How long can you wait for CMMI Compliance?

Manage your software development in guaranteed compliance with the CMMI® NOW!

processMax includes all the necessary policies, procedures, guidelines, criteria, templates, and forms in role-based, step-by-step instructions, ready for use—everything you need for CMMI compliance. Integrated with robust document management and workflow, processMax is the intranet web-based solution for effective and efficient management of your software development projects. processMax also includes a complete measurement and reporting capability. Measurement collection and report generation are fully automated—giving managers real-time, fact-based insight into all aspects of project progress. processMax is the proven solution—as verified by more than 50 successful independent appraisals.

www.pragmasystems.com

pragma Systems Corporation, 1810 Samuel Morse Reston, VA 20190
877.838.PMAX E-mail: info@pragmasystems.com
processMax is a registered trademark of pragma SYSTEMS Corporation.
CMMI is a registered mark of Carnegie Mellon University. Copyright © 2005 pragma SYSTEMS Corporation.
Just what is “Software Archaeology” anyway?

Is it just an analogy? A useful metaphor? A software engineering process? A software engineering discipline? A technology? Or perhaps, a combination of these things? It appears that the software community has not yet converged on a singular meaning for the term and its scope as evidenced by the statements of our current authors and others cited in my references.

Hunt and Thomas[1] claim that archaeology offers a pretty good analogy for the activities that one performs when tasked with fixing or revising legacy code and, in general, addressing the common problems of trying to understand someone else’s code. The major difference between software archaeology and real archaeology is that the objects of our efforts do not have to be a thousand years old. They use the terminology of real archaeology to describe useful techniques for working with someone else’s software.

Booch[3] defines software archaeology as “the recovery of essential details about an existing system sufficient to reason about, fix, adapt, modify, harvest, and use that system itself or its parts.”

The authors featured in this issue of Software Tech News offer additional, yet differing, perspectives on the subject. In the first article titled “Reverse Engineering and Software Archaeology”, Ralph Johnson (co-author of the seminal book, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995) asserts that “software archaeology is a point of view, not a set of technologies” and that, although it uses the same tools as reverse engineering, it has a much longer-range purpose. He claims that software archaeology is not a means to an end, but rather an effort to learn about software just to understand it. He goes on to describe some projects that exemplify this perception and then discusses the importance of reverse engineering and software archaeology before presenting lessons he has learned along the way.

In the next article, titled “Software Archaeology”, Andy Schneider, defines software archaeology as “the systematic study of past software systems by the recovery and examination of remaining material evidence, such as code, test, and design documents.” He then focuses on the shortcomings or limitations of the software archaeology and history metaphors. He outlines some principles of history that are applicable to software systems and then uses those principles to identify key techniques, which, he asserts, can assist the inexperienced software archaeologist in their quest. He concludes with a conviction that software archaeologist and

An advertisement appearing on the House of C website[2] is titled “Software Archaeology” and contains the following text, “We specialize in making things work, particularly integrating old data with new interfaces. It doesn’t matter how old, creaky, and poorly documented the product or system you’re fighting with - we can figure it out.”

continues on page 2
software historian are merely roles adopted at various points in time by the software professional in their everyday work, and that they are, therefore, merely useful metaphors for examining best practices and nothing more.

Grace Lewis, Edwin Morris and Dennis Smith, from the Software Engineering Institute (SEI), assert that software archaeology investigates and rehabilitates legacy systems so that their architecture can be discovered and their code reused. Their article titled “Migration of Legacy Components to Service-Oriented Architectures,” discusses what it means to create services from legacy components and then summarizes a project where the SEI helped a program office make decisions about migrating legacy components as services within a service-oriented architecture (SOA). They also discuss the Service-Oriented Migration and Reuse Technique (SMART) developed by the SEI as a method for evaluating legacy components for their potential to become services in an SOA.

The last article, an interview with Sellam Ismail, Curator of Software for the Computer History Museum (CHM), presents yet another perspective on software archaeology, focusing on software preservation. The primary mission of the CHM has been to preserve hardware; software was often donated with the hardware but its preservation was an afterthought. Ismail describes what the CHM is now doing to organize and maintain the software they have, and he talks about its uses. While others interested in archaeology focus on studying the artifacts in order to build a better system, or use some of the components elsewhere, the CHM focuses on getting the software to run in its existing state on equipment for which it was built. Much of the value in this software preservation comes from the capability it provides to get at data that was recorded in outdated proprietary formats. CHM gets few requests from software archaeologists seeking to study the software in its holdings. Most of the artifacts are executable code for operating systems and printers and tape readers, etc., not design and requirements documents.

It seems that we are approaching a fork in the road on our software archaeology journey. One camp is looking way down the road and way out in time, seeking to preserve classic software, to leave a historical record for future generations to study, and focus on what artifacts to preserve. The other camp is pragmatic, focusing on tasks that enable them to understand existing software (without regard to age) so that they can use it (or its parts), fix it, or get ideas from it to develop other software. The following figure highlights the characteristics, concerns and focus of these two camps.

Regarding software preservation, the largest obstacle, at the moment, is the lack of a central source code repository. Booch[3] estimates that cumulative source lines of code (SLOC) will reach 750,000,000,000 by the end of 2005. What would it take to store all of that, together with its related artifacts in such a way that it is useful for archaeological digs? Another major obstacle is the lack of a significant user base. Kaplan[4] lays out the frustrating history of software preservation attempts and failures noting, in retrospect, that two fundamental issues could not be satisfactorily addressed.

“First, while all participants agreed that software history is important, that awareness of it should be raised, and that it must be documented, participants simply could not identify a solid user base of any justifiable proportion. Second, as participants stated over and over again, ‘preserving software’ is much more than an act of accumulation. It means conserving, organizing, researching, cataloging, and presenting materials in ways that researchers can use. To do otherwise is simply hoarding. And no individual institution or consortium of institutions has been able to balance these two issues.”

Some people view the Internet Archive as a potential model for establishing software archives. It already stores screen shots of Web sites and other artifacts of the digital age. Adding source code to the mix would be easy enough, says staff software preservationist Simon Carless. Unfortunately, legal issues and aging copy-protection continues on page 3
tion mechanisms make it difficult to provide a decent record of historic programs. Carless says the Digital Millennium Copyright Act (CDMCA) clouds the current preservation landscape.[5]

Some authors, including Booch[3], Hunt and Thomas[1], suggest that as developers, we should leave a legacy for future archaeologists (anyone else who needs to examine the software at any point in time) who may some day look at our “polished” code and say “What were they thinking?”, by creating those artifacts that future reviewers will need.

Will there be a convergence among the software community about what software archaeology is and is not? Is convergence important or even necessary? Regardless of the answers, and regardless of the age of the software, it is clear that we can effect the outcome of the archaeologist by increasing our awareness of the information needs of those looking at our software in the future, and creating helpful software artifacts when we build our code. However, the ROI of such efforts has yet to be determined.

References

About the Author
Ellen Walker, a DACS Analyst, is currently developing a series of publications on software “best practices” as part of the DACS Gold Practice Initiative. She has spent the past 20 years as a software developer in various roles spanning the entire software life cycle including project management of multiple business process re-engineering efforts within the DoD community.

She is also experienced with assessment initiatives such as the Capability Maturity Model for Software (CMM-SW) and the quality management practices of the New York State Quality Award program. Ellen has an MS in Management Science (State University of New York (SUNY) at Binghamton), and bachelor degrees in both Computer Science (SUNY – Utica/Rome) and Mathematics (LeMoyne College).
I often study software systems written by someone else. When I was an undergraduate, I learned compilers by reading some books and a couple of Pascal compilers. When I came to the University of Illinois as a young professor in 1985, I wanted to learn how object-oriented programming changed the way people developed software, especially how it enabled the development of reusable software, so I studied a variety of frameworks. I developed some with my students, but I learned just as much studying ones that other people developed. For the last ten years, my focus has been as much on software patterns as on frameworks, and I have continued to study other systems. I can learn several systems in the time it would take to build one. Since my job is to learn about software rather than to produce it, I spend as much time studying existing systems as developing new ones.

Although there are many differences between my life as an academic and the life of a typical software developer, one similarity is the need to understand someone else’s software. Most programmers spend most of their time working with software written by someone else. Even when they think they are starting a new project, they usually end up spending a lot of time trying to figure out one of the packages that they are using. Documentation is rarely good enough, so they will usually have to learn the design by experimentation. If they are using open source software then they can learn the design by reading the code, though this is making a virtue of necessity, since open source software usually doesn’t have much documentation.

Studying software is serious business and should be treated seriously. Unfortunately, many developers think of it as a necessary evil. They are paid to write new software, and they consider time spent reading old software as wasted. However, old software is valuable. It is valuable to users and it is valuable to software developers. Like anything of value, we need to understand it so we can take care of it and make use of it. As software lasts longer, the ability to learn an existing software system becomes more important. Learning the design of an existing system is much more than a necessary evil; it is crucial to modern software development. Learning a software system can be hard work, but sometimes it is rewarded with flashes of insight and moments of beauty.

One of the signs that old software has become important is that we have invented language to talk about it. We talk about a “legacy” project, which deals with software that we inherited from someone else, in contrast to a “green field” project, in which we get to choose a new architecture and make the basic design decisions. We have to “reverse engineer” software with inadequate documentation. We have to “reengineer” software when needed changes are too large for a simple patch. People use terms like “software archaeology” to emphasize either that there are no documents or experts who can explain the system, or that the system is very old. But software archaeology is different from extreme reverse engineering. This article will show the difference between reverse engineering and software archaeology and explain how both of them are important to software developers, but in different ways.

Reverse Engineering

Software archaeology is closely related to reverse engineering and reengineering, but it is not the same. Wikipedia gives a popular definition of reverse engineering when it says that reverse engineering is figuring out what a system does without source code. But the IEEE Technical Council on Software Engineering (http://tcse.org/revengr) gives a broader definition that is more often used by textbook authors, which is that reverse engineering is any process of analyzing a system and creating a representation at a higher level[1]. While this includes converting a binary to C, it also includes developing a UML class diagram for a Java program, making a flow chart for a Fortran program, or documenting the interfaces of a package.

*continues on page 5*
Reengineering

Reengineering is “the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form” [1]. Reengineering almost always requires reverse engineering. This is partly because few systems are documented as well as they should be, and partly because the reason that the system is being reengineered is because something was not taken into account during its design, and if it was documented then it would have been taken into account.

Reverse engineering does not imply that there is no documentation. It implies that there is not enough of the right kind of documentation. Object Oriented Reengineering Patterns [2] by Demeyer, Ducasse, and Nierstrasz is an outstanding book on software reverse engineering and reengineering. One of its main points is to use interviews with users and maintainers when possible and to read available documentation. Reverse engineering is not only needed when you have a binary written in an unknown language, it is also important when you are trying to make a change to a large system that is different from any previous changes. Reverse engineering is sometimes needed even for systems with an active development community and thousands of pages of documentation.

Software Archaeology

“Software archaeology” conjures up images of people poring over documents in a forgotten language, looking at broken artifacts whose purpose is unknown. But, software archaeology is more than looking at old code. Most programmers add features to old code every day. Age is not always related to obscurity. Some systems built last year are more obscure and hard to understand than many old systems. Most current programmers were not even born when Unix was first released to the public, yet a lot of the code in modern Unix kernels dates to that time and Unix Release 6 is easily understood by modern Unix kernel programmers.

The tools and techniques used for software archaeology are the same as those used for reverse engineering. But while reverse engineering has an immediate purpose, software archaeology has a much longer-range purpose. We reverse engineer a system because we want to improve or replace it. Maybe we just need to improve the documentation to improve maintenance costs. Maybe we want to change a part of the system and do not know it well enough to plan the change. Maybe we want to make a new system but want to make sure that it is as good as the old so we need a specification of the old system. All these are examples of reverse engineering. Reverse engineering is a means to an end. Eventually we will stop it and move on to “more important things”.

Archeologists unearth old civilizations because they are curious about how people lived and they want to better understand how we became the people we are. They are not looking to solve current problems like overpopulation, though books like Collapse [3] show that archaeology can tell us a lot about these problems. Archaeology is general and long-term, and is definitely not a means to an end.

Similarly, software archeologists are trying to learn about software, not to solve an immediate problem.

Software archaeology is a point of view, not a set of technologies.

We believe that better understanding of software will lead to better software. We don’t study existing systems to blindly copy them, but to learn from them. We study systems to learn patterns, to learn different architectural styles, to see principles and techniques in action.

One of the best examples of software archaeology is Grady Booch’s project to write a handbook for software architects (http://www.booch.com/architecture). The core of this project is studying a hundred important software systems, such as Photoshop, the Google search engine, AWACS, and 5ESS. The book will have a few pages about each project, describing the key architectural features of each. It will also

continues on page 6
have a collection of architectural patterns that Grady Booch expects to discover along the way. This is a very challenging project, but one that will make a tremendous impact if it is successful.

A more mundane example is a recent project of mine to learn more about security. I decided to study a secure system to see why it was so secure. So, Munawar Hafiz and I looked at qmail, a mail transport agent that is a competitor to sendmail. Since its introduction in 1997, qmail has a perfect security record. We compared its design to that of sendmail and found a number of differences that led to better security. The designs and our analysis of what makes qmail so secure are reported in Munawar Hafiz’s MS thesis [5].

Since qmail is a system that is heavily used and that is probably understood by many people, you might argue that it was not a case of software archaeology. However, you’d be wrong. Like most open source software, qmail doesn’t have much documentation. The only documentation that comes with the system is man pages. They described the interfaces to the components of qmail. We learned a little from web pages written by others, very little directly from the author of qmail, and learned the most about the design by reading the code. It was a classic case of reverse engineering, but we didn’t do it to add a feature or fix a bug. We reverse engineered qmail so we could learn why it was so secure and learn lessons we could apply to other systems.

**Why Reverse Engineering Is Important**

All programmers need to know how to reverse engineer a software system. They need those skills when they join an on-going project, or when they want to learn to use a new component. Software engineering texts tend to treat reverse engineering as a minor issue, something that only a few people need to know. In reality, reverse engineering is a crucial skill, probably more important than being able to design a new system. Maintenance programming is all about figuring out what a system does and how to change it. New programmers are often assigned to maintenance until they prove themselves. So, if they can’t figure out what an existing system does, they will never be given the chance to design a new system. Most new systems are built from reusable components, so even if a programmer is designing a new system, it is important to be able to study the components and to infer how to use them. On large projects, crucial design decisions are usually made by a small group, and most of the team spends more time figuring out the existing system than designing new features. Thus, many developers are more likely to use reverse engineering skills than design skills.

During the last few decades, maintenance and software reuse have both increased. However, the quality of software documentation has not increased that much, so the need to reverse engineer software has increased. It is good that we have more software that is worth maintaining. The cost of building good software that is worth keeping is that we have to maintain it. The amount of maintenance will probably go up. Unless there is a break-through in the quality of documentation, this means that reverse engineering will be even more important in the future.

**Why Software Archaeology Is Important**

Software archaeology is less important than reverse engineering. Reverse engineering has immediate benefits while software archaeology has longer-term benefits. Nevertheless, the benefits of software archaeology are real and can be large.

One of the main reasons I study systems is to discover patterns. Patterns are always discovered by looking at existing systems. Often, these systems are not learned for the purpose of discovering patterns. Instead, the discovery of patterns is a side-effect of learning how to use a library or change a system. For example, the four of us who wrote *Design Patterns* had all been interested in frameworks design and had studied many frameworks. We were not studying the frameworks to learn patterns, but we had learned them anyway. However, most software developers will not work on more than one or two systems per year, and many developers will work on a system for several years before moving to

---

*continues on page 7*
another. Moreover, these systems might only use well-known patterns. To find new patterns, you have to look in new places. This is why I studied qmail. None of the systems that I knew were very secure. If I wanted to learn about security then I would have to look for other systems.

Another reason to study a system is to confirm a pattern or to learn more about it. When I’m trying to write a pattern, people often tell me about other systems that have the same pattern, and I’ll look at them to see whether it really is the same pattern and to learn variations and complications. Design is about trade-offs, and every pattern has its dark side. I try to find a system where a pattern is abused so that I can learn when not to use it.

There are many other reasons to study existing systems. You can learn about a class of programs by reading a few examples, as I did with compilers. You can learn an architectural style or how to achieve a particular software quality, like reliability or portability. The best way to fight a patent that you think is improper is to find prior art, and that often requires digging through old software. If you want to understand how software evolves, you not only have to look at old software, you have to look at several versions. Practicing software archaeology is a great way to learn new design ideas. It is not for everybody, since many software developers just want to focus on getting the job done. But it is important for software developers who want to get better at doing their job.

**Lessons Learned**

So far, I have been fairly philosophical and have focused on definitions rather than tools and techniques. I’d like to leave you with two concrete suggestions that follow from what I’ve said. The first is how people can learn to read software. The second is how to make software easier to read.

There are few courses on how to study software, and most schools do not teach it. The University of Illinois (where I teach) is not an exception. Several of us try to teach code reading as a side-effect of our courses by giving students software packages that they are required to understand and to change. But we do not spend much time explaining how to approach a large system and other professors might teach the course differently.

Some of our graduates are experts at learning large software systems. These are usually those students who have worked on several open source software projects. They have learned several large systems and know that they can do it again. But these students almost always picked up their skills outside of class.

Therefore, if you want to make sure that developers know how to learn software systems, you can’t depend on their college education. I recommend two books on the subject of studying software; “Code Reading: The Open Source Perspective” by Diomidis Spinellis [6] and “Object Oriented Reengineering Patterns” by Demeyer, Ducasse, and Nierstrasz [2]. The first book is more code-oriented and Unix and C oriented. The second is focused on larger systems, object-oriented systems, and on reengineering a system once you have learned it. People with a lot of experience reading software will probably know most of the material in these books, but they will still be able to pick up some useful techniques. Books are only a start, of course. Learning to read software takes practice. However, the books are a guide that will make learning faster.

Is reverse engineering necessary? Perhaps software developers can provide documentation that is good enough that the people who come after them do not have to read the software, but only need to read the documentation. Although this is possible in theory, in practice good documentation is too expensive for all but the most popular software. Developers try to automate documentation with tools like JavaDoc. These tools are useful, but tend to produce detailed, low-level documentation. Developers still have to reverse engineer these documents to learn the purpose and the architecture of the system.

Reverse engineering will always be necessary. Even though more and more software is long-lived, most software dies young. It

*continues on page 8*
doesn’t make sense to develop expensive documentation for software before we know whether it is needed. So, we will continue to have under-documented software and people will continue to add documentation to software long after it has been written.

The most important documentation is the high-level documentation. It is relatively easy to reverse engineer flow charts and UML diagrams. There are tools that help automate them. But the high-level documentation is more about the intent of the designers than the actual code, and so is hard to derive from the code. If a system is well-designed and there is good high-level documentation, then it is relatively easy to figure out the low-level details from the code. But the high-level documentation is hard to produce without close communication with the developers.

High-level documentation is more valuable than low-level documentation. Low-level documentation is easy to produce, but it quickly becomes out of date. High-level documentation requires more thinking, but it is less likely to become obsolete as new features are added. Low-level documentation tends to be large and so, often, it is not read. Newcomers are much more likely to read a 20 page high-level document than a 200 or 2000 page low-level document. So, high-level documentation is more likely to be correct and is more likely to be read.

Reverse-engineering is always going to be an important skill. However, we can build our systems in such a way that they are easier to reverse-engineer by documenting the goals and assumptions of the system and its architecture. This would not only make the systems more maintainable, it would help future generations of software archeologists.

References

About the Author
Professor Johnson is on the faculty of the Department of Computer Science at the University of Illinois. He is the leader of the UIUC patterns/Software Architecture Group and the coordinator of the senior projects program for the department. His professional interests cover nearly all things object-oriented, especially frameworks, patterns, business objects, Smalltalk, COM and refactoring. He received his PhD and MS from Cornell University and his BA from Knox College.

e-mail: johnson@cs.uiuc.edu
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL 61801
(217) 244-0093
(217) 244-6869- fax
Are you getting **All** you can from your software investment?

The DACS ROI Dashboard

**Impact on Quality (% defect reduction)**

75th Percentile: 90

Oklahoma City Air Logistics Center, Test Program Set and Industrial Automation line observed the defect density decrease 91%, from 3.3 to 0.3 Defects per KSLOC, while achieving CMM Level 4.

**Fact Information**

- From 1999 to 2000, Oklahoma City Air Logistics Center, Test Program Set and Industrial Automation line observed the following changes:
  - Total development time was 13 months measured before the improvements and 12 months measured after the improvements.
  - Number of defects observed per kilo lines of code (KLOC) decreased from 3.3 defects per KLOC measured before the improvements to 0.3 defects per KLOC measured after the improvements.
  - Total person hours measured for the improvement efforts was 730 thousand person hours measured after improvements.

**Background:**

- Data collection was conducted over a period of 12 months.
- The defect density was measured using the industry standard defect measurement method.

**Source:**


**Access the DACS ROI Dashboard!**

http://www.thedacs.com/databases/roi/

**Technologies Covered:**

- SEI CMM/CMMI
- SEI Team Software Process (TSP)
- SEI Personal Software Process (PSP)
- Inspections
- Reuse
- Cleanroom
- And Many More!

**Graphs Showing Impact Of Software Technologies On:**

- ROI
- Productivity
- Quality

**Summarizes Facts From Open Literature**

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

http://iac.dtic.mil/dacs
Software Archaeology

By Andy Schneider, BP Plc. and Pete Windle, BJSS Ltd.

Introduction

For us to discuss software archaeology productively we must first define it. A naive definition can be derived thus:

**software archaeology** Pronunciation Key (sôftwâr ärk-l-j)

*The systematic study of past software systems by the recovery and examination of remaining material evidence, such as code, tests and design documentation.*

Using archaeology as a metaphor allows us to reason about how we recover and examine material evidence relating to software projects. As useful as this is, it is by no means the entire story. Many times there exist individuals to interview, and primary and secondary evidence to investigate. These types of activities owe more to historical practice than they do to archaeology. One of the reasons for this is that existing communities in organizations ensure that the investigation of systems is as much about people, existing practice and precedent as it is about tools and artifacts ‘dug out of the ground’. This emphasis on people and historical practice is less evident in existing software archaeology literature.

As an illustration, consider the UK constitution. The UK constitution defines the form, structure, activities, character, and fundamental principles by which UK society, law making institutions and government institutions operate. Famously, though, it is not written down. Instead it consists of a number of bodies, namely the monarchy, the executive, parliament and the civil service, supported by both precedent and law. A student of the UK constitution is rather like someone trying to understand a software system. The student has to talk to experts, study paperwork, identify ‘urban myths’, analyze past behaviours and cope with the fact that, whilst they investigate it, the constitution is changing. The authors find this process, the process of historical study, to be as compelling an analogy as archaeology when considering practical techniques for studying software systems.

With this in mind, this paper will take a more historical and people orientated perspective. Firstly, we will outline some of the limitations of the archaeology and historical metaphors. We will then go on to outline certain principles of history, as applicable to software systems, and then use these principles to drive out key techniques for understanding systems. Hopefully we will leave you, the reader, with practical techniques and another way of thinking about understanding systems.

The Trouble with a Metaphor

It is important that any metaphor is seen for what it is – a starting point or a framework for describing the properties of the object of interest. It is not to be used as a reference or to define truth. One can say that a hierarchical software structure is like the branches of a tree; one cannot then infer that software is made of wood. Fowler goes into this in some detail[3].

So then, what are the limitations of our archaeological and historical metaphors?

Software is unlike archaeology in many ways. One of the main contentions that the authors have is that archaeology is essentially an academic subject. Its practical techniques are there to divine information from the past using the artifacts left behind, and this is for no reason other than the discovery of knowledge. In contrast, the authors believe that the role of software archaeologist is useful as part of a software practitioner’s everyday skill set. As Booch notes in [1], software archaeology provides the ability to “[recover] essential details about an existing system sufficient to reason about, fix, adapt, modify, harvest, and use that system itself or its parts.”

Also, a large part of archaeology is dedicated to maintaining the condition of a site or dig. Software is inherently easy to protect from external damage – even if you don’t have a source control system, you can always burn a copy onto a CD.

We contend, therefore, that historical techniques are just part of the toolkit to be applied to software problems. The main difference continues on page 10
here is one of timescale – the information to be gleaned from software historical studies in our context is usually much closer to the present, giving more access to primary sources.

The differentiation between archaeology and history is no doubt one that the respective disciplines have debated extensively. For our purposes, we will define history as “the study of the past”. At that point, archaeology becomes that subset of history that is largely concerned with the study of the concrete artifacts left behind by a subject of interest. History encompasses archaeology then, and adds more of a bias towards considering the human aspects of the subject in hand.

Applying that to our software paradigm, we afford equal weight to the surrounding culture as to its artifacts.

Types of Evidence

Historians subdivide evidence into primary and secondary sources.

Primary sources can be material things such as artifacts, tools, constructions or remains (although in software, “material” can include intangibles – source and object code, log files, etc). Alternatively, they could be contemporaneous written records – documents, specifications, e-mails.

Finally, interviews conducted with the protagonists or direct witnesses to events are themselves primary sources. The oral history of software teams tends to be a very rich source of information (albeit some of it apocryphal or cargo cult[2] invocation). This is for several reasons, but mainly because most software projects are delivered with a number of undocumented deltas to the original requirements and/or design.

Cargo cult science is a term used by Richard Feynman in his 1974 Caltech [http://en.wikipedia.org/wiki/Caltech> commencement address to describe work that has the semblance of being scientific, but is missing “a kind of scientific integrity, a principle of scientific thought that corresponds to a kind of utter honesty”. Feynman cautioned that to avoid becoming cargo cult scientists, researchers must first of all avoid fooling themselves, being willing to question and doubt their own theories and their own results, and to investigate possible flaws in a theory or an experiment (from http://en. wikipedia.org/wiki/Cargo cult science).

Secondary sources are texts and documents which are derivative of the actual event – not actually part of the phenomenon but later analysis or interpretation of it. As we will discuss later, the production of secondary sources is an important part of the software historian’s milieu.

When considering these sources it is often useful to evaluate their quality before spending significant time reading them. Some questions a good historian may ask when judging software related sources would be:

- Why was the source created?
- Does it look like the source was ad-hoc and rushed or well thought and pre-planned?
- Did the author actually work on the system?
- Who was the target of the document?
- Was the source intended to be for public consumption or private to the team?
- Was the document in current use?
- How long after the system/module was written was the source produced?

Exploring the Past

A good historian recognises that the past is different from the present. This means avoiding researching a system with the aim of reinforcing existing expectations but rather, allowing the system to teach you how it really is. For example, when encountering a method called createInstance you may assume from the name that it creates instances of some structure or object and therefore pass it by. However, the method could be doing anything. There are examples of poor naming all over the IT world. Worse, by passing over the method you may have missed a lesson; that ‘instance’ in this system refers to some specific concept (such as a point in time) rather than the meaning you have ascribed to it.

With the best will in the world, you cannot hope to know all there is to know about a system and its past, so it is worth remembering that the past you see is only part of the

continues on page 11
Be conscious of signifiers

In semiotics [5], people distinguish between the signified, the concept being communicated and the signifier, the token that represents the concept. The signifier and the signified together are known as a sign. The much used example of this is the White Hat in old cowboy movies. Viewers of these movies know that the White Hat stands for the Good Guy. If you are not familiar with this sign, or apply it outside of the genre then you may miss something or worse, misunderstand the communication.

Pattern names are signifiers and so are software terms such as ‘instance’. So, be conscious that sets of signs used in the past may not be the same as those you are using. Or, put more bluntly, ‘assumption is the mother of all screw-ups’.

picture. Consider the functionality someone developed in the past. Ask yourself “was that the functionality that was intended?” – moreover, ask yourself “was that the functionality that was actually wanted?” If you assume that the code correctly implements the required functionality then you may be missing a bug or a requirements gap.

The functionality of the system you are studying is and has been part of a wider system, combining software, users and support (at a minimum). Good historians always consider multiple perspectives and contexts before drawing conclusions.

The past is not static. When seeking to understand a system, recognise that you can both examine it now, at this instance, and you can also examine it over a series of points in time. A historian will always look at what came before and after when assessing a particular point in time. Consider collating data such as:

- Complexity metrics per source file per point in time. This allows you to see how broad characteristics of the system have changed over time. Correlating large changes (above and beyond a general trend) with stories from people who know the history of the system can give key insights into how and why the system evolved as it did.
- Code size per source file per point in time. This metric allows you to draw conclusions about the amount of refactoring that has occurred. Areas where the amount of code rises, then falls, then rises again can indicate ‘tending of the garden’ and, as such you can use this information (in concert with other data) to help prioritise where you will spend your time.

Other useful metrics are:

- Check-ins per source file. This identifies the most volatile source files, and therefore those under active update. This can target initial stabilisation and cure strategies.
- Defects per subject area. Use this to demarcate areas with a relatively high defect density. Areas with high defect densities will likely only become more defective if they are changed. Use the data you have gathered to steer around such problem areas (unless you have good reason to enter that is).

Whilst you obtain this data on a per-source file basis, be sure to choose metrics that can be rolled up to per-module or per-package views. A broad view is the most useful when initially starting out. Tools such as Maven1 can help here, as they provide a cockpit through which you can observe and drill down into metrics.

It doesn’t always do what it says on the tin

On encountering some code written in the past, seek to both understand the code and determine if the code correctly implements the functionality. If you cannot determine the latter, be sure to explicitly recognise that fact. Sources of functional definition (apart from a requirements document) are often the support team or the users.

1 http://maven.apache.org
Points in time

Gather metrics that vary over time to obtain an understanding of the evolution of the system. Correlate specific events in the results with any spoken or written history of the project to enrich your understanding of why things have evolved the way they did.

Cultures

Examining a system is inextricably linked with gaining familiarity with the group of people who create (or created), use and support the system. There is much to be gained from examining some of the popular ethnography literature (e.g. [6], [7] and [8]). There isn’t space in this article to spend much time on the subject, but we have included a limited number of practices to ensure a reasonable breadth of coverage.

Many cultures deny that an outsider can ever understand them. They maintain that their treasure is only for members; this is particularly the case in some tribes. Recognise that groups that form around legacy systems can exhibit many tribal characteristics. Understand that whilst the tribe may be the final authority on their perceptions and the terms they use, the past and its evolution are open for you to look into. Because of the tribal nature, it is important to use their language, to be initiated and to adopt their customs - without becoming pickled\(^2\) (i.e. ‘going native’) or failing to question the customs.

Each group of people will have their own interpretation of

Speak the lingo

Paying attention to the terminology used by existing legacy teams allows you to communicate with them on their terms. This helps communication, improves rapport and shows you are actively listening.

history; listening to only one (e.g. the development team) can result in inaccurate bias creeping in to your research process.

Seek Multiple Sources

Acquire multiple perspectives on the system to reduce the sensitivity of your finding to bias.

You can extend this practice to observing how conversations with individuals vary over time. For example, one of the authors was interviewing someone on a legacy team about the deployment process. The process sounded incredibly complex and the reasons for the complexity seemed valid, but somewhat tortuous. The author went back for repeated conversations and noticed as time went on that the explanations changed or simplified. This indicated that

Question Customs

Responses to questions of the form ‘well, it just works that way’ are often clues of where to spend your time when examining a system. These responses may point to Cargo Cult\(^2\). Identifying these misconceptions can be useful pointers to fragile or complex areas of the system that need understanding well.

Gather Objective Data

Where possible gather quantifiable data to validate hearsay. E.g. use profile tools to determine where there was more to the story than was being told. The differences pointed to where to explore and the author changed his tack as a result.

Update and Extend the Existing Literature

This is generally not a valid option in historical circles – a historian who merely rewrites other people’s books to “correct” interpretations would not be seen as contributing much.

When understanding software systems, however, there is usually the potential to at least use existing documentation to bootstrap your work.

Catalogue existing design and requirements documentation. Do not treat it as gospel truth – consider potential levels of accuracy, bias and how up-to-date it has been kept. Consider which parts are still useful and which approaches are worth extending.

It may be that a slash and burn strategy is required should the existing documentation be woefully inadequate or glaringly inaccurate. Not every document contains enough value to be worth recovering.

Conversely, it may be that although inaccurate in places, some existing documentation is “good enough” – it is still worth briefly recording where it is not, however.

Cultures have a strong interest in their own pedigree. This can

\(^2\)The system thinker Jerry Weinberg once observed that “cucumbers get more pickled than the brine gets cucumbered”.

Data & Analysis Center for Software (DACS)
result in cultural chauvinism, where certain groups compete to show they are the ‘oldest’, ‘fastest’, ‘youngest’ etc. To do this they will often, sometimes without knowing it, manipulate history - to show things in a certain light. It is important therefore to rely not only on artifacts from a team, but also actively engage with external stakeholders to gather data. For example, if you are interested in performance, ask the support team; measure the response times, do not just rely on anecdotes from the team. See Multiple Sources above.

Documenting the Past

There are several different techniques for documenting the past. This section details some of the forms of output and methods of production that the authors have found useful.

Conclusion

The authors have looked at applying some of the principles of history to the study of existing software systems, and have tried to outline some practices to assist the student of those systems.

Whilst the archaeological metaphor looks initially compelling, we believe that the essentially human nature of software and its comparatively contemporaneous nature make the history metaphor a more interesting fit.

Finally, it should be stressed that both the software archaeologist and software historian are merely roles that are adopted at one time or another by the software professional throughout the course of their everyday work, and should not be taken as anything more than useful metaphors for examining best practices in the field.

References


Start With A Clean Sheet

In a way, the mirror image of the previous point. When approaching a large and unknown architecture with many moving parts, it is generally best to start with a blank sheet of paper.

This can then be iteratively filled in as your understanding expands – the blanks within the diagram then become a map of where further investigation is required.

On an old map of the world, these places would be marked “Here Be Dragons”, perhaps.

Good diagrams for forming these high-level views include:
• Activity diagrams – especially for core transaction pipelines.
• Sequence diagrams – for understanding interactions between object graphs.
• Package and/or class diagrams – for establishing an idea of structure.

Don’t Trust Their Focus

The subjects that the original authors of sources considered interesting may not be the events that you are interested in now. A corollary of “The Past Is Not The Present”.

The low-level concerns of the parish council rarely make front page news, but may well generate screeds of text.

In context, then, the original documentation may wax lyrical at some length about their optimistic locking model. Providing that it works, you may never need to touch it and therefore understanding the intimate detail may be a waste of your time. Of course, this is providing that it works.

The key has to be to understand your own requirements at that moment, and search out the documentation that meets them.

Make a Timeline

Draw a history timeline showing significant events that occurred during the lifetime of the development and use this to remind you of the context within which people worked.

This helps because it forces you to think about why things happened and when they happened - it’s a framework for thinking about the past in a dynamic fashion.

Know What You Don’t Know

Documenting what you don’t know allows people who follow you to determine where they need to spend their time when investigating the system and to determine between validated fact and hypothesis.

continues on page 14
Migration of Legacy Components to Service–Oriented Architectures

By Grace Lewis, Edwin Morris, and Dennis Smith, Software Engineering Institute

1. Introduction

Software archaeology investigates and rehabilitates legacy systems so that their architecture can be discovered and their code reused. An increasingly popular approach to software archaeology has been to leverage the value of legacy systems by exposing all or parts of it as services within a service-oriented architecture (SOA).

A service-oriented architecture is a collection of services with well-defined interfaces and a shared communications model. A service is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through this loosely coupled, often asynchronous, message-based communication model [2, 4]. Systems or applications that are called “service-based” use the functionality provided by these services as part of their mission. For example, when a person makes a reservation through Travelocity, what appears to be a single web-based application actually involves the complex orchestration of a set of “services” from a number of sources. These services may include authentication, flight schedules, reservations, hotel searches, and credit card validation.

SOAs offer the promise of enabling legacy systems to work together, presumably without making significant changes. In theory, a developer would simply have to create a well-defined common interface to the legacy system so that it could offer its services to other systems or applications. This migration is a complex task, particularly when services will execute within a tightly constrained environment.

SOA migration tasks can be considered from a number of per-

Software Archaeology

Continued from page 13.

Questioning.html

Further Reading

About the Authors

Andy Schneider is the Lead Integration Architect for BP’s Oil Trading & Supply systems. Andy is an industry exponent of agile development techniques with over 15 years of relevant experience in the IT industry. He has extensive experience in application and systems architecture, project management and software delivery. Andy regularly publishes papers and presents on subjects such as technical leadership and systems development at conferences such as OOPSLA and SPA.

Email: Andy Schneider [andrew_schneider_uk@yahoo.co.uk]

Pete Windle is a technical project delivery specialist, consulting for BJSS (http://www.bjss.co.uk). He lives in a software development commune in Islington, London.

Email: Pete Windle [peter.windle@bjss.co.uk]
Migration of Legacy Components
Continued from page 14.

In practice, constructing services from existing systems is neither easy nor automatic.

perspectives including that of the user of the services, the SOA architect, or the service provider. This paper focuses on the service provider. Section 2 briefly discusses what it means to create services from legacy components. Section 3 summarizes a recent engagement where the SEI helped a program office make decisions about migrating legacy components as services within an SOA. Section 4 outlines the Service-Oriented Migration and Reuse Technique (SMART) method for evaluating legacy components for their potential to become services in an SOA. Section 5 provides conclusions and discusses next steps.

2. Creation of Services From Legacy Components

Enabling a legacy system to work within Web services is sometimes relatively straightforward. Web service interfaces are based on well-known standards and are set up to receive messages, parse their content, invoke legacy code, and optionally wrap the results as a message to be returned to the sender. Many modern development environments provide tools to help in this process, and commercial organizations are employing these environments to expose their business processes to the world.

However, characteristics of legacy systems, such as age, language, and architecture, as well as of the target SOA, can complicate the task. This is particularly the case when migrating to highly constrained SOAs such as those being proposed for some DoD systems. DoD migrations to SOAs will likely rely less on automation, and more on careful analysis of the feasibility and magnitude of the effort involved that goes beyond just the technical aspects.

Such an analysis needs to consider:

1. Requirements from potential service users. It is important to know what applications would use the services and how they would be used. For example, what information will be exchanged? In what format?

2. Technical characteristics of the target environment, such as messaging technologies, communication protocols, service description languages, and service discovery mechanisms.

3. Characteristics of the architecture of the legacy system such as dependencies on commercial products or specific operating systems.

4. The effort involved in writing the service interface. Even if it is expected that the legacy system will remain intact, code must receive the service request, translate it into legacy system calls, and produce a response.

5. The effort involved in the translation of data types, which may be small for basic data types and newer legacy systems, but very large in the cases such as audio, video, graphics, and other complex data types, or in legacy programming languages that do not provide capabilities for building XML documents.

6. The effort required to describe the services so that they can be discovered and used appropriately. This may require information about qualities of service (e.g., performance, reliability, and security) or service level agreements (SLAs).

7. The effort involved in writing initialization code that prepares the service to take requests, and operational procedures that must be followed for deployment of services.

8. Estimates of cost, difficulty,

---

1The most common (but not only) form of SOA is that of Web services, in which (1) service interfaces are described using Web Services Description Language (WSDL), (2) payload is transmitted using Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP), and optionally (3) Universal Description, Discovery and Integration (UDDI) is used as the directory service [4].

2XML (eXtended Markup Language) is currently the most common format for message payload within SOAs.

continues on page 16
and risk. The information gathered in the previous points should provide for more realistic estimates.

3. Support for Program Office Plans for Migration to SOA

The SEI recently worked with a DoD program office to analyze the potential for migrating a set of legacy components from a command and control (C2) system to form services for an SOA. The program office recognized that if a selected set of components from their C2 system were converted to application domain services (ADS), they may have applicability for a broad variety of purposes. Our role was to perform a preliminary evaluation of the feasibility of this conversion.

We initially met with the government program office and the contractors who had developed the system. At this meeting we were given an overview and history of the systems, migration plans, and the drivers for the migration. We were also given a brief orientation to the target SOA along with system documentation. This initial meeting was followed up by in-depth interviews and architectural analysis of portions of the system. The following paragraphs summarize what we learned about the target SOA, the current system, the gap between the current state and that required for the SOA, and our suggestions to the program office.

Understanding the Target SOA

The target SOA was studied through an analysis of available documentation and through a meeting with the developers. The target SOA is currently under development using a variety of commercial products and standards, along with significant custom code. The effort is focused on satisfying a number of specific quality attributes important to the DoD, such as performance, security, and availability. In order to meet these needs, the SOA will impose constraints on potential services. Because the SOA is still under development, the specifications for how to deploy and write services had not yet been fully defined. The target SOA is illustrated in Figure 1.

Figure 1 shows that the SOA includes common services (CS) that are to be used by user applications and ADSs. The SOA owns the common services. The environment allows for a set of application domain services (ADS) which derive their requirements from user applications. Program offices are invited to submit proposals for services to meet these requirements, either by building them from scratch or by migrating them from legacy components. These requirements then need to be analyzed in detail and matched to existing functionality to determine what can be used as-is, what has to be modified, and what has to be new development.

Even though the full details of compliant services for the target SOA have not yet been worked out, the target SOA imposes a number...
Migration of Legacy Components

Continued from page 16.

of constraints on organizations that are developing ADSs from existing legacy components. Some of the constraints/requirements for developers of ADSs include:

1. An ADS needs to be self-contained, that is, it should be able to be deployed as a single unit. In this specific target SOA, services need to be stand-alone and of small granularity so that they can be deployed as needed on standardized and often limited-resource platforms such as handheld devices. In a legacy component, functionality that has been identified as part of a service needs to be fully extracted from the system, including code that corresponds to shared libraries or the core of a product line.

2. In the target SOA, an ADS has to be able to be deployed on a Linux operating system. As a result, Windows-based legacy components could be a problem, especially if there are dependencies on the operating system through direct system calls or if there is a dependency on commercial products that are only available for Windows.

3. All services will share a common data model and all data will be accessed through a Data Store common service. Services will no longer define internal data and all data will be defined as part of the common data model. An ADS will use the Discovery common service to find and connect to other services.

4. If the ADS will rely on other services, code to discover and connect to these services will have to be written. Once the service is developed it needs to be advertised. Other applications that wish to use this service will perform a discovery on the available services and choose which service(s) they desire to use.

5. An ADS will use the Communications common service for communicating with other services. The target SOA provides tools for generating data readers and data writers that will take incoming and outgoing data and format it accordingly.

Understanding the Existing Capabilities

We met with the contractor and program office representatives to learn about the system, focus our investigation on a limited number of legacy components, and to select criteria for further screening. The current system, written in C++ on a Windows operating system, had a total of about 800,000 lines of code and 2500 classes (the rough equivalent of modules in the object-oriented paradigm). In addition, the system had dependencies on a commercial database and a second product for visualizing, creating, and managing maps. Both commercial prod-
ucts have only Windows versions.

The program office had completed a preliminary identification of potential services that could be built from components of the legacy system. Using the screening criteria, we selected seven of these services, containing 29 classes, as our focus since they offered potentially high payoff for our effort. Together with the contractor and program office representatives, we developed characteristics for analyzing the potential reusable components. These characteristics reflected both the base characteristics provided by the Options Analysis for Reuse (OAR) technique developed at the SEI and our knowledge of the necessary characteristics of services operating within the target SOA [1].

The characteristics included:
- Size
- Complexity
- Level of documentation
- Coupling
- Cohesion
- Number of base classes
- Programming standards compliance
- Black box vs. white box suitability
- Scale of changes required
- Commercial mapping software dependency
- Microsoft dependency
- Support software required

The characteristics provided input on potential reusable components. These characteristics form the basis for the more detailed analysis discussed below.

**Analyze the Gap**

Given the known and projected constraints of the target SOA, we performed a preliminary analysis of the legacy components to determine their suitability for reuse as services and the amount of effort and risk that would be involved. This analysis consisted of three parts: 1) a gap analysis of the changes to the legacy components that would be necessary for migration to the SOA, 2) an informal evaluation of code quality, 3) an architecture reconstruction to obtain a better understanding of the set of undocumented dependencies between different parts of the legacy system. The results of these analyses allowed us to define a service migration strategy based on the risks due to the unknown future state of the target SOA. These analyses are outlined below.

During the gap analysis we considered the candidate legacy components in terms of the characteristics that were developed earlier. The characteristics provided input on level of difficulty and risk factors. We identified dependencies of the selected classes on the commercial mapping software, the commercial database, and Windows. The contractor provided estimates for converting the components into services, based on a set of simplifying assumptions on the actual make-up of the target SOA and the final set of user requirements.

A summary of this initial analysis of converting the selected components to services is illustrated in Figure 3.

Using the existing contractor, the level of difficulty of making these changes would be low to

<table>
<thead>
<tr>
<th>Services</th>
<th>Number of Services</th>
<th>Number of Classes</th>
<th>Size (LOC)</th>
<th>Effort (MM)</th>
<th>Level of Difficulty</th>
<th>Level of Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected Services</td>
<td>7</td>
<td>16</td>
<td>16,163</td>
<td>12.5</td>
<td>Low to Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Base Classes</td>
<td>4</td>
<td></td>
<td>2,199</td>
<td>4</td>
<td>Low to Medium</td>
<td>Low</td>
</tr>
<tr>
<td>“Coupled” Classes</td>
<td>3</td>
<td></td>
<td>5,388</td>
<td>1</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Total</td>
<td>23</td>
<td></td>
<td>23,750</td>
<td>17.5</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Figure 3. Results of Initial Analysis

3Base classes are those from which the classes in the service are inheriting properties. “Coupled” classes are those that contain code that is used by the classes in the service. It is important to account for these as they represent code that needs to be migrated.

continues on page 19
medium, and the risk would be low because of their familiarity with the systems. These determinations were made by having detailed discussions with the contractor where we explained the target SOA requirements and constraints and they would analyze the specific changes to be made to the code. However, inadequacies were found in architecture documentation such that there remained a number of gaps in our understanding of the system. In addition, the contractor underestimated the amount of code used by the potential services.

To get a better understanding of these issues we performed a code analysis and architecture reconstruction. We first analyzed the code through a code analyzer “Understand for C++”. This analysis provided:

- Data dictionary
- Metrics at the project, file, class, and function level
- Invocation tree
- Cross reference for include files, functions, classes/types, macros and objects
- Unused functions and objects

The code analysis enabled us to validate the input from the contractor and to produce input for an architecture reconstruction tool that would identify dependencies between portions of the code. From the code analysis, we found that the code was better organized and documented at the code level than most code that we have seen. However, as mentioned earlier, there were inconsistencies in the quality and documentation between different parts of the code that made the analysis complicated.

Because we needed to get a better understanding of dependencies between different parts of the legacy code, we performed an architecture reconstruction with a tool called ARMIN. This analysis, a form of software archaeology, provided a look at the software architecture of the “as-built” system [3, 6].

In our analysis, we were interested in:

- Dependencies between services and user interface code — user interface code needs to be removed from services given that the users of these services will have their own user interfaces
- Dependencies between services and the commercial mapping software — this code has to be clearly identified so that decisions can be made as to how to replace this functionality if Windows products are not allowed in this environment
- Dependencies between services — there were two migration projects taking place at the same time; if these projects shared code it would make no sense to treat them as separate projects
- Dependencies between services and the rest of the code that mainly represented the data model — this would prove the importance of the data model as well as the underestimation of the code used by the services

The analysis was able to identify a substantial number of undocumented dependencies between portions of the code. These enabled a more realistic understanding of the scope of the migration effort. It also documented dependencies with the commercial mapping software and database. These are a potential concern in the target environment.

The architecture reconstruction also provided evidence that the system data model is potentially a valuable reusable component that had not been identified during the initial analysis. However, this finding was tempered by the fact that in the target SOA environment, the eventual common data model will preclude the use of the current data model. The common data model will likely be the result of negotiation among many interested parties.

**Developing a Strategy for Service Migration**

In general, we found that the legacy code represents a set of components with significant reuse potential. The largest risk in reusing the legacy components concerns the fact that the target SOA has not been fully developed. While its overall structure has been defined, many of the specific mechanisms for interacting with it are still pending. Thus, it is not yet clear what the ultimate requirements will be for a service. Based on these general observations, the recommended migration strategy can be summarized in the following steps:

*continues on page 20*
1. Require the contractor to update the software architecture documentation and standardize comments in the code.
2. Work with the developers of the target SOA to define what is meant by a compliant service.
3. Work closely with the team within the target SOA group that is defining the data model to understand its contents and influence it as necessary.
4. Find out if there the vendor has plans for a Linux version of the mapping software or if the target SOA group has plans for a mapping common service to replace the current Windows mapping software.
5. Interact with potential application developers that will be using the services to understand their requirements and develop appropriate service interfaces.
6. Recalculate cost and effort of migration based on complete set of code dependencies and new understanding of user requirements and SOA constraints.
7. Understand the commonality between the current service migration effort and a second forthcoming similar migration project to a different target SOA.

4. Service-Oriented Migration and Reuse Technique (SMART)

We had previously employed this general approach and these techniques on other architecture and reuse investigations—but not as an integrated approach for making decisions on migration to services. Because of interest from other customers, we are in the process of formalizing the approach and techniques into the Service-Oriented Migration and Reuse Technique (SMART). A Technical Note that explains the technique is now available on the SEI Website (see http://www.sei.cmu.edu/publications/documents/05.reports/05tn029.html). The C2 example provides a general outline of SMART, which details the following steps:

1. Establish stakeholder context.
2. Describe existing capabilities.
3. Describe the future service-based state.
4. Analyze the gap between service-based state and existing capabilities.
5. Develop strategy for service migration

SMART is not intended to replace system engineering activity. It provides a preliminary analysis of the viability of migrating to services, migration strategies available, and the costs, risks, and associated confidence ranges for each strategy. The organization must still pursue an appropriate engineering strategy [5].

5. Conclusions and Next Steps

Determining how to expose functionality as services can have substantial complexity. Software archaeology, in this context, has to go beyond understanding the architecture of a system and reusing code. Other concerns, such as the stakeholder context and SOA development plans will greatly influence the way in which code is reused.

Our report to the C2 client, while not definitive, pointed out a number of issues that they had not previously considered. The type of disciplined analysis that we performed supported our recommendations and provided the client an invaluable amount of information. For example, a by-product of this work was a better understanding of the target SOA on behalf of the client. The developer of the target SOA also benefited from the process because it provided a better understanding of the potential needs and challenges for the services that would use the target SOA. Given our recommendations, mainly that the target SOA is still under development, the client decided to defer the migration decision—a decision that could have been costly two years from now.

An early version of SMART was applied in the C2 example just described. This early version differed from the current structure of SMART because there were various guides and outputs that had not been formalized. However, similar concepts were applied informally. We are currently updating the prototype process with the following goals:

• Improve the breadth and consistency of information gathered about the engineering effort necessary to change the legacy artifact into a ser-
vice. The Service Migration Interview Guide is the first tool intended for this purpose. By incorporating significant technical “know how” into the SMIG, we also further an ultimate goal of transitioning the technique to other users.

- Incorporate decision rules on when it is most useful to include the code analysis and architecture reconstruction steps as part of the process.
- Develop machine support for capturing and analyzing data gathered during the SMART process. This will entail building templates for major artifacts.
- Develop techniques and criteria for determining when a SMART team has captured sufficient information to complete the analysis process.
- Establish a mechanism to capture the net effect of SMART on migration efforts. This information is essential for continued evolution and improvement of SMART.

References


About the Authors

Grace Lewis is a senior member of technical staff at the Software Engineering Institute (SEI) of Carnegie Mellon University (CMU), where she is a part of the Integration of Software-Intensive Systems (ISIS) Initiative. Grace is currently working in the areas of constructive interoperability, service-oriented architectures, Web services, modernization of legacy systems, and model-driven architecture. Her latest publications include several reports published by Carnegie Mellon on these subjects and a book in the SEI Software Engineering Series. Grace has over fifteen years of experience in Software Engineering. She is also a member of the technical faculty for the Master in Software Engineering program at CMU. Grace holds a B.Sc. in Systems Engineering and an Executive MBA from Icесь University in Cali, Colombia; as well as a Master in Software Engineering from Carnegie Mellon University.

Email: glewis@sei.cmu.edu

Edwin Morris is a Senior Member of the Technical Staff at the Software Engineering Institute, assigned to the Integration of Software-Intensive Systems (ISIS) Initiative. He is currently investigating approaches to achieving technical interoperability between complex systems and programmatic interoperability between the organizations that build and maintain them. Previous activities involved improving processes and techniques for the evaluation and selection of COTS products, and the development of the COTS Usage Risk Evaluation (CURE) technology. Before coming to the SEI, Ed developed custom operating systems for embedded microprocessors along with support tools to predict and monitor the performance of real time systems. Ed holds a B.A. and an

continues on page 22
This article is a synopsis of several email and phone conversations, occurring in May and June of 2005, between Sellam Ismail, Curator of Software for the Computer History Museum (CHM) and Ellen Walker, DACS Analyst. It has been organized into a series of Questions and Answers for your reading convenience. The questions are not in any particular order.

1- What, if anything, does the CHM have to do with Software Archaeology?

Software archaeology involves digging into an existing (often perceived as ancient) code base to recover understanding of algorithms, but if the code is trapped on out-moded computer media, for which the means to read it is no longer available, the digging must begin at a deeper level, where the code being investigated is buried on old disks, tapes, or even punched cards. These old forms of storage are like a tomb for the program that lay within, and the first step in gaining access is solving the riddle of how to read them.

The CHM maintains a collection of vintage computers and related hardware. As a historical computer consultant specializing in vintage computing technology, I am often called upon to read programs and data from old computer media that runs the gamut from punched cards to paper tape to strange and bizarre tape formats that many people today either haven’t even heard of or hardly remember. Each job I get is an interesting challenge, often times requiring many hours of investigative work to determine the format of the media and to piece together a functional system capable of reading the media. 

Software Preservation at the Computer History Museum

http://www.computerhistory.org/

Interview with Sellam Ismail, Curator of Software

Migration of Legacy Components

Continued from page 21.

M.A. in Psychology from University of Connecticut, as well as an M.S. in Computer Science from Bowling Green State University. Email: ejm@sei.cmu.edu

PhD Dennis Smith is a senior member of the technical staff and Lead of the Integration of Software-Intensive Systems (ISIS) Initiative. This initiative was launched in October, 2003 and focuses on developing and applying methods, tools and other technologies that enhance the effectiveness of complex networked systems and systems of systems. Previously, he was a member of the Product Line Systems Program and technical lead in the effort for migrating legacy systems to product lines. In this role his team developed the method Options Analysis for Reengineering, OAR, to support reuse decision-making. He has published a variety of books, articles and technical reports, and has given talks and keynotes at conferences and workshops. Dennis was the co-editor of the IEEE and ISO recommended practice on CASE Adoption, and has been general chair of two international conferences, IWPC99 and STEP99. Dennis holds an M.A. and PhD from Princeton University, and a B.A. from Columbia University.

Email: dbs@sei.cmu.edu
covering the bits. For this, I draw upon my vast collection of vintage computers: over 2,000 with the oldest being an original PDP-8 from 1965.

Most recently I was called upon to recover actual archaeological data from a set of VHS tapes at the Mel Fisher Museum in Key West, Florida. In the late 1970s, a company called Alpha Microsystems (one of the first microcomputer companies which is still in business today) pioneered a system for storing data on standard VHS tapes using ordinary video cassette recorders. At that time, the system was deemed to be a practical and elegant solution to the problem of backing up entire hard drives, which were then counted in the single megabytes but were getting bigger by leaps and bounds each year (much like our current trend).

When Mel Fisher discovered the Atocha treasure ship in 1986, which sunk off the Florida Keys in 1622, one of his first priorities was to make a proper record of the finds pulled up from the wreck. The sheer magnitude of the motherlode (for example, there were over 100,000 silver coins recovered) required a flexible and efficient solution for documenting and cataloguing each artifact. Fisher hired a computer consultant who designed a system to digitally photograph each coin so that a visual record could be made. The photos were taken hundreds at a time and stored to the hard disk of the digital camera station. Once the hard drive was full, the processed photos were backed up to VHS tapes, the hard drive was cleared, and the next batch was processed. The result was over 150 tapes consisting of tens of thousands of digital photographs. The tapes were then stored away for safe keeping and quietly forgotten.

The Mel Fisher Museum recently re-discovered the tapes (now fifteen years after their creation) and realized they had no way to read them. The equipment to process the tapes had long since vanished. Worse yet, the tapes held the only photographs of the silver coins that were pulled from beneath the ocean. My firm was hired to recover the data from the tapes. After 4 years and thousands of dollars in effort, we were able to track down and assemble the necessary hardware and software to read these tapes and convert the images to a modern graphics format (they were stored in a proprietary 16-level grayscale). The project involved dozens of hours of searching for interface cards, special VCRs, old software, ancient versions of DOS, and properly antiquated (i.e. slow) PCs to make everything work.

Unfortunately, this situation is quite common. Organizations have historically not considered the ramifications that the obsolescence of

UNIVAC Punched Tape (1960s)
Computer control instructions are contained on punched paper tape from an early UNIVAC plant in Utica, NY. Try to imagine what the debugging process would be for this code. How would one proceed to dig into it? How would the process differ from your current process? The image is provided by courtesy of the Department of Information Systems, London School of Economics website (see http://is.lse.ac.uk/History/UNIVAC-PunchedTape.htm).
computer media can have. Before it is realized, media that holds perhaps thousands of man-hours of computer code and data can be put at risk when the last unit of a particular Zip drive leaves the assembly line. The issue has plagued government and private sector entities for decades. Computer technology advances so quickly that computer media can be outmoded suddenly and without warning. An organization that does not have a proper plan for the obsolescence of its data stores is one day going to face the same problem.

Through its acquisition of outdated hardware and software, the CHM is providing linkage to computer technologies of the past. The scope of usage of our computer artifacts, including software, can be whatever we, the community of software archaeologists, want it to be.

**2 - What is the Museum currently doing regarding the collection and cataloguing of software?**

In the past, the Museum tended to collect software as an afterthought. Software would usually come in as part of a hardware donation. As such, not much discretion was used in determining what software artifacts the Museum should be accepting, and as a result we ended up with a lot of incredible artifacts (such as the operating system and programs for the MIT Whirlwind) as well as a lot of rubbish, such as entirely non-interesting driver disks and random media with unknown stuff on it. Some of this “unknown stuff” may yet prove to be very historically significant, but without the proper context to go along with it, we will have to do a lot of investigating to separate the wheat from the chaff.

Fortunately, the Museum recognized this deficit in properly collecting software and created the position of Software Curator. I was tapped to fill the position and have since been establishing collecting guidelines for software as well as building out the infrastructure for properly maintaining the software collection, including physical storage, cataloguing, and access.

To that end, I first started by organizing the Museum’s existing holdings. We have two rooms devoted exclusively to software. I’ve set up the necessary shelving and have arranged the various software artifacts in a structured manner to make the cataloguing process more efficient. I added the proper facilities for a Software Collection Catalog to the Museum’s database, defining the fields and developing a data dictionary to use as reference for populating each record. I also developed a Software Collection Taxonomy, which ultimately serves the purpose of assisting researchers in finding the type of software they are looking for. By using the taxonomy to categorize software, researchers can, for instance, request all titles in our collection that have something to do with spreadsheets, and from there they can perform a more refined search.

We’ve begun the cataloguing process and plan to have it completed by the end of the summer. We have over a thousand packaged software titles (e.g. commercial software), and thousands of other artifacts including software on common media such punched cards, paper tape, magnetic tape, floppy disk, etc., and on more exotic media such as magnetic film and program rods (steel rods with holes drilled into them to represent bits). Our software collection also includes source listings on paper. We have software going back to the early 1950s and as late as from within the last several years.

I have initiated a project (outside of the CHM) called FutureKeep to develop a universal media imaging format so that the data on media of any type can be described and encoded digitally in a manner that will allow the original data image to be reconstructed at a future date if the need or desire ever arises. It is also being designed to serve as a universal image format for simulators. The specification for this format also takes into account the quickly advancing nature of computing and storage technology and the fact that media of today will be outmoded and difficult to source in mere months or years or, at most decades. The intent is to create archives that will (hopefully) withstand the test of time and be readable and useable centuries from now.

continues on page 25
3 - Can you talk about your plans for preserving the software, and the reality of where you are with it right now?

As for preservation of the software (i.e. the code itself), this is a major undertaking and we are currently studying the issues.

In the meantime, I am currently formulating a plan for the creation of a “transcoding” lab at the Museum. We’ve settled on the term “transcoding” to describe the process of extracting information from one medium and storing it on another, in this case a centralized server where all the bits can be conveniently managed. The lab will contain all the hardware needed to read all the various media we have in our software collection. We will begin methodically transcoding all the media in our software archive as soon as the lab is ready for action. This cannot come soon enough as we are in somewhat of a race against time with a lot of the artifacts (i.e. the physical media) in the software collection, some of which are stored on disks and tapes which are either at or well beyond their theoretical lifespan (though I should add in practice we find that magnetic media is more durable than once predicted) and some of which cannot be currently read by conventional means.

4 - How does the storage media impact the preservation of your software?

We must keep in mind that no one truly knows just how long magnetic media will really last, just like we don’t truly know how long CD or DVD media will really last as the technology is still relatively too new to have data from the real world. CD media is thought to have a lifespan of 100 years, but those estimates are based on accelerated testing, and they certainly don’t apply to the cheap commodity CD-R media you buy off the shelf today (which can last anywhere from years to seconds). Floppy disks were thought to have a lifespan of 15-20 years, but I am finding that disks even 30 years old still read just fine, while 3.5” disks manufactured in the late 1990s die between the time it takes me to copy a file to one of them and then walk over to a PC to which I’m trying to transfer that file. A big factor is the quality of the manufacturing process of the media and, having studied this issue, I always recommend that people research the media they are buying to store data for the long-term as they may have a rude awakening in the not too distant future.

5 - How are you addressing copyright issues and the proprietary nature of items in your collection?

We have taken the first steps towards developing an access policy for our potential audience of researchers and hobbyists. This policy will take into account the fact that some of our software artifacts are considered sensitive or are proprietary and could be used for competitive advantage. The issue of copyrights is a big can of worms wriggling and writhing about, waiting for someone to come along and open it up and uncover the icky sliminess contained within. We, of course, have to be very sensitive about these issues and, to that end, we have fields in our database that flag certain artifacts as being under embargo or not for general release. The proprietary nature of some of...
our artifacts is a relatively easy issue to deal with, but the copyright issues are a far greater concern and, given the direction these issues are currently heading, we may well have to keep some of our digital software artifacts locked down for a good long time to come, which we feel would be a loss to society. In fact, the way copyright law in the United States currently stands, we may even be breaking the law if we circumvent copy protection mechanisms on old, obsolete and no longer published software in order to archive it. This is an issue on which the Museum has commented to the US Copyright Office. We are hoping to get a permanent waiver of the Digital Millennium Copyright Act (DMCA) for the Museum and similar institutions so that preservation efforts can be exempted from the DMCA. This is an unfortunate example of where the provisions of the DMCA clearly fall short of what they were trying to accomplish, and indicates a lack of foresight by the drafters of that law.

6 - How do you decide what software artifacts to keep?

One of the more important tasks I’ve completed since joining the staff of the Museum is developing a Software Selection Criteria document to guide the collections department in deciding what software donation offers to accept or decline. As I mentioned previously, most software typically comes in as a part of physical artifact donation and would automatically be moved to the software room and placed on a shelf. Now, all software is vetted against the Software Selection Criteria to ensure the software has certain historic characteristics that make it a good candidate for long-term preservation. Proper historical preservation for artifacts of any kind is an intensive task that requires lots of resources, both human and otherwise, so the intent of the Software Selection Criteria is to make sure the software we are accepting into the collection is worthy of that expenditure of resources. The criteria address various purposes or value that a software artifact might confer to the Museum, such as the obvious historical merit, or assisting in curatorial efforts (i.e. something useful in completing a collection or including in exhibits), and so on.

7 - How is hardware preservation different from software preservation? To what extent does software preservation depend on hardware preservation? Can the two be separated?

The main differences are the size of the artifacts and the resources required to manage them. Hardware, especially older computers from the Paleolithic era of computing (i.e. the 1950s), requires lots of space and therefore funding to acquire and store. Software, on the other hand, takes up much less space, and once the bits are safely rendered in a digital format, the original media could theoretically be tossed, though for the time being, the Museum’s policy is to retain the original media as the ultimate archival medium, at least until it proves to be utterly useless in holding data (i.e. magnetic tapes in which the magnetic coating is flaking off the base substrate). Even then, of course, the original media may retain some value as cultural artifacts in their own right. This is a consideration that has been especially promoted, for example, by the Smithsonian Institution.

As far as hardware and software as individual artifacts, each cannot be properly understood without the other. True, you can successfully display a computer that is just sitting there powered down and not being anything other than a hulk of metal and plastic, but it is not really telling the whole story. On the other hand, how does one present software without actually executing it on its native hardware, or in the very least under simulation? It’s an interesting conundrum, especially when one considers the effort required to resurrect a decades old computer system. The Museum has so far restored to working order an IBM 1620, a DEC PDP-1, and an IBM 1401 is currently undergoing restoration. However, without software, the effort that went into the restoration of these machines would be pointless. It is only when you sprinkle in the magic of software that the machine does anything particularly interesting or useful. Software is

continues on page 27
indeed the soul of the computer.

With regard to preservation, hardware is needed to preserve software only to the extent that the bits must be recovered from the original media before the media lose their ability to retain information. And while it’s nice to be able to run software on its native hardware, it is not always practical to do so. Machine restorations take lots of time and effort by highly skilled individuals, not to mention a significant amount of money (the IBM 1401 restoration has so far consumed several thousand dollars, the majority of which was kindly donated by Museum supporters). On the other hand, once we have the software in a modern digital format, we can always utilize simulators to recreate the effective feel of operating historic software. Some simulators try to recreate the full experience, with the recorded sounds of whirring disk or tape drives and teletypes pecking away, and while it’s a far cry from the experience of running Spacewar! on an actual PDP-1 and watching the front panel lights blink away, it’s better than not having any experience. I imagine at some point in the future we’ll be able to synthesize a more realistic experience, say something like what they have in the imaginary Holodeck in the Star Trek series, but for now we have to mostly rely on simulators for historic computers from the 1950s and 1960s. Fortunately, we can still enjoy the wealth of minicomputer and microcomputer software on native platforms since these classes of machines from the early 1970s onwards are relatively easy to setup and maintain in working condition. However, they too will someday be impractical to run, if only because their mechanical parts get worn out, or components die, or bits in ROM fade away.

8 - How is the software that you have preserved currently being used and by whom (what types of patrons)?

Currently, we don’t get many requests from outside our organization to access the software, but of the few requests we do get, most are from attorneys seeking specific titles as prior art for patent infringement lawsuits. We do, however, utilize assets in our software archive internally to support exhibits we set up and also, to give our volunteer restoration teams something to run on the machines they resurrect. On request, we provide the researcher access to the physical media itself, and we would assist them in actually retrieving or executing the software on either suitable hardware or, if that’s not possible or practical, on simulators.

9 - What is your plan for preventing your software holdings from becoming obsolete as technology whizzes by? What is the cost that you incur to record the history of computing?

We are establishing tools, procedures and guidelines that we hope have fully taken into account all the various and germane parameters to ensure that our efforts remain relevant for some time to come. Of course, these tools, procedures and guidelines will require periodic revamping to keep up with changing technology and evolving methods of software storage and preservation, but we believe we have started off with a very good precedent that should carry us through at least the next five years and give us time to evaluate and plan for the next step.

Have something to donate to CHM?
Complete the web form at http://www.computerhistory.org/collections/donateArtifact/
Or
Call the Museum at 650/810-1010
The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer’s regular, hub-mounted, CD drive. You’ll use this great resource over and over again. It’s FREE, just for filling out our brief survey on the next page!

http://iac.dtic.mil/dacs/
Software Tech News Subscriber Survey

1. Which volume of the Software Tech News did you receive? STN 8:3 Secure Software

2. When did you receive the newsletter? (month/year)

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)

4. How satisfied were you with the APPEARANCE of the newsletter?

5. How satisfied were you with the OVERALL QUALITY of the newsletter?

6. How satisfied were you with the ACCURACY of the address on the newsletter?

7. Approximately how much of the newsletter do you read?

8. Would you read this newsletter in an E-mail newsletter format?

9. How did you request the product or service?

10. Would you recommend the DoD Software Tech News to a colleague?

11. What topics would you like to see this newsletter devoted to?

Comments (Optional)

Contact Information (Optional*)

Name:
Position/Title:
Organization:
Office Symbol:
Address:
City: State: Zip Code:
Country: E-mail:
Telephone: Fax:

Functional Role:

Organization Type: Air Force Army Navy Other DoD
Commercial Non-Profit Non-US US Government FFR&D Other

*Note: You must give us your address to receive the CD.
About the DoD Software Tech News

STN Editorial Board

Philip King
Editor
ITT Industries, DACS

Paul Engelhart
DACS COTR
Air Force Research Lab (IFED)

Morton A. Hirschberg
Editorial Board Chairman
Army Research Lab (retired)

Ellen Walker
ITT Industries, DACS

Thomas McGibbon
DACS Director
ITT Industries, DACS

Article Reproduction

Images and information presented in these articles may be reproduced as long as the following message is noted:

“This article was originally printed in the DoD Software Tech News, Vol. 9, No. 2. Requests for copies of the referenced newsletter may be submitted to the following address:

Philip King, Editor
Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Phone: 800-214-7921
Fax: 315-334-4964
E-mail: news-editor@dacs.dtic.mil


In addition to this print message, we ask that you send us three copies of any document that references any article appearing in the DoD Software Tech News.

About This Publication:

The DoD Software Tech News is published quarterly by the Data & Analysis Center for Software (DACS). The DACS is a DoD sponsored Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC). The DACS is technically managed by Air Force Research Laboratory, Rome, NY and operated by ITT Industries, Advanced Engineering and Sciences Division.

Cover Design by Joseph Barbaccia,
ITT Industries

To Subscribe to this Publication Contact:

Phone: 800-214-7921
Fax: 315-334-4964
E-mail: news-editor@dacs.dtic.mil
Web: www.dacs.dtic.mil

Distribution Statement:
Unclassified and Unlimited
Advertisement

The DoD Software Tech News is now accepting advertisements for future newsletters. In addition to being seen by the thousands of people who subscribe to the DoD Software Tech News in paper copy, the Tech News will also be placed on the Data & Analysis Center for Software’s website (http://iac.dtic.mil/dacs/), exposing your product, organization, or service to hundreds of thousands of additional eyes.

Interested in learning more? For rates, layout information, and requirements contact:

Philip King, STN Editor
Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Phone: (800) 214-7921
Fax: (315) 334-4964
E-mail: news-editor@dacs.dtic.mil