Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

From Transience to Persistence in Object Oriented Programming

1School of Computer, Wuhan University, Wuhan 430072, China
2Department of computer science, Huazhong Normal University, Wuhan 430079, spain
(2013)

Abstract

Object-oriented programs (like most programs) evolve over time and it would be ideal if we could capture persistent parts of the programs early on and then derive the transient versions of the program from the persistent part. In our view, the object-oriented community is moving in this direction through its work on software architecture and patterns. Capturing the persistent parts of a program allows us to better maintain the integrity of the program during evolution.

Software Architecture

The goal of software architecture is to capture the persistent parts of the program and to derive the transient versions using architecture refinement. A number of architecture description languages are under development. Common to many of those architecture description languages is the concept of components and connections between them. Work in this area is summarized in the Architecture Guide.

The work at Northeastern on software architecture is distinguished by using adaptive programming (AP) to capture architecture. This has the advantage that architectural refinements can be automated in many cases. An adaptive program tries to capture the persistent parts of an object-oriented program by using traversals and context objects. Traversals are specified succinctly taking advantage of the structure of object-oriented programs. Work on AP is collected in the adaptive programming book and recent developments are described in Collaborative Behavior Modification.

The work on adaptive programming has connections to many other approaches such as Open Implementation. Those connections are further explored in a description of AP with five patterns. The Inventor's Paradox pattern abstracts from the specifics of AP and tries to describe something which Gregor Kiczales calls Aspect-Oriented Programming (AOP). In AOP a program is described by several loosely-coupled aspects which are woven together into a combined program. In my view, the goal of AOP is to make programs robust to changes in one aspect and to avoid redundancy in the programs. AOP strives to describe the aspects of an

application in such a way that the volatile aspects are localized and so that changes to those aspects don't have too many repercussions. This leads to a better description of the persistent parts of the application. A definition of AOP is under preparation at Xerox PARC and the above summary might not be correct.

In the application of AOP to AP we have three aspects: The structural aspect (class graph), the navigation aspect (traversal specifications), and the behavior modification aspect (context objects). The three aspects are described by three little languages, one per aspect. Adaptive programs are robust since they are robust to changes in the class graph and also robust to changes in the context objects. AP eliminates redundancy since the class graph information is not repeated many times in the program.

AP is programming by hooks which loosely couple the parts together. The program for each part defines hooks into other parts. When all parts are given, the hooks are combined, if necessary, by weaving instructions. (we borrow the term "weaving" from AOP; but AOP is more general than AP.) The hooks are used to formulate the weaving algorithm. In the simplest form of AP, the hooks are classes and edges in the class graph to which we refer selectively to express context objects and traversals.

One feature of interesting applications of AP is that the weaving algorithm is sophisticated enough so that from weave(A1,A2,A3) it is not possible to uniquely determine a part even if all other parts are given. On the other hand, the weaving algorithm cannot be too sophisticated since it must be fast. For example, in simplest form of AP with class graphs and traversal specifications, it is usually impossible to uniquely determine the traversal specifications from the object-oriented program, the class graph and the context objects.

The typing of adaptive programs is a challenge. We call a program (A1,A2,*) type-correct if there exists a part A3 such that weave(A1,A2,A3) is type-correct. Jens Palsberg is currently studying the typing of adaptive programs: Given an adaptive program consisting of traversal specifications and context objects, is there a class graph which creates a type-correct object-oriented program.

The typing of adaptive programs is influenced by the generality of the weaving algorithm. If the weaving algorithm needs to make many restrictions regarding

what can be woven together, the typing question becomes more difficult. It is important that the weaving algorithm is general purpose. See the work on compiling adaptive programs Efficient Implementation of AP , and Automata Theoretic Compilation.

The work on meta-object-protocols is also concerned, to some degree, with persistence versus transience and with software architecture. The idea is that the persistent parts of the program are expressed at the base level while the transient parts are expressed at the meta level. The meta level provides building blocks for the base level and ideally most changes to the program can be accomplished by changing the meta objects and programs. See URL: http://www.parc.xerox.com/spl/projects/oi/

Patterns

Work on patterns in software development has been popularized by the design pattern book [GOF]. Patterns can be classified in different categories: programming, design, architectural and organizational. Patterns contribute to capturing the persistent parts of a program by describing often used solutions to problems in a context.

The work on design patterns will develop in a number of directions. New patterns will be added to the catalog of well known patterns. Patterns of different levels of abstraction will be developed. Finally, well known patterns will be supported directly by tools. To develop those tools, it is necessary to have a notation to describe patterns so that a tool can operate with patterns. Tasks which can be automated are: analyzing patterns, customizing patterns, composing patterns, retrieving patterns.

Analyzing means to check whether the pattern is ``type-correct'' and customizing means to check for proper application of a pattern to a program and to automatically create the code prescribed by a pattern.

The emerging field of architecture description languages can play an important role in producing models and tools to make work with patterns even more productive. An architecture description language ought to be expressive enough to express design patterns and even some architectural patterns.

However there are high-level patterns which will not (yet) lend themselves to automation. For example, the description of AP with five patterns leads to high-level patterns, called Class Graph, Structure-shy Object, Structure-shy Traversal and Context which would be hard to describe formally and would be hard to automate. What kind of features do architecture description languages need to express design patterns?

- Pattern Abstraction One important feature which stands out is a facility to express patterns abstractly, without referring to unimportant details. Yet the patterns must be customizable to lead to specific execution behavior. Customization adds details and creates an executable program which controls the interaction specified by the pattern. The concepts of AP offer new ways to describe patterns succinctly.

- Applicability Constraints A pattern consists of a partial object interaction specification and an applicability condition. By ``applicability condition'' we mean restrictions on the set of entities that collaborate for the purpose of the interaction. The applicability condition specifies the contexts in which the pattern may be applied. Therefore, an architecture description language needs to express applicability constraints for patterns.

Conclusions

More work is needed to explore techniques which allow us to better express the persistent properties of program families. The ideas behind AP should prove useful to formulate software architectures and patterns at a higher level of abstraction.

At the programming language level, it would be useful to integrate the AP concepts of traversals and contexts into the programming language Java.

Doc2

3. Architectural Goals and Constraints

Key networking protocol standards such as TCP/IP and HTTP and the markup language HTML have driven the explosive growth of the Web. For the next phase of growth, industries can use XML to enable the sharing of data across multiple systems and XML Web Services to enable rich integration of distributed

applications. These technologies will streamline the development process for creating a highly interactive and interoperable suite of software tools for communities of medical professionals.

With this in mind we may envision an end-result as seen from a human user similar to the diagram below, alternate views are possible depending on the role of the user; producer, consumer etc.

Diagram…..

The diagram denotes how the services implemented according to the MedBiquitous standards can be utilized by the Portal infrastructure. We see 3 different implementation styles for the information service in the example above:

• Style #1. This information service interfaces to an existing application and presents that applications information to the community according to MedBiquitous technical standards.

• Style #2. This is a service that bypasses the existing application (due perhaps to an inability to interface through the application itself) and serves information directly from local repositories.

• Style #3. This is an aggregation service, it communicates to two or more downstream information services and aggregates content according to some predefined rules.

• We also see the ability for one portal to access and present information from another portal, although the ICE content syndication standards are more likely than direct screen-scraping technology.

3.1 Standards

The MedBiquitous consortium has chosen a set of technology standards that support the implementation of a federated information network. These standards are those that underpin the Internet as well as in the emerging area of Web Services. These standards were chosen because of their open and platform neutral

manner thus allowing an open and flexible choice for the implementer. The following diagram shows how the MedBiquitous technology standards (indicated by *) build upon one another and upon this existing infrastructure of Internet technologies.

Note that the above diagram is a snapshot of the current state of the technology; detailed design documents in each area will outline the specific versions of standards to be used in each area.

MedBiquitous standards provide the means to integrate applications and data across the Internet, enabling rich collaboration among many organizations in professional medicine. MedBiquitous development efforts build upon existing Internet protocols, Web protocols, and emerging Web services standards and include the following technologies:

• XML Payload Standards for the data and documents produced by medical communities and their business partners.

• XML Web Services Standards to enable the delivery and integration of applications for medical communities.

• Portal Standards for integrating content and applications from various sources into a cohesive and usable presentation for physicians.

• Security Standards and guidelines for exchanging authentication and access data among disparate systems and organizations.

Doc 3

An Introduction to Software Architecture

1. Introduction

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality

to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

This is the software architecture level of design. There is a considerable body of work on this topic, including module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms. In addition, an implicit body of work exists in the form of descriptive terms used informally to describe systems. And while there is not currently a well-defined terminology or notation to characterize architectural structures, good software engineers make common use of architectural principles when designing complex software. Many of the principles represent rules of thumb or idiomatic patterns that have emerged informally over time. Others are more carefully documented as industry and scientific standards.

It is increasingly clear that effective software engineering requires facility in architectural software design. First, it is important to be able to recognize common paradigms so that high-level relationships among systems can be understood and so that new systems can be built as variations on old systems. Second, getting the right architecture is often crucial to the success of a software system design; the wrong one can lead to disastrous results. Third, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives. Fourth, an architectural system representation is often essential to the analysis and description of the high level properties of a complex system.

2. From Programming Languages to Software Architecture

One characterization of progress in programming languages and tools has been regular increases in abstraction level—or the conceptual size of software designers building blocks. To place the field of Software Architecture into perspective let us begin by looking at the historical development of abstraction techniques in computer science.

2.1. High-level Programming Languages

When digital computers emerged in the 1950s, software was written in machine language; programmers placed instructions and data individually and explicitly in the computer's memory. Insertion of a new instruction in a program might require

hand-checking of the entire program to update references to data and instructions that moved as a result of the insertion. Eventually it was recognized that the memory layout and update of references could be automated, and also that symbolic names could be used for operation codes, and memory addresses. Symbolic assemblers were the result. They were soon followed by macro processors, which allowed a single symbol to stand for a commonly-used sequence of instructions. The substitution of simple symbols for machine operation codes, machine addresses yet to be defined, and sequences of instructions was perhaps the earliest form of abstraction in software. In the latter part of the 1950s, it became clear that certain patterns of execution were commonly useful—indeed, they were so well understood that it was possible to create them automatically from a notation more like mathematics than machine language. The first of these patterns were for evaluation of arithmetic expressions, for procedure invocation, and for loops and conditional statements. These insights were captured in a series of early high-level languages, of which Fortran was the main survivor.

Higher-level languages allowed more sophisticated programs to be developed, and patterns in the use of data emerged. Whereas in Fortran data types served primarily as cues for selecting the proper machine instructions, data types in Algol and it successors serve to state the programmer's intentions about how data should be used. The compilers for these languages could build on experience with Fortran and tackle more sophisticated compilation problems. Among other things, they checked adherence to these intentions, thereby providing incentives for the programmers to use the type mechanism. Progress in language design continued with the introduction of modules to provide protection for related procedures and data structures, with the separation of a module's specification from its implementation, and with the introduction of abstract data types.

2.2. Abstract Data Types

In the late 1960s, good programmers shared an intuition about software development: If you get the data structures right, the effort will make development of the rest of the program much easier. The abstract data type work of the 1970s can be viewed as a development effort that converted this intuition into a real theory. The conversion from an intuition to a theory involved understanding

• the software structure (which included a representation packaged with its primitive operators),

• specifications (mathematically expressed as abstract models or algebraic axioms),

• language issues (modules, scope, user-defined types),

• integrity of the result (invariants of data structures and protection from other manipulation),

• rules for combining types (declarations),

• information hiding (protection of properties not explicitly included in specifications).

The effect of this work was to raise the design level of certain elements of software systems, namely abstract data types, above the level of programming language statements or individual algorithms. This form of abstraction led to an understanding of a good organization for an entire module that serves one particular purpose. This involved combining representations, algorithms, specifications, and functional interfaces in uniform ways. Certain support was required from the programming language, of course, but the abstract data type paradigm allowed some parts of systems to be developed from a vocabulary of data types rather than from a vocabulary of programming-language constructs.

2.3. Software Architecture

Just as good programmers recognized useful data structures in the late 1960s, good software system designers now recognize useful system organizations. One of these is based on the theory of abstract data types. But this is not the only way to organize a software system. Many other organizations have developed informally over time, and are now part of the vocabulary of software system designers. For example, typical descriptions of software architectures include synopses such as (italics ours):

•“Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers.”[8]

•“Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client server model for the structuring of applications.”[9]

•“We have chosen a distributed, object-oriented approach to managing information.” [10]

•“The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors. . . . A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program.”

REFERENCES

[1] J. Campbell, "Speaker recognition: a tutorial," *Proc. IEEE*, vol. 85, pp. 1437–1462, Sept. 1997.

[2] D. A. Reynolds, T. Quatieri, and R. Dunn, "Speaker verification using adapted Gaussian mixture models," *Digital Signal Processing*, vol. 10, no. 1–3, pp. 19–41, 2000.

[3] D. A. Reynolds, "Comparison of background normalization methods for text-independent speaker verification," in *Proc. Eurospeech*, 1997.

[4] D. A. Reynolds and R. C. Rose, "Robust text-independent speaker identification using Gaussian mixture speaker models," *IEEE Trans. Speech Audio Processing*, vol. 3, no. 1, pp. 72–83, 1995.

[5] J. L. Gauvain and C.-H. Lee, "Maximum a posteriori estimation for multivariate Gaussian mixture observations of Markov chains," *IEEE Trans. Speech Audio Processing*, vol. 2, pp. 291–298, Apr. 1994.

[6] E. Bocchieri, "Vector quantization for the efficient computation of continuous density likelihoods," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1993, pp. 692–695.

[7] K. M. Knill, M. J. F. Gales, and S. J. Young, "Use of Gaussian selection in large vocabulary continuous speech recognition using HMMs," in *Proc. Int. Conf. Spoken Language Processing*, 1996.

[8] D. B. Paul, "An investigation of Gaussian shortlists," in *Proc. Automatic Speech Recognition and Understanding Workshop*, 1999.

[9] T. Watanabe, K. Shinoda, K. Takagi, and K.-I. Iso, "High speed speech recognition using tree-structured probability density function," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1995.

[10] J. Simonin, L. Delphin-Poulat, and G. Damnati, "Gaussian density tree structure in a multi-Gaussian HMM-based speech recognition system,"

[11] T. J. Hanzen and A. K. Halberstadt, "Using aggregation to improve the performance of mixture Gaussian acoustic models," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1998.

[12] M. Padmanabhan, L. R. ahl, and D. Nahamoo, "Partitioning the feature space of a classifier with linear hyperplanes," *IEEE Trans. Speech Audio Processing*, vol. 7, no. 3, pp. 282–288, 1999.
in *Proc. Int. Conf. Spoken Language Processing*, 1998.

[13] R. Auckenthaler and J. Mason, "Gaussian selection applied to text-independent speaker verification," in *Proc. A Speaker Odyssey—Speaker Recognition Workshop*, 2001.

[14] J. McLaughlin, D. Reynolds, and T. Gleason, "A study of computation speed-ups of the GMM-UBM speaker recognition system," in *Proc. Eurospeech*, 1999.

[15] S. van Vuuren and H. Hermansky, "On the importance of components of the modulation spectrum of speaker verification," in *Proc. Int. Conf. Spoken Language Processing*, 1998.

[16] B. L. Pellom and J. H. L. Hansen, "An efficient scoring algorithm for Gaussian mixture model based speaker identification," *IEEE Signal Processing Lett.*, vol. 5, no. 11, pp. 281–284, 1998.

[17] J. Oglesby and J. S. Mason, "Optimization of neural models for speaker identification," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1990, pp. 261–264.

[18] Y. Bengio, R. De Mori, G. Flammia, and R. Kompe, "Global optimization of a neural network—hidden Markov model hybrid," *IEEE Trans. Neural Networks*, vol. 3, no. 2, pp. 252–259, 1992.

[19] H. Bourlard and C. J. Wellekins, "Links between Markov models and multilayer perceptrons," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 12, pp. 1167–1178, Dec. 1990.

[20] J. Navrátil, U. V. Chaudhari, and G. N. Ramaswamy, "Speaker verification using target and background dependent linear transforms and multi-system fusion," in *Proc. Eurospeech*, 2001.

[21] L. P. Heck, Y. Konig, M. K. Sonmez, and M. Weintraub, "Robustness to telephone handset distortion in speaker recognition by discriminative feature design," *Speech Commun.*, vol. 31, pp. 181–192, 2000.

[22] A. Dempster, N. Laird, and D. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *J. R. Statist. Soc.*, vol. 39, pp. 1–38, 1977.

[23] K. Shinoda and C. H. Lee, "A structural Bayes approach to speaker adaptation," *IEEE Trans. Speech Audio Processing*, vol. 9, no. 3, pp. 276–287, 2001.

[24] K. Fukunaga, *Introduction to Statistical Pattern Recognition.* New York: Academic, 1990.

[25] J. C. Junqua, *Robust Speech Recogntion in Embedded Systems and PC*

[26] U. V. Chaudhari, J. Navrátil, S. H. Maes, and R. A. Gopinath, "Transformation enhanced multi-grained modeling for text-independent speaker recognition," in *Proc. Int. Conf. Spoken Language Processing*, 2000.

[27] Q. Lin, E.-E. Jan, C. W. Che, D.-S. Yuk, and J. Flanagan, "Selective use of the speech spectrum and a VQGMM method for speaker identification," in *Proc. Int. Conf. Spoken Language Processing*, 1996.

[28] S. Raudys, *Statistical and Neural Classifiers: An Integrated Approach to Design.* New York: Springer, 2001.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*. Cambridge, MA: MIT Press, 1986, pp. 318–364.

[30] [Online] Available: http://www.nist.gov/speech/tests/spk/index.htm.

[31] J. Pelecanos and S. Sridharan, "Feature warping for robust speaker verification," in *Proc. A Speaker Odyssey—Speaker Recognition Workshop*, 2001.

[32] B. Xiang, U. V. Chaudhari, J. Navrátil, N. Ramaswamy, and R. A. Gopinath, "Short-time Gaussianization for robust speaker verification," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 2002.

[33] G. R. Doddington, M. A. Przybocki, A. F. Martin, and D. A. Reynolds, "The NIST speaker recognition evaluation—overview, methodology, systems, results, perspective," *Speech Communication*, vol. 31, pp. 225–254, 2000.

Bing Xiang (M'03) was born in 1973 in China. He received the B.S. degree in radio and electronics and M.E. degree in signal and information processing from Peking University in 1995 and 1998, respectively. In January, 2003, he received the Ph.D. degree in electrical engineering from Cornell University, Ithaca, NY.

From 1995 to 1998, he worked on speaker recognition and auditory modeling in National Laboratory on Machine Perception, Peking University. Then he entered Cornell University and worked on speaker recognition and speech recognition in DISCOVER Lab as a Research Assistant. He also worked in the Human Language Technology Department of IBM Thomas J. Watson Research Center as a summer intern in both 2000 and 2001. He was a selected remote member of the SuperSID Group in the 2002 Johns Hopkins CLSP summer workshop in which he worked on speaker verification with high-lelvel information. In January, 2003, he joined the Speech and Language Processing Department of BBN Technologies where he is presently a Senior Staff Consultant-Technology. His research interests include large vocabulary speech recognition, speaker recognition, speech synthesis, keyword spotting, neural networks and statistical pattern recognition.

Toby Berger (S'60–M'66–SM'74–F'78) was born in New York, NY, on September 4, 1940. He received the B.E. degree in electrical engineering from Yale University, New Haven, CT in 1962, and the M.S. and Ph.D. degrees in applied mathematics from Harvard University, Cambridge, MA in 1964 and 1966, respectively.

From 1962 to 1968 he was a Senior Scientist at Raytheon Company, Wayland, MA, specializing in communication theory, information theory, and coherent signal processing. In 1968 he joined the faculty of Cornell University, Ithaca, NY where he is presently the Irwin and Joan Jacobs Professor of Engineering. His research interests include information theory, random fields, communication networks, wireless communications, video compression, voice and signature compression and verification, neuroinformation theory, quantum information theory, and coherent signal processing. He is the author/co-author of Rate Distortion Theory: A Mathematical Basis for Data Compression, Digital Compression for Multimedia: Principles and Standards, and Information Measures for Discrete Random Fields.

Dr. Berger has served as editor-in-chief of the IEEE TRANSACTIONS ON INFORMATION THEORY and as president of the IEEE Information Theory