Wisdom is the ability to apply knowledge and understanding gained through experience to new problems, using the insight of that experience to effectively identify solutions. We commonly think of wisdom applying to experienced individuals, but the term can also apply to software development organizations. Software systems are products of human knowledge and experience, and their development relies heavily on communication between individuals. Leveson notes that research in software engineering has been primarily directed towards the technical aspects of software development, ignoring the critical cognitive and social behaviors of the people involved. Part of the reason for this may be the lack of a common language for theorizing and communicating about the process of software development, a language that is capable of expressing not only the technical details of a development project, but also the individual and organizational attributes and behaviors that are also important to the success or failure of the project.

This document provides supplemental information and references for ACMSE06 Poster ID# 56. Also attached is the extended abstract to be published in the conference proceedings. This poster presents a preliminary outline of the author’s dissertation research proposal. The remainder of this document is organized as follows:

1 Problem Statement

The Standish group, has identified six key characteristics of unsuccessful (abandoned or incomplete, over budget, late, excessive defects, etc.) software system development projects:

- **Weak Executive Support.** Executive support requires justification, for example, why the software is needed and what value it will bring to the organization. Executives are paid for their experience and leadership — their wisdom — and their ability to maintain the larger perspective of the organization as a whole.

- **Poor User Involvement.** If the software does not fulfill the needs of the people who will use it, how can it be a success? Users understand what the software is supposed to do (or at least what it appears to do) from an immediate perspective. They also know what makes a particular piece of software effective, although they may not be able to explain why.

- **Inexperienced Project Leadership.** Project leaders are the primary channel of communication between those who actually perform the work and those who design and guide the large-scale objectives.
Their problems are largely those of people — allocating and motivating individuals and groups to accomplish tasks for the good of the project. Project leaders rely on their own past experiences (good and bad), as well as on the knowledge passed along from other managers.

- **Ambiguous Objectives.** Everyone involved in a software development project must have an understanding of the project’s goals, although this understanding may take different forms based on the various perspectives of the individuals. When the objectives are not clear, or the perspectives are not synchronized, the common vision is distorted, expectations and priorities diverge, and the project is handicapped.

- **Uncontrolled Project Scope.** Closely related to ambiguous objectives, uncontrolled scope can also lead to unsuccessful projects. Defining clear objectives for a software system, and maintaining these goals in a common vision are essential to preventing uncontrolled growth and expansion.

- **Change.** Change is inevitable, and experienced professionals expect change. Dealing with change requires experience and communication. Experience provides insights into potential outcomes, while communication shares this knowledge with others involved in a project.

The common thread that runs through these causes of unsuccessful software projects is the failure to capture, communicate, and apply the knowledge and understanding gained from past experience to new problems. Much of this knowledge and experience is held internally by individuals. This is the tacit knowledge — things we know but cannot readily put into words — that is often the key to unlocking solutions to newly encountered problems. Polanyi argues that this tacit knowledge is, in fact, the essence of scientific discovery\[23\]. The problem then becomes one of capturing and reusing this internal, tacit knowledge, in conjunction with the codified and explicit knowledge held by individuals and organizations, to identify problems and discover solutions.

In the domain of software development, this problem is compounded by the diversity of people involved in a particular software project. Each group of people brings a different perspective to the table. These perspectives may be different views of the same goals or objectives, or they may represent widely dissimilar expectations. Unifying these views into a common understanding of what the system is intended to be and do is difficult at best, particularly since there is not a common language in which these perspectives can be expressed. The goal of this research is to propose such a common language, justifying its purpose, structure, and usage based on results from other disciplines as well as empirical studies directed at investigating the effectiveness of this approach in the process of software development.

## 2 Approach

Storytelling is one of the oldest means of passing knowledge and experience on to others. The idea of using this approach in software development is common: the Unified Modeling Language (UML) provides notation for describing required use cases of a system, and eXtreme Programming relies on the collection of user stories to extract functional requirements. Software itself is also like a story where the executable code is the “script” and the users and other agents that interact with the system are the “actors.” The process of creating a software system is also a story, often of epic proportion, involving people, processes, and code. The development of a common language that integrates the cognitive, social, and technical perspectives would support the creation of narratives documenting software projects. These stories would provide something that has been missing, for the most part, from software development: a historical record that is written as a project is going on, and is reviewable after the project ends.

To achieve this goal, I propose a four-step approach starting with the definition of a schema-based meta-language based upon established theories of human communication and cognition. The second step
will be the development of visual and electronic representations of the elements of this language structure. Visual representations will be designed to maximize comprehension and communication, while electronic representations will be structured to support the creation of “dictionaries” of language elements as well as automated processing of language constructs. Step three initiates the evolution of a vocabulary for this language, starting with the translation of existing design, process, and organizational patterns. Vocabulary development continues with the documentation of other observed patterns of structure and behavior. The ultimate goal of this research is to construct narratives describing software system development projects. These stories will integrate human interactions and development methodologies with the technical details of system design and implementation.

- **Language:** Define syntactic and semantic structure of a metaphoric and metonymic design language.

Metaphor and metonymy have been identified as the two key mechanisms of human discourse\[11, 10\]. Since the goal is to evolve a language that facilitates effective communication, this language must incorporate these mechanisms. Metaphor is defined as a figure of speech where an expression is used to refer to something that it does not literally denote in order to suggest a similarity. For example, *raining cats and dogs* means that it is raining very hard, not that cats and dogs are falling from the sky. Metonymy, on the other hand, is the substitution of one word or phrase with another, closely associated term. In the U.S., the word *Washington* is often used to refer to the Federal government because the government is centered in Washington, DC. These two literary devices are not mutually exclusive: metaphor used within metonymic text can provide powerful semantic emphasis unattainable with either device alone\[16\]. Lakoff refocuses the concept of metaphor away from strict literary usage and onto empirical study of everyday metaphor. The definition of the term is also updated: metaphors are “general mappings across conceptual domains...the way we conceptualize one mental domain in terms of another.”\[14\] Petrie and Oshlag consider the metaphor to be a central tool for education, providing an important bridge between old and new knowledge, critical to triggering cognitive change and motivating disinterested students\[22\]. The connections to software design have not gone unnoticed, either: Nobel, Biddle, and Tempero discuss possible relationships between metaphor, metonymy, and software design\[20\].

Metaphor and metonymy are critical devices for a language describing software systems and their development. Software systems themselves are usually metaphors for a tangible, real-world process, yet they also create their own dynamic reality as a result of their execution. Everyone involved in the process of developing a software system must also engage in learning tasks as they integrate various perspectives and information with their own knowledge and experience. Verbal and written discourse are critical to this learning process.

The uniqueness of this approach lies in the application of cognitive and discourse research to the task of defining an integrated organizational, process, and design language. High-level programming languages, particularly object-oriented languages are highly metaphoric: classes and interfaces, for example, provide behaviors associated with the abstract class/interface names, allowing a programmer to work at a level closer to natural language. Modeling languages like UML build upon these abstractions with visual symbolism. This is where a critical break can occur: the symbolism of modeling languages may only be understood by the technical people involved in the development project. The direct cognitive connection between the technical models and stakeholder expectations or requirements is absent, forcing the less technically adept to “trust” the technical notation (and the people behind it).

Bridging this cognitive gap between technical and non-technical representations of system design can enable the transfer of both explicit, codified knowledge as well as internally held tacit knowledge and experience among a more diverse collection of stakeholders in software development projects. Designers, programmers, and testers can be more informed about customer and user expectations. Users, customers, and other dependent stakeholders could understand the strengths, weaknesses, and
limitations of the system. In general, a consensus about what the system is supposed to do and what conditions it will operate in can be evolved more readily than with current modeling languages.

- **Representation:** Develop visual and digital representations of language constructs.

An effective visual representation for this language is essential for human understanding. Individuals and groups must be able to use the language to effectively communicate about the system under development. Current modeling languages have evolved out of common usage, but no studies have been found that attempt to clearly define how expressive and understandable these languages are to a population as diverse as those involved in the development of a large software system. Developing a visual representation for this design language through studying the communicative effectiveness of the representation rather than its ability to map closely to actual source code is a departure from traditional approaches to software engineering, but one that is necessary.

The transitions between phases in a generalized software development process are commonly marked by the production of artifacts intended to transfer the knowledge and understanding gained in the earlier phase to the individuals responsible for the next set of tasks. Cognitively, this involves the transfer of knowledge between frames of reference or schemas. Brewer and Nakamura define schemas as “higher-order cognitive structures that have been hypothesized to underlie many aspects of human knowledge and skill,” noting that they are critical to the interaction between old and new knowledge.

Gick and Holyoak found that schemas are a key mechanism for the analogical transfer of knowledge between problem domains. The central role schemas play in reading comprehension, enabling and supporting recall and reconstruction of unlearned or forgotten text elements, has been demonstrated by Anderson and Pearson.

In software development, we have “design patterns” that resemble codifications of cognitive schemas, but these have been developed based solely on anecdotal evidence and arbitrary choices of format and content. The majority of the research on design patterns in the computer science and software engineering literature has ignored schema theory research from the cognitive sciences. I believe that this is a critical mistake, and that this existing body of research evidence on schema theory can significantly strengthen and enhance our understanding of why design patterns work (and why they do not). Furthermore, our application of design patterns to software development prefers to deal with them as single units, or at best in small groups tightly coupled to each other. Alexander’s seminal works on patterns and architecture require patterns to be used to create a “language” describing the generation of an architectural design. I disagree with Alexander on this point: the description should not be called a “language” but rather it is a story constructed using his patterns as vocabulary and structure. Taken in this context, his premise is much more understandable (this is an example of how the cognitive schemas for “language” and “story” condition how we think about a statement).

Alexander’s pattern structure and representation are also somewhat vague and abstract, although that was his intent - patterns were not to be absolute and deterministic rules, but generative guides. Gamma, Helm, Johnson, and Vlissides, Coplien, and many others have defined their own pattern formats, but these were also constructed based on the immediate needs of a particular organization. A significant portion of this research will be devoted to understanding and identifying visual representations of patterns that facilitate the transfer of knowledge. The extensive literature on both schema theory and metaphor/metonymy will be leveraged, as will the empirical methods used to evaluate cognitive processes.

A digital representation is also necessary if we wish to build collections of language elements and be able to exchange and manipulate the elements of these collections programmatically. While the electronic representation need not be closely coupled with the visual presentation, understanding how to effectively construct and represent patterns and schemas should provide insights into efficient constructs for digital storage. At the same time, we must also remember that human beings are the ones who must create these structures, and the cognitive connection between the human and electronic cannot be ignored.
The ability to collect, store, and recall explicit and tacit knowledge and experience in software development has many benefits. Collections of design patterns (e.g., [8, 6, 18, 24]) are exactly this: codification of knowledge and experience to enable reuse. The literature on design patterns has been growing steadily over the past decade, although this research is largely anecdotal and observational rather than being derived from closely controlled empirical studies. While efforts have been made to formalize the structure of design patterns ([12], [13], and [19] for example), this work has taken a distinctly “computer science” perspective, and has not attempted to address the communicative effectiveness of the formal structures. Aside from developing visual and electronic pattern/schema representations, this proposed research will also contribute to our understanding of how to study the creation and use of patterns.

**Vocabulary:** Document common design decision, development process, and organizational structure patterns.

Once a language structure and effective visual representations have been formulated and identified, we can begin building a vocabulary. The first step in this process will be the translation of existing patterns into the new representation and language structure. This will be followed by the documentation of other patterns intimately involved in software development: methodologies, organizational structures, common group attributes and behaviors, developer/stakeholder interactions, etc. This vocabulary will facilitate the construction of stories in this schema/pattern language; narratives that describe the ongoing development of a software system. These descriptions will be multilayered, reflecting the many concurrent tasks that are actually happening, but will enable individuals to select a particular perspective on the story and examine the project from that viewpoint.

The common perspective on design patterns is that they can only be used in either isolation or in small, closely coupled groups. I would argue otherwise. Patterns and schemas are composite entities: small-scale patterns are grouped by association or relation to form larger-scale patterns. Patterns of ink on paper may for recognizable letters, patterns of letters form words, patterns of words form sentences, etc. Similarly, there are fundamental software concepts: loops/iteration, conditional and unconditional branching, assignment, instantiation, scope, etc. We combine these basic patterns to implement data structures and algorithms, combine these to build modules, and eventually the large, complex patterns we call software systems. To manage the variety of tasks needed to accomplish this, we have developed methodologies to structure and control the software development process, more patterns within patterns. Software systems are developed, maintained, and used by human beings, with a variety of organizational and social structures: more patterns within patterns. We already have names for many of these tasks, activities, relationships, behaviors, etc. What is missing is the common framework in which we can integrate our knowledge and experience, our wisdom, towards effectively finding solutions to new problems.

**Narrative:** Describe software system development projects, integrating human interactions, development methodology, and software design.

Current approaches to capturing the evolutionary process of software development rely on tracking the changes in the assortment of documents generated within the process, including the actual source code. Identifying and collecting these changes only provides a limited amount of information: which document was changed; what was removed, added, or modified; when the change occurred; who made the change. Responsibility for describing why a change was made rests on the person who submits the modification to the change management system. Different organizations can have different standards for this documentation. Constructing the overall view of how a system is developed and why various decisions are made is difficult with this limited information.

However, if we view the process of developing software systems as a story, a solution can be visualized. All but the simplest narratives are layered literary constructs, and even simple stories can have a variety of interpretations depending upon the reader’s perspective. These layered meanings are built through metaphor and metonymy, leveraging different interpretive schemas. Multiple storylines weave
their way through the central theme, reinforcing certain aspects while deemphasizing others, all the while interacting and modifying each other and building towards the climactic event: the release of the software system to the customer.

The ultimate goal of this research is to facilitate the capture of software development projects in a narrative form. This narrative will be created from sequences and layers of patterns and schemas. Sequences will trace particular tasks, processes, or other chains of events. Layers will capture different levels of detail pertinent to different perspectives on the process, organization, and system under development. When a change occurs, it will affect a particular sequence of patterns, and the other sequences and layers closely related to the modified sequence will reflect the reason for the change, and the mechanism by which the change was made.

For example, consider a design and implementation change made because of a defect detected during testing. One sequence will be describing (telling the story) of the testing process, including the failures. The design and implementation sequences interact with the testing sequence, and change due to the detected defect. The source code to be changed will itself be part of a sequence of events that lead to its existence, allowing the sequence of decisions originally made to write it to be traced, providing additional information about the defect and possible ramifications of proposed fixes.

Narratives are a common and fundamental mechanism human beings use to share knowledge and experience — we are storytellers. To attempt to disassociate ourselves from that heritage is a disservice to our abilities to teach and learn with each other. Multi-level narrative descriptions of software development projects offer the ability to capture the entire history of a project, technically, socially, and cognitively. As we collect these stories and build a true literature of software development, we may be able to identify those stories with lasting quality, beauty, and elegance, just as we do with literature, music, and art. We could then begin to theorize about why some works are better than others (and why some are worse), and put our efforts towards emphasizing what is good.

3 Proposed Empirical Studies

Note: The first study is under development at the present time. The current plan is to have a pilot study complete by May 2006.

1. Compare comprehension and applicability of existing design patterns with and without explanation of specific syntactic and semantic relationships between elements of pattern schema.
   Goals:
   - Identify critical relationships within pattern schema
   - Demonstrate improved usability when relationships are well-defined

2. Compare visual representations of pattern formats
   Goal:
   - Identify structural and presentational elements that support or hinder the use of patterns as knowledge collection and transfer entities

3. Compare collections of patterns with and without defined sequential and layered orderings, in all cases given defined syntactic and semantic structure and using best visual representation formats
   Goals:
   - Identify key layering and sequencing relationships
- Demonstrate improved communicability of layered and sequential pattern sets

4. Evaluate the effectiveness of pattern-based narrative descriptions integrating system design and implementation with organizational structure and development methodology against traditional documentation methods

Goal:
- Demonstrate that pattern-based narratives are more communicative to a broader range of stakeholders in a software system project

References


[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.


