

SIMPLIFYING THE RELATIONSHIP BETWEEN PROGRAMMERS AND COMPUTERS

An Interactive Qualifying Project Proposal

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

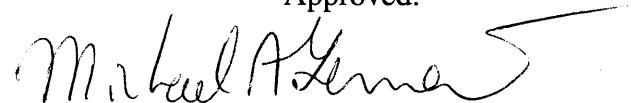
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

by

Benjamin Holt
Ryan Kenyon
Alexander Goodrich

Date: April 30, 2001

Approved:



MICHAEL A. GENNERT

ABSTRACT

This project will examine the ways software developers interact with each other through the source code they write. Specifically, we will study guidelines, some of which are in widespread use today and others that have failed to gain acceptance, that are intended to help make source code easier for other programmers to understand. Finally, by examining these practices in the broader context of human linguistics, we will develop improved techniques to facilitate this form of communication.

Table Of Contents

Abstract.....	1
Table Of Contents.....	2
1. Authorship.....	
2. Introduction.....	
.	
3. Background.....	
.	
4. Understanding The Meaning of Code.....	
.....	
5. Developing Meaningful Code.....	
.....	
6. Methodology.....	
.....	
7. Case Studies.....	
..	
8. Conclusion.....	
.	
9. References.....	

1. Authorship

Abstract

Table Of Contents

1. Authorship - rmk
2. Introduction – bjh, rmk
3. Background – arg, bjh
4. Understanding The Meaning of Code
 - 4.1 Linguistics: Interresting Observations – bjh
 - 4.2 Readability – bjh
 - 4.3 Program Cognition - bjh
5. Developing Meaningful Code
 - 5.1 How Style Impacts Meaning – rmk
 - 5.2 Literate Programming – arg
 - 5.3 EXtreme Programming - bjh
6. Methodology - bjh
 6. Case Studies
 - 6.1 An Examination of Literate Programming Tools – arg
 - 6.2 7.2 Meaning conventions used in the Windows SDK® - rmk
8. Conclusion - rmk

2. Introduction

Throughout their careers, programmers must not only write code, but also read it. It is often as challenging to interpret the meaning, intension, or goal of programmer's code as it was for that programmer to write it. Sometimes it is more challenging. The need for understanding the meaning of code varies from task to task. Code must be modified in order to upgrade, modify or debug software components. If code cannot be understood, then programmers cannot work with it -- it must be rewritten from scratch.

Collaboration amongst programmers can help them to understand the project's code. Tools also exist which may help, such as syntax-coloring editors, but the programmer should write the code well to begin with will to provide a solid working foundation.

It is difficult and time consuming to strive to write perfectly clear code with unmistakable meaning. Developers usually try, to some extent, to write their programs in a way that helps others (or themselves) understand and work with the code later. Formatting standards and style guidelines abound, but generally fail to address how developers can make their code more meaningful.

The term meaningful is not often applied to program source code; this is symptomatic of the particular problem this project seeks to address. To illustrate further, writing style can make a remarkable difference in the clarity and accuracy of a reader's understanding of the ideas an author seeks to convey through a piece of prose. Similarly, we wish to

explore ways in which programmers can improve their coding style to enhance communication with other developers.

The benefits of highly readable code are many, varied, and far-reaching. It is fairly obvious that well written code is easier to debug, but it is astonishing how much difference this makes, even for the person who wrote the program, even while it is still fresh in their mind! In our professional experience, the most effective debugging technique is to write well.

Moreover, when writing conscientiously, programmers are better able to maintain the various aspects of the problem in their minds and so make fewer mistakes. This is even more important if they are just starting to work with a codebase, picking an old project back up, or trying to reuse an old routine, since they must build this mental model quickly and accurately.

A related benefit, which may not be quite so obvious, is that good code is usually faster to write. This may or may not be true for very small programs, but it becomes critical on larger projects with complex interactions between components, particularly when they involve collaboration with other developers.

Finally, hard-to-read programs are jokingly called 'job security code' because, according to legend, people used to write programs poorly so that, since no one else could understand that system, they would be too valuable to ever lay off. Regardless of whether

anyone ever actually used that strategy, companies have come to understand the benefits of well written code and one of the best ways to increase job security today is to build a reputation for writing good code.

Given the tremendous advantages of good coding practices, numerous techniques have been developed, but so far the computer programming community has not been terribly satisfied with the results. In order to improve on these, we need to do three things: first, we must learn what others have tried, both successfully and unsuccessfully, to solve this problem. Then, we must examine the actual practices which good developers use to write clear code. Lastly, we will study the communicative mechanisms at work, and develop a set of techniques that improve the flow of this communication while, hopefully, avoiding the pitfalls encountered by previous solutions.

3. Background

In the course of our studies, we uncovered various techniques that programmers have used over the course of the past few decades. The first revolution in programming occurred in the early 1970s, when developers first started codifying the experience gathered by the first pioneers of programming. The “structured programming” paradigm introduced the programming world to the concepts of design, analysis, and maintenance.

[Ben86]

The developer community has since then been concerned with the direction that program development has progressed. Other problems needed to be resolved, especially areas in program cognition and understanding.

Early experience with code maintenance has shown that programs and the tools that are used to write these programs must be standardized, easy to learn, and well documented. The end goal of this monumental effort is to write code that is simple and easy to understand by any developer. We make the claim that program code should be as easy to understand as natural language.

In the next few sections, we will explore some of the techniques used by developers over the years to get closer and closer to this ultimate goal.

4. Understanding The Meaning of Code

4.1 Linguistics: Interesting Observations

Although many programming language design techniques, such as Chomsky grammars, for instance, originated from the study of 'natural' linguistics, typical computer science curricula and textbooks mention this only as a minor curiosity, if at all! More to the point, most programming languages are very rarely designed to incorporate aspects similar to those observed in natural language.

Nevertheless, many programming languages do, indeed, exhibit a number of the characteristics of natural language. For example, variables are a lot like nouns and function calls are analogous to verb phrases. The connections run deeper, however. In natural language, nouns and verbs (along with adjectives and adverbs) are classified as 'content words' and considered to be 'open classes', meaning that new ones are added to a given language and others drop out of use continuously. Certain other grammatical categories (such as demonstratives or articles, like 'a' or 'the' in English) are grouped together as 'function words', and are generally 'closed classes' because they change very rarely. [ADH84] Similarly, programming languages typically do not acquire or eliminate keywords unless they are undergoing an exceptional revision or splitting off a separate new language.

Also noteworthy, in spite of the fact that precise grammars are readily available for virtually every programming language devised, they are universally taught and learned through techniques very similar to those used for acquiring a second natural language. [OGD97] Specifically, even after students have been exposed to concepts and notation of grammars, they are still typically introduced to programming languages with example fragments (much like simple sentences or vocabulary words), specific constructs (akin to idiomatic expressions), and generalized, 'fuzzy', grammatical rules (similar to, for instance, pronoun usage or standard verb conjugations in a foreign language class). [OGD97] If they ever see the full, exact, grammar at all, it is almost always after they have become proficient with programming in that language.

4.2 Readability

Reading is an amazing and hard-won ability requiring years to develop. While reading programs is not a particularly large step beyond the general skill, it is largely ignored by computer science and software engineering curricula. Students are expected to just 'pick it up' and, to some extent, they do; however, the process could probably be greatly accelerated by guidance and cultivation.

Comprehension of a computer program is a very complex process, and at present, there are only a few rough theories about how it works. Whatever the underlying mechanism, there are many 'tricks' expert programmers use to help unravel an unfamiliar program, and there are many pitfalls which instruction could steer novices around. Furthermore, it is a skill that improves greatly with practice, and students are typically not required to read programs nearly as often as software engineers. Finally, the ability to read and understand programs improves a developer's writing proficiency as well.

There are a number of factors which influence code readability, most of which are within a programmer's control, at least to some degree. On the other hand, most of the same issues apply to programming languages and libraries as well. While one can write reasonably readable programs in virtually any language, some require more effort than others to achieve the same level of readability; indeed, sometimes it seems like it would be easier to just re-write the language!

For example, C never was a particularly graceful or language, but it worked pretty well for a lot of things, so it stuck and grew and evolved. Recently, a great deal of new syntax and semantics were added in the creation of C++; none of this has been done in a consistent or organized manner and this has continually eroded what little elegance the language may have had, and sharply degraded its readability. Werther and Conway felt that it was time for a major overhaul; preserving the semantics of the language, but re-designing the syntax to be consistent, easy to read, and reduce the incidence of some of the most common errors.

They call their new language 'SPECS' (Significantly Prettier and Easier C++ Syntax), and the improvement is dramatic! We believe that their design choices indicate (or comprise) some very useful principles/heuristics for improving readability in general; a few of the most important are as follows:

Maintain consistency; this is the single most important readability principle, but it is often tricky to engineer, and difficult to maintain. One aspect of this is repetition and predictability of patterns or motifs. For example: SPECS uses "keyword identifier : typeid {definition}" for declarations and definitions of all sorts; variables are "obj foo : int" for instance, functions are "func bar : (void -> int) { return 42; }"

Another part of this, a component important enough to break out separately, is that items and constructs should resemble each other in proportion to the similarity of their purpose; things that are similar should look alike (type creation for typedef, struct, enum, for

instance), things that are related should have a similar format but be distinguished in some significant way (variable vs function definitions for instance), and things that are different should be markedly distinct, relative to how different they are (related things should have the same format but a different keyword for instance, but a comparison should not look like an assignment!)

Finally, some heuristics we observed the SPECS syntax exhibiting:

Put most important information up front, like flavor of declaration, identifier, and type; this makes it quicker and easier to spot when scanning through the code.

Things should flow - progress - generally in temporal order, sometimes from specific to general, or whatever makes sense for the situation, but it should be deliberate.

4.3 Program Cognition

"Program Understanding or Code Cognition is a central activity during software maintenance, evolution, and reuse. Some estimate that up to 50% of the maintenance effort is spent trying to understand code. Thus a better grasp of how programmers understand code and what is most efficient and effective can lead to a variety of improvements: better tools, better maintenance guidelines and processes, and documentation that supports the cognitive process." [VV94]

It is pretty clear that program understanding requires a developer to integrate his or her existing knowledge, particularly knowledge computer science, the specific language and platform involved, and the domain of the problem that the program was meant to solve. This integration, ultimately, produces new knowledge: an understanding of precisely what the program does, how the system goes about it, and often a better understanding of the problem, the language and platform, even computer science in general. [VV94]

How this integrated information is represented, how the developer's mind constructs and maintains its model of the system being studied, and how to facilitate the process most effectively are not nearly so straightforward.

There are several common threads in the various theories that have been developed to answer these questions. These threads each represent the stages a particular aspect of this process goes through as it touches on each of the above questions.

The first, and most concrete, of these threads concerns the textual representation, and how it is organized; specifically, the 'chunks' of source code that represent meaningful ideas, how those chunks are located, processed, and how to make them stand out most clearly. The consensus is that 'beacons' such as formatting conventions, descriptive names, and common implementation techniques, play a key role in this process; most programming style guides are about improving these beacons. [VV94]

Above, and growing out of, the chunks are the concepts implemented in the blocks of code; these ideas get organized into hierarchical, cross-referenced, knowledge structures called 'plans'. These plans span all the strata of knowledge relating the relevant bits of each to the system and its operation, thus creating the developer's understanding of its implementation. [VV94]

Both the exploration of the code and the development of the plans are driven by 'strategies' for the formation and testing of hypotheses. Strategies vary greatly, and are often influenced by the code itself and the hypotheses the developer is seeking to verify. For example, a developer might take a top-down approach by starting with the main function proceeding on the ones it calls; in so doing, he or she may proceed in an orderly fashion, or not. Also, each block of code may be taken at the face value of any beacons the developer recognizes, examined in detail, or re-examined if there is reason to question a previous interpretation. [VV94]

So what does this all mean? Basically, that programmers should write using a consistent formatting style which sets coherent units apart as easily recognizable blocks and choose variable and function names that are descriptive of their purpose; that had better not surprise any developers!

On the other hand, experimental evidence has revealed some techniques that tend to be more effective for program comprehension than others have. They tend to take a relatively systematic, breadth-first approach, typically with the aid of specialized schemas which they have accumulated from experience with other programs. They are generally also comparatively flexible about their approach, and question their assumptions more readily. [VV94]

5. Developing Meaningful Code

5.1 How Style Impacts Meaning

It is not possible to write high level code without employing some degree of consistency or style. There are many ways to format a given construct in code. Some programmers are very particular about the conventions they use when writing code, while others may put little or no thought into it.

When writing code, it is important to consider the impact it will have on future programmers' ability to understand and work with it. When writing any programmatic construct, a developer should always select from a reasonable list of possible style

conventions. The convention that they choose should most expressive of what the programmer wishes to communicate about the code block.

5.1.1 Naming

Before we had compilers or assemblers, programs had to be written in the language of their target machine. Machine languages use designations, ordinals, or handles to refer to objects. As human beings, we prefer to assign names to the objects we work with.

Whether these objects are software entities or real world entities, naming them can allow us better understand their relationships with each other and perhaps ourselves. High level computer software development environments allow us to bind names to entities.

Computer systems will ignore the names we choose for our software entities, as long as they comply with the semantics of the programming language. Good naming is the most fundamental step in writing meaningful code, because there are no constraints on what we choose to name our objects. They are independent of the code's operation. The name of a variable, property, parameter, function, class, package, namespace, library, project, module, or machine should be chosen which best reflects it's purpose. If a developer purposely chooses a cryptic name, they will impede other developers from using the entity. In general, long, friendly, and descriptive names invite programmers to utilize a named entity in code. Entities which should be used with special care or not at all should not perhaps be given the most friendly names to document their matter.

Developers may choose names that possess special meaning in order to help them remember. This is acceptable as long as other developers can relate to it as well or at

least understand why the name was chosen. A developer working with another's code may not initially understand the nature of a given name. But if the name is unique or perhaps has a nice ring to it, then it may cause them to bind a story or scenario with that name which better helps them to understand the purpose of the named entity. Even if the developer has a different emotional response to the name than the original developer did, they are still just as likely to be able to form an effective memory device in their own mind for understanding and working with entities of the given name.

Used properly, names can describe any number of attributes about an entity. An entity's type is often described using Hungarian Notation. Hungarian Notation is a variable naming convention which consists of prepending a short type code before a variable name suffix.

Names can also indicate a base type which a class has been derived from, or an interface which a class implements. They can suggest that an entity implements a common paradigm. For example, if an object is retrieved through a 'get' function, one might expect there to be a release function as well. One might feel comfortable applying a 'delete' or 'destroy' function to an object only if that object was retrieved through a previous call to a 'create' function. Function parameters beginning with 'i' are probably an index of something, where 'h', 'id', or 'key' may be used to signify a handle, identification, or key of something.

5.1.2 Distribution

Developers may choose to group a set of components based on their relevance to each other. Very complex hierarchical class libraries can become easier to understand in doing such. Nesting a component within another implies that that component belongs to its parent in some way.

When a library of code is expanded upon, parts of it may have to be rewritten. If a nested component becomes shared among multiple components, its root in the hierarchy should be moved to a location that satisfactorily expresses its relevance to its super and sub components. Sometimes the only obvious place for the component will be the root of the entire library. In such cases developers should decide if it is really worth losing all hierarchical information, or if they should just leave it alone. Sometimes these sorts of decisions can seem too philosophical or even religious. In such cases the issue should be ignored and considered trivial to the benefit of the overall development process.

5.1.3 Formatting

Tabulation of text provides a convenient way for programmers to express class and data scopes in their code. In general the rule is simple: indent blocks of code relative to their parent's block when the scope, frame, or namespace is sub classed. Labels that designate GOTO targets in C should be aligned in the same column for any given function because they can be called from any point in that function.

Procedure declarations with very long parameter lists can utilize tabulation when wrapping the list to a new line. Typically it makes sense to indent the wrapped portion of

the parameter list beyond the first line of the declaration, so that it does not hide the declarator. (Some programmers may prefer to see a single parameter on each line, which others may wish to crush as much code onto each line as they can.) The same idea can be applied to class declarations that require more than one line for the base class list.

In C/C++ header files, tabulation can be applied to prototypes to indicate hierarchical relationships among the classes, functions, etc. There are many clever ways in which tabulation can be used to indicate a hierarchy amongst objects in a program.

5.2 Literate Programming

Another approach we took in researching the problem of how developers can express more meaning through their code is a method called literate programming. Literate programming is a philosophy of writing code as if it were literature. To put this in context with the current paradigms of programming that are available for use, such as object-oriented programming and structured programming, literate programming approaches development of code from the standpoint that we are trying to explain to other humans what the code does rather than instruct the computer on what to do. [Knu84]

Donald Knuth in the 1980s intended that literate programmers approach their programs as if they were works of art. He envisioned programmers as essayists seeking to write comprehensible programs, such that every programmer reading the work would fully understand the nuances involved and the design philosophy that went into it. [Knu84]

This involves treating every program being worked with as a technical paper intended for publishing. Literate programs can be reviewed by other programmers with ease, with feedback and criticism can be found to be similar to the world of literature where fellow authors discuss styles of authorship. [BKM86]

Literate programming is not by any means a stand-alone paradigm that cannot be combined with existing styles and methods of programming. In contrast with traditional programming paradigms, literate programming should be used to enhance code readability. Literate programming does not require specific language support. Although the original tool (WEB) developed by Donald Knuth was focused around the use of the Pascal programming language, there are many other tools that have been created over the years that allow literary programs to be used in any language, such as CWEB, FWEB, noweb, nuweb, and LEO. There are many other tools, but during the course of our research, we will only evaluate a small sampling.

The initial proposal describing this new paradigm of programming by Donald Knuth involved a system he referred to as WEB. This WEB file encapsulates the program source and documentation. This system decomposed a WEB file into a printable document and a program that is directly compiled and executed by the computer. The intention is to force the programmer into thinking about programs at a higher, more abstract level, and let the computer parse this information into executable code. The programmer can no longer deal directly with the compiler source that is the output of the

WEB file; the output is made as garbled as possible in order to force the programmer into dealing only with the encapsulated documentation and source file. [Ben86]

This approach affords the programmer several advantages. One of the advantages that is provided by this new paradigm is an integrated feel for design strategies, which gives the programmer the ability to attack the problem by using a bottom-up approach or a top-down manner, as code can be arranged in entirely any arbitrary manner [Gra99]. Only when the program is being compiled for execution is it rearranged to fit into program order. This is a very useful advantage from the perspective of software engineering, as it allows the developer to design a new project incrementally in any fashion desired.

Another advantage offered by literate programming techniques is the ability to divide code in the project into easily manageable chunks, a method known as divide and conquer. Literate programming also forces the developer to think about readability on a global scale, by referencing variable names and function names in an index and table of contents. Visual aids and algorithmic descriptions are included in the same document that the code is based from, encapsulating both design and code into one easily referenced document. Literate programming makes extensive use of natural language throughout its implementation, allowing developers to understand code descriptions easily, as humans are more readily accepting of plain language descriptions than cryptic commands. [Gra99]

These advantages, when taken together, benefit not only the developer who is currently working on the project, but also programmers who may be looking over and maintaining the code years down the line. In addition, a fully realized literate program would also allow newly added developers on a project team to get up to speed quickly with the work currently in progress with minimal delay.

Given these advantages over the more traditional methods and tools of programming, this approach seems ideal for the role which we intended. However, we find a surprising industry reluctance to accept this new methodology of programming, even though it was introduced nearly 20 years ago! This reluctance is examined by browsing several internet newsgroups on the subject. A few developers have expressed their thoughts as to why they think literate programming has not taken off immediately in the community. One software developer named Lee Wittenberg explained a few points as to why he believes the paradigm has not reached mass popularity.

The first point the developer made was that most people simply do not know that the programming technique exists. Very few classrooms across the country teach literate programming techniques to beginning computer science students. The only exposure programmers have to literary programming comes across through professional and academic journals. Unfortunately, many programmers are not exposed to these journals. The second point made by the developer is made with regards to programmer laziness. He puts forth the argument that many programmers only “pay lip-service” to the ideals of structured programming. In addition, these programmers cannot be bothered with taking

the time to construct a fully working design or do a lot of work up front before diving into programming. The final argument is one of inertia. Programmers tend to work with what they're familiar with and do not inherently enjoy learning a new style at all once set in their ways.

Another developer named Kristopher Johnson put forth some arguments regarding lack of literate programming support in common tools such as profilers, debuggers, compilers, and visual editors. He also makes the point that many literate programming tools were developed with the UNIX environment in mind, and that tools developed with other operating systems and hardware platforms in mind are not as polished or useful.

Other assorted developer experience with the literate programming philosophy included notes on how several of the tools commonly used in literate programming were difficult to learn. This difficulty in learning how to use the tools is passed down from not only the programmers that were directly involved with coding the project in the first place, but to those who maintain the project years down the line.

With these developer opinions in mind, we decided to research academic and professional journals to see what those who were intimately familiar with the paradigm saw as the inherent weaknesses of literate programming.

One problem that was mentioned was not brought up at all in the developer newsgroups. This problem, however, is related to Lee Wittenberg's point that programmers are inherently lazy. Most developers would find it too bothersome and time-consuming to

write literate programs for short coding projects [Gra99]. The emphasis of literate programming is to write extensive documentation. Hence, if a developer is writing a quick and dirty tool that may be used once or twice before being discarded, spending a significant amount of time creating a work of art for a 200 line program seems ludicrous.

The problem regarding the lack of suitable tools was confirmed as well. Until very recently, each of the tools available for literate programming were language dependent, and none were WYSIWYG. In addition, such tools had a steep learning curve in which many of the features that were provided with the tool were not even used. [Ram94]

When taken together, these problems helped prevent literate programming from becoming main stream. The possibility then exists that in order to overcome several of the hurdles, students need to be taught literate programming starting from the beginning of their careers in order for the paradigm to be effective. Several developers had already taken this idea one step further by conducting a study on the subject.

These advocates of literate programming have demonstrated that teaching beginning and novice programmers how to program in a literate manner from the start was highly beneficial. The developers found that these students had an easier time learning problem solving skills. In addition, they found that students that had little background in programming often did better in the course than the students that already had some programming knowledge. This can be attributed to the third point made by one of the developers regarding programmer resistance to change. Finally, programmers exposed to

the literate programming technique were overall more successful in future classes and work than those who had not been exposed to the paradigm. [CDL95]

5.3 eXtreme Programming

extreme programming, abbreviated XP, is a software development methodology designed to improve the performance of relatively small teams developing custom software. The driving goal of XP can be summed up in two words: embrace change. Kent Beck, the 'father' of extreme programming articulates the problem particularly well:

"Everything in software changes. The requirements change. The design changes. The business changes. The technology changes. The team changes. The team members change. The problem isn't change, per se, because change is going to happen; the problem, rather, is the inability to cope with change when it comes." //28//

Therefore, XP is designed to create an environment which makes change easily manageable and minimizes risk while placing as little additional burden on the development team as possible. To achieve this, XP brings together some relatively well-known practices and a few guiding principles, all implemented to their respective extremes (hence the name) and all working together to help the team produce the maximum functionality with the least overhead.

Crucial to the organization of any team, an underlying theme of this report, and one of the core tenets of the extreme programming 'philosophy' is communication. In fact Beck grounds the entire process in this principle:

"The first value of XP is communication. Problems with projects can invariably be traced back to somebody not talking to somebody else about something important" //29-30//

XP takes a particularly unusual approach to the aspects of communication this report is primarily investigating: the communication between collaborating developers, and the role program code plays in that interaction.

To elaborate, peer code review is generally considered to encourage this, so an XP team peer reviews everything all the time by writing all code in dynamic partnerships. The technique is called 'pair programming', and helps to keep everyone on the team familiar with the entire codebase; individual members may be more at home with some parts of the system than others, but working with code you have seen before is always easier than exploring a component for the very first time. The pairing also strongly discourages unclear programming style because the act of writing the code becomes integrated into a real-time dialogue. As Beck describes it:

"Code also gives you a chance to communicate clearly and concisely. If you have an idea and explain it to me, I can easily misunderstand. If we code it together, though, I can see in the logic you write the precise shape of your ideas."

Furthermore, the source code will always be the most up-to-date documentation of any given piece of functionality, especially in combination with reasonable coding standards to maintain consistency and a strict ethic of writing programmatic tests to illustrate the use of that functionality and insure that it remains operational.

Thus, XP emphasizes both coding standards and tests, because they also facilitate several other aspects of the process as well as documenting the code, and it dispenses with external documentation because of the high maintenance costs and comparatively low return of such documentation. Moreover, the more frequently the code is changed, the more dramatically the cost of keeping documentation in step with the functionality increases.

Though rigorous testing and pair programming sound like a significant amount of overhead, both directly support the ultimate goal of producing useful, maintainable, high quality software. Even if one categorizes them as overhead, teams applying these techniques, particularly as parts of the complete XP method, are consistently far more productive and, closely related, have much higher morale than they did when they were working within a more traditional Requirements-Analysis-Design-Implementation-Testing-Production methodology.

Revisiting program understanding for a moment, it does not appear that Beck or any of his colleagues performed or drew upon any formal program cognition experiments when

they codified the techniques and principles of XP, but their collective intuition, honed by years of experience, guided them to replace external formal documentation with a set of practices which match up exactly with the characteristics formal studies have shown to optimize code cognition.

Specifically, consistent formatting standards, consistent designs and implementations which primarily use simple -- straightforward -- algorithms. XP also provides two other aids, neither of which has been formally analyzed yet: a team of coworkers with a strong cooperation ethic, all of whom are at least acquainted with the codebase, and a flexible, easy-to-use, testing and experimentation harness which can be used not only to verify a hypothesis, but also to insure that the routine in question retains that functionality in future revisions.

6. Methodology

In order to learn more about how developers in the industry interact with the code which they and their colleagues write, we conducted a survey. Due to time constraints, the scope of the survey was relatively limited, and targeted towards experienced developers that we believed possessed significant expertise and insight into the issues that we explored. It was not meant to be a widespread survey to discover what most programmers' experiences are when working with existing source code; rather we sought a collection of case studies illustrating how exceptional developers who deal with these problems constantly, and have developed techniques to best cope with them.

To this end, we designed the survey to have both very specific questions and many opportunities to elaborate on the answers given and make additional comments. We then wrote a simple perl program to collect the results through an HTML form, which we made accessible on the World Wide Web. We then sent email to contacts in companies known to be experts in their respective areas of the industry explaining our project and asking for their input.

6.1 Survey Results

Because our target population was so specific we only received five responses, three of them from developers at the same company, one from a manager at another company and one from a student.

Such a small, unevenly distributed, set of returns is, of course, not a representative sample; though it is useful as a group of case-studies.

The numerical results and basic analyses are in the table below.

		Response Category							
Question	R1	R2	R3	R4	R6	Category %		Note:	
Company	C1	C1	C1	C2	St.	60%	C1		
						20%	C2		
						20%	Student		
1	75	50	50	25	25	20%	75	(% of programming time spent trying to understand existing code)	
						40%	50		
						40%	25		
2	Y	Y	N	Y	N	60%	Y	(company coding standard)	
3	4	4	--	4	--	100%	4	("it comes in pretty handy")	
4	4	4	--	4	--	100%	4	(follow it "most of the time")	
5	Y	Y	Y	Y	Y	100%	Y	(do something on your own)	
6	None	Some	None	None	None	80%	None	(feedback about coding style)	
7	Y	Y	N	Y	Y	80%	Y	(do anything outside the code)	
8	4	4	--	5	5	50%	4	(follow those: "most of the time"	
						50%	5	"always")	
9	1	1	3	2	1	60%	1	(most useful: "company standard"	
						20%	2	"own coding style"	
						20%	3	"other practices")	

When we designed the survey, we decided that, because of the small anticipated sample size, the most useable results would be obtained by relatively coarse-grained, categorical, data. Therefore our analysis focuses on grouping, comparison, and bringing out the particularly strong relationships.

The first important observation was that they are all pretty similar, no obvious outliers (except for response R5 which was completely blank, and so has been omitted from the table and the calculations).

Secondly, the respondents from company C1 all spend at least half of their programming time working on understanding existing code, while the manager from company C2 and the student both spend only about a quarter of their programming time on this.

It also seems that company coding standards are fairly well accepted, followed regularly, and are widely regarded as extremely helpful.

One of the strongest patterns, however, is that the respondents all make some sort of individual effort to improve the readability of the code they write, even though most receive no feedback to indicate if they are, indeed, producing good, readable, code.

The write-in responses also showed some interesting patterns; summed up in the following table categorized by whether the respondent wrote 'Lots' (200 characters or more), 'Some' (1 to 199 characters), or 'None'.

Write -In	Response Category					Response % (1 or more characters)	Note:
	R1	R2	R3	R4	R6		
1	L	S	--	S	--	100%	(elaborate on Q2)
2	L	S	L	S	S	100%	(elaborate on Q5)
3	L	S	N	N	N	40% *	(elaborate on Q6)
4	L	S	--	S	S	100%	(elaborate on Q7)
5	L	S	L	S	N	80%	(elaborate on Q9)
6	L	N	N	N	N	20%	(tell us about your development process)

* Some of the respondents may have believed that this field was not applicable if they answered 'None' to question 6; we feel that it was still quite applicable, and besides the other obvious interpretation would have yielded a 200% response rate!

The most striking pattern is the consistency of individual respondents; all write-ins answered by an individual received approximately the same amount of attention. On the other hand there was no significant correlation between particular questions and the length of the answer, so we opted to group the length categories together when computing the response rates.

Although it is good that most of the response rates on the write-in questions were at or above eighty percent, we were hoping for more in-depth responses, particularly on the final one; however the single response we received for it was excellent.

7. Case Studies

In this section we shall expose tools, libraries, and segments of code owned by real world software vendors.

7.1 An Examination of Literate Programming Tools

In this case study, we will examine three of the programming tools used in the literate programming environment. Their characteristics, similarities, advantages and disadvantages, and ease of use will all be considered in comparison to each other and to the possible programmer desiring to learn the literary paradigm.

7.1.1 CWEB

This tool is an adaptation of the original WEB system that Donald Knuth created for the Pascal programming language. As such, it runs in the same manner, using a tangle/weave operation to produce source and pretty printable documentation from a single “CWEB” file. CWEB was written specifically to fit the nuances of C++, Java, and ANSI C programming languages. It is widely available on all platforms – including the

Macintosh, Windows, Amiga, and UNIX environments. The CWEB system requires familiarity with not only the WEB system, but the TeX typesetting language. Hence, the learning curve is somewhat high, but there are many features available for use as a result, including pretty-printing and extensive cross-referencing and indexing capabilities. The complexity of this system is fairly high and is not recommended for the novice programmer, especially when there are easier tools that can be used, such as noweb.

7.1.2 FWEB

FWEB was originally a spin-off of the CWEB program, intended for the Fortran programming language, but it has evolved into much more than that. FWEB supports multiple programming languages including C, C++, Fortran, Ratfor, and TeX. FWEB can also run using other programming languages in a special language independent mode. FWEB also allows programming using multiple languages at once, which can be useful when mixing two languages together. FWEB support across multiple platforms is widespread. As long as the platform provides an ANSI C compiler, FWEB is supported on that system. As FWEB is based around the same WEB system as CWEB, the complexity is high, as is the learning curve. For programmers already familiar with the WEB system, learning FWEB is easy. FWEB is recommended for users that are looking for language independence with the features of CWEB.

7.1.3 noweb

noweb was developed as an answer by Norman Ramsey to the complaints of many programmers about the difficulties in beginning literate programming. The requirements were simple: the tool had to be easy to learn, simple, and language independent. Keeping in mind the problem of programmer acceptance outlined in the literate programming section, this program was designed to address the complaints of the developer community.

noweb strips away much of the functionality of the WEB system, leaving the core fundamentals of literate programming. The ability to pretty print documentation is no longer present, but this removes the burden of the programmer having to learn the TeX typeset language. noweb keeps the ability for the programmer to divide his/her work into chunks that can be edited out of order, along with the ability to index and cross-reference these chunks and other identifiers. Unfortunately, noweb is only available on UNIX and MS-DOS platforms, leaving Macintosh users with the choice of using the CWEB system only.

7.2 Meaning conventions used in the Windows SDK®

7.2.1 Input and Output Parameters

The Windows SDK Hungarian Notation for function parameters uses a 'p' or 'lp' in front of a variable name to indicate that it is a pointer. For function parameters in the SDK, names meeting this qualification remind programmers that this is an output parameter.

The programmer understands that they must supply the address of a variable which will have its value set upon returning from the call.

```
BOOL GetWindowRect(  
    HWND hWnd,    // handle to window  
    LPRECT lpRect // address of structure for window coordinates  
);
```

Most parameter names which do not begin with 'p' or 'lp' are not pointers, and therefore programmers can safely assume that the parameter acts as input to the function.

7.2.2 Accessors and Mutators

Windows SDK tends to begin functions with 'Get' to indicate that it accesses an item in a collection, and conversely 'Set' to indicate that it modifies some item.

>> for example: GetActiveWindow(), GetCurrentProcess()

In any software model which manages access to members of a collection of objects, the design must define a way to refer to the objects in the collection. In many cases, collections will closely resemble an array, a map, a bag, or a set. Parameter and function names used in the Windows SDK help the programmer to understand the type of collection they are dealing with. For example, accessors to map or table entries tend to take parameters beginning with 'h' or 'hKey'. Array accessors tend to take parameters beginning with 'i' or 'n'. The following declaration shows a function which first accesses a Window object from a handle, then sets the index of a double word value within the object:

```
LONG SetWindowLong(  
    HWND hWnd,    // handle of window  
    int nIndex,   // offset of value to set  
    LONG dwNewLong // new value
```

);

7.2.3 Functions which must be used in pairs

Functions which must be used in pairs often participate in paradigm which requires the user to release a handle or object back to the system when they are finished using it.

Windows SDK tends to begin functions with 'Release' to indicate that it releases a handle previously obtained through a 'Get' function. 'Begin' and 'End' or 'Enter' and 'Leave' are also prepended to function names to indicate that the functions must be used with each other:

BeginPaint(...)/EndPaint(...), EnterCriticalSection(...)/LeaveCriticalSection(...)

7.2.4 Functions which manage allocation from a specific pool

Windows SDK allocation functions tend to be appended with 'Alloc', 'Free' or 'Realloc'. The programmer can tell what he or she is allocating or freeing by peeking at the preceding token in the function name.

for example: HeapAlloc(...) / HeapFree(...) / HeapRealloc(...), VirtualAlloc(...) / VirtualFree(...)

7.2.5 Synchronization Functions

Windows SDK synchronization functions in general contain 'Wait', 'Lock', and 'Reset'

for example: WaitForMultipleObjects(...) / ResetEvent(...) and VirtualLock(...) / VirtualUnlock(...)

The remaining tokens in the function name describe what is being locked or unlocked.

7.2.6 Object Creation and Destruction Functions

Windows SDK tends to couple creation and destruction functions by prepending 'Create' or 'Destroy' to the function suffix.

for example: `CreateWindow(...)`, `DestroyWindow(...)` and `CreateMenu(...)`,
`DestroyMenu(...)`

The remaining tokens in the function name describe what is being created or destroyed.

7.2.7 Class Names

Windows COM designates all interfaces by prepending an 'I' in front of the class name.

for example: `IDispatch`, `IDataObject`, `IClassFactory`

Microsoft MFC concatenates base class names in some cases to form the new qualified class name.

for example: `CStdioFile` has `CFile` as a base class.

Microsoft MFC prepends every class name in it's library with a 'C'.

7.2.8 Hungarian Notation Conventions For Windows SDK

Prefix Data type

b Boolean

by byte or unsigned char

c Char

cx / cy short used as size

dw DWORD, double word or unsigned long

fn Function

h Handle

i int (integer)

l Long

n short int

p a pointer variable containing the address of a variable

s string

sz ASCIIZ null-terminated string

w WORD unsigned int

x, y short used as coordinates

8. Conclusion

Computer code written by human beings is designed to be interpreted by a computer, but can be oriented for human interpretation as well. This document describes how programmers can read, write, and think about code which is meaningful to humans. Programmers can write more meaningful code by thinking about the implications of formatting their constructs in various ways. Learning to read code is a valuable skill that can reduce the amount of external documentation, and questions aimed at the programmer of the original work.

9. References

==== ARG =====

[Ben86] Jon Bentley, Programming Pearls – literate programming, *Communications of the Association for Computing Machinery*, 29(5):364-369, May 1986, CODEN CACMA2. ISSN 0001-0782

[Knu84] Donald E. Knuth, Literate Programming, *The Computer Journal*, 27(2):97-111, May 1984

[BKM86] Jon Bentley, Donald E. Knuth, and Doug McIlroy Programming pearls - A literate program. *Communications of the Association for Computing Machinery*, 29(6):471-483, June 1986. CODEN CACMA2. ISSN 0001-0782.

[Gra99] Mike Gradman, Literate Programming, <http://www.cs.tamu.edu/people/mgradman/>, December 1999

[CDL95] Bart Childs, Deborah Dunn, and William Lively. Teaching CS/1 courses in a literate manner. *TUGboat*, 16(3):300-309, September 1995.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97-105, September 1994. CODEN IESOEG. ISSN 0740-7459.

=====

==== BJH =====

[VV94] Anneliese, Von Mayrhauser and Vans, A. M.. Program Understanding - A Survey. *Technical Report CS-94-120* Colorado State University, August 1994.

[ADH84] Akmajian, A., Demers, R.A., and Harnish, R.M. (1984) Linguistics: An Introduction To Language And Communication (6th ed.) Cambridge, MA: The MIT Press.

[Beck00] Beck, K (2000) Extreme Programming Explained: Embrace Change. Reading, MA: Addison Wesley Longman, Inc..

[DN90] Deimel, L. and Naveda, J. (1990) Reading Computer Programs: Instructor's Guide and Exercises. CMU/SEI-90-EM-3

[OGD97] O'Grady, W., and Dobrovolsky, M. (1997) Contemporary Linguistics: An Introduction (3rd ed.) M. Aronoff (Ed.). New York: St Martin's Press.

[WC96] Werther, B. and Conway, D. (1996) A Modest Proposal: C++ Resyntaxed
Technical Report 95/240, Dept. Computer Science, Montash University

=====