into modules,
   adding separation of concerns.

-- Top-level object representing entry point into
   system.
class System

  operations
```

```
    public Init: () ==> ()
    Init() == is not yet specified;

    public Dispatch: seq of char * seq of char ==> ()
    Dispatch(controller, action) == is not yet
       specified;

end System

-- Since the security module was defined in some detail
    in the specificaiton model,
-- it is unchanged in the component-level model.
class Security
  types
    public AccessType = <read> | <edit> | <delete>;
    public AccessObject = User | Group;
    public Permission = map (AccessObject * Document)
       to (map AccessType to bool);

  instance variables
    -- The administrative users
    public administrators: set of User:= {};
    public groups: set of Group:= {};
    public permissions: Permission:= {|->};

  operations

    -- Check if a user has permission to access a
       document.
    -- Checks both user and group permissions, giving
       precedence to
    -- individual user permissions.
    public
    PermissionCheck(u: User, d: Document, t: AccessType
       ) r: bool
    ext rd permissions, groups
    post
      r =
        if mk_(u, d) in set dom permissions then
          permissions(mk_(u, d))(t)
        else
          exists g in set groups &
            mk_(g, d) in set dom permissions and
            u in set g.members and
            permissions(mk_(g, d))(t) = true;
    -- Set a new permission
    public
    SetPermission(obj: AccessObject, d: Document, perm:
        AccessType, status: bool)
    ext wr permissions
    post permissions(mk_(obj, d))(perm) = status;

--      public
--      RemovePermission(obj: AccessObject, doc: Document
   , perm: AccessType)
--      ext wr permissions
--      post not exists x in set rng permissions & dom x
   = perm;

end Security

-- Controller for users, defining key operations on
   user accounts.
class UserController

  operations

    public Create: () ==> ()
    Create() == is not yet specified;

    public Delete: () ==> ()
    Delete() == is not yet specified;

    public Login: () ==> ()
    Login() == is not yet specified;

    public Logout: () ==> ()
    Logout() == is not yet specified;

end UserController

-- Controller for groups, defining key operations on
   user groups.
class GroupController

  operations

    public Create: () ==> ()
    Create() == is not yet specified;

    public Delete: () ==> ()
    Delete() == is not yet specified;

    public AddUser: () ==> ()
    AddUser() == is not yet specified;

    public DeleteUser: () ==> ()
    DeleteUser() == is not yet specified;

end GroupController

-- Controller for documents, defining key operations on
    documents.
class DocumentController

  operations

    public List: () ==> ()
    List() ==  is not yet specified;

    public Create: () ==> ()
    Create() == is not yet specified;

    public Delete: () ==> ()
    Delete() == is not yet specified;

    public Edit: () ==> ()
    Edit() == is not yet specified;

    public Read: () ==> ()
    Read() == is not yet specified;

    public SetPermissions: () ==> ()
    SetPermissions() ==  is not yet specified;

    public ChangeOwner: () ==> ()
    ChangeOwner() ==  is not yet specified;

end DocumentController

-- View class representing UI layer.
-- At the current level of detail, all this does is
   output something.
class View
  operations
    public Display: () ==> ()
    Display() == is not yet specified;
```

```
end View

-- Defines the pattern used for the data access objects
   .
class ActiveRecord
  operations
    public Insert: () ==> ()
    Insert() == is not yet specified;

    public Update: () ==> ()
    Update() == is not yet specified;

    public Delete: () ==> ()
    Delete() == is not yet specified;
end ActiveRecord

-- Represents documents with self-persistence.
class Document is subclass of ActiveRecord
  instance variables
    public title: seq of char:= "";
    public content: seq of char:= "";
    public owner: User;
```

```
end Document

-- Represents user accounts with self-persistence.
class User is subclass of ActiveRecord
  instance variables
    public username: seq of char:= "";
    public password: seq of char:= "";

    inv len username > 0;

end User

-- Represents user groups with self-persistence.
class Group is subclass of ActiveRecord
  instance variables
    public name: seq of char:= "";
    public members: set of User:= {};

    inv len name > 0;

end Group

-- Defines some of the key tables in the database.
```

```
class Database
  types
    public SecurityRecord = Security`AccessObject *
       Document * Security`AccessType * bool;

  instance variables
    public users: set of User:= {};
    public groups: set of Group:= {};
    public documents: set of Document:= {};
    public permissions: set of SecurityRecord:= {};

    inv (forall d in set documents & d.owner in set
       users) and
        (forall g in set groups & forall u in set g.
          members & u in set users);

  operations

end Database
```

B.3 Detailed Specification

B.3.1 Controller Classes

Listing B.3: Controllers/controller.vdmpp

```
-- Base action controller class.
-- Initializes database connection and checks for a
   logged-in user.
class ActionController is subclass of PHP, BaseObject

  instance variables
    protected current_user: [User]:= nil;
    protected db: DB:= new DB();
    protected postdata: Array:= POST;
    public view: View:= new View();

  operations

    protected
    ActionController: () ==> ActionController
    ActionController() == (
      current_user:=
        if Auth`CURR_USER in set dom SESSION then
          SESSION(Auth`CURR_USER)
        else
          nil;
    );
    public
    Redirect: String ==> Response
    Redirect(url) == is not yet specified;

    public
    Redirect: String * BaseObject ==> Response
    Redirect(url, obj) == is not yet specified;

end ActionController
```

Listing B.4: Controllers/frontcontroller.vdmpp

```
-- The application front controller. This represents
   the single point
-- of entry to the application. It performs common
   initialization
-- and dispatches requests to separate controllers
   based on the
-- required action.
class FrontController is subclass of PHP

  instance variables
```

```
functions

  private
  GetController: String -> ActionController
  GetController(ctlname) ==
    cases ctlname:
      "user" -> new UserController(),
      "document" -> new DocumentController(),
      "group" -> new GroupController(),
      "index" -> new IndexController(),
      "" -> new IndexController(),
      others -> new ErrorController()
    end;

  -- Split the URL into its component paths, i.e.
  -- return the itens between slashes as a list.
  private
  SplitURL(url: String) ret: seq of String
  pre len url > 2 and '/' in set elems url and hd url
      <> '/' and url(len url) <> '/' -- URL must have
      2 components, e.g. 'a/b'
  post len ret >= 2;
```

```
operations

  private
  CallAction: ActionController * String * seq of
     String ==> Response
  CallAction(ctl, actname, data) == is not yet
     specified;

  -- Parse the URL into a controller and action and
     dispatch
  public
  Dispatch: String ==> Response
  Dispatch(url) == (
    dcl ctl: ActionController,
        urldata: seq of String;
    urldata:= SplitURL(url);
    ctl:= GetController(urldata(1));
    -- Only selected methods of the UserController
       allow anonymous logins.
    -- For all others, redirect to the login page.
    if not isofclass(UserController, ctl) and Auth`
       CURR_USER not in set dom SESSION then (
      return ctl.Redirect("/user/login/");
      ) else
        return CallAction(ctl, urldata(2), tl urldata);
    );

end FrontController
```

Listing B.5: Controllers/documentcontroller.vdmpp

```
class DocumentController is subclass of
   ActionController

  functions
    public
    toArray: set of DB`UserPermissionRow -> Array
    toArray(row) == is not yet specified;

    public
    toArray: set of DB`GroupPermissionRow -> Array
    toArray(row) == is not yet specified;

  operations

    public
    List: () ==> Response
```

```
List() == (
  dcl doc: set of Document:= Document`GetAll(),
      showdocs: set of Document,
      acl: ACL:= new ACL();
  showdocs:= {d | d in set doc & acl.HasPermission(
     current_user, d, <read>) };
  view.Load("document_list", PHP`toObj(showdocs));
  return view.Render()
)
pre current_user <> nil;

public
Read: nat ==> Response
Read(id) == (
  dcl doc: [Document]:= Document`GetById(id),
      acl: ACL:= new ACL();
  if acl.HasPermission(current_user, doc, <read>)
     then (
    view.Load("document_view", doc);
    return view.Render()
  ) else
    return self.Redirect("/error/denied/")
)
```

```
pre current_user <> nil;

public
Edit: nat ==> Response
Edit(id) == (
  dcl doc: [Document]:= Document`GetById(id),
      acl: ACL:= new ACL();
  if acl.HasPermission(current_user, doc, <edit>)
     then (
    if {"title", "body"} subset dom POST then (
      doc.title:= POST("title");
      doc.content:= POST("body");
      doc.Update();
      return self.Redirect("/document/view/", doc)
    ) else (
      view.Load("document_edit");
      return view.Render()
    )
  ) else
    return self.Redirect("/error/denied/");
);

public
```

```
Delete: nat ==> Response
Delete(id) == (
  dcl doc: [Document]:= Document`GetById(id),
      acl: ACL:= new ACL();
  if acl.HasPermission(current_user, doc, <delete>)
     then (
    doc.Delete();
    return self.Redirect("/document/list")
  ) else (
    return self.Redirect("/error/denied/")
  )
);

public
Create: () ==> Response
Create() == (
  dcl doc: Document:= new Document(),
      acl: ACL:= new ACL();
  if {"title", "body"} subset dom POST then (
    doc.title:= POST("title");
    doc.content:= POST("body");
    doc.owner:= current_user;
    doc.Insert();
```

```
      acl.SetDefaultPermissions(doc);
      return self.Redirect("/document/view/", doc)
    ) else (
      view.Load("document_create");
      return view.Render()
    )
  )
  pre current_user <> nil;

  public
  ChangeOwner: nat ==> Response
  ChangeOwner(id) == (
    dcl doc: [Document]:= Document`GetById(id),
        usr: [User];
    if doc = nil then
      return self.Redirect("/error/invalid_document
         /");
    if not current_user.IsAdministrator() then
      return self.Redirect("/error/denied/");
    if "username" in set dom POST then (
      usr:= User`GetByName(POST("username"));
      if usr <> nil then (
        doc.owner:= usr;
```

```
          doc.Update();
          return self.Redirect("/document/view/", doc)
        ) else (
          view.Load("document_change_owner");
          return view.Render()
        )
      ) else (
        view.Load("document_change_owner");
        return view.Render()
      )
    );

    public
    SetUserPermission: nat ==> Response
    SetUserPermission(id) == (
      dcl doc: [Document]:= Document`GetById(id),
          usr: [User],
          acl: ACL:= new ACL(),
          perm: [ACL`Permission],
          status: bool;
      if doc = nil then
        return self.Redirect("/error/invalid_document
           /");
```

```
   if not current_user.IsAdministrator() and
      current_user <> doc.owner then
     return self.Redirect("/error/denied/");
   if {"user", "perm", "status"} subset dom POST
      then (
     usr:= User`GetByName(POST("user"));
     perm:= acl.ToPermission(POST("perm"));
     status:= POST("status") = "granted";
     if usr <> nil and perm <> nil then (
       acl.SetPermission(usr, doc, perm, status);
       return self.Redirect("/document/view/", doc)
     ) else (
       view.Load("set_user_permission");
       return view.Render()
     )
   ) else
     view.Load("set_user_permission");
     return view.Render()
);

public
SetGroupPermission: nat ==> Response
SetGroupPermission(id) == (
dcl doc: [Document]:= Document‘GetById(id),
      grp: [Group],
      acl: ACL:= new ACL(),
      perm: [ACL‘Permission],
      status: bool;
if doc = nil then
   return self.Redirect("/error/invalid_document
       /");
if not current_user.IsAdministrator() and
    current_user <> doc.owner then
   return self.Redirect("/error/denied/");
if {"group", "perm", "status"} subset dom POST
    then (
   grp:= Group‘GetByName(POST("user"));
   perm:= acl.ToPermission(POST("perm"));
   status:= POST("status") = "granted";
   if grp <> nil and perm <> nil then (
      acl.SetPermission(grp, doc, perm, status);
      return self.Redirect("/document/view/", doc)
   ) else
      view.Load("set_group_permission");
      return view.Render()
) else
```

```
    view.Load("set_group_permission");
    return view.Render()
);

public
ChangePermissions: nat ==> Response
ChangePermissions(id) == (
  dcl doc: [Document]:= Document`GetById(id),
      userPerms: set of DB`UserPermissionRow,
      groupPerms: set of DB`GroupPermissionRow;
  if (doc = nil) then
    return self.Redirect("/error/invalid_document
       /");
  groupPerms:= db.SelectGroupPermissionByDocument(
     doc.id);
  userPerms:= db.SelectUserPermissionByDocument(doc
     .id);
  view.Load("document_change_group_permissions",
     toArray(groupPerms));
  view.Load("document_change_user_permissions",
     toArray(userPerms));
  return view.Render();
);
```

```
end DocumentController
```

Listing B.6: Controllers/groupcontroller.vdmpp

```
class GroupController is subclass of ActionController

  functions
    public toArray: set of Group -> Array
    toArray(groups) == is not yet specified;

    public toArray: [Group] -> Array
    toArray(groups) == is not yet specified;

  operations

    public
    List: () ==> Response
    List() == (
      dcl groups: set of Group:= Group`GetAll();
      if not current_user.IsAdministrator() then
        return self.Redirect("/error/denied/");
      view.Load("group_list", toArray(groups));
      return view.Render();
```

```
);

public
Show: String ==> Response
Show(groupname) == (
  dcl g: [Group]:= Group`GetByName(groupname);
  if not current_user.IsAdministrator() then
    return self.Redirect("/error/denied/");
  view.Load("group_show", toArray(g));
  return view.Render();
);

public
Create: () ==> Response
Create() ==  (
  dcl grp: Group:= new Group();
  if not current_user.IsAdministrator() then
    return self.Redirect("/error/denied/");
  if {"group_name", "description"} subset dom POST
     then (
    grp.name:= POST("group_name");
    grp.description:= POST("description");
    grp.Insert();
    return self.Redirect("/group/list/");
  ) else (
    view.Load("group_create");
    return view.Render();
  );
);

public
AddUser: String ==> Response
AddUser(groupname) == (
  dcl g: [Group]:= Group`GetByName(groupname),
      u: [User],
      ret: bool:= false;
  if not current_user.IsAdministrator() then
    return self.Redirect("/error/denied/");
  if g = nil then
    return self.Redirect("/error/invalid/");
  if "username" not in set dom POST then (
    view.SetMessage("failure");
  ) else (
    u:= User`GetByName(POST("username"));
    if u = nil then (
      view.SetMessage("failure");
    ) else (
      ret:= g.Add(u);
      if ret then view.SetMessage("success");
      if not ret then view.SetMessage("failure");
    );
  );
  view.Load("group_show");
  return view.Render();
);

public
RemoveUser: String ==> Response
RemoveUser(groupname) == (
  dcl g: [Group]:= Group`GetByName(groupname),
      u: [User],
      ret: bool:= false;
  if not current_user.IsAdministrator() then
    return self.Redirect("/error/denied/");
  if g = nil then
    return self.Redirect("/error/invalid/");
  if "username" not in set dom POST then (
    view.SetMessage("failure");
  ) else (
        u:= User‘GetByName(POST("username"));
        if u = nil then (
          view.SetMessage("failure");
        ) else (
          ret:= g.Remove(u);
          if ret then view.SetMessage("success");
          if not ret then view.SetMessage("failure");
        );
      );
      view.Load("group_show");
      return view.Render();
    );

end GroupController
```

Listing B.7: Controllers/usercontroller.vdmpp

```
class UserController is subclass of ActionController
  functions
    public
    toArray: set of User -> BaseObject
    toArray(users) == is not yet specified;

  operations
```

```
public
Create: () ==> Response
Create() == (
  dcl usr: User:= new User();
  if current_user = nil or not current_user.
     IsAdministrator() then
    return self.Redirect("/error/denied/")
  else if not {"username", "password", "confirm"}
     subset dom POST then (
    if POST("password") = POST("confirm") then (
      usr.username:= POST("username");
      usr.SetPassword(POST("password"));
      usr.Insert();
      -- Reload view with success message
    );
    view.Load("user_create");
    return view.Render()
  )
)
pre current_user <> nil;

public
```

```
List: () ==> Response
List() == (
  dcl users: set of User:= User`GetAll();
  if not current_user.IsAdministrator() then
    return self.Redirect("/error/denied/");
  view.Load("user_list", toArray(users));
)
pre current_user <> nil;

public
ChangePassword: PHP`String ==> Response
ChangePassword(username) == (
  dcl usr: [User]:= User`GetByName(username);
  if current_user = nil or not current_user.
     IsAdministrator() then (
    return self.Redirect("/error/denied/")
  );
  if usr = nil then (
    return self.Redirect("/error/invalid_user/");
  );
  if not {"password", "confirm"} subset dom POST
     and
         POST("password") = POST("confirm") then (
    usr.SetPassword(POST("password"));
    usr.Save();
    return self.Redirect("/user/list/");
  ) else (
    view.Load("user_password");
    return view.Render();
  );
)
pre current_user <> nil and current_user.
   IsAdministrator();



public
Login: () ==> Response
Login() == (
  dcl logged_in: bool:= false;
  if not {"username", "password"} subset dom POST
     then (
    view.Load("user_login");
    return view.Render()
  ) else (
    logged_in:= Auth`Login(POST("username"), POST("
       password"));
```

```
        if logged_in then (
          current_user:= User`GetByName(POST("username
             "));
          return self.Redirect("/document/list/")
        ) else (
          view.Load("user_login");
          return view.Render()
        )
      )
    )
    pre current_user = nil;



    public
    Logout: () ==> Response
    Logout() == (
      current_user:= nil;
      Auth`Logout();
      return self.Redirect("/user/login/")
    );

end UserController
```

Listing B.8: Controllers/indexcontroller.vdmpp

```
class IndexController is subclass of ActionController


  operations
    public
    List: () ==> Response
    List() == (
      view.Load("index");
      return view.Render()
    );


end IndexController
```

Listing B.9: Controllers/errorcontroller.vdmpp

```
class ErrorController is subclass of ActionController


  operations


    public
    PermissionDenied: () ==> Response
    PermissionDenied() == (
      view.Load("error_permission_denied");
      return view.Render()
    );

    public
    InvalidData: () ==> Response
    InvalidData() == (
      view.Load("error_invalid_data");
      return view.Render()
    );

end ErrorController
```

B.3.2 Model Classes

Listing B.10: Models/document.vdmpp

```
class Document is subclass of DataModel, BaseObject

  instance variables
    public id: nat;
    public title: PHP`String:= "";
    public content: PHP`String:= "";
    public owner: User;

  operations
```

```
Document: DB`DocumentRow ==> Document
Document(row) == (
  id:= row.id;
  title:= row.title;
  content:= row.content;
  owner:= User`GetByName(row.owner);
);

Document: PHP`String * PHP`String * User ==>
   Document
Document(t, c, o) == (
  title:= t;
  content:= c;
  owner:= o
);

private static
DocumentExists(docid: nat, dbconn: DB) r: bool
post r = exists d in set dbconn.Documents & d.id =
   docid;

public static
GetById: nat ==> [Document]
```

```
GetById(docid) == (
  dcl d: DB:= new DB();
  if DocumentExists(docid, d) then
    return new Document(d.SelectDocument(docid))
  else
    return nil
);

public static
GetAll:() ==> set of Document
GetAll() == (
  dcl dbconn: DB:= new DB();
  return {new Document(row) | row in set dbconn.
     Documents}
);

public
Insert: () ==> ()
Insert() == (
  dcl row: DB`DocumentRow:= mk_DB`DocumentRow(self.
     id, self.title, self.content, self.owner.
     username);
  db.InsertDocument(row);
);

public
Update: () ==> ()
Update() == (
  dcl row: DB`DocumentRow:= mk_DB`DocumentRow(self.
     id, self.title, self.content, self.owner.
     username);
  db.UpdateDocument(self.id, row);
);

public
Save: () ==> ()
Save() == (
  if exists row in set db.Documents & row.id = self
     .id then
    Insert()
  else
    Update()
);

public
Delete: () ==> ()
```

```
    Delete() == (
      db.DeleteDocument(self.id)
    );
end Document
```

Listing B.11: Models/group.vdmpp

```
class Group is subclass of PHP, DataModel

  instance variables
    public name: String:= "";
    public description: String:= "";

  operations

    public
    Group: DB`GroupRow ==> Group
    Group(row) == (
      name:= row.name;
      description:= row.desc;
    );

    public
    IsMember(u: User) r: bool
```

```
ext rd db, name
post r = (exists row in set db.GroupMembers & row.
   user = u.username and row.group = name);

public
GetMembers() r: set of User
ext rd db, name
post r = {User`GetByName(gm.user) | gm in set db.
   GroupMembers & gm.group = name};

public
Add: User ==> bool
Add(u) == (
  return db.InsertGroupMember(mk_DB`GroupMemberRow(
     self.name, u.username));
)
post db.GroupMembers = db.GroupMembers union {mk_DB
   `GroupMemberRow(self.name, u.username)};

public
Remove: User ==> bool
Remove(u) == (
  return db.DeleteGroupMember(self.name, u.username
     );
)
post db.GroupMembers = db.GroupMembers \ {mk_DB`
   GroupMemberRow(self.name, u.username)};

public static
GetByName: String ==> [Group]
GetByName(grpname) == (
  dcl d: DB:= new DB();
  if exists g in set d.Groups & g.name = grpname
     then
    let row in set d.Groups be st row.name =
       grpname in
    return new Group(row)
  else
    return nil
);

public static
GetAll: () ==> set of Group
GetAll() == (
  dcl dbconn: DB:= new DB();
```

```
    return {new Group(row) | row in set dbconn.Groups
       }
  );

  public
  Insert: () ==> ()
  Insert() == (
    dcl row: DB`GroupRow:= mk_DB`GroupRow(self.name,
       self.description);
    db.InsertGroup(row);
  );

  public
  Update: () ==> ()
  Update() == (
    dcl row: DB`GroupRow:= mk_DB`GroupRow(self.name,
       self.description);
    db.UpdateGroup(self.name, row);
  );

end Group
```

Listing B.12: Models/user.vdmpp

```
-- Represents a user in the system.
-- Fulfills requirements: 1
class User is subclass of BaseObject, DataModel


  values
    -- Length for pasword hashes, we assume MD5.
    PW_HASH_LENGTH: int = 32;
    AUTH_TOK_NAME: PHP`String = "auth";


  instance variables
    public username: PHP`String;
    public password: PHP`String;


  functions
    -- Calculate a hash value for a string
    public static
    hash(data: PHP`String) r: PHP`String
    pre  len data > 0
    post len r = PW_HASH_LENGTH ;


  operations
```

```
private static
UserExists(name: PHP`String, d: DB) r: bool
post r = exists u in set d.Users & u.username =
   name;



public
User: () ==> User
User() == (
  username:= "";
  password:= "";
);

public
User: DB`UserRow ==> User
User(row) == (
  username:= row.username;
  password:= row.password;
);

-- Change the user's stored password to use the
   given string.
public
```

```
SetPassword(pass: PHP`String)
ext wr password
post password = hash(pass) ;

public
CheckPassword(pass:PHP`String) r: bool
ext rd password
post r = (hash(pass) = password);

-- Get the list of groups to which this user
   belongs
public
GetGroups() r: set of Group
ext rd db, username
post r = {Group`GetByName(g.group) | g in set db.
   GroupMembers & g.user = username};

-- Determine if this user is an administrator
public
IsAdministrator() r: bool
ext rd db, username
post r = (exists g in set GetGroups() & g.name = "
   administrators");
```

```
public static
GetByName: String ==> [User]
GetByName(name) == (
  dcl d: DB:= new DB();
  if UserExists(name, d) then
    let r in set d.Users be st r.username = name in
    return new User(r)
  else
    return nil
);

public static
GetAll: () ==> set of User
GetAll() == (
  dcl d: DB:= new DB();
  return {new User(r) | r in set d.Users}
);

public
Insert: () ==> ()
Insert() == (
  dcl row: DB`UserRow:= mk_DB`UserRow(self.username
     , self.password);
  db.InsertUser(row);
);

public
Update: () ==> ()
Update() == (
  dcl row: DB`UserRow:= mk_DB`UserRow(self.username
     , self.password);
  db.UpdateUser(self.username, row);
);

public
Save: () ==> ()
Save() == (
  if exists row in set db.Users & row.username =
     self.username then
    Insert()
  else
    Update()
);
```

```
end User
```

B.3.3 Other Classes

Listing B.13: activerecord.vdmpp

```
class ActiveRecord is subclass of DB


  types
    public DataRow = map PHP`String to PHP`Scalar;


  instance variables
    protected db: DB:= new DB();
    public id: nat:= 0;


  operations


    public
    Save: () ==> ()
    Save() == is subclass responsibility;


    public
    Update: () ==> ()
    Update() == is subclass responsibility;
    public
    Insert: () ==> ()
    Insert() == is subclass responsibility;

end ActiveRecord
```

Listing B.14: auth.vdmpp

```
-- This class handles authenticating a user session
-- against the application.
class Auth is subclass of PHP

  values
    public CURR_USER: String = "current_user";
    TOKEN_NAME: String = "logged_in";
    COOKIE_NAME: String = "login_token";

  instance variables

  functions

  operations

    -- Determine if a valid login cookie is present.
```

```
public static
HasLoginCookie() r: bool
post r = (exists c in set ClientCookies &
    c.name = COOKIE_NAME and c.cookieIsValid() );

-- Determine if a valid login token is present in
   either
-- the session or a client cookie.
public static
IsLoggedIn() r: bool
post r = (SESSION(TOKEN_NAME) = true or
   HasLoginCookie() );

-- Log in the user and set the authentication token
   .
-- Takes an optional parameter to "remember" the
   user
-- login in a client cookie.
public static
Login: String * String * bool ==> bool
Login(uname, pw, remember) == (
  dcl usr: [User];
  usr:= User`GetByName(uname);
```

```
    if usr = nil then (
      return false
    ) else if usr.password = pw then (
      SESSION:= SESSION ++ {Auth`CURR_USER |->usr.
         username};
      if remember then Cookie`SetCookie(COOKIE_NAME,
         usr.username);
      return true
    ) else return false
  );

-- Overload for default value of optional third
   parameter.
-- Would this be better expressed as a nullable
   parameter?
public static
Login: String * String ==> bool
Login(uname, pw) == return Login(uname, pw, false);

--- Destroy the user's authentication tokens.
public static
Logout()
post TOKEN_NAME not in set dom SESSION and
```

```
        not HasLoginCookie() ;


end Auth
```

Listing B.15: cookie.vdmpp

```
class Cookie


  instance variables
    public name   : PHP`String:= "";
    public val    : PHP`String:= "";
    public expire : PHP`Timestamp:= 0;
    public domain : PHP`String:= "";

  functions
    public
    now: () -> int
    now() == is not yet specified;

  operations


    public
    Cookie: PHP`String * PHP`String ==> Cookie
    Cookie(n, v) == (
      name:= n;
      val:= v;
    );

    public
    Cookie: PHP`String * PHP`String * PHP`Timestamp ==>
        Cookie
    Cookie(n, v, e) == (
      name:= n;
      val:= v;
      expire:= e;
    );

    public
    cookieIsValid() r: bool
    ext rd expire
    post r = (expire < now()) ;

    public static
    SetCookie: PHP`String * PHP`String ==> ()
    SetCookie(cname, cval) == is not yet specified;

    public static
```

```
    SetCookie: PHP`String * PHP`String * nat ==> ()
    SetCookie(cname, cval, cexpire) == is not yet
       specified;


end Cookie
```

Listing B.16: datamodel.vdmpp

```
class DataModel is subclass of PHP


  types


    public DataRow = DB`UserRow | DB`DocumentRow | DB`
       GroupRow | DB`GroupMemberRow |
                     DB`UserPermissionRow | DB`
                        GroupPermissionRow;


    public DataTable = DB`UserTable | DB`DocumentTable
       | DB`GroupTable | DB`GroupMemberTable |
                       DB`UserPermissionTable | DB`
                          GroupPermissionTable;


  instance variables
    protected db: DB:= new DB();
  operations



end DataModel
```

Listing B.17: db.vdmpp

```
class DB is subclass of PHP


  types
    public UserRow ::
        username: String
        password: String
    inv row == len row.password  = 32 and
               forall c in set elems row.password &
                  isUpper(c) or isDigit(c);

    public DocumentRow ::
        id: nat
        title: String
        content: String
        owner: [String];
```

```
public GroupRow ::
    name: String
    desc: String;

public GroupMemberRow ::
    group: String
    user: String;

public UserPermissionRow ::
    document_id: nat
    username: String
    permission: ACL`Permission
    granted: bool;

public GroupPermissionRow ::
    document_id: nat
    group_name: String
    permission: ACL`Permission
    granted: bool;

public GroupDocumentRow ::
    group: GroupRow
    document: DocumentRow;
```

```
public UserTable = set of UserRow
inv usrtbl == forall r, s in set usrtbl & r.
   username = s.username => r = s;

public DocumentTable = set of DocumentRow
inv doctbl == forall r, s in set doctbl & r.id = s.
   id => r = s;

public GroupTable = set of GroupRow
inv grptbl == forall r, s in set grptbl & r.name =
   s.name => r = s;

public GroupMemberTable = set of GroupMemberRow
inv grpmemtbl == forall r, s in set grpmemtbl &
(r.group = s.group and r.user = s.user) => r = s;

public UserPermissionTable = set of
   UserPermissionRow
inv userpermtbl == forall r, s in set userpermtbl &
(r.document_id = s.document_id and r.username = s.
   username
```

```
                                          and r.permission = s
                                             .permission) => r
                                             = s;

  public GroupPermissionTable = set of
     GroupPermissionRow
  inv grppermtbl == forall r, s in set grppermtbl &
  (r.document_id = s.document_id and r.group_name = s
     .group_name
                                          and r.permission = s
                                             .permission) => r
                                             = s;

instance variables

  public Users: UserTable:= {};
  public Documents: DocumentTable:= {};
  public Groups: GroupTable:= {};
  public GroupMembers: GroupMemberTable:= {};
  public UserPermissions: UserPermissionTable:= {};
  public GroupPermissions: GroupPermissionTable:= {};

  inv
```

```
-- Foreign key Documents to Users
(forall d in set Documents & d.owner = nil or (
   exists u in set Users & d.owner = u.username) )
   and
-- Foreign key GroupMembers to Users and to Groups
(forall m in set GroupMembers & (exists u in set
   Users & u.username = m.user) and
                                (exists g in set
                                   Groups & g.name
                                   = m.group) ) and
-- Username and password not null
(forall u in set Users & u.username <> "" and u.
   password <> "") and
-- Unique group name
(card {g.name | g in set Groups} = card Groups) and
-- Group name not null
(forall g in set Groups & g.name <> "") and
-- Group permission foreign keys to Groups and
   Documents
(forall p in set GroupPermissions & (exists g in
   set Groups & g.name = p.group_name) and
                                    (exists d in
                                       set
```

```
                                                         Documents &
                                                         d.id  = p.
                                                         document_id)
                                                          ) and
    -- User permission foreign keys to Users and
       Documents
    (forall p in set UserPermissions & (exists u in set
        Users & u.username = p.username) and
                                                     (exists d in set
                                                         Documents &
                                                        d.id  = p.
                                                        document_id)
                                                        );


  functions


    static public
    isUpper(c: char) r: bool
    post r = (c in set {'A','B','C','D','E','F','G',
                        'H','I','J','K','L','M','N',
                        'O','P','Q','R','S','T','U',
                        'V','W','X','Y','Z'});
    static public
    isLower(c: char) r: bool
    post r = (c in set {'a','b','c','d','e','f','g',
                       'h','i','j','k','l','m','n',
                       'o','p','q','r','s','t','u',
                        'v','w','x','y','z'});

    static public
    isDigit(c: char) r: bool
    post r = (c in set
       {'0','1','2','3','4','5','6','7','8','9'});

  operations

    public
    SelectUser(key: String) r: UserRow
    ext rd Users
    post r = iota u in set Users & u.username = key;

    public
    UpdateUser(key: String, data: UserRow)
    ext wr Users
```

```
post let row = iota u in set Users & u.username =
   key in
         row.username = data.username and row.
            password = data.password;


public
InsertUser(u: UserRow)
ext wr Users
pre  not exists ur in set Users & ur.username = u.
   username
post Users = Users~ union {u};


public
DeleteUser(key: String)
ext wr Users, Documents, GroupMembers
post Users = {u | u in set Users~ & u.username <>
   key} and
     -- ON DELETE SET NULL action on foreign key
        from Documents
     Documents = {d | d in set Documents~ & d.owner
        <> key} union
                 {mk_DocumentRow(d.id, d.title, d.
                    content, nil) |
                d in set Documents~ & d.owner =
                   key} and
     -- ON DELETE CASCADE action on foreign key
        from GroupMembers
     GroupMembers = {gm | gm in set GroupMembers~ &
        gm.user <> key};

public
SelectDocument(id: nat) r: DocumentRow
ext rd Documents
post r = iota d in set Documents & d.id = id;

public
SelectDocumentByOwner(owner: String) r: set of
   DocumentRow
ext rd Documents
post r = {d | d in set Documents & d.owner = owner
   };

public
SelectDocumentsByGroup(group: String) r: set of
   GroupDocumentRow
ext rd Documents, Groups, GroupMembers
```

```
post r = {mk_GroupDocumentRow(g, d) | g in set
   Groups, d in set Documents, gm in set
   GroupMembers &
                                               gm.user = d.
                                                  owner and
                                                 gm.group =
                                                  g.name
                                                 and g.name
                                                  = group};

public
SelectReadableDocuments(user: String) r: set of
   DocumentRow
ext rd Documents, GroupMembers, UserPermissions,
   GroupPermissions
post r = {d | d in set Documents &
                (exists up in set UserPermissions &
                   up.document_id = d.id and
                                    up.username = user
                                       and up.
                                       permission =<
                                       read> and up.
                                       granted) or
                (exists gp in set GroupPermissions &
                  gp.document_id = d.id and
                                gp.group_name in
                                  set {g.group | g
                                   in set
                                  GroupMembers & g
                                  .user = user}
                                  and
                                gp.permission = <
                                  read> and gp.
                                  granted)};

public
InsertDocument(d: DocumentRow)
ext wr Documents
    rd Users
pre  exists u in set Users & u.username = d.owner
post Documents = Documents~ union {d} and
     -- Increasing auto-increment key
     forall row in set Documents~ & d.id > row.id;

public
UpdateDocument(key: nat, data: DocumentRow)
```

```
ext wr Documents, Users
pre  exists u in set Users & u.username = data.
   owner
post let row = iota d in set Documents & d.id = key
    in
         row.id = data.id and row.title = data.
            title and
         row.content = data.content and row.owner =
             data.owner;

public
DeleteDocument(id: nat)
ext wr Documents, UserPermissions, GroupPermissions
post Documents = {d | d in set Documents~ & d.id =
   id} and
     UserPermissions = {up | up in set
        UserPermissions~ & up.document_id <> id}
        and
     GroupPermissions = {gp | gp in set
        GroupPermissions~ & gp.document_id <> id};

public
SelectGroup(name: String) r: GroupRow
```

```
ext rd Groups
post r = iota g in set Groups & g.name = name;


public
InsertGroup(g: GroupRow)
ext wr Groups
post Groups = Groups~ union {g};


public
UpdateGroup(key: String, data: GroupRow)
ext wr Groups
post let row = iota g in set Groups & g.name = key
   in
         row.name = data.name and row.desc = data.
            desc;


public
DeleteGroup(name: String)
ext wr Groups, GroupMembers, GroupPermissions
post Groups = {g | g in set Groups~ & g.name <>
   name} and
     -- Cascading delete on GroupMembers table.
```

```
    GroupMembers = {gm | gm in set GroupMembers~ &
        gm.group <> name} and
    -- Cascading delete on GroupPermissions table.
    GroupPermissions = {gp | gp in set
        GroupPermissions & gp.group_name <> name};

public
SelectGroupMembersByUser(user: String) r: set of
    GroupMemberRow
ext rd GroupMembers
post r = {gm | gm in set GroupMembers & gm.user =
    user};

public
SelectGroupMembersByGroup(group: String) r: set of
    GroupMemberRow
ext rd GroupMembers
post r = {gm | gm in set GroupMembers & gm.group =
    group};

public
DeleteGroupMember(group: String, user: String) r:
    bool
```

```
ext wr GroupMembers
post (GroupMembers = {gm | gm in set GroupMembers~
    & gm.group <> group and gm.user <> user}) or
   not r;

public
InsertGroupMember(gm: GroupMemberRow) r: bool
ext wr GroupMembers
post (GroupMembers = GroupMembers~ union {gm}) or
   not r;

public
SelectUserPermissionByUser(user: String) r: set of
   UserPermissionRow
ext rd UserPermissions
post r = {up | up in set UserPermissions & up.
   username = user};

public
SelectUserPermissionByDocument(id: nat) r: set of
   UserPermissionRow
ext rd UserPermissions
```

```
post r = {up | up in set UserPermissions & up.
   document_id = id};

public
InsertUserPermission(up: UserPermissionRow)
ext wr UserPermissions
post UserPermissions = UserPermissions~ union {up};

public
DeleteUserPermission(docid: nat, user: String)
ext wr UserPermissions
post UserPermissions = {up | up in set
   UserPermissions~ & up.document_id = docid and up
   .username = user};

public
SelectGroupPermissionByUser(group: String) r: set
   of GroupPermissionRow
ext rd GroupPermissions
post r = {gp | gp in set GroupPermissions & gp.
   group_name = group};

public
```

```
    SelectGroupPermissionByDocument(id: nat) r: set of
        GroupPermissionRow
    ext rd GroupPermissions
    post r = {gp | gp in set GroupPermissions & gp.
        document_id = id};

    public
    InsertGroupPermission(gp: GroupPermissionRow)
    ext wr GroupPermissions
    post GroupPermissions = GroupPermissions~ union {gp
        };

    public
    DeleteGroupPermission(docid: nat, group: String)
    ext wr GroupPermissions
    post GroupPermissions = {gp | gp in set
        GroupPermissions~ & gp.document_id = docid and
        gp.group_name = group};
end DB
```

Listing B.18: perms.vdmpp

```
class ACL is subclass of PHP
```

```
values
  DEFAULT_ACCESS: AccessControlList = {|->};

types

  public Permission = <read> | <edit> | <create> | <
     delete>;
  public AccessObject = Group | User;
  public PermissionList = map Permission to bool;
  public AccessControlList = map AccessObject to
     PermissionList;

instance variables
  private db: DB:= new DB();

functions

  public
  ToPermission(s: String) ret: [Permission]
  post ret = cases s: "read"   -> <read>,
                      "edit"   -> <edit>,
                      "create" -> <create>,
                      "delete" -> <delete>,
                others    -> nil
        end;

operations

  public
  HasPermission(u: [User], d: [Document], p:
     Permission) r: bool
  ext rd db
  post r = if u = nil or d = nil then
    false
    else (
    (exists row in set db.UserPermissions &
      row.username = u.username and row.document_id =
          d.id and row.permission = p and row.granted
        )
    or
    (exists row in set db.GroupPermissions &
      Group`GetByName(row.group_name) in set u.
         GetGroups() and row.document_id = d.id and
         row.permission = p and row.granted)
    );
```

```
public
SetPermission(u: User, d: Document, p: Permission,
   granted: bool)
ext wr db
post exists row in set db.UserPermissions &
     row = mk_DB`UserPermissionRow(d.id, u.username
        , p, granted);

public
SetPermission(g: Group, d: Document, p: Permission,
    granted: bool)
ext wr db
post exists row in set db.GroupPermissions &
     row = mk_DB`GroupPermissionRow(d.id, g.name, p
        , granted);

public
SetDefaultPermissions(d: Document) == (
  dcl admins: Group:= new Group(iota r in set db.
     Groups & r.name = "Administrators");
  SetPermission(d.owner, d, <read>, true);
    SetPermission(d.owner, d, <edit>, true);
    SetPermission(d.owner, d, <delete>, true);
    SetPermission(admins, d, <read>, true);
      SetPermission(admins, d, <edit>, true);
      SetPermission(admins, d, <delete>, true);
  );

  public
  GetUserPermissions: Document ==> set of DB`
     UserPermissionRow
  GetUserPermissions(doc) == (
    return db.SelectUserPermissionByDocument(doc.id);
  );

  public
  GetGroupPermissions: Document ==> set of DB`
     GroupPermissionRow
  GetGroupPermissions(doc) == (
    return db.SelectGroupPermissionByDocument(doc.id)
       ;
  );

end ACL
```

Listing B.19: php.vdmpp

```
-- Top-level class to represent native PHP
   functionality.
-- This includes standard data types, functions, and
   superglobals, etc.
-- This serves to represent the runtime environment of
   the system.
class PHP


 types
   -- Standard string definition
   public String = seq of char;
   -- The UNIX timestamp is the standard PHP date/time
       value
   public Timestamp = int;
   -- Here Object is a general purpose base type.
   public Object = BaseObject | Array | String | int |
       real | bool | token;
   -- Scalar primitives
   public Scalar = String | int | real | bool | token;

   -- PHP allows arrays to be indexed by string or
      integer
```

```
    public ArrayIndex = int | String;
    public Array = map ArrayIndex to Object;

    public ResponseCode = <ok> | <redirect>;

    public Response ::
      header: ResponseCode
      body   : String;

  instance variables
    -- Server superglobals
    public static POST: Array:= {|->};
    public static GET: Array:= {|->};
    public static COOKIE: Array:= {|->};
    public static SESSION: Array:= {|->};
    public static SERVER: Array:= {|->};
    public static ENV: Array:= {|->};

    -- Client side cookies, used for setting
    public static ClientCookies: set of Cookie:= {};

  functions
```

```
public
toArray: set of BaseObject -> Array
toArray(s) == is not yet specified;


public
toObj: set of BaseObject -> BaseObject
toObj(s) == is not yet specified;


-- Determine class name of a particular object.
-- This maps to PHP's get_class() function
public
GetClassName: BaseObject -> String
GetClassName(obj) == is not yet specified;


-- Create an instance of a class based on the class
     name.
-- This is equivalent to 'new $classname()' in PHP
public
ObjectFromString(classname: String) ret: BaseObject
post GetClassName(ret) = classname;


-- Determines if a class with a particular name is
    defined.
```

```
-- Maps to PHP class_exists() function.
public
ClassExists: String -> bool
ClassExists(classname) == is not yet specified;

-- Function for PHP's implicit casting of strings
   to booleans.
public
ToBool(s: String) ret: bool
post ret = not s in set {"", "0"};

public
MethodExists: BaseObject * String -> bool
MethodExists(obj, methodname) == is not yet
   specified;

-- Takes an object and a string and calls the
   method
-- on the object with that name.
-- Equivalent to '$obj->$methodname()' in PHP.
public
CallMethod: BaseObject * String -> Object
CallMethod(obj, methodname) == is not yet specified
```

```
    pre MethodExists(obj, methodname);

  operations

    -- Interpreter initialization
    public
    PHP()
    ext wr SERVER
    post SERVER <> {|->} ;

    public static
    time() r: Timestamp
    post r > 0 ;



    public
    Init: () ==> Response
    Init() == (
      dcl fc: FrontController;
      fc:= new FrontController();
      return fc.Dispatch(SERVER("REQUEST_URI"));
    );
```

```
end PHP


class BaseObject
 -- Empty class - used for inheritance purposes.
end BaseObject
```

Listing B.20: view.vdmpp

```
class View is subclass of PHP


 instance variables


   body: String:= "";
   messages: String:= "";


 operations


   -- Takes a template name and returns its content.
   private
   GetTemplate(name: String) text: String
   pre  name <> ""
   post text <> "";
```

```
-- Load a view template and append it to the
   response body
public
Load: String ==> ()
Load(viewname) == (
  body:= body ^ messages ^ GetTemplate(viewname)
);

public
Load: String * PHP`Object ==> ()
Load(viewname, data) == is not yet specified;

public
SetMessage: String ==> ()
SetMessage(msg) == (
  messages:= messages ^ msg;
);

-- Return a response for the loaded views.
public
Render: () ==> Response
Render() == (
  return mk_Response(<ok>, body);
```

```
    );


end View
```

C System Code

C.1 Controller Classes

Listing C.1: controllers/DocumentController.php

```
<?php
class DocumentController extends ActionController {

  public function __construct($viewPath) {
    parent::__construct($viewPath);
  }

  public function getlist() {
    $acl = new ACL();
    $docs = Document::GetAll();
    $allowed_docs = array();
    foreach ($docs as $doc) {
      if ($acl->HasPermission($this->current_user, $doc
          , ACL::READ)) {
        $allowed_docs[] = $doc;
      }
    }
    $this->view->load('document_list', array('
        allowed_docs'=>$allowed_docs));
    $this->view->render();
  }

  public function view($id) {
    $doc = Document::GetById($id);
    $acl = new ACL();
    if ($p = $acl->HasPermission($this->current_user,
        $doc, ACL::READ)) {
      $this->view->load('document_view', array('doc'=>
          $doc));
      $this->view->render();
    } else {
      $this->redirect('/error/denied');
    }
  }
```

```
public function edit($id) {
  $doc = Document::GetById($id);
  $acl = new ACL();

  if (! $acl->HasPermission($this->current_user, $doc
     , ACL::READ)) {
    $this->redirect('/error/denied');
  }

  $required_fields = array('title', 'body');
  if ($this->arrayContains($required_fields, $_POST))
      {
    $doc->title = $_POST['title'];
    $doc->content = $_POST['body'];
    $doc->Update();
    $this->redirect('/document/view/'.$doc->id);
  } else {
    $data = array('doc'=>$doc, 'action'=>'edit');
    $this->view->load('document_edit', $data);
    $this->view->render();
  }
}
```

```
public function delete($id) {
  $doc = Document::GetById($id);
  $acl = new ACL();

  if (! isset($_POST['confirm'])) {
    $this->view->load('document_delete', array('doc'
      => $doc, 'title'=> 'Delete document'));
    $this->view->render();
  } elseif ($acl->HasPermission($this->current_user,
     $doc, ACL::DELETE)) {
    $doc->Delete();
    $this->redirect('/document/list/');
  } else {
    $this->redirect('/error/denied/');
  }
}

public function create() {
  $doc = new Document();
  $acl = new ACL();

  // Do we need a permission check here?
  $required_fields = array('title', 'body');
```

```
    if ($this->arrayContains($required_fields, $_POST))
        {
      $doc->title = $_POST['title'];
      $doc->content = $_POST['body'];
      $doc->owner = $this->current_user;
      $doc->insert();
      $acl->SetDefaultPermissions($doc);
      $this->redirect('/document/view/'.$doc->id);
    } else {
      $data = array('doc'=>$doc, 'action'=>'create');
      $this->view->load('document_edit', $data);
      $this->view->render();
    }
  }

  public function changeOwner($id) {
    $doc = Document::GetById($id);

    if (! $doc) {
      $this->redirect('/error/invalid/');
    }

    if (! $this->current_user->isAdministrator()) {
```

```
    $this->redirect('/error/denied/');
  }

  if ($this->arrayContains(array('username'), $_POST)
     ) {
    $usr = User::GetByName($_POST['username']);
    if (! $usr) {
      $this->redirect('/error/invalid/');
    } else {
      $doc->owner = $usr;
      $ret = $doc->Update();
      if ($ret) $this->redirect('/document/view/'.$id
          );
      else $this->view->setMessage("Error␣setting␣
          document␣owner", "error");
    }
  } else {
    $users = User::GetAll();
    $data = array(
      'users'=> $users,
      'doc'=> $doc,
      'title'=> 'Change␣Document␣Owner'
    );
```

```
      $this->view->load('document_changeowner', $data);
      $this->view->render();
    }
  }

  public function changePermissions($id) {
    $db = new DB();
    $doc = Document::GetById($id);
    if (! $doc) {
      $this->redirect('/error/invalid/');
    }
    $user_perms = $db->SelectUserPermissionByDocument(
       $id);
    $group_perms = $db->SelectGroupPermissionByDocument
       ($id);
    $data = array('user_perms'=>  $this->
       groupPermsByField($user_perms, 'username'),
            'group_perms'=> $this->groupPermsByField(
              $group_perms, 'group_name'),
            'doc'=> $doc,
            'title'=> 'Change document permissions');
    $this->view->load('document_changepermissions',
       $data);
    $this->view->render();
  }

  protected function groupPermsByField($rowset,
     $fieldname) {
    $ret = array();
    foreach ($rowset as $row) {
      if (! isset($ret[$row[$fieldname]])) {
        $ret[$row[$fieldname]] = array();
      }
      $ret[$row[$fieldname]][$row['permission']] = $row
         ['granted'];
    }
    return $ret;
  }

  public function setUserPermission($id) {
    $doc = Document::GetById($id);
    $acl = new ACL();

    if (! $doc) {
      $this->redirect('/error/invalid/');
    }
```

```
if (! $this->current_user->isAdministrator() &&
    $this->current_user->username != $doc->owner->
      username) {
  $this->redirect('/error/denied/');
}

if ($this->arrayContains(array('user', 'perm', '
   status'), $_POST)) {
  $usr = User::GetByName($_POST['user']);
  $perm = ACL::toPerm($_POST['perm']);
  $status = $_POST['status'];
  if ($usr && $perm) {
    $acl->SetPermission($usr, $doc, $perm, $status)
      ;
    $this->view->setMessage("Permissions␣updated",
      'success');
    $this->redirect('/document/view/'.$id);
  } else {
    $this->view->setMessage('Error␣-␣User␣or␣
      permission␣not␣valid', 'error');
    $this->redirect('/document/changePermissions/'.
      $id);
      }
    } else {
      $this->view->setMessage('Error␣-␣Missing␣
          parameters', 'error');
      $this->redirect('/document/changePermissions/'.
          $id);
    }
  }

  public function setGroupPermission($id) {
    $doc = Document::GetById($id);
    $acl = new ACL();

    if (! $doc) {
      $this->redirect('/error/invalid/');
    }

    if (! $this->current_user->isAdministrator() &&
        $this->current_user->username != $doc->owner->
          username) {
      $this->redirect('/error/denied/');
    }
    if ($this->arrayContains(array('group', 'perm', '
        status'), $_POST)) {
      $group = Group::GetByName('user');
      $perm = ACL::toPerm($_POST['perm']);
      $status = $_POST['status'] == 'granted';
      if ($group && $perm) {
        $acl->SetPermission($group, $doc, $perm,
            $status);
        $this->redirect('/document/view/'.$id);
      } else {
        $this->view->load('set_group_permission');
        $this->view->render();
      }
    } else {
      $this->view->load('set_group_permission');
      $this->view->render();
    }
  }
}
```

Listing C.2: controllers/ErrorController.php

```
<?php
class ErrorController extends ActionController {
     public function __construct($viewPath) {
         parent::__construct($viewPath);
     }

     public function denied() {
         $this->view->load('error_denied');
         $this->view->render();
     }

     public function invalid() {
         $this->view->load('error_invalid');
         $this->view->render();
     }
}
```

Listing C.3: controllers/GroupController.php

```
<?php
class GroupController extends ActionController {
    public function __construct($viewPath) {
    parent::__construct($viewPath);
  }

    public function getlist() {
        if (! $this->current_user->isAdministrator()) {
            $this->redirect('/error/denied/');
        }

        $groups = Group::GetAll();
        $this->view->load('group_list', array('groups'
           => $groups));
        $this->view->render();
    }

  public function show($name) {
    if (! $this->current_user->isAdministrator()) {
            $this->redirect('/error/denied/');
        }

        $group = Group::GetByName($name);
        $this->view->load('group_show', array('group'=>
            $group));
        $this->view->render();
  }

    public function create() {
        $req_fields = array('group_name');
```

```
if ( ! $this->current_user->isAdministrator() ) {
  $this->redirect('/error/denied/');
} elseif (!$this->arrayContains($req_fields, $_POST
   )) {
  $this->view->load("group_create");
        $this->view->render();
} else {
  $grp = new Group();
  $grp->name = $_POST['group_name'];
  $grp->description = isset($_POST['description'])
     ? $_POST['description'] : '';
  $grp->insert();
        $this->redirect('/group/getlist/');
}
}

public function addUser($name) {
    if (! $this->current_user->isAdministrator()) {
        $this->redirect('/error/denied/');
    }

    $g = Group::GetByName($name);
    if (! $g) {
        $this->redirect('/error/invalid/');
    }

    $req_fields = array('username');

    if (! $this->arrayContains($req_fields, $_POST)
        ) {
  $this->view->setMessage("Missing␣username");
} else {
        $u = User::GetByName($_POST['username']);
        if ($u) {
            $ret = $g->Add($u);
    $msg = $ret ? "Added␣user␣".$u->username."␣to␣"
       .$g->name : "Failed␣to␣add␣user";
    $this->view->setMessage($msg, $ret ? "success"
       : "error");
        } else {
    $this->view->setMessage("User␣'".$_POST['
       username']."'␣does␣not␣exist", "error");
  }
    }
$this->redirect('/group/show/'.$g->name);
$this->view->render();
```

```
}

public function removeUser($name) {

    if (! $this->current_user->isAdministrator()) {
        $this->redirect('/error/denied/');
    }

    $g = Group::GetByName($name);
    if (! $g) {
        $this->redirect('/error/invalid/');
    }

    $req_fields = array('username');

    if ($this->arrayContains($req_fields, $_POST))
       {
        $u = User::GetByName($_POST['username']);
        if ($u) {
            $g->Remove($u);
            $this->redirect('/group/list/');
        } else {
            $this->redirect('/error/invalid/');
```

```
                }
            } else {
                $this->view->load("group_remove_user");
                $this->view->render();
            }
        }
}
```

Listing C.4: controllers/IndexController.php

```
<?php
class IndexController extends ActionController {
    public function __construct($viewPath) {
    parent::__construct($viewPath);
  }

    public function getlist() {
        $this->view->load('index');
        $this->view->render();
    }
}
```

Listing C.5: controllers/UserController.php

```
<?php
class UserController extends ActionController {
```

```
public function __construct($viewPath) {
  parent::__construct($viewPath);
}

public function create() {
  $req_fields = array('username', 'password', '
     confirm');
  if ( empty($this->current_user) ||
     ! $this->current_user->isAdministrator() ) {
    $this->redirect('/error/denied/');
  } elseif ($this->arrayContains($req_fields, $_POST)
     ) {
    $usr = new User();
    $usr->username = $_POST['username'];
    $usr->setPassword($_POST['password']);
    if ($usr->checkPassword($_POST['confirm'])) {
      $usr->insert();
      $data = array('name'=>$usr->username);
      $this->view->setMessage('Successfully␣created␣
         user␣'.$usr->username, 'success');
    } else {
      $this->view->setMessage('Error␣creating␣user␣'.
         $usr->username, 'error');
    }
    $this->view->load("user_create", $data);
  } else {
    if (! empty($_POST)) {
      $this->view->setMessage('Error␣creating␣user␣-␣
         missing␣fields', 'error');
    }
    $this->view->load("user_create");
  }
  $this->view->render();
}

public function changePassword($username) {
  if ( empty($this->current_user) ||
     ! $this->current_user->isAdministrator() ) {
    $this->redirect('/error/denied/');
  }

  $usr = User::GetByName($username);
  if (! $usr) {
```

```
  $this->view->setMessage('Specified user does not 
     exist', 'error');
  $this->redirect('/user/list/');
}

$keys_set = $this->arrayContains(array('password',
    'confirm'), $_POST);

if ( $keys_set && $_POST['password'] == $_POST['
    confirm'] ) {
  $usr->SetPassword($_POST['password']);
  $usr->Save();
  $this->view->setMessage("Password changed", "
     success");
  $this->redirect('/user/list/');
} elseif ( $keys_set && $_POST['password'] !=
    $_POST['confirm']) {
  $this->view->setMessage("Passwords do not match")
     ;
}
$this->view->load('user_password', array('title'=>
    'Change password'));
$this->view->render();
```

```
}

public function login() {
  if ($this->current_user) {
    $this->redirect('/document/getlist/');
  }

  if ($this->arrayContains(array('username', '
     password'), $_POST)) {
    $logged_in = Auth::login($_POST['username'],
       $_POST['password']);
    if ($logged_in) {
      $this->current_user = $_SESSION[User::CURR_USER
          ];
      $this->redirect('/document/getlist/');
    } else {
      $this->view->setMessage("Invalid username or
          password.");
    }
  }
  $this->view->load("user_login", array('title'=> '
     User login'));
  $this->view->render();
  }

  public function logout() {
    unset($this->current_user);
    unset($_SESSION[Auth::CURR_USER]);
    $this->redirect('/user/login/');
  }

  public function getlist() {
    if ( empty($this->current_user) ||
        ! $this->current_user->isAdministrator() ) {
      $this->redirect('/error/denied/');
    }

    $users = User::GetAll();
    $this->view->load('user_list', array('title'=> '
       User␣list', 'users'=> $users));
    $this->view->render();
  }

}
```

C.2 Library Classes

Listing C.6: lib/ACL.php

```
<?php
class ACL {

  const READ = 'read';
  const WRITE = 'edit';
  const DELETE = 'delete';
  const CREATE = 'create';

  private $db = null;

  public function __construct() {
    $this->db = new DB();
  }

  public function HasPermission($user, $document, $perm
     ) {
    if (! $user || ! $document) return false;
    $user_has_perm = $this->db->SelectUserPermission(
       $user->username, $document->id, $perm);
    if ($user_has_perm !== null) {
      return $user_has_perm == 1;
    } else {
      $group_has_perm = $this->db->
         SelectGroupPermission($user->username,
         $document->id, $perm);
      return $group_has_perm == 1;
    }
  }

  public function SetPermission($user, $document, $perm
     , $granted) {
    $data = array('document_id'=> $document->id,
           'permission'=> $perm,
           'granted'=> $granted);
    if (is_a($user, 'User')) {
      $data['username'] = $user->username;
      $has_perm = $this->db->SelectUserPermission($user
         ->username, $document->id, $perm);
      if ($has_perm !== null) {
        $this->db->UpdateUserPermission($user->username
           , $document->id, $perm, $granted);
      } else {
        $this->db->InsertUserPermission($data);
      }
    } else {
      $data['group_name'] = $user->name;
      $has_perm = $this->db->SelectGroupPermission(
         $user->name, $document->id, $perm);
      if ($has_perm !== null) {
        $this->db->UpdateGroupPermission($user->name,
            $document->id, $perm, $granted);
      } else {
        $this->db->InsertGroupPermission($data);
      }
    }
  }

  public function SetDefaultPermissions($doc) {
    $admins = Group::GetByName('Administrators');
    $this->SetPermission($doc->owner, $doc, ACL::READ,
       true);
    $this->SetPermission($doc->owner, $doc, ACL::WRITE,
        true);
    $this->SetPermission($doc->owner, $doc, ACL::DELETE
       , true);
    $this->SetPermission($admins, $doc, ACL::READ, true
        );
    $this->SetPermission($admins, $doc, ACL::WRITE,
        true);
    $this->SetPermission($admins, $doc, ACL::DELETE,
        true);
  }

  public static function toPerm($val) {
    if (in_array($val, self::getPermissionList())) {
      return $val;
    } else {
      return null;
    }
  }

  public static function getPermissionList() {
    return array(ACL::CREATE, ACL::DELETE, ACL::READ,
        ACL::WRITE);
  }

}
```

Listing C.7: lib/ActionController.php

```
<?php
class ActionController {

  protected $current_user;
  protected $db;
  protected $view;

    public $baseUrl = '';
  public $rootUrl = '/';
    public $useQueryString = false;

  protected function __construct($viewPath) {
    $this->db = new DB();
        $this->view = new View($viewPath, $this);
    if (isset($_SESSION[Auth::CURR_USER])) {
      $this->current_user = User::GetByName($_SESSION[
          Auth::CURR_USER]);
    } else {
      $this->current_user = null;
    }
    $this->view->current_user = $this->current_user;
  }
```

```
/**
 * Utility method for calculating array subset
    based on keys.
 *
 * @param array $keys   Array of key values for
    which to check
 * @param array $array  The array in which to check
     for them
 * @return bool True if the array contains all the
    requested keys,
 *              false otherwise.
 */
protected function arrayContains(array $keys, array
    $array) {
    foreach ($keys as $key) {
        if (! isset($array[$key])) return false;
    }
    return true;
}

public function route($ctl, $act, $data=null) {
    if ($this->useQueryString) {
            $ret = $this->baseUrl."?c=$ctl&a=$act";
            if ($data !== null) $ret .= "&d=$data";
        } else {
            $route = array($ctl, $act);
            if ($data !== null) $route[] = $data;
            $ret = $this->baseUrl . '/' . implode('/',
               $route);
        }
        return $ret;
    }

  public function link($res) {
    return $this->rootUrl . $res;
  }

  public function redirect($url) {
    if ($this->baseUrl) {
      $url = $this->baseUrl.$url;
    }
    header('Location:␣'.$url);
    exit;
  }
}
```

```
?>
```

Listing C.8: lib/ActiveRecord.php

```
<?php
abstract class ActiveRecord {

    public $id;

    public function Update() {

    }

    public function Insert() {

    }

    public function Save() {
        if ($this->id) {
            $this->update();
        } else {
            $this->insert();
        }
    }
}
```

Listing C.9: lib/Auth.php

```
<?php
class Auth {
    const CURR_USER = 'curr_user';

    public static function login($username, $password)
       {
        $logged_in = false;
        $user = User::GetByName($username);
        if ($user && $user->checkPassword($password)) {
            $_SESSION[User::CURR_USER] = $user->
               username;
            return true;
        } else {
            return false;
        }
    }

    public static function logout() {
        unset($_SESSION[User::CURR_USER]);
    }
}
```

Listing C.10: lib/DB.php

```
<?php
class DB {

  protected $dbconn;

  public function __construct() {
    $this->dbconn = new PDO($GLOBALS['db']['dsn'],
                $GLOBALS['db']['username'],
                $GLOBALS['db']['password']);
    $this->dbconn->setAttribute(PDO::ATTR_ERRMODE, PDO
       ::ERRMODE_EXCEPTION);
  }

  /**
   * Convenience function to execute a select query and
       return the result set.
   * @param string  $sql  The SQL query to execute.
   * @param array   $data The parameters to execute the
       prepared query.
```

```
 */
protected function runSelect($sql, $data=array()) {
  //try {
    $query = $this->dbconn->prepare($sql);
    $query->execute($data);
    return $query->fetchAll();
  //} catch (PDOException $e) {
  //  error_log('Error on line '.__LINE__.' of '.
      __FILE__.': '.
  //          $e->getMessage());
  //  return null;
  //}
}

/**
 * Convenience function to fetch a single row.
 * @param string $sql The SQL query to execute.
 * @param array $data The data to execute the
    prepared query.
 */
protected function runSingleSelect($sql, $data) {
  //try {
    $query = $this->dbconn->prepare($sql);
```

```
      $query->execute($data);
      $ret = $query->fetch();
      return $ret;
    //} catch (PDOException $e) {
    //  error_log('Error on line '.__LINE__.' of '.
        __FILE__.': '.
    //          $e->getMessage());
    //  return null;
    //}
  }

  /**
   * Convenience function to fetch a scalar value.
   * @param string $sql The SQL query to execute.
   * @param mixed $data The data in the single result
       column, or null for an empty result set.
   */
  protected function runScalarSelect($sql, $data) {
    //try {
      $query = $this->dbconn->prepare($sql);
      $query->execute($data);
      $ret = $query->fetchColumn();
      return $ret === false ? null : $ret;
    //} catch (PDOException $e) {
    //   error_log('Error on line '.__LINE__.' of '.
        __FILE__.': '.
    //           $e->getMessage());
    //   return null;
    //}
  }

  protected function runQuery($sql, $data) {
    //try {
      $query = $this->dbconn->prepare($sql);
  //var_dump($query); exit;
      return $query->execute($data);
    //} catch (PDOException $e) {
    //   error_log('Error on line '.__LINE__.' of '.
        __FILE__.': '.
    //           $e->getMessage());
    //   return false;
    //}
  }

  public function SelectUser($username) {
    $sql = 'SELECT * FROM Users WHERE username = ?';
    return $this->runSingleSelect($sql, array($username
        ));
  }

  public function UpdateUser($key, $data) {
    $sql = "UPDATE␣Users␣SET␣username␣=␣?,␣password␣=␣?
        ␣WHERE␣username␣=␣?";
    $params = array($data['username'], $data['password'
        ], $key);
    return $this->runQuery($sql, $params);
  }

  public  function InsertUser($data) {
    $sql = "INSERT␣INTO␣Users␣(username,␣password)␣
        VALUES␣(?,␣?)";
    $params = array($data['username'], $data['password'
        ]);
    return $this->runQuery($sql, $params);
  }

  public function DeleteUser($key) {
    $sql = "DELETE␣FROM␣Users␣WHERE␣username␣=␣?";
    $params = array($key);
    return $this->runQuery($sql, $params);
  }

  public function SelectAllDocuments() {
    $sql = "SELECT * FROM Documents";
    return $this->runSelect($sql);
  }

  public function SelectDocument($id) {
    $sql = 'SELECT * FROM Documents WHERE id = ?';
    $params = array($id);
    return $this->runSingleSelect($sql, $params);
  }

  public function SelectDocumentByOwner($owner) {
    $sql = 'SELECT * FROM Documents WHERE owner = ?';
    $params = array($id);
    return $this->runSelect($sql, $params);
  }

  public function SelectDocumentsByGroup($group) {
    $sql = 'SELECT * FROM Documents WHERE owner IN '.
```

```
                '(SELECT␣user_name␣FROM␣GroupMembers␣WHERE␣
                    group_name␣=␣?)';
    $params = array($group);
    return $this->runSelect($sql, $params);
  }

  public function SelectReadableDocuments($user) {
    $sql = 'SELECT␣*␣FROM␣Documents␣AS␣d␣WHERE␣EXISTS␣'
            .
                '(SELECT␣*␣FROM␣UserPermissions␣AS␣up␣WHERE␣
                    '.
              '␣up.document_id␣=␣d.id␣AND␣up.username␣=␣?␣
                  AND␣'.
              "␣up.pemission␣=␣'read'␣and␣up.granted␣=␣1)␣OR
                  ␣" .
              '(SELECT␣*␣FROM␣GroupPermissions␣AS␣gp␣WHERE␣'
                  .
              '␣gp.group_name␣IN␣(SELECT␣group_name␣FROM␣
                  GroupMembers␣WHERE␣user␣=␣?)␣AND␣'.
              '␣gp.document_id␣=␣d.id␣AND␣gp.group_name␣=␣?␣
                  AND␣'.
              "␣gp.pemission␣=␣'read'␣and␣gp.granted␣=␣1)";
    $params = array($user, $user, $user);
    return $this->runSelect($sql, $params);
  }

  public function UpdateDocument($key, $data) {
    $sql = 'UPDATE Documents SET id = ?, title = ?, 
       content = ?, owner = ? WHERE id = ?';
    $params = array($data['id'], $data['title'], $data[
       'content'], $data['owner'], $data['id']);
    return $this->runQuery($sql, $params);
  }

  public function InsertDocument($data) {
    $sql = 'INSERT INTO Documents (title, content, 
       owner) VALUES (?, ?, ?)';
    $params = array($data['title'], $data['content'],
       $data['owner']);
    # Need to add note to paper on handling auto-id.
    $ret = $this->runQuery($sql, $params);
    if ($ret) return $this->dbconn->lastInsertId();
    else return false;
  }

  public function DeleteDocument($key) {
    $sql = 'DELETE FROM Documents WHERE id = ?';
    $params = array($key);
    return $this->runQuery($sql, $params);
  }

  public function SelectAllGroups() {
    $sql = 'SELECT * FROM Groups';
    return $this->runSelect($sql);
  }

  public function SelectAllUsers() {
    $sql = 'SELECT * FROM Users';
    return $this->runSelect($sql);
  }

  public function SelectGroup($name) {
    $sql = 'SELECT * FROM Groups WHERE name = ?';
    $params = array($name);
    return $this->runSingleSelect($sql, $params);
  }

  public function InsertGroup($data) {
    $sql = 'INSERT␣INTO␣Groups␣(name,␣description)␣
        VALUES␣(?,␣?)';
    $params = array($data['name'], $data['description'
        ]);
    return $this->runQuery($sql, $params);
}

public function UpdateGroup($key, $data) {
    $sql = 'UPDATE␣Group␣SET␣name␣=␣?,␣description␣=␣?␣
        WHERE␣name␣=␣?';
    $params = array($data['name'], $data['description'
        ]);
    return $this->runQuery($sql, $params);
}

public function DeleteGroup($key) {
    $sql = 'DELETE␣FROM␣Groups␣WHERE␣name␣=␣?';
    $params = array($key);
    return $this->runQuery($sql, $params);
}

public function SelectAdminMembershipByUser($user) {
    $sql = "SELECT␣CASE␣WHEN␣EXISTS␣".
```

```
              "(SELECT * FROM GroupMembers WHERE user_name
                 = ? AND group_name = 'Administrators')
                 THEN 1 ".
           "ELSE 0 END";
    $params = array($user);
    return $this->runScalarSelect($sql, $params);
  }

  public function SelectGroupMembership($user, $group)
     {
    $sql = "SElECT CASE WHEN "  .
              "EXISTS (SELECT * FROM GroupMembers WHERE " .
           "user_name = ? AND group_name = ?) THEN 1"  .
           "ELSE 0 END";
    $params = array($user, $group);
    return $this->runScalarSelect($sql, $params);
  }

  public function SelectGroupMembersByUser($user) {
    $sql = 'SELECT *
            FROM GroupMembers AS gm JOIN Groups AS g ON
     gm.group_name = g.group_name
        WHERE user_name = ?';
    $params = array($user);
    return $this->runSelect($sql, $params);
  }

  public function SelectGroupMembersByGroup($group) {
    $sql = 'SELECT␣*
␣␣␣␣␣␣␣␣␣␣␣␣FROM␣GroupMembers␣AS␣gm␣JOIN␣Users␣AS␣u␣ON␣
   gm.user_name␣=␣u.username
␣␣␣␣␣␣␣␣WHERE␣group_name␣=␣?';
    $params = array($group);
    return $this->runSelect($sql, $params);
  }

  public function InsertGroupMember($data) {
    $sql = 'INSERT␣INTO␣GroupMembers␣(group_name,␣
       user_name)␣VALUES␣(?,␣?)';
    $params = array($data['group_name'], $data['
       user_name']);
    return $this->runQuery($sql, $params);
  }

  public function DeleteGroupMember($group, $user) {
    $sql = 'DELETE FROM Group WHERE group_name = ? AND 
       user_name = ?';
    $params = array($data['group_name'], $data['
       user_name']);
    return $this->runQuery($sql, $params);
  }

  public function SelectUserPermissionByUser($user) {
    $sql = 'SELECT * FROM UserPermissions WHERE 
       username = ?';
    $params = array($user);
    return $this->runSelect($sql, $params);
  }

  public function SelectUserPermissionByDocument($id) {
    $sql = 'SELECT * FROM UserPermissions WHERE 
       document_id = ?';
    $params = array($id);
    return $this->runSelect($sql, $params);
  }

  public function SelectUserPermission($user, $docid,
     $perm) {
```

```
    $sql = 'SELECT granted FROM UserPermissions WHERE '
         .
            'username = ? AND document_id = ? AND
               permission = ?';
    $params = array($user, $docid, $perm);
    return $this->runScalarSelect($sql, $params);
  }

  public function InsertUserPermission($data) {
    $sql = 'INSERT INTO UserPermissions '.
            '(document_id, username, permission, granted
               ) VALUES (?, ?, ?, ?)';
    $params = array($data['document_id'],
            $data['username'],
            $data['permission'],
            $data['granted']);
    return $this->runQuery($sql, $params);
  }

  public function DeleteUserPermission($docid, $user) {
    $sql = 'DELETE FROM UserPermissions '.
            'WHERE document_id = ? AND username = ?';
    $params = array($docid, $user);
    return $this->runQuery($sql, $params);
  }

  public function UpdateUserPermission($user, $docid,
     $perm, $granted) {
    $sql = 'UPDATE␣UserPermissions␣SET␣granted␣=␣?␣
       WHERE␣' .
             'username␣=␣?␣AND␣document_id␣=␣?␣AND␣
                permission␣=␣?';
    $params = array($granted, $user, $docid, $perm);
    return $this->runQuery($sql, $params);
  }

  public function SelectGroupPermissionByUser($group) {
    $sql = 'SELECT␣*␣FROM␣GroupPermissions␣WHERE␣
       group_name␣=␣?';
    $params = array($group);
    return $this->runSelect($sql, $params);
  }

  public function SelectGroupPermissionByDocument($id)
     {
```

```
    $sql = 'SELECT * FROM GroupPermissions WHERE
        document_id = ?';
    $params = array($id);
    return $this->runSelect($sql, $params);
  }

  public function SelectGroupPermission($group, $docid,
      $perm) {
    $sql = 'SELECT * FROM GroupPermissions WHERE ' .
           'group_name = ? AND document_id = ? AND
               permission = ?';
    $params = array($group, $docid, $perm);
    return $this->runScalarSelect($sql, $params);
  }

  public function InsertGroupPermission($data) {
    $sql = 'INSERT INTO GroupPermissions '.
           '(document_id, group_name, permission,
               granted) VALUES (?, ?, ?, ?)';
    $params = array($data['document_id'],
            $data['group_name'],
            $data['permission'],
            $data['granted']);
    return $this->runQuery($sql, $params);
  }

  public function DeleteGroupPermission($docid, $group)
      {
    $sql = 'DELETE FROM GroupPermissions '.
            'WHERE document_id = ? AND username = ?';
    $params = array($docid, $group);
    return $this->runQuery($sql, $params);
  }

  public function UpdateGroupPermission($group, $docid,
      $perm, $granted) {
    $sql = 'UPDATE GroupPermissions SET granted = ? 
        WHERE ' .
            'group_name = ? AND document_id = ? AND 
                permission = ?';
    $params = array($granted, $group, $docid, $perm);
    return $this->runQuery($sql, $params);
  }
}
```

Listing C.11: lib/FrontController.php

```
<?php
class FrontController {

  public $useQueryString = false;
  public $baseUrl = '';
  public $rootUrl = '/';
  public $viewPath = '';

  private function getControllerStatic($ctlname) {
    switch ($ctlname) {
      case 'user':
        return new UserController($this->viewPath);
      case 'document':
        return new DoucmentController($this->viewPath);
      case 'group':
        return new GroupController($this->viewPath);
      case 'index':
      case '':
        return new IndexController($this->viewPath);
      default:
        return new ErrorController($this->viewPath);
    }
```

```
}

private function getControllerDynamic($ctlname) {
  $controller_name = ucwords($ctlname ? $ctlname : '
     index') . "Controller";
  if (class_exists($controller_name, true)) {
    return new $controller_name($this->viewPath);
  } else {
    return new ErrorContorller();
  }
}

private function getController($ctlname) {
  $ret = $this->getControllerDynamic($ctlname);
  $this->configureController($ret);
  return $ret;
}

private function CallAction($ctl, $actname, $data) {
  if ($actname == 'list' && method_exists($ctl, '
     getlist')) {
    $actname = 'getlist';
  }
```

```
    call_user_func_array(array($ctl, $actname), $data);
  }

  private function splitUrl($url) {
    if (strpos($url, $this->baseUrl) === 0) {
      $url = substr($url, strlen($this->baseUrl));
    }
    $url = trim($url, '/');

    if ($this->useQueryString) {
      return array($_GET['c'], $_GET['a'], $_GET['d']);
    } else {
      $data = trim($url, '/');
      return explode('/', $url);
    }
  }

  protected function configureController($ctl) {
    $ctl->baseUrl = $this->baseUrl;
    $ctl->rootUrl = $this->rootUrl;
    $ctl->useQueryString = $this->useQueryString;
  }
  public function dispatch($url) {
    $urldata = $this->splitUrl($url);
    $ctl = $this->getController($urldata[0]);
    if ( ! is_a($ctl, 'UserController') &&
       ! isset($_SESSION[Auth::CURR_USER]) ) {
      $ctl->redirect('/user/login/');
    } else {
      $this->CallAction($ctl, $urldata[1], array_slice(
         $urldata, 2));
    }
  }
}
```

Listing C.12: lib/View.php

```
<?php
class View {

  public $viewPath = "";
  public $current_user = null;
  public $controller = null;

  private $body = '';
  private $filename = '';
private $data = array();

public function __construct($path='', $ctl=null) {
  $this->viewPath = $path;
  $this->controller = $ctl;
}

/**
 * Load a template and set data.
 * @param string $viewname  The name of the view file
     to load, minus extension
 * @param array  $data     An associative array of
    view data
 */
public function load($viewname, $data = null) {
  $this->filename = $viewname;
  $this->data = $data;
}

public function route($ctl, $act, $dat='') {
  return $this->controller->route($ctl, $act, $dat);
}
```

```
public function link($res) {
  return $this->controller->link($res);
}

public function setMessage($message, $type="info") {
  if (! isset($_SESSION['messages'])) $_SESSION['
     messages'] = array();
  $_SESSION['messages'][] = array($message, $type);
}

protected function getMessages() {
  $ret = @$_SESSION['messages'];
  unset($_SESSION['messages']);
  return $ret;
}

/**
 * Renders the view to the client.
 *
 * The header.php and footer.php view files are
    wrapped around the loaded
 * view automatically.
 */
```

```
  public function render() {
    if ($this->data) {
      extract($this->data);
    }
    if (! isset($title)) {
      $title = 'DocMan␣Document␣Manager';
    }
    $session_messages = $this->getMessages();
    include $this->viewPath.DIRECTORY_SEPARATOR.'header
        .php';
    include $this->viewPath.DIRECTORY_SEPARATOR.$this->
        filename.'.php';
    include $this->viewPath.DIRECTORY_SEPARATOR.'footer
        .php';
  }

}
```

C.3 Model Classes

Listing C.13: models/Document.php

```
<?php
class Document extends ActiveRecord {
```

```
public $id = null;
public $title = '';
public $content = '';
public $owner = null;

protected $db;
public static $dbconn;

public function __construct($row=null) {
  $this->db = self::getDb();
  if ($row) {
    $this->id = $row['id'];
    $this->title = $row['title'];
    $this->content = $row['content'];
    $this->owner = User::GetByName($row['owner']);
  }
}

protected static function getDb() {
  if (! self::$dbconn) {
    self::$dbconn = new DB();
  }
```

```
    return self::$dbconn;
  }

  static public function GetAll() {
    $db = self::getDb();
    $rows = $db->SelectAllDocuments();
    $ret = array();
    foreach ($rows as $row) {
      $ret[] = new Document($row);
    }
    return $ret;
  }

  static public function GetById($id) {
    $db = self::getDb();
    $row = $db->SelectDocument($id);
    return new Document($row);
  }

  public function Insert() {
    $data = array('title'=>$this->title,
                  'content'=>$this->content,
                  'owner'=>$this->owner->username);
    $last_id = $this->db->InsertDocument($data);
    $this->id = $last_id;
  }

  public function Update() {
    $data = array('id'=>$this->id,
            'title'=>$this->title,
            'content'=>$this->content,
            'owner'=>$this->owner->username);
    return $this->db->UpdateDocument($this->id, $data);
  }

  public function Delete() {
    $this->db->DeleteDocument($this->id);
  }
}
```

Listing C.14: models/Group.php

```
<?php
class Group {

  public $name = '';
  public $description = '';
```

```
private $db = null;

public function __construct($datarow=null) {
  $this->db = new DB();
  if ($datarow) {
    $this->name = $datarow['name'];
    $this->description = $datarow['description'];
  }
}

static public function GetAll() {
  $db = new DB();
  $rows = $db->SelectAllGroups();
  $ret = array();
  foreach ($rows as $row) {
    $ret[] = new Group($row);
  }
  return $ret;
}

static public function GetByName($name) {
  $db = new DB();
```

```
    $row = $db->SelectGroup($name);
    return new Group($row);
  }

  public function IsMember($user) {
    $ret = $this->db->SelectGroupMembership($user,
        $this->name);
    return $ret == 1;
  }

  public function GetMembers() {
    $members = $this->db->SelectGroupMembersByGroup(
        $this->name);
    $ret = array();
    foreach ($members as $u) {
      $ret[] = new User($u);
    }
    return $ret;
  }

  public function Add($user) {
    $data = array('user_name'=> $user->username,
              'group_name'=> $this->name);
    return $this->db->InsertGroupMember($data);
  }

  public function Remove($user) {
    return $this->db->DeleteGroupMember($this->name,
       $user->username);
  }

  public function Insert() {
    $data = array('name'=> $this->name,
            'description'=> $this->description);
    return $this->db->InsertGroup($data);
  }

  public function Update() {
    $data = array('name'=> $this->name,
            'description'=> $this->description);
    return $this->db->UpdateGroup($this->name, $data);
  }

}
```

Listing C.15: models/User.php

```
<?php
# TODO: Need to account for
class User {
  const PW_HASH_LENGTH = 32;
  const AUTH_TOK_NAME = 'auth';
  const CURR_USER = "curr_user";

  public $username = '';
  public $password = '';

  private $db;

  public function __construct($datarow=null) {
    $this->db = new DB();
    if ($datarow) {
      $this->username = $datarow['username'];
      $this->password = $datarow['password'];
    }
  }

  public function SetPassword($pass) {
    $this->password = $this->hash($pass);
}

public function checkPassword($pass) {
  return $this->password == $this->hash($pass);
}

// Enforce that passwords cannot be empty.
protected function hash($pass) {
  if (strlen($pass) == 0) {
    throw new Exception('Password␣too␣short');
  }
  return md5($pass);
}

public function GetGroups() {
  $members = $this->db->SelectGroupMembersByUser(
      $this->username);
  $groups = array();
  foreach ($members as $member) {
    $groups[] = new Group($member);
  }
  return $groups;
}
```

```
public function isAdministrator() {
  return $this->db->SelectAdminMembershipByUser($this
     ->username);
}

public function Insert() {
  $data = array('username'=> $this->username,
           'password'=> $this->password);
  return $this->db->InsertUser($data);
}

public function Update() {
  $data = array('username'=> $this->username,
           'password'=> $this->password);
  return $this->db->Updateuser($this->username, $data
     );
}

public function Save() {
  $user = $this->db->SelectUser($this->username);
  if ($user === null) {
    return $this->Insert();
    } else {
      return $this->Update();
    }
  }

  static public function GetByName($name) {
    $db = new DB();
    $row = $db->SelectUser($name);
    if ($row) {
      return new User($row);
    } else {
      return null;
    }
  }

  static public function GetAll() {
    $db = new DB();
    $rows = $db->SelectAllUsers();
    $ret = array();
    foreach ($rows as $row) {
      $ret[] = new User($row);
    }
    return $ret;
  }
}
```

C.4 View Templates

Listing C.16: views/document_changeowner.php

```
<?php include 'document_options.php'; ?>
<form action="<?=$this->route('document',␣'changeowner
   ',␣$doc->id)?>" method="post">
    <div>
        <em>"<?=$doc->title?>"</em> currently owned by
           <?=$doc->owner->username?>.<br />
        <label for="username">Set document owner:</
           label>
        <select name="username" id="username">
            <?php foreach ($users as $user): ?>
            <option<?php if ($user->username == $doc->
                owner->username): ?> selected="selected"
                <?php endif; ?>>
                 <?=$user->username?>
            </option>
            <?php endforeach; ?>
        </select>
        <input type="submit" value="Save" />
    </div>
</form>
```

Listing C.17: views/document_changepermissions.php

```
<?php
include 'document_options.php';

$granted_str = 'Granted';
$denited_str = 'Denied';
$granted_img = $this->link('images/accept.png');
$denied_img = $this->link('images/cross.png');
?>
<script type="text/javascript">
$(document).ready(function () {
  $('#userPermList a.edit').click(function () {
    var username = $(this).parent().parent().children()
       .first().html();
    var type = $(this).parent().attr('class');
    var permval = $.trim($(this).prev().attr('alt')) ==
        '<?=$granted_str?>' ? 1 : 0;
    $('#username').val(username);
    $('#userPerm').val(type);
    $('#userStatus').val(permval);
    return false;
  });

  $('#groupPermList a.edit').click(function () {
    var username = $(this).parent().parent().children()
       .first().html();
    var type = $(this).parent().attr('class');
    var permval = $.trim($(this).prev().attr('alt')) ==
        '<?=$granted_str?>' ? 1 : 0;
    $('#group').val(username);
    $('#groupPerm').val(type);
    $('#groupStatus').val(permval);
    return false;
  });
});
</script>

<h3>Edit permissions for "<a href="<?=$this->route('
   document', 'view', $doc->id)?>"><?=$doc->title?></a>
   "</h3>

<h4>User Permissions</h4>
```

```
<table id="userPermList">
  <tr>
    <th>Username</th>
    <th>Read</th>
    <th>Edit</th>
    <th>Delete</th>
  </tr>
  <?php foreach ($user_perms as $user=>$perms): ?>
  <tr>
    <td class="username"><?=$user?></td>
    <td class="read">
      <img class="value" alt="<?=@$perms['read']?
         $granted_str:$denided_str?>" src="<?=@$perms['
         read']?$granted_img:$denied_img?>" />
      <a href="#" class="edit"><img src="<?=$this->link
         ('images/page_edit.png')?>" title="Edit" alt="
         Edit" /></a>
    </td>
    <td class="edit">
      <img class="value" alt="<?=@$perms['edit']?
         $granted_str:$denied_str?>" src="<?=@$perms['
         edit']?$granted_img:$denied_img?>" />
      <a href="#" class="edit"><img src="<?=$this->link
         ('images/page_edit.png')?>" title="Edit" alt="
         Edit"/ ></a>
    </td>
    <td class="delete">
      <img class="value" alt="<?=@$perms['delete']?
         $granted_str:$denied_str?>" src="<?=@$perms['
         delete']?$granted_img:$denied_img?>" />
      <a href="#" class="edit"><img src="<?=$this->link
         ('images/page_edit.png')?>" title="Edit" alt="
         Edit"/ ></a>
    </td>
  </tr>
  <?php endforeach; ?>
</table>
<div id="userEdit">
  <form action="<?=$this->route('document',␣'
      setUserPermission',␣$doc->id)?>" method="post" id=
      "userPerms">
    <label for="username">Username</label>
    <input name="user" id="username" type="text" />
    <select name="perm" id="userPerm">
      <option value="read">Read</option>
      <option value="edit">Edit</option>
      <option value="delete">Delete</option>
    </select>
    <select name="status" id="userStatus">
      <option value="1">Granted</option>
      <option value="0">Denied</option>
    </select>
    <input type="submit" value="Set" />
  </form>
</div>

<h4>Group Permissions</h4>
<table id="groupPermList">
  <tr>
    <th>Group</th>
    <th>Read</th>
    <th>Edit</th>
    <th>Delete</th>
  </tr>
  <?php foreach ($group_perms as $group=>$perms): ?>
  <tr>
    <td class="group"><?=$group?></td>
    <td class="read">
      <img class="value" alt="<?=@$perms['read']?
          $granted_str:$denied_str?>" src="<?=@$perms['
          read']?$granted_img:$denied_img?>" />
      <a href="#" class="edit"><img src="<?=$this->link
          ('images/page_edit.png')?>" title="Edit" alt="
          Edit" /></a>
    </td>
    <td class="edit">
      <img class="value" alt="<?=@$perms['edit']?
          $granted_str:$denied_str?>" src="<?=@$perms['
          edit']?$granted_img:$denied_img?>" />
      <a href="#" class="edit"><img src="<?=$this->link
          ('images/page_edit.png')?>" title="Edit" alt="
          Edit"/ ></a>
    </td>
    <td class="delete">
      <img class="value" alt="<?=@$perms['delete']?
          $granted_str:$denied_str?>" src="<?=@$perms['
          delete']?$granted_img:$denied_img?>" />
      <a href="#" class="edit"><img src="<?=$this->link
          ('images/page_edit.png')?>" title="Edit" alt="
          Edit"/ ></a>
    </td>
```

```
  </tr>
  <?php endforeach; ?>
</table>
<div id="groupEdit">
  <form action="<?=$this->route('document',␣'
     setGroupPermission',␣$doc->id)?>" method="post" id
     ="groupPerms">
    <label for="group">Group</label>
    <input name="group" id="group" type="text" />
    <select name="perm" id="groupPerm">
      <option value="read">Read</option>
      <option value="edit">Edit</option>
      <option value="delete">Delete</option>
    </select>
    <select name="status" id="userStatus">
      <option value="1">Granted</option>
      <option value="0">Denied</option>
    </select>
    <input type="submit" value="Set" />
  </form>
</div>
```

Listing C.18: views/document_delete.php

```
<h3>Confirm delete</h3>
<p>Are you sure you want to delete "<?=$doc->title?>"
    ?</p>
<div class="confirmButtons">
  <form method="post" action="<?=$this->route('document
      ', 'delete', $doc->id)?>">
    <input type="submit" name="confirm" value="Yes" />
  </form>
  <form method="post" action="<?=$this->route('document
      ', 'view', $doc->id)?>">
    <input type="submit" name="cancel" value="No" />
  </form>
</div>
```

Listing C.19: views/document_edit.php

```
<?php include 'document_options.php'; ?>
<form action="<?=$this->route('document', $action, $doc
    ->id)?>" method="post">
    <div>
        <label for="title">Title:</label>
        <input type="text" name="title" id="title"
            value="<?=htmlentities($doc->title)?>"/>
    </div>
    <div>
        <label for="body">Content:</label>
        <br />
        <textarea name="body" id="body␣" cols="50" rows
            ="30"><?=$doc->content?></textarea>
    </div>
    <div>
        <input type="submit" value="Submit" />
    </div>
</form>
```

Listing C.20: views/document_list.php

```
<h1>Document Listing</h1>
<ul>
<?php foreach ($allowed_docs as $doc): ?>
    <li><a href="<?=$this->route("document",␣"view",␣
        $doc->id)?>"><?=$doc->title?></a></li>
<?php endforeach; ?>
</ul>
```

Listing C.21: views/document_options.php

```
<div id="options">
    <h2>Actions:</h2>
```

```
<ul>
    <li><a href="<?=$this->route('document',␣'view
        ',␣$doc->id)?>"><img src="<?=$this->link('
        images/page.png')?>" alt="View" title="View"
         /></a></li>
    <li><a href="<?=$this->route('document',␣'edit
        ',␣$doc->id)?>"><img src="<?=$this->link('
        images/page_edit.png')?>" alt="Edit" title="
        Edit" /></a></li>
    <li><a href="<?=$this->route('document',␣'
        delete',␣$doc->id)?>"><img src="<?=$this->
        link('images/page_delete.png')?>" alt="
        Delete" title="Delete" /></a></li>
    <li><a href="<?=$this->route('document',␣'
        changePermissions',␣$doc->id)?>"><img src="
        <?=$this->link('images/page_key.png')?>" alt
        ="Change␣Permissions" title="Change␣
        Permissions" /></a></li>
    <?php if ($this->current_user->isAdministrator
        ()): ?>
    <li><a href="<?=$this->route('document',␣'
        changeOwner',␣$doc->id)?>"><img src="<?=
        $this->link('images/group_key.png')?>" alt="
```

```
            Change␣Owner" title="Change␣Owner" /></a></
            li>
        <?php endif; ?>
    </ul>
</div>
```

Listing C.22: views/document_view.php

```
<?php include 'document_options.php'; ?>
<h2 class="title"><?=$doc->title?></h2>
<div class="byline">Owned by <?=$doc->owner->username
   ?></div>
<div id="content">
    <?php
    // Account for non-HTML documents.
    if (strip_tags($doc->content) == $doc->content) {
        echo nl2br($doc->content);
    } else {
        echo $doc->content;
    }
    ?>
</div>
```

Listing C.23: views/error_denied.php

```
<div>
```

```
    <h2>Access denied</h2>
    <p>You do not have access to this page.
    <?php if ($this->current_user): ?>
    Please contact the system administrator if you
       believe this is an error.
    <?php else: ?>
    Please log in and try again.
    <?php endif; ?>
    </p>
</div>
```

Listing C.24: views/footer.php

```
<div class="footer" style="visibility:hidden">
  Icons courtesy of <a href="http://www.famfamfam.com/
     lab/icons/silk/">Fam Fam Silk</a>.
</div>
</body>
</html>
```

Listing C.25: views/group_create.php

```
<form action="<?=$this->route('group',␣'create')?>"
   method="post">
    <div>
        <label for="group_name">Group name:</label>
        <input id="group_name" name="group_name" type="
            text" />
    </div>
    <div>
        <label for="description">Description:</label>
        <br />
        <textarea name="description" id="description"
            cols="30" rows="10"></textarea>
    </div>
    <div>
        <input type="Submit" value="Create" />
    </div>
</form>
```

Listing C.26: views/group_list.php

```
<h1>Group list</h1>
<ul>
    <?php foreach ($groups as $g): ?>
    <li><a href="<?=$this->route('group',␣'show',␣$g->
        name)?>"><?=$g->name?></a></li>
    <?php endforeach; ?>
</ul>
```

Listing C.27: views/group_show.php

```
<div>
    <form method="post" action="<?=$this->route('group
        ',␣'removeuser',␣$group->name)?>">
        <label for="user_remove">Group Members:</label>
        <br />
        <select name="" id="user_remove" multiple="
            multiple">
             <?php foreach ($group->GetMembers() as
                $user): ?>
             <option value="<?=$user->username?>"><?=
                $user->username?></option>
             <?php endforeach; ?>
        </select>
        <br />
        <input type="submit" value="Remove" />
    </form>
    <form method="post" action="<?=$this->route('group
        ',␣'adduser',␣$group->name)?>">
        <label for="user_add">Add User:</label>
        <input id="user_add" type="text" name="username
            " />
        <input type="submit" value="Add" />
    </form>
</div>
<div><a href="<?=$this->route("group", "getlist")?>">
    Back to group list</a></div>
```

Listing C.28: views/header.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 
    Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.
    dtd">
<html>
<head>
  <title><?=@$title?></title>
  <link type="text/css" rel="stylesheet" href="<?=$this
      ->link('style.css')?>" />
  <script type="text/javascript" src="http://code.
      jquery.com/jquery-1.4.3.min.js"></script>
</head>
<body>
  <div id="header">
    <h1>DocMan Document Manager</h1>
    <ul>
      <?php if ($this->current_user): ?>
```

```
<li><a href="<?=$this->route("document",␣"getlist
    ")?>"><img src="<?=$this->link('images/
    page_white_stack.png')?>" title="Document␣List
    " alt="Document␣List" /></a></li>
<li><a href="<?=$this->route("document",␣"create"
    )?>"><img src="<?=$this->link('images/page_add
    .png')?>" title="Create␣Document" alt="Create␣
    Document" /></a></li>
<?php if ($this->current_user && $this->
    current_user->isAdministrator()): ?>
<li><a href="<?=$this->route("user",␣"getlist")?>
    "><img src="<?=$this->link('images/user.png')
    ?>" title="List␣Users" alt="List␣Users" /></a
    ></li>
<li><a href="<?=$this->route("user",␣"create")?>"
    ><img src="<?=$this->link('images/user_add.png
    ')?>" title="Create␣User" alt="Create␣User"
    /></a></li>
<li><a href="<?=$this->route("group",␣"getlist")
    ?>"><img src="<?=$this->link('images/group.png
    ')?>" title="List␣Groups" alt="List␣Groups"
    /></a></li>
```

```
      <li><a href="<?=$this->route("group", "create")?>
         "><img src="<?=$this->link('images/group_add.
         png')?>" title="Create Group" alt="Create 
         Group" /></a></li>
      <?php endif; ?>
      <li><a href="<?=$this->route("user", "logout")?>"
         ><img src="<?=$this->link('images/user_go.png
         ')?>" title="Log out" alt=="Log out" /></a></
         li>
      <?php endif; ?>
    </ul>
  </div>
  <?php if ($session_messages): ?>
    <?php foreach ($session_messages as $msg): ?>
      <div class="<?=$msg[1]?>"><?=$msg[0]?></div>
    <?php endforeach; ?>
  <?php endif; ?>
```

Listing C.29: views/index.php

```
<ul>
    <li><a href="<?=$this->route("document", "getlist")
       ?>">Document List</a></li>
    <li><a href="<?=$this->route("document",␣"create")
        ?>">Create Document</a></li>
    <?php if ($this->current_user->isAdministrator()):
        ?>
    <li><a href="<?=$this->route("user",␣"list")?>">
        List Users</a></li>
    <li><a href="<?=$this->route("user",␣"create")?>">
        Create User</a></li>
    <li><a href="<?=$this->route("group",␣"list")?>">
        List Groups</a></li>
    <li><a href="<?=$this->route("group",␣"create")?>">
        Create Group</a></li>
    <?php endif; ?>
</ul>
```

Listing C.30: views/user_create.php

```
<div>
    <form method="post" action="">
        <div>
            <label for="username">Username</label>
            <input type="text" id="username" name="
                username" />
        </div>
        <div>
            <label for="username">Password</label>
            <input type="password" id="password" name="
                password" />
        </div>
        <div>
            <label for="confirm">Confirm</label>
            <input type="password" id="confirm" name="
                confirm" />
        </div>
        <div>
            <input type="submit" value="Create" />
        </div>
    </form>
</div>
```

Listing C.31: views/user_list.php

```
<h1></h1>
<ul>
  <?php foreach ($users as $user): ?>
  <li>
    <a href="<?=$this->route('user',␣'changepassword',␣
        $user->username)?>">
      <?=$user->username?>
    </a>
  </li>
  <?php endforeach; ?>
</ul>
```

Listing C.32: views/user_login.php

```
<form action="<?=$this->route('user', 'login')?>"
    method="post" id="login_form">
      <div>
          <label for="username">Username</label><input id
              ="username" name="username" type="text" />
      </div>
      <div>
          <label for="password">Password</label><input id
              ="password" name="password" type="password"
              />
      </div>
      <div><input id="submit" type="submit" value="Log In
          " /></div>
</form>
```

Listing C.33: views/user_password.php

```
<div>
```

```
    <form method="post" action="">
        <div>
            <label for="username">Password</label>
            <input type="password" id="password" name="
               password" />
        </div>
        <div>
            <label for="confirm">Confirm</label>
            <input type="password" id="confirm" name="
               confirm" />
        </div>
        <div>
            <input type="submit" value="Set␣Password"
               />
        </div>
    </form>
</div>
```

C.5 Auxiliary Files

Listing C.34: db.sql

```
-- Database initialization script for MySQL.
-- Static permission listing. This enforces the valid
    values
-- for permissions in the UserPermissions and
    GroupPermissions.
CREATE TABLE Permissions (
   permission VARCHAR(10) NOT NULL PRIMARY KEY
) Engine=InnoDB;
INSERT INTO Permissions VALUES ('read'), ('edit'), ('
    create'), ('delete');

CREATE TABLE Users (
   username VARCHAR(255) NOT NULL,
   password CHAR(32) NOT NULL,
   PRIMARY KEY (username)
) Engine=InnoDB;

-- Add the default administrator account
INSERT INTO Users (username, password) VALUES ('admin',
     MD5('admin'));

CREATE TABLE Documents (
   id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
   title TEXT NOT NULL,
   content TEXT NOT NULL,
   owner VARCHAR(255),
   PRIMARY KEY (id),
   FOREIGN KEY (owner) REFERENCES Users(username)
     ON UPDATE CASCADE ON DELETE CASCADE
) Engine=InnoDB;

CREATE TABLE Groups (
   name VARCHAR(255) NOT NULL,
   description TEXT NOT NULL,
   PRIMARY KEY (name)
) Engine=InnoDB;
INSERT INTO Groups (name, description) VALUES ('
    Administrators', 'System␣administrators');

CREATE TABLE GroupMembers (
   group_name VARCHAR(255) NOT NULL,
   user_name VARCHAR(255) NOT NULL,
   PRIMARY KEY (group_name, user_name),
   FOREIGN KEY (user_name) REFERENCES Users(username)
     ON UPDATE CASCADE ON DELETE CASCADE,
   FOREIGN KEY (group_name) REFERENCES Groups(name)
     ON UPDATE CASCADE ON DELETE CASCADE
```

```
) Engine=InnoDB;
INSERT INTO GroupMembers (group_name, user_name) VALUES
    ('Administrators', 'admin');

CREATE TABLE UserPermissions (
  document_id INTEGER UNSIGNED NOT NULL,
  username VARCHAR(255) NOT NULL,
  permission VARCHAR(10) NOT NULL,
  granted TINYINT(1),
  PRIMARY KEY (document_id, username, permission),
  FOREIGN KEY (document_id) REFERENCES Documents(id)
        ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY (username) REFERENCES Users(username)
        ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY (permission) REFERENCES Permissions(
     permission)
        ON UPDATE CASCADE ON DELETE CASCADE
) Engine=InnoDB;

CREATE TABLE GroupPermissions (
  document_id INTEGER UNSIGNED NOT NULL,
  group_name VARCHAR(255) NOT NULL,
  permission VARCHAR(10) NOT NULL,
  granted TINYINT(1),
  PRIMARY KEY (document_id, group_name, permission),
  FOREIGN KEY (document_id) REFERENCES Documents(id)
        ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY (group_name) REFERENCES Groups(name)
        ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY (permission) REFERENCES Permissions(
     permission)
        ON UPDATE CASCADE ON DELETE CASCADE
) Engine=InnoDB;
```

Listing C.35: index.php

```
<?php
session_start();
error_reporting(E_ALL);
date_default_timezone_set('America/New_York');

function __autoload($className) {
  $locations = array('controllers', 'lib', 'models');
  foreach ($locations as $loc) {
    $path = dirname(__FILE__).DIRECTORY_SEPARATOR.$loc.
       DIRECTORY_SEPARATOR.$className.'.php';
    if (file_exists($path)) {
      require_once $path;
      return;
    }
  }
}

$db['dsn'] = 'mysql:host=localhost;dbname=docman';
//$db['dsn'] = 'sqlite:'.dirname(__FILE__).
   DIRECTORY_SEPARATOR.'docman.db';
$db['username'] = 'docman';
$db['password'] = 'namcod';

$fc = new FrontController();
$fc->baseUrl = '/index.php';
$fc->rootUrl = '/';
$fc->viewPath = dirname(__FILE__).DIRECTORY_SEPARATOR.'
   views';
$fc->dispatch(isset($_SERVER['REQUEST_URI']) ? $_SERVER
   ['REQUEST_URI'] : $_SERVER['PHP_SELF']);
```

Listing C.36: style.css

```
/*
 Styles for the page header
```

```
*/

#header h1 {
    font-size: 14pt;
    display: inline;
    margin: 5px;
    padding: 0;
}

#header ul {
    margin: 5px;
    padding: 0;
    display: inline;
}

#header ul li {
    display: inline;
    margin-left: 5px;
}

#header {
    clear: both;
    border: thin solid black;
    background-color: #6699FF;
}

.info, .error, .success {
    margin: 4px;
    padding: 2px;
    border: thin solid black;
    text-align: center;
}

.info {
    background-color: yellow;
}

.success {
    background-color: green;
}

.error {
    background-color: red;
}
```

```
/*
 Footer styles
*/
.footer {
    font-size: 50%;
    text-align: center;
    margin-top: 100px;
}

/* Confirm buttons */
.confirmButtons form {
    display: inline;
}

/*
 Styles for per-page control links.
*/

#options {
    clear: both;
    background-color: #C8C8C8;
    border-left: thin solid black;
    border-right: thin solid black;
    border-bottom: thin solid black;
}

#options ul {
    margin: 0 0 0 5px;
    padding: 0;
    display: inline;
}

#options ul li {
    display: inline;
    margin: 0;
}

#options h2 {
    font-size: 12pt;
    display: inline;
    margin: 0 0 0 5px;
    padding: 0;
}

.clear {
    clear: both;
```

```
}

/* Login form styles */
#login_form {
    width: 260px;
    margin: 10px auto;
}

#login_form > div {
    clear: both;
    padding-top: 4px;
}

#login_form > div > label {
    float: left;
    width: 125px;
    text-align: right
}

#login_form > div > input {
    float: right;
    width: 125px;
    text-align: left;
```

```
}

#login_form > div > #submit {
    width: 6em;
    margin: 5px auto;
    float: none;
    display: block;
    text-align: center;
}

/* Document page styles */
.title {
    margin-bottom: 5px;
}
.byline {
    font-size: 70%;
    margin-bottom: 2em;
}

/* User permission page */

#userPermList td, #groupPermList td {
    text-align: center;
```

```
}


#userPermList td.username, #groupPermList td.username {
    text-align: left;
}
```

Bibliography

[AC03] Peter Amey and Roderick Chapman. Static verification and extreme programming. In *SigAda '03: Proceedings of the 2003 annual ACM SIGAda international conference on Ada*, pages 4–9, New York, NY, USA, 2003. ACM Press.

[Ame02] Peter Amey. Correctness by construction: Better can also be cheaper. *Crosstalk Magazine*, 2002.

[BH95a] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Softw.*, 12(4):34–41, 1995.

[BH95b] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[BH05] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial* *critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[BJ78] Dines Björner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, London, UK, 1978. Springer-Verlag.

[BN94] Manfred Broy and Greg Nelson. Adding fair choice to dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 16(3):924–938, 1994.

[BoLS09] U.S. Department of Labor Bureau of Labor Statistics. Occupational outlook handbook. `http://www.bls.gov/oco/`, 2009.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995.

[CF98] Baudouin Le Charlier and Pierre Flener. Specifications are necessarily informal or: some more myths of formal methods. *J. Syst. Softw.*, 40(3):275–296, 1998.

[CGR93] D. Craigen, S. Gerhart, and T.J. Ralston. An international survey of industrial applications of formal methods (volume 1: Purpose, approach, analysis and conclusions, volume 2: Case studies). Technical Report NIST GCR 93/626-V1 & NIST GCR 93-626-V2 (Order numbers: PB93-178556/AS &

PB93-178564/AS), National Inst. of Standards and Technology, Gaithersburg, MD., National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.

[Cha00] Roderick Chapman. Industrial experience with spark. *Ada Lett.*, XX(4):64–68, 2000.

[CMCP+99] Emanuele Ciapessoni, Piergiorgio Mirandola, Alberto Coen-Porisini, Dino Mandrioli, and Angelo Morzenti. From formal models to formally based methods: an industrial experience. *ACM Trans. Softw. Eng. Methodol.*, 8(1):79–113, 1999.

[CN00] Ana Cavalcanti and David A. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Trans. Softw. Eng.*, 26(8):713–728, 2000.

[dB94a] Roberto Souto Maior de Barros. Deriving relational database programs from formal specifications. In *FME '94: Proceedings of the Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*, pages 703–723, London, UK, 1994. Springer-Verlag.

[dB94b] Roberto Souto Maior de Barros. *On the Formal Specification and Derivation of Relational Database Applications.* PhD thesis, University of Glasgow, 1994.

[DS09] Eric Dash and Brad Stone. Credit card processor says some data was stolen. `http://www.nytimes.com/2009/01/21/technology/21breach.html`, 2009.

[DSRJ04] Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 165–174, Boston, Massachusetts, USA,, July 11-14, 2004 2004. ACM.

[EB04] Bruce Edmonds and Joanna J. Bryson. The insufficiency of formal design methods – the necessity of an experimental approach - for the understanding and control of complex mas. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 938–945, Washington, DC, USA, 2004. IEEE Computer Society.

[FF96] Kate Finney and Norman Fenton. Evaluating the effectiveness of z: the claims made about cics and where we go from here. *J. Syst. Softw.*, 35(3):209–216, 1996.

[FLM$^+$05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proc. Symp. Applied Mathematics*, pages 19–32. Am. Mathematical Soc., Am. Mathematical Soc., 1967.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[GL80] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, 2(4):564–579, 1980.

[Gri81] David B. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.

[HC02] Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Softw.*, 19(1):18–25, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hol97] C. Michael Holloway. Why engineers should consider formal methods. In *Proceedings of the* 16^{th} *AIAA/IEEE Digital Avion-*

ics Systems Conference, volume 1, pages 1.3–16 – 1.3–22, Irvine CA, October 1997.

[Jac98] Michael Jackson. Formal methods and traditional engineering. *J. Syst. Softw.*, 40(3):191–194, 1998.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[KHCP00] Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Trans. Softw. Eng.*, 26:675–686, August 2000.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml: A notation for detailed design, 1999.

[PH97] Shari Lawrence Pfleeger and Les Hatton. Investigating the influence of formal methods. *Computer*, 30:33–43, February 1997.

[Pol01] Fiona Polack. A case study using lightweight formalism to review an information system specification. *Software – Practice and Experience*, 31(8):757–780, 2001.

[SHW02] Carol Smidts, Xin Huang, and James C. Widmaier. Producing reliable software: an experiment. *J. Syst. Softw.*, 61(3):213–224, 2002.

[Sli04] Carol Sliwa. Blueprint for code automation. `http://www.computerworld.com/developmenttopics/development/story/0,10801,91383,00.html`, 2004.

[SM03] A. C. Simpson and A. P. Martin. Supplementing the understanding of z: a formal approach to database design. In *BCS 2003: Proceedings of the BCS Teaching Formal Methods workshop*, 2003.

[Smi00] Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[SW03] Donna C. Stidolph and E. James Jr. Whitehead. Managerial issues for the consideration and use of formal methods. In *FME*, pages 170–186, 2003.

[Zei02] Alan Zeichick. Modeling usage low; developers confused about uml 2.0, mda. `http://www.sdtimes.com/content/article.aspx?ArticleID=26637`, 2002.