Software Cultures and Evolution

Working effectively with a legacy software program requires software engineers to view it in the context in which its developers created it.

Václav Rajlich
Wayne State University

Norman Wilde
Michelle Buckellew
University of West Florida

Henry Page
Micro Systems

To work effectively with legacy code, software engineers need to understand a legacy computer program's *culture*—the combination of the programmer's background, the hardware environment, and the programming techniques that guided its creation.

Software systems typically pass through a series of stages.[1] During the initial *development* stage, software developers create a first functioning version of the code. An *evolution* stage follows, during which developmental efforts focus on extending system capabilities to meet user needs. During the *servicing* stage, only minor repairs and simple functional changes are possible. In the *phase out* stage, the system is essentially frozen, but it still produces value. Finally, during the *close down* stage, the developers withdraw the system and possibly replace it.

Most of the tasks in the evolution and servicing phases require program comprehension—understanding how and why a software program functions in order to work with it effectively. Effective comprehension requires viewing a legacy program not simply as a product of inefficiency or stupidity, but instead as an artifact of the circumstances in which it was developed. This information can be an important factor in determining appropriate strategies for the software program's transition from the evolution stage to the servicing or phase out stage.

DEFINING SOFTWARE CULTURE

Tylor[2] defined human culture as "...that complex whole which includes knowledge, belief, art, morals, law, custom, and any other capabilities and habits acquired by man as a member of society." The programmers who created legacy software used the knowledge and customs appropriate for the time and circumstances in which they were working. Just as an anthropologist needs background information to correctly interpret human cultures, having information about the environment in which developers produced a given piece of legacy software helps software engineers understand it.

Imagine a software engineer attempting to reengineer the statements in the example in Figure 1, drawn from Convert, a Fortran program from the 1970s. A recent computer science graduate raised on Unix or Windows would recognize that the program reads in some parameters and then prints them out. But the first write statement using the 9000 FORMAT is meaningless. Why does a line containing just the digit "1" precede the output? However, a programmer from the 1970s is likely to recognize this as output for a line printer. Such printers typically used the first column of output as a carriage control character, interpreted by special hardware and used with perforated paper tape to control printing. A "1" typically forced a skip to the top of a new page, a "0" forced a double space, and so on. This convention was reasonable given existing hardware because it reduced the number of lines transmitted from computer to printer.

This is a typical example of the conventions that impact comprehension of legacy programs. Although programmers in modern computer installations rarely encounter line printers, successful reengineering of such code requires knowledge of the line printer convention. Reengineering this program could require discarding the first character of every record.

Changes in programmer background, hardware

0018-9162/01/$10.00 © 2001 IEEE

```
      WRITE (IOUT,9000)
      READ (45,9010,IOSTAT=IOS) IV,IDONT,ISORT,IASCI,KPRNT,IBIN,FACT,
     *  PTSS,PSTT
      WRITE (IOUT,9020) IV,IDONT,ISORT,IASCI,KPRNT,IBIN,FACT,PTSS,PSTT
9000 FORMAT ('1')
9020 FORMAT ('  IV = ',I5,'  IDONT = ',I5,'  ISORT = ',I5,
     *   '  IASCI = ',I5,'  KPRNT = ',I5/,'  IBIN = ',I5,'  FACT = ',
     *   F5.1,'  PTSS = ',F5.1,'  PSTT = ',F5.1)
```

Figure 1. Short quiz for recent computer science graduates: What does the first line of this code do?

properties, problem-solving techniques, software processes, and programming practices have resulted in several major shifts during the history of mainstream software engineering. The concept of what comprises a "good" program has changed radically over time.

CULTURE CONTRASTS

Our case studies demonstrate that the software culture strongly affects both the comprehension and evolution of software for large systems.[3] In these studies, we used a legacy Fortran system from the 1970s and a more modern C application from the early 1990s.

The Fastgen geometric modeling system is a suite of Fortran 77 programs that developers can use to construct models of solid objects such as vehicles and aircraft from primitives such as triangles, spheres, cylinders, donuts, boxes, wedges, and rods. The US Air Force uses Fastgen to model the interactions between weapons and targets by tracing rays representing explosions or projectiles.[4] Convert, a Fastgen preprocessor that consists of a single Fortran 77 source file of 2,335 lines (raw line count), is representative of the 1970s Fortran software culture. Originally developed in 1978 for a mainframe environment, Convert expands simplified geometric model input and transforms models into the formats other tools require. Software developers have updated and maintained Convert many times to keep pace with the introduction of different hardware platforms.

NCSA Mosaic,[5] one of the first widely distributed and used Web browsers, was developed in the National Center for Supercomputing Applications at the University of Illinois during the mid-1990s. After four years of work and several released versions, NCSA suspended work on this application in 1997. However, source code is still publicly available (http://www.ncsa.uiuc.edu), and programmers still use it for case studies of program comprehension techniques. Mosaic, which is fairly representative of the mid-1990s C culture, is a single program consisting of approximately 100,000 lines of well-structured, fairly well-commented C code in approximately 180 C source files.

Table 1 summarizes the main differences between Fastgen and Mosaic.

System partitioning

Fastgen programs share files of intermediate results. Convert typically preprocesses a geometric model and then feeds the output into other programs for simulation, analysis, or display. Early software often used this kind of partitioning, obviously motivated by the limited memory available in most computers. This places a greater burden on the user, who must execute each program individually and run them in the correct order with the correct parameters. One benefit is that the developers wrote very accurate user documentation for the individual Fastgen programs.

In comparison, 1990s developers implemented Mosaic as a monolithic program in which users either select options from a menu or the software executes them in response to the input data.

Modularity

In keeping with modern practices, Mosaic's developers implemented a large number of C functions,

Table 1. Summary of Fastgen and Mosaic cultural differences.

System	Partitioning	Modularity	Control flow	Obsolete program plans
Fastgen 1970s Fortran	System of programs that share files	Large noncohesive subroutines Global data in large common blocks	Unstructured, tangled, many GO TOs	Input/output in batches Scratch files Data packed into integers Binary files
Mosaic 1990s C	Single monolithic program	Small cohesive functions Little use of global data	Structured	

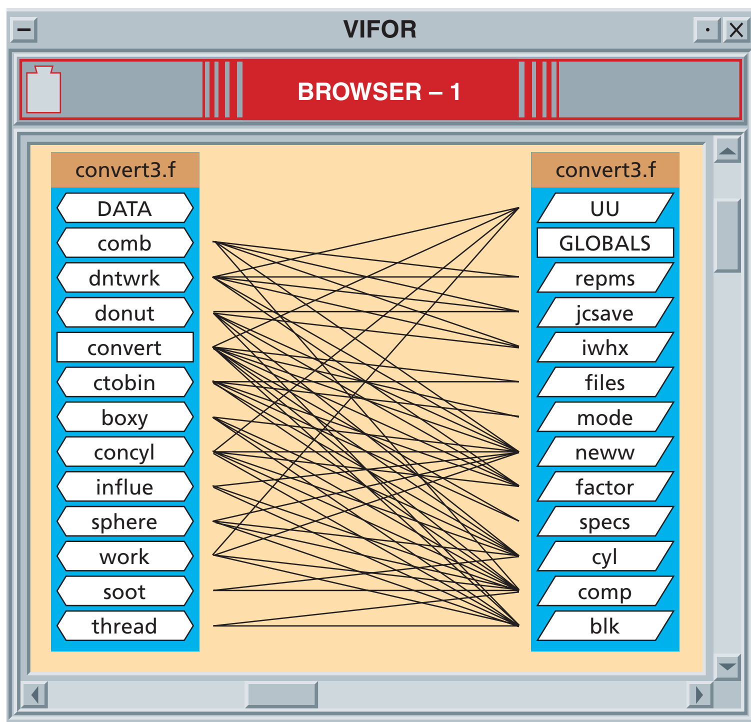

Figure 2. Common subroutine use in Convert. In this display from the Vifor Fortran analyzer program, subroutines appear in the left column and the program's common blocks appear in the right column. A line indicates that the subroutine references the block.

typically with high cohesion. Relatively short descriptions summarize each function's purpose. In contrast, Fastgen's developers implemented it before programmers accepted the importance of cohesion. Subroutines are large, and they mix many unrelated issues, which makes comprehension difficult because the subroutines no longer serve as logical chunks of program code. Convert holds most of the data in large global common blocks that many subroutines share, as Figure 2 shows.[6] In comparison, Mosaic uses few global variables that are well-coupled with the relevant processing.

Convert's poor modularity had a substantial impact on program comprehension. In C, programmers use the functions as the basic chunks for program comprehension—understanding each function is a step in the comprehension methodology. Programmers could not do this with Convert because the subroutines lack the necessary cohesion.

Control flow

Although modern control flow structures are available in Fortran77, Convert relied extensively on using `GO TO`. The developers may have implemented the first version of Convert in an earlier version of Fortran and then ported it into Fortran77. The frequent use of `GO TO`s complicates reconstructing program plans and makes code reading difficult, particularly for today's programmers, who are accustomed to structured control flow. For example, the main loop that reads and processes a geometric model's components uses `GO TO`s that jump both forwards and backwards over hundreds of lines of code. This creates a complex structure that seems at first to be totally arbitrary. In contrast, Mosaic expresses program plans in structured control flow constructs and rarely uses C's `goto`.

Obsolete program plans

Obsolete program plans solve hardware and operating system problems that no longer exist. The line printer carriage control character is one example, but Convert provides others.

Input/output batching. Convert reads and processes geometric models in batches of 200 records. Batching made execution in old mainframe operating systems more efficient because the system swapped a job that was doing I/O out of memory. A simple read-process-read-process cycle was very inefficient because it caused multiple swaps. However, I/O batching provides little benefit on more modern PC systems. The batch cycle complicates the code because processing breaks at the arbitrary 200th record, not at any logical point such as at the end of a model component.

Consider how this might affect an attempt to reengineer Convert. The software engineer who does not understand the plan's purpose might continue using the 200-record cycle, just in case it conceals some important functionality of the program.

Scratch files. Convert opens at least seven scratch files for intermediate results as soon as it starts processing. Because old computers had limited local memory, Convert stores all data of large or unknown size-such as parts of a geometric model-on a scratch tape, and continually rewinds and rewrites these tapes. This greatly complicates the understanding of Convert's "mirroring" feature. Mirroring simplifies input of a geometric model by allowing the user to input just one half of a symmetric component, while the system generates the other half. The program writes the component to be mirrored to the scratch tape and then reads it back in. On the second pass, the system reexecutes much of the Convert code, but this time with data from the scratch tape instead of the original input. Understanding mirroring relies on understanding the scratch tape plan.

Packing data into an integer. In early computers, input data was largely constrained to punched cards, which imposed a limit of 80 characters per record. To save record space and main memory, programmers sometimes used complicated encodings to pack data together. For example, Convert packs the following data into a single integer for each geometric component:

- First digit: a code for the kind of component (triangle, donut, sphere, box, and so on);
- Second and third digits: the component thickness in hundredths of an inch;
- Fourth digit: a space code, to specify what is adjacent to the component; and
- Sign: the modeling mode (volume or plate).

Convert contains code that picks this integer apart to extract the individual items.

Binary input/output. A primary use of Convert is to convert a geometric model from ASCII to binary format for use by other programs in the Fastgen suite. Because the models can be quite large, using binary input reduced both storage space and I/O time significantly on earlier machines. However, this is less beneficial for current PCs because much more memory is available.

The programmers who implemented Convert solved legitimate problems of their time, and often solved them well. However, hardware and operating systems properties have continued to change so radically that these plans are now an obsolete and dysfunctional reminder of the past. Mosaic also contains many complicated program plans, but they do not appear to be as obsolete at the moment. In a few years, Mosaic too is likely to appear strange and dated to newly trained programmers.

RECENT CULTURE CHANGES

Antipatterns are an example of a recent software programming change. William Brown and colleagues[7] described a collection of antipatterns that programmers should avoid. Controversially, one of the antipatterns is functional decomposition, which was widely used in software architecture as late as the early 1990s. The Mosaic system we describe uses functional decomposition. Brown's book clearly states that functional ecomposition is a bad strategy and that new projects should not use it. Of course, this offers little help to the programmer who must maintain a program from the functional decomposition culture.

Harry Sneed[8] argued that program code is the result of the programmer's thought patterns, which in turn depend on the programmer's experience. Understanding and maintaining the programs requires deciphering these patterns.

Code decay, the accumulation of factors that make software changes difficult, drives a software system from the evolution phase to the servicing phase. Loss of expertise is the main reason for code decay, and culture change is a major contributor. Recent computer science graduates invariably received training in the C/Unix, object-oriented, or component-based culture, remaining unaware of programming methods used several decades ago. Older programmers may still retain knowledge of these methods, but as time progresses, this source of expertise is disappearing.

Yet organizations still use programs implemented decades ago, and—as the Y2K problem showed—in some cases, these programs are mission-critical. The large investment these systems represent makes continuing evolution a necessity, but understanding the code becomes progressively more difficult. Consequently, organizations are forced into a servicing or phase-out strategy that emphasizes minimal change, with a potentially high cost of losing business flexibility.

Sometimes software culture differences are so large that programmers who are evolving old programs must learn the old cultures. As a first step, programmers need to document, through interviews and code study, some of the main practices that were considered acceptable in different periods and environments. To prepare them for encountering legacy code written for a very different environment, the education of newly trained programmers should help them develop an appreciation and tolerance for old cultures.

Hardware and operating systems have changed so radically that legacy software programs are now a dysfunctional reminder of the past.

Programmers sometimes try to reengineer old code to bring it up to date. However, simply translating an old program into a new language or restructuring the code won't help. Obsolete program plans that solve problems that no longer exist are a troublesome cultural difference that translation or restructuring won't solve. Because these plans are variable and have complicated interactions, programmers cannot identify and remove them automatically. Documentation of the old culture will facilitate the task of programmers who must perform this task manually.

The problem of software cultural change will not go away in the future. For example, the National Aeronautics and Space Administration is planning very long term space probes, which will have missions lasting 50 years or more. Such space vehicles will undoubtedly require a large volume of software, both onboard and as ground support. Presumably, programmers will use today's best software best software practices for these programs, but it would be foolish to assume that the software will not require major evolution over such a time period. In the year 2050, where will NASA find trained software engineers willing to immerse themselves in the archaic ways of the early 21st century? ✷

Acknowledgment

This work was partially supported by the Air Force Office of Scientific Research under grant number F49620-99-1-0057.

References

1. V.T. Rajlich and K.H. Bennett, "A Staged Model for the Software Life Cycle," *Computer*, July 2000, pp. 66-71.
2. E. Tylor, *Primitive Culture*, Harper and Row, New York, 1958.
3. N. Wilde et al., "A Case Study of Feature Location in Unstructured Legacy Fortran Code," *Proc. 5th European Conf. Software Maintenance and Reengineering*,

IEEE CS Press, Los Alamitos, Calif., 2001, pp. 68-76.
4. E.D. Aitken et al., *A Guide to Fastgen Target Geometric Modeling: User's Manual*, ASI Systems, Fort Walton Beach, Fla., 1993.
5. NCSA Software Development Group, "NCSA Mosaic Home Page," http://www.ncsa.uiuc.edu/SDG/Software/Mosaic (current Aug. 2001).
6. V. Rajlich et al., "Vifor: A Tool for Software Maintenance," *Software Practice and Experience*, Jan. 1990, pp. 67-77.
7. W.J. Brown et al., *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, New York, 1998.
8. H. Sneed, "Human Cognition of Complex Thought Patterns—How Much Is Our Perception of the Present Determined by Our Experience of the Past?" *Proc. 6th Int'l Workshop on Program Comprehension* (IWPC 98), IEEE CS Press, Los Alamitos, Calif., 1998, http://cds.unina.it/~iwpc98/keynote.html.

Václav Rajlich *is a full professor and former chair in the Department of Computer Science at Wayne State University. His research interests include software change, evolution, comprehension, and maintenance. Rajlich received a PhD in mathematics from Case Western Reserve University. He is a member of the IEEE Computer Society and the ACM. Contact him at vtr@cs.wayne.edu.*

Norman Wilde *is a full professor of computer science at the University of West Florida. His research interests include software maintenance and program comprehension. Wilde received a PhD in mathematics and operations research from the Massachusetts Institute of Technology. He is a member of the IEEE and the IEEE Computer Society. Contact him at nwilde@uwf.edu.*

Michelle Buckellew *is a software engineer at Lockheed Martin Integrated Systems in Orlando, Fla. Her work involves developing software for the Joint Air-to-Surface Standoff Missile (JASSM). She received an MS in software engineering from the University of West Florida. Contact her at Michelle.Buckellew@lmco.com.*

Henry Page *is a senior software engineer at Micro Systems, Fort Walton Beach, Fla. Page received an MS in software engineering from the University of West Florida. He has participated in several case studies in software maintenance and program comprehension. Contact him at hpage@gomicrosystems.com.*