ALPHA DRAFT

Writing programs and documentation to meet reader expectations

Aamod Sane

Icon Computing
11940 Jollyville Road, Suite 300-N,
Austin, TX 78759
email: *sane@iconcomp.com*

May 19, 1998

Abstract

Rules of thumb such as "place important material at the end of a sentence" are commonly used in writing. Such guidelines describe reader expectations that good writers satisfy. This paper extends this idea to the study of reader expectations for program readers. The paper presents a pattern language of reader expectations, and exhibits its use in diverse cultures, including literate programming, extreme programming, TEX programming and Unix programming. The approach offers a perspective that may help improve documentation practices in different contexts.

1 Shepherding Notes

The Plop guidelines suggest that long papers indicate 10 pages to review. For this paper, skim the section titles. Try the details of pages 1-3, 4-8, and 13-14

2 Introduction

Effective writing fulfils reader expectations. So writing recommendations [23, 26] include knowledge about reader expectations codified as rules of thumb. One such rule, for example, is that readers emphasize material that arrives at the end of a sentence. Therefore, writers should place new material near the end of a sentence to engage the readers attention. Another rule is that the beginning of a sentence creates expectations for what follows, and a connection to the previous material. Violating this rule creates drifting, incohesive paragraphs. A working knowledge of such patterns of reader expectations improves writing.

Are there analogous rules or patterns of reader expectation for programs? If there are, then they perhaps suggest clearer answers to the always difficult questions about the proper form of program organization and documentation. This paper is a preliminary attempt at uncovering the pattern language of program reader expectations. The language links techniques for program organization and documentation to readers needs.

The patterns are classified into four categories:

Readers Purpose Unlike books, a program reader is also a writer, perhaps of the same program. The readers purpose, typically, is to use or modify the program. The degree of desired modification leads to differing expectations about documents.

Readers Methods Books assist readers with tables of contents and indices. But typical programs are intricate structures that demand correspondingly sophisticated reading aids.

Readers Culture The language and cadence of a book is often an accurate indicator of its period of origin. Similarly, programs bear the stamp of the programmers culture in their choice of names, program structure and code organization. A reader from a culture will expect writers to observe its mores.

Reading Style A readers learning style, kinesthetic, visual, abstract, concrete, etc., determines her ideal documentation.

We study these patterns at work by looking at different programming cultures: Literate programming, Extreme Programming, TEX programs, Unix programming, Research programming cultures, and product programming cultures. Each culture comes with different expectations, and leads to different ways of organizing programs and documentation.

3 The Readers Purpose

A reader reads a program in order to: use it as a library, make casual modifications, learning in order to be an expert, or is living with the program and evolving it. For our purposes, these four distinctions suffice.

3.1 Reader Is User

The reader uses the program as a library. In this context, the reader constructs library objects, and invokes a method and expects that the advertised transformations will happen. He is not interested in the internals of the program, but cares about completeness, consistency, predictability, accuracy, and typically, efficiency.

Therefore: Describe the pre-and-post conditions of methods, and their errors. Keep the values consistent: if you return "-1" for errors, do so consistently. Keep the externally visible names consistent. Give examples of use; such examples should be enough for the programmer to get started. Highlight and comment on the rough spots.

Examples: The Unix standard library is a good example; it is by and large consistent, and the manual pages are accurate. But there are classic problems, like `getchar()`, where the name suggests that the return value is a character, but in fact it is an integer. One of the best features of Unix documentation is its bugs section, and situations like the getchar bug are usually mentioned. The java documentation is complete and consistent in its style due to automatic generation. But in their first release they had problems with inconsistent names for accessing instance variables. Since then, `get()` and `put()` prefixes have been used uniformly to access variables. The bugs sections are not prominent, but the public newsgroups and their frequently-asked-questions lists compensate to a degree. In neither case are examples supplied with the documentation, and books pick up the slack. On the other hand, the Java Generic Library (JGL) supplies both the interface documentation and examples. Following its intellectual progenitor, the C++ Standard Template Library (STL), JGL also specifies performance metrics for time. Quirks and bugs are addressed in its frequently-asked-questions, and via the documentation of STL.

3.2 Reader is Proto-Expert

The reader is, perhaps, a newly hired programmer working on a project. He will spend time learning the code; it is in the interest of the project to reduce the learning curve. You will need to account for differences in reading styles 6 and reading methods 4. The project will have engendered a culture, and the reader must adjust to it 5. Often, the best way is to simply encourage the reader to talk to expert programmers. However, before you can *Just Ask*, the proto-expert needs to acquire some shared background.

Therefore: Supply the reader with a guide to the overall architecture, and local style guides and naming conventions. Make the connection of the architecture to the code evident. Ensure that the code actually follows the conventions. The program organization can be critical in understanding the program, so present a rationale for the organization. Where third party products and tools are involved, produce interaction and collaboration diagrams. Record knowledge about the quirks of these products. Diagram complex aspects of the system, especially those that involve concurrency and distribution. Leave a trail of critical design decisions for the system. Unit tests and functional tests are an important form of documentation. A novice can begin to understand system behavior by tracing execution during tests. Diagrams are often out date; the tests had better be up-to-date. A good programming environment is critical to exploring programs. Failing this, cross reference tools will help.

Examples: In many companies, the overall architecture of the product, the analysis of features of competing products, and customer requirements are available in some form. Style guides and the use of diagrams is commonplace; equally often, the guides and diagrams are relics of infrequent interactions with management, and developers rely on code: code never lies. Parnas and the patterns movement have emphasized the importance of design decisions. Patterns have emerged as an effective way to capture design issues and are seeing increasing use. The importance of tests as documentation is recognized in projects such as Chrysler Comprehensive Compensation 7.2. Exploratory environments such as Smalltalk and Emacs excel at revealing program organization. To a lesser extent, the so-called Integrated Development Environments help, but usually suffer in comparison with Smalltalk or even Emacs. Tools like Ctags that cross reference are common. They are endlessly reinvented because often users are unaware of them. At least in the case of Java, the Javadoc tool is intrinsic to the culture.

3.3 Reader Is Writer

Reader is writer, sometimes of different pieces of the program. The program is evolving, and each reader is expert at one aspect, less so at others. In this situation, on the one hand the need for documentation is supreme, but there is no time to do detailed documents. You do not need documents, you need telepathy, you need documentation leprechauns[1] , you need photographic memory.

Therefore: Adopt practices that approximate these requirements. These practices include:

Pair Programming : Programmers should program in pairs, one person programming, the other looking over his shoulder. This sounds uncomfortable, but practitioners report otherwise.

[1] Leprechauns are creatures from Celtic mythology. There is supposed to be one behind you, just where you cannot see it. If you glimpse one in a mirror, it becomes your slave. In my mythology, a documentation leprechaun would be one who *is* your slave, follows you around, and documents code while you are writing it.

You commit fewer errors as the other person detects them, and two people know the system intimately. Pair programming is an integral part of Extreme Programming, discussed in Section 7.2.

Pauseful Programming : After breakneck sessions of programming, upon attaining reasonable stability, pause and step back from programming. Explain the design and implementation to someone else, write up the essence, record the design decisions. Here, you act as your own documentation leprechaun.

Programming Diaries : Diaries are used to record interesting self-debates about design possibilities, whiteboard designs, changes to program pieces written by other people, interesting bugs and debugging techniques, and so on. This goes hand in hand with Pauseful Programming, and approximates photographic memory. It is especially useful when the programmer is away from the code, and reverts to *Reader is Proto-Expert*.

Where the programmer is an expert, documentations needed by the proto-expert are less useful. For example, cross reference tools, diagrams and other devices are far slower than memory, although not to be disdained.

Examples: Pair programming has an ancient but murky history. There are reports of pair programming going back to Multics. In recent days, Kent Beck and Ward Cunningham have been enthusiastic promoters of this idea. Pauseful programming is a pattern practiced in research cultures. You let the issues sink in, meditate on the problems and solution, and often interesting ideas bubble up. Discussions and explanations well known techniques for gaining perspective on your work. A famous example of programming diary is Knuths diary and the resulting paper, The Errors of TEX [12] Diaries have been recommended by Watts Humphrey for his Personal Software Process [8]. Emacs includes a special mode for keeping change logs of programs; they may be accompanied by design notes.

3.4 Reader is Other

Other roles in programming include team leader, tester, release manager and so on. People in these roles have to assess the state of the project, ensure that internal guidelines are met, track changes in the projects, know about program bugs and features, manage developer responsibilities, and so on. Depending on the size of the group, these responsibilities may distributed in various ways.

Therefore: A different set of documents, connected more loosely the program, are necessary. These include an overview of the past and the future, the history and the target. They also include bug tracking and the changes in the project from week to week. Depending on the size and culture of the group, these documents may range from separate programs that track requirements, bugs, testers and so on. Other alternatives include the bazaar development and extreme programming styles that are less formal but involve practices such as frequent code releases, pair programming, refactoring and so on 7.2.

Note: A proper discussion of this pattern should probably take place in the context of organization patterns.

3.5 Reader is Casual Modifier

This is a peculiar situation that arises in the open-source software culture of promiscuous code sharing. This culture trades in Emacs-lisp, Unix, X, or TEX programs borrowed from the net. The program is modified in some way, and once the reader is satisfied, left untouched for years. The reader may also export the change back to the writer. The reader in this situation is resourceful, and comfortable with the culture where the program is born 5.

Therefore: Do not gratuitously violate cultural mores, such as naming conventions, programming and documentation styles, program organization styles. Clearly mention the library versions and the machines where your program is tested. Surprisingly, virtually no explicit documentation is needed in these cases.

Examples: The Emacs-Lisp library is a successful example of this style of hit-and-run programming. It is a convention to use names separated by hyphens, and use prefixes to distinguish functions that define a program. Comments are included as part of function definition. Extensions to the program are affected by hook methods. Emacs-Lisp programs are typically editing modes that build on a standard infrastructure for support modes. There are standards for writing help files, file comments, and for keeping track of various programs. The documentation of the underlying Emacs is usually the only explicit documentation, but it is an exceptional example of the pattern *Reader is User*. Adhering to standards, the source code together with the superb programming environment is sufficient to communicate the essence of the code. The enthusiasm of programmers triumphs over "minor" details like lack of packages and dynamic scope. Even more extreme examples are found in the case of Unix, X, and TEX programs. The Linux system is also an example, although modifications to the the all-important kernel are rigidly filtered by its creator. The systems described here are even more chaotic, as they are not so critical as the kernel.

4 Reading Methods

These patterns present techniques used to read programs. Program documentation and organization should take these techniques into account, but remember that the reading methods evolved together with approaches to program organization. You can equally interpret these patterns as guidelines for readers.

A reader typically looks for architectural overviews and program organization to begin understanding a program. She will subsequently look for the basic classes, the class hierarchies, the data flow, the creational aspects and critical choices in the design. Uncommon constructions in programs will need explanations. Good names will help a reader grasp a program. Examples, especially unit tests and functional tests are good targets for a reader. The patterns below consider how a writer can support these activities.

4.1 Overview

Usually, to understand a program the reader must have knowledge of its primary function, and of the principal features of its architecture. This remains true for recursive sub-parts of the program.

Therefore: When it is natural, capture the overall function in a Controller object that is part Facade [6] and part Mediator [6]. In other cases, maintain records of scenarios and collaboration diagrams that highlight the essential components and their interactions. Often, the test driver of a program will reveal the overall program organization.

Examples: The importance of overall architecture is well known. There are efforts underway to classify and formalize software architectures [21].

4.2 Units Are Cohesive

The structure of program units like packages (Java, Ada), directories and makefiles (Unix), categories (Smalltalk), suggest a suitable program decomposition for a reader.

Therefore: Ensure that program units are cohesive: that is, their subcomponents implement a clearly defined aspect of the overall program. Motivate and document the decomposition you have chosen. Follow the rules of your programming culture.

Examples: Booch [2], Martin [16] and others argue the importance of units for program understanding, deployment, and release.

4.3 Why This And Not That

Often, a particular design is easier understand when contrasted with other plausible designs that are not applicable. The other designs are eliminated because of non-functional requirements like efficiency or perceived extensibility. It is also important to remember the reasons for rejecting the other designs, because in time the original assumptions may become invalid.

Therefore: Record the design decisions. The structure of pattern descriptions suggests a suitable form. Discuss the pro-and-con aspects, and the forces that motivate the decision. Include consequences.

Examples: Parnas [18] has argued for the need for rationalizing designs. Design Patterns are examples of code-level design decisions that are typically explained by contrasting the bad practice against good practice. The choice of external technology will often be accompanied by justifications about cost and effectiveness.

4.4 Inverse Analysis and Design

In understanding a program, a reader analyzes a program to uncover the design and its connection to the problem being solved. The reader deduces use cases, role models, scenarios and patterns from the program.

Therefore: Make the connection to analysis and design artifacts evident in your code. Indicate the important classes, class hierarchies and collaborations of your program. Make patterns evident using comments or object names. Use factories to create objects. Package and functional tests should document the aspects of scenarios and use cases that they test.

Examples: One oft-quoted advantage of object-oriented programming is the continuity from analysis to implementation. It follows that programs ought to be un-analyzable to reveal their connection to analysis and design artifacts. Pattern recognition tools [20] have been built to aid in program analysis. It is commonplace to use tests as examples of the use of programs. With many programs, it is the best place to begin analyzing a program.

4.5 Here Be Dragons

Depending on the readers sophistication, techniques like Generic Programming, Functional programming with heavy use of recursion, Meta Object systems, Concurrency and Distributed Protocols might appear unusual. If such a user is also modifying code, the results can be traumatic.

Therefore: Realize that sooner or later a comparative novice will sully your code. Defend against it by including references to standard works, and instructions to ask someone before attempting to modify unusual code.

Examples: It is common to include references to textbooks for unusual algorithms. With the coming of Java, concurrent programming has become widespread, but not widely understood. Books such as Concurrent Programming in Java [15] help, but inexperienced developers underestimate the subtleties of concurrent programming.

4.6 Open Sesame

The right names for objects and operations help make complex designs understandable.

Therefore: Consider books like Smalltalk Best Practices Patterns [1] that suggest ways to choose good names. Use style guides for selecting names of variables. Keep a thesaurus and dictionary at hand.

Examples: Experience suggests that one of the greatest benefits of development processes is to produce agreement regarding notation, and to synchronize program names with business concepts. Many methods emphasize the use of data dictionaries to keep terminology consistent. It is folklore that good programmers have above-average vocabulary and writing skills.

4.7 Examples are Documentation, alias Guided Tour

The control flow and roles of objects are more easily understood when we see a program in action. A debugger can be use to trace example execution. This serves as a guided tour of the program structure.

Therefore: Provide examples and test cases for your program. They should be documented so that each example may be connected with some aspect of the problem the program solves.

Examples: Virtually all frameworks come with illustrative examples. In Smalltalk, Lisp and other cultures with interpretive environments, a traditional way to understand a program is to set a breakpoint and inspect interesting objects, function parameters, and variable values. The Squeak web site contains “bon mots”, little examples that illustrate aspects of the Smalltalk framework.

4.8 Code must Communicate

Readers usually have very focused interest in portions of the code. For those portions, they will need details. But detail in the form of comments is distracting and hard to keep consistent with the code. For the rest, they would like an overview to see if it interferes with the fragment of interest.

Therefore: Supply readers with birds-eye view documentation and various diagrams for the rest of the code. At a line by line level, the code must be self documenting, using expressive names (*Open Sesame*), standard idioms(*Programming Idioms*), etc. Consider books like Smalltalk Best Practices Patterns [1] that teach principles of communicative programming.

Examples: After understanding the architecture overview, programmers often directly proceed to the code. Then they employ methods like *Examples are Documentation*. It is all too common for analysis and design diagrams to be out of date, so the best option is to *Just Ask*.

4.9 Just Ask

Because the real product of programming is the program, analysis and design does not keep pace with the code, customer requirements change for one reason or another, so that analysis and design documents get out of sync with the code. Even when there are up-to-date documents, explanations by other people are often more effective after a preliminary shared background is acquired.

Therefore: Just talk to the other programmers, and ask them to explain the design.

Examples: This is by far the most common method of reading programs. A proto-expert can become an expert faster if he participates in programming with experts, and interacts with them. When *Reader is Writer* and the program is evolving rapidly, people can share knowledge far faster than it can be written down. Especially when you practice pair programming, documentation is redundant to the pair, and needed only for the outside world.

4.10 Technology Testbed

Often programs use external frameworks that are not available in source form, or interact with operating system facilities. The quirks of these infrastructure facilities, such as configuration, installation, and so on are hard to understand and rationalize. Infrastructure bugs are usually hardest to track down. The readers job is simplified if the infrastructure can be studied in isolation with dummy programs.

Therefore: Create a technology testbed, and document it thoroughly. Keep track of product versions, updates, and patches from the technology source. Maintain “hello world” programs that are suitable for testing.

Examples: The classical C “hello world” is such a testbed. When you have run hello world, you know how to use the compiler. C++ compilers are known for their incompatibilities and mysterious behavior. So it is common to keep feature test programs to ensure that the programmer understands the language and the compiler. The TCP suite comes with programs like ping and ttcp (trivial tcp) that exercise features of the network.

4.11 The Usual Suspects

One of the strongest aids in reading a program is to know the culture where the program was born. Readers will automatically look in "the usual places" to understand code.

Therefore: Follow the style of names, program organization, and programming idioms of the culture.

Examples: Programs written by traditional C programmers include makefiles make targets like "install" "clean". Typically, directories will be used as packages, and header files export names. Idiomatic control structures shape the code. Programs in the lisp community are typically data-driven programs expressed as composed functions. They are not shy about recursive control structures, despite the presence of iterative ones. Overuse of iteration in a lisp program feels awkward. Over time, books and tools support these artifacts. For example, the C lint tool, similar tools that include the Scott Meyers [17] C++ guidelines.

4.12 Environment and Tool Idioms

Programming environments entail their own idioms. A skilled reader uses these without a second thought.

For instance, in interpretive environments, a common way to understand programs is to follow its execution in a debugger. They also make it possible to develop and test code in small bites, virtually line by line. For instance, in emacs, if you don't know what a function does, copy it from some other code and try it out It takes a while for readers without experience in an interpretive environment to adopt this style.

In a compiled environment too, you can use the compiler as a (slow) search engine. For instance, if you want to understand what a method does, first rename it to something improbable, and try to compile the code. The compiler will obligingly produce all callers that depend on the method. Then you go look them up.

Similar odd idioms are useful in debugging. On Unix, if you are compiling a program in one directory and executing it in another, make sure that no program with a similar name exists on the path prior to the compiled program. This seemingly trivial "tip" is extremely useful; more often than one would expect, I have rescued novices and (tired) experts with this one.

Regrettably, not enough of these idioms are documented. They are often too arcane, and prone to obsolence; tricks involving the old Unix debugger 'adb', for example, would scarcely be relevant now. On the other hand, every new environment has its dark corners. Increasingly, frequently asked questions lists and newsgroups on the Internet are the best avenues for such idioms.

5 Readers Culture

Typically, culture constitutes such things as program organization, use of the environment, naming schemes, and so on. The following patterns help a writer adjust to the programming culture.

5.1 The Bible

A programming culture usually gathers around languages or systems that are useful. The initial descriptions of such systems often set the tone of the culture, and become common knowledge.

Therefore: A new writer should seek out the source texts. Other texts include collections of advice, traps and pitfalls that become apparent after long use.

Examples: There are many examples: the Kernighan and Ritchie [10], C bible, followed by books from Andrew Koenig [13] and later Peter van der Linden [24], documenting best practices and pitfalls. Similar C++ pairs include Stroustrup [22], and Scott Meyers [17]. For Smalltalk, the combination might be the "Blue" Book [7], the Smalltalk image itself, and Beck [1].

5.2 Naming conventions

Naming conventions are important as shorthand, and sometimes to avoid obscure bugs. Often, tools in the environment will depend on them. Readers associate semantics with names in certain forms.

Therefore: Study the project guidelines for naming, and follow them. If in doubt, consult the local guru.

Examples: The C culture is full of such conventions. A particularly arcane one forbids the use of names with leading underscores, because libraries often use them. So user names with leading underscores can confuse the macro processor, linker, and so forth. Another such convention is to use all capitals for preprocessor macros. Beyond mere conventions, good names are critical in understand programs: see *Open Sesame*.

5.3 Programming styles

Programs may be written in different styles, such as functional, generic, or use architecture styles [21] like pipe-and-filter, implicit invocation, and so on. To maintain coherence as the program evolves, the changes should respect the styles.

Therefore: Where possible, refer to standard sources of architectural styles, such as Buschmann [3]. Garlan [21]. Use "bibles" that describe, for instance, functional or logic programming. Use idioms of the style in standard ways.

Examples: The C++ Standard Template Library [4] is intended to be an extensible library framework. It uses a functional style, in a way that elegantly extends the basic C++ imperative style. The papers written by the architects show how STL embodies a style called generic programming. Object oriented languages are usually versatile enough to emulate other styles such as functional programming. The Smalltalk collections library makes use of this style using Smalltalk blocks. Books such as Smalltalk best practice patterns [1] show how to emulate the "native" Smalltalk styles.

5.4 Program organization

Styles of program organization are sensitive to the culture and the kind of tools native to that culture. Often this knowledge is implicit within the community, existing only in the form of programs.

Therefore: Use older programs from the culture as templates for creating new ones.

Examples: In the emacs-lisp community, standard organization requires that the program consist of interactive functions to be invoked by the user, variables that the user can configure, hook functions that support minor function extensions and interact with other library programs. There are corresponding conventions: hook function names typically end with '-hook', user variables are defined with 'defvar' and documented in a certain way, interactive functions use the special form '(interactive)' and so forth. As time passes, these conventions become enshrined in tools. For instance, Smalltalk the language derives its organizing conventions like Categories and private methods via Smalltalk the environment. In emacs, the built-in help functions rely on the writer supplying documentation for the user-definable variables. Similar conventions in the C world were mentioned in *The Usual Suspects*. They too get enshrined in tools: in places where code follows the "K&R" conventions, you can search for function names easily, because they will be found one line lines whose first character is an open brace. Such conventions are arcane but useful.

5.5 Programming Idioms

We differentiate programming idioms from styles for convenience. The term idioms is applied to individual lines of code, whereas styles may characterize program architecture. Idiomatic code is readily understood, is often supported by tools, and helps avoid common mistakes.

Therefore: Use and disseminate programming idioms. Use examples from old code.

Examples: Idioms overlap with best practices and programming conventions. The diversity of the Unix world has led to various little languages their quirky idioms. For example, in C, use `if (1 == i)` instead of the more natural `if (i == 1)`. This idiom helps detect the common typing mistake where you type = instead of ==. Another idiom involves embedding macros as `do macro while (0)` so that the the semicolon after the macro is gobbled up. C++ has equally strange idioms, enough to fill books [17].

5.6 Organization culture

This pattern involves support documentation. It is included primararily for completeness, because I think proper discussion requires the context of organization issues discussed in organization patterns.

The program development organization needs documents that are important to readers other than programmers; see *Reader is Other*. Common documents include program Change Logs, Bug reports, Requirements, Project Planning, and others. Bug Reports and Change Logs arise out of necessity even in the hacker culture. Requirements and Project Planning depend on organization patterns.

6 Reading Styles

Different people use different kinds of documentation effectively. In the following, we relate known patterns of learning styles to different types of documentation. The connections established here between learning style and kinds of document are speculative, based on informal discussions with programmers. The examples, though, are from observing programmers at work, and there are no hard and fast distictions; people operate in different modes at different times.

6.1 Global

Global learners like to get the global picture first.

Therefore: Provide the reader with overall architecture diagrams, the history and the end goal of the project. It should be concise, and help predict the rest of the system.

Examples: After getting the gist of the architecture, some programmers will ignore documentation, try out their own designs, and match it against the implementation. If the design fulfills their expectations, they understand it immediately. If it does not, they may get argumentative.

6.2 Sequential

Sequential learners will consumer information patiently in the order presented.

Therefore: Such readers like reference manuals, lists, and books.

Examples: When asked to learn Emacs or Perl, such a person will head for the bookstore to buy the book, instead of mucking in the more chaotic documentation on the net, or trying out the program without any manual. In one case, after gaining some experience with C++, a programmer was seen to purchase the Annotated Reference Manual, read it from end-to-end, and acquire working knowledge of the arcane aspects of C++.

6.3 Inductive

Inductive learners generalize from particular experiences.

Therefore: They prefer instance diagrams and interaction diagrams as the starting point.

Examples: These programmers will work through examples, draw instance diagrams and interaction diagrams, and grow an understanding of the programs. I have observed such programmers have trouble with class diagrams: they intend to draw instance diagrams, draw them as classes, and then get confused. A simple remedy appears to work: name the instances "Bob" and "Fred", and they proceed.

6.4 Deductive

Deductive readers like to go from generalizations to particulars.

Therefore: Such readers prefer class diagrams and role models. They will deduce the instance diagrams and interactions. The names of the classes will be critical to a generalist. They prefer documentation as structured essays like the various pattern forms.

Examples: These people complain about "implementation details" and would prefer not to be bothered with them, unless they are well thought out and connected to the overall architecture.

6.5 Kinesthetic

Kinesthetic people learn from doing.

Therefore: These readers will prefer examples and code that they can change. They will prefer code and written descriptions to diagrams.

Examples: Kinesthetic programmers find it difficult to draw and see the need for diagrams. They would rather you read code, or look at a formal specification. They have to learn to explain themselves in diagrams.

6.6 Visual

Visual learners prefer diagrams to specifications and code.

Therefore: They use diagrams not merely to communicate, but use them for thinking.

Examples: Diagrammatic notations are essential for communication because they avoid details of code. So even the kinesthetic programmer learns to diagram. But visual people draw diagrams for themselves and shy from specifications. In academia, there is abiding interest in programming with pictures.

7 Patterns at work in different cultures

"A body can get used to anything, even being hanged", goes an Irish folk saying. The study of programming culture shows fascinating adaptations to a wide variety of situations. In the examples, we see how the patterns presented above appear in concert.

7.1 Literate Programming

Literate programming takes the view that programs should be literature. The programmer writes a document, English interspersed with code, where code is merely precise restatement of the English. Knuth, the inventor of literate programming, hoped that people would then publish code much as they publish books, and people would buy them for Sunday afternoon reading. This would "solve" the documentation problem, because you would produce documentation with code. Let us see the interplay of the patterns in this setting.

Readers Purpose In this culture, if *Reader is User*, his needs are met. If *Reader is Proto-Expert*, he would certainly be glad about the documentation. However, if *Reader is Writer*, then the overhead of crafting English together with the code involves very high overhead. This is exacerbated because of the speed of changes to the implementation.

Reading Methods The reading methods can in principle be supported in Literate Programming. However, the *Environment and Tool Idioms* are violated by introducing the unusual code-with-text format. None of the common environments or tools is geared to support literate programming. Two other reading patterns *Code must communicate* and *Just ask* render literate programming redundant, at least at a line-by-line level.

Reader and Program Culture Literate programming introduces a new element into *Program Organization*. Programming cultures will adapt to changes if they are beneficial, but failure to satisfy some of the other patterns means that cultural inertia will triumph.

Reading Styles Literate programming with the current tool enforces the explanation style concocted by the developer on the reader. Some readers may have to adapt to the writers style.

Typically, literate programming has seen use in situations where *Reader is User* and and *Reader is Not Writer*, so that the code stabilizes. These cases include Knuth's "TEX The Program" [11], Fraser and Hanson's [5] book on compiler construction. There are reports of industrial practice [19], to bring users up to speed — *Reader is proto-Expert.*

The Java libraries, where again *Reader is User*, practice semi-literate programming. They do not write documents; they write programs and strip out the comments, inverting Knuth's intent.

7.2 Extreme Programming

Where Literate Programming says you should write documents not code, Extreme Programming says write *Code that communicates*, but do not document beyond necessity. See [9] for further details. Here, the documentation patterns play out as follows.

Readers Purpose In this culture, no *Reader is User.* If *Reader is Proto-Expert*, he is encouraged to *Just ask* and read *Code that communicates.* If *Reader is Writer*, he is encouraged to follow *Pair Programming.*

Project history is implicit among the programming team.

Reading Methods This culture encourages non-documentation reading methods. *Open Sesame* is used to ensure good names. *Examples and Tests are Documentation*; in fact, tests may be written before the actual programs. *Code Must Communicate*, and project best practices must be followed. A reader learns a program using *Just ask*, together with *Pair Programming*

Culture based reading methods dominate, with the patterns *The usual suspects* and *Environment and Tool Idioms.*

Reader and Program Culture A reader is supposed to be conversant with the culture. He should follow *The Bible*, *Programming Idioms*, *Naming Conventions*, rigorously. As an oral culture, Extreme Programming relies on shared values and idioms for communication.

Reading Styles Readers rely on adaptation to the culture to overcome reading style concerns.

Patterns such as test and example driven programming have appeared in other projects. Culture-steeped communication between programmers is common in the Unix, Emacs, TEX and other "free software" communities.

7.3 TEX Programming

TEX is a word-processing system designed by Knuth (also the inventor of Literate Programming). With probably the best automatic layout and word breaking algorithms, meticulous attention to aesthetics especially for mathematical formulae, TEX is perhaps the best documentation system for scientific writing. However, the macro language of TEX is another story:

- It is call-by-text language, with much of its intelligence in different ways for substituting arguments in `def`ined macros.
- A peculiar form of `if` and `let` statements
- A few integer registers and integer operations

- Built in text and layout algorithms
- Other control structures, including loops, are self-modifying programs.

The community that has emerged around this system exhibits a strange mixture of our reading and writing patterns and culture.

Most *Readers are Users*, however since document markup may involve minor forms of programming, often *Readers are proto-experts* to various degrees. The quirks of language mean that it is hard to be more than a proto-expert. This leads to the evolution of a few *wizards*, who perform acts of magic such as emulating structured programming within TEX[14], getting TEX and postscript to cooperate to produce versatile figure drawing macros [25] and so on. However, virtually all these feats are individual performances; the pattern *Reader Is Writer* is rare.

As for reading methods, there are countless books on using TEX (and its successor, LATEX. Except the TEXbook, none that I have ever seen explain the TEX language. But the source of the TEXbook, and that for its successor LATEX are available, and so are other examples, so the pattern *Examples are Documentation* exists. Recently, many of them include documentation targeted to the user.

Otherwise, communication about TEX programs is conducted entirely within the culture established by the TEX infrastructure and the TEXbook. There are instances of the patterns *Programming Idioms*, *Naming conventions*.

The readers need skills in all reading styles; it would appear that predominantly visual readers would not make any headway with TEX the language.

7.4 Unix Programming

For typical public Unix programs, the patterns are

Readers Purpose For Unix programs, all *Readers are Casual Modifier*, at the very least. Some *Readers are Writers*. Documentation is sparse, and most communication is based on shared culture. Projects are passed from writer to reader over time, as the writer loses interest and moves on.

Reading Methods Non-documentation reading methods are well supported. *Units are cohesive*: the directory as unit philosophy is well established in the culture. *Examples are Documentation*; examples also double as tests. You are encouraged to *Just ask*, usually over email. Culture based reading methods are also supported. *The usual suspects* and the *Environment and Tool Idioms* are rigorously followed.

Reader and Program Culture Books such as K&R, Kernighan and Pike have set the standards. Code will follow *Naming Conventions*, *Program Organization*, and *Programming Idioms*. Much new code emerges by modifying old programs, and the conventions are perpetuated.

Sometimes, small but interesting changes happen: in the C world multi-word names were separated by underscores; over the years, probably under the impact of Smalltalk and object-orientation, this has changed to use mixedCase names.

Reading Styles The rare programmer will include ascii art depicting data structures.

The other parts of Unix, such as its libraries and utilities have followed patterns where *Reader is User*. The libraries have by and large suitable documentation for a user. Books have supplied the necessary examples.

The story of the utilities is one of early success followed by failure (in a sense). They violated the reader expectations discussed in *Reader is User*: documentation became inconsistent and unwieldy among different utilities and Unix versions. The text processing utilities and the different shells acculturated a variety of inconsistent options and features like barnacles. There were several reactions against this. Usually, users stuck with a very few, well known options, and pretend the barnacles under the waterline did not exist. The GNU project scraped the barnacles, standardized a help system for the options, wrote extensive documentation, and regained some much needed order. The Perl community enshrined selected barnacles in a single language, removing inconsistencies, for instance, among the different regular expression syntaxes.

7.5 Research Programming Cultures

In research cultures, the program is not the product; the products are papers describing innovative architecture, optimizations, and new programming models. So communication about the program is informal. Virtually all *Readers are Writers*. They practice varying degrees of *Pair Programming*. More importantly, they practice *Pauseful Programming*, stepping back from the code and describing its novelties. Predominant reading methods are *Overview*, *Inverse Analysis and Design*, *Why This Not That*, *Examples are Documentation*, and *Just Ask*. The *Environment and Tool Idioms* typically are those of the Unix culture. The Readers culture is also typically that of Unix. The Reading styles will often decide research directions. People with visual tendencies develop animation and case tools and do performance studies; people with kinesthetic, deductive tendencies seem to have a feel for formalisms.

7.6 Product Programming Cultures

The observations below pertain to companies that build products in the consumer market, not turnkey or fixed-bid systems. The companies I have observed do not follow formal processes.

Readers Purpose In this setting, some *Readers are proto-Experts*, some *Readers are Writers*, and some are non-programmers. Documentation that includes competitor analysis, potential customers, overall architecture, and realization plans is available, satisfying the patterns *Reader is Other*. Documentation of the program itself is sparse; most communication is based on shared culture.

Reading Methods Usually, the patterns include *Overview*, *Units are Cohesive*, *Here Be Dragons*, *Examples are Documentation*, *Just Ask*, *Technology Testbed*, *The Usual Suspects*, and *Environment and Tool Idioms*. The patterns *Code must Communicate*, *Why this and not that*, are not followed as much as you might hope. Testing is usually separate from programming, even in different departments. Comprehensive Unit Tests and conscious use of Test Cases as documentation is rare.

Reader and Program Culture Program culture patterns are followed; they are almost second nature to mature developers. Organizations also evolve local guidelines, and *Organization Culture* about issues like bug reports and change logs.

Reading Styles Whiteboard diagrams are common. On occasion, they are transferred to drawing tools.

8 Conclusion

This paper presented patterns about the expectations of those who read programs. The patterns were classified according to the purpose of the reader, his culture, different methods of reading, and learning styles. We then presented instances of these patterns at work in different programming cultures.

This study offers a perspective on documentation, and identifies the kinds of documentation that different readers need. This is only the first step; as yet no opportunity for applying these patterns has presented itself. The hope is that designers of programming teams, documentation support tools and others will be able to use them. Perhaps further pattern mining may reveal better techniques for educating proto-experts, or show how to design documentation that is appropriate for the context, so that it is easier to maintain project continuity when programmers leave.

Other programming cultures include Visual Basic or Excel programming, Macintosh programming, the process-centered cultures espoused by organizations such as SEI, and the cultures of fixed-bid projects. A study of these cultures should also prove interesting.

9 Acknowledgements

I thank Kent Beck and John Heintz for their comments.

References

[1] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[2] Grady Booch. *Object-oriented Analysis and Design with Applications.* Benjamin/Cummings, California, 1994.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture : A System of Patterns.* John Wiley & Sons, New York, 1996.

[4] Alexander Stepanov David R. Musser, Atul Saini. *Stl Tutorial & Reference Guide : C++ Programming With the Standard Template Library.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[5] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[7] Adele Goldberg and David Robson. *Smalltalk 80 : The Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[8] Watts Humphrey. *A Discipline for Software Engineering.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[9] Ron Jeffries, Kent Beck, and Ward Cunningham. *Extreme Programming*. www.armaties.com, 1997.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language : ANSI C Version*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[11] Donald Ervin Knuth. *Tex : The Program*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[12] Donald Ervin Knuth. The errors of TEX. *Software Practice and Experience*, 19(7):607–685, July 1989.

[13] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[14] Leslie Lamport. *Latex : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[15] Doug Lea. *Concurrent Programming in Java : Design Principles and Patterns*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[16] Robert Cecil Martin. *Designing Object-Oriented C++ Applications : Using the Booch Method Discipline*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[17] Scott Meyers. *Effective C++ : 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.

[18] David L. Parnas. A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering*, February 1986.

[19] Norman Ramsey. Literate programming on a team project. *Software Practice and Experience*, July 1991.

[20] Mohlalefi Sefika, Aamod Sane, and Roy Campbell. Monitoring compliance of a software system with its high-level design models. In *18th International Conference on Software Engineering*, 1996.

[21] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.

[23] William Strunk and E. B. White. *Elements of Style*. Allyn and Bacon, 1995.

[24] Peter van der Linden. *Expert C Programming : Deep C Secrets*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[25] Timothy van Zandt and Denis Girou. Inside pstricks. *TUGboat*, 15(3):239–246, September 1994.

[26] Joseph M. Williams. *Style: Ten Lessons in Clarity and Grace*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.