Application development as an engineering discipline: Revolution or evolution?

by N. Bieberstein

The title question is answered differently according to the nature of the person being asked. A talented person with a new solution to a particular problem in the existing technology may be a revolutionary, gathering devoted followers who spread the new idea. Such leaders and their followers then propagate paradigm shifts promoting the one answer, the "silver bullet" to solve all problems. When we look more closely, in most cases only a single aspect was solved; we were not given a whole new way to develop software. This confirms the position of the traditionalists, who continue to keep and protect what is well known. In the end, in application development as in any other discipline, evolution is driven by new inventions and kept on track by the conservatives among us. This essay reflects on the historical path of software development toward an engineering discipline. It also introduces the papers collected for this theme on application development, which demonstrate this progress in selected areas.

A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible. There are no prima donnas in engineering.

—Freeman Dyson[1]

One has to look out for engineers—they begin with sewing machines and end up with the atomic bomb.

—Marcel Pagnol[2]

During the last 30 years we have experienced several waves of ever-improving programming languages, numbered by generations that represent distinct degrees of abstraction. More than once we were introduced to new concepts marking the start of a new application development era: expert systems or artificial intelligence, approaches to simplify the human-computer interaction with languages like SIMULA or Smalltalk (originally intended to allow the user to do "small talk" with the computer), or GUI (graphical user interface) builder approaches, among others. Visual Basic** became one of the most widespread of the GUI builders, although BASIC as a programming language was disdained by professional software writers.

Was this software engineering? *Engineering* means "the application of scientific and mathematical principles to practical ends such as the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems."[3]

A computer system is a machine that operates according to mathematical principles, so any way to communicate with such an engine requires a language that follows strict grammatical rules and semantics, an algebra. Hence computer science was derived from mathematics as an independent subject. To *engineer* means "to skillfully or shrewdly manage an enterprise."[3] The engineering of software then includes all aspects of the software life cycle, and today we have techniques, methods, and methodologies that address the specification, verification, deployment, and testing of application systems, as well as their development.

0018-8670/97/$5.00 © 1997 IBM

As in other disciplines, each advance in application development acquired a number of followers and a fast-growing (and sometimes quickly vanishing) market. But with each advance, a few key ideas and elements were adapted by mainstream developers and introduced into the currently accepted approach. The technical papers selected for this issue do not postulate radical paradigm shifts, but illustrate good ideas for improving application development within a given environment. For this theme, we chose to focus on what has found acceptance, rather than to weigh one approach against another. To measure the progress toward industrial software development, experience reports and field studies are necessary. With this in mind, our theme was initiated and the authors invited to contribute.

Two of our papers show how existing approaches can be extended with object-oriented techniques, one using a construction from parts framework and the other using objects in COBOL. Another paper provides a treatise on how to solve the issue of object persistence, either directly in an object database, or bridged using the gateway-based object persistence (GOP) concept, or with object-relational databases, where relational database management systems (RDBMSs) are enriched with functions managed and executed under a single control system. Innovative ways to debug code are required within such a closed system, and one paper shows a research approach for debugging stored procedures and user-defined functions in Database 2*/Common Server (DB2*/CS) for UNIX**-based platforms.

A way to structure and build applications by making them independent, at least to some degree, of control flow as well as data is explained by another paper. Here the extracted control flow is modeled for and implemented in a separate workflow system. This approach changes the structure of programming systems, and thus the application development process, quite radically, although it still uses and relies on conventional (this includes object-oriented) development for the concrete data transformations, data stores, and presentations. We see here the potential for an industrial approach: a plug-in solution with reuse of well-defined components.

To determine progress on the industrialization scale, reflections on methods and techniques are needed. So one paper reports experiences using an approach that applies object thinking and formal structuring to the development process of banking applications. An empirical study investigating the effect of different business models on object-oriented and "classical" software development shows some results that make us think (". . . the classical approach shows higher on-time delivery than OO . . ."[4]), and another paper reports the findings of an extensive long-term study on the effects of using computer-aided software engineering (CASE) tools (". . . this implies that using lower CASE tools increases the likelihood to run over scheduled time tables . . ."; ". . . CASE tool usage is related to higher satisfaction . . ."[5]). These two field studies make clear that there are human beings involved in defining and realizing the functionality and behavior of computer systems. This leads to several questions that I want to discuss in this essay.

Revolution vs evolution

Looking back, we find that almost all approaches to improve the way we specify computer programs were based on a desire to abstract repeatable steps and to provide an easier-to-learn and more understandable language for communicating with the machine. Today it is taken for granted that formal computer instructions are generated from higher-level languages that are based on a set of words derived from the English language. In general, this is still true for any 3GL (third-generation language),[6] 4GL, and object-oriented programming language, whether compiled or interpreted. Because of the higher-speed processors available today, interpretative systems have become more competitive than in the past and can play to their strength, the flexibility allowed by run-time binding and textual interpretation. Java**, the latest "hit" on the programming language market, exploits this quite well, no longer even bound to a single central processing unit.

In parallel with the process that generated higher-level programming languages, which concentrated on processing data and controlling program execution, abstraction occurred in the way data were stored, managed, and made accessible. We can follow this development from first sequential, then indexed sequential systems, over relational database management systems and other DBMSs, to the recent structured query language (SQL) standard[7] that allows databases to manage and execute functions on the data they contain. Based on the scheme used to define the structure for the stored data and to access the data, languages were developed to be as close to natural language as possible.

Adoption and adaptation of ideas from other disciplines often leads to improvements. Tools to de-

fine and access data in databases show two trends: abstractions manifested in graphical representations based on Chen's[8] entity-relationship approach (primarily used for design), and tools using modern graphical user interfaces with predefined categories and so-called "intelligent" wizards that allow point-and-click user interfaces (to easily access the data). Built on these tools are decision support tools that exploit data mining concepts.[9] Star Trek's "Mr. Data"[10] is not yet perfectly incarnated, but we are getting closer.

When the contemporary approach shows severe shortcomings, newer concepts are then developed and invented. Those with a revolutionary mind-set use the new concept as their vehicle to "fight the establishment." They have good arguments, because the new idea solves a serious problem. This allows them to quickly gain followers, especially when the new approach shows very good results in solving the particular problem. So the new message is spread, and at every opportunity the flaw in the existing solution is pointed out. However, the traditionalists will soon find good reasons to defend their old approach, proclaiming its strengths and detecting weaknesses in the new solution.

The paper on object persistence[11] in this issue examines such a situation. A need to manage complex structures and to store the newly introduced objects gave birth to object-oriented databases. Soon these were challenged by their inability to match the mass data management capabilities of the older solutions in other data storage systems. We find similar situations in nearly all disciplines. The dialectic between the new, revolutionary approach and the old system drives the evolution of the industry. A new idea, however, also provides "buzz words" that are soon applied as labels for old things. So fashion is created, as we have most recently encountered with the term "object-oriented."

Is the term "objects" misleading?

During this decade we focused on *objects* for software development. This noun has its root in the medieval Latin language, where *objectum* means a "thing put before the mind," literally translated, "something that is thrown in front of you."[3]

We perceive things with our senses and observe certain behavior and relationships among them. We map what is experienced by our senses onto given, or learned, patterns using certain rules. According to Whorf,[12] these patterns are determined by the linguistic system the individual was brought up with. He shows that quite different schemes of grammars generate different types of perception. In our mind we always deal with abstractions of what we name "the real world" according to our first adapted pattern. With that we build models using abstractions to understand certain details filtered from the infinite universe. In each discipline, whether engineering or science, we encounter models: of buildings, machines, or the human body, of plants, or the world and the universe; and the use of models is further extrapolated to explain political structures and legal situations. Most of the models we encounter today are constructed based on the patterns of the Indo-European (Western) linguistic system.

The objects we observe are not really "objectively" perceived, but are based on Western abstractions.

This means that the languages we use to communicate with a computer system, and the basic foundations of our hardware and software, are built on a common way of abstracting. In other words, for people raised in the Western cultures, the object in the mind and in models, such as in a computer program, conform to our adapted way to abstract. For people from other cultures this is not the case, but to contribute in the modern world they adopt the Western system in its wholeness.[13] Hence the objects we are dealing with are reflected in our mind as Western abstractions. What we recognize as the thing thrown in front of us, the objects we observe, are not really "objectively" perceived. How would a computer system look if it were based on the Chinese linguistic system?

Therefore we all have a common understanding of objects, but the underlying language pattern is not the same for everyone. Also, people have individual preferences linked to the predominance of a single sense, or to a combination of a few senses. This is true not only for blind or deaf persons; all of us are stamped by our very first experiences of the envi-

ronment. The pattern we find in it determines our way of thinking and further behavior. For more details, see Vester.[14]

Some of us are very good at expressing ourselves in words, others can draw pictures, or differentiate tones or tastes, etc., in an exceptional way. Similarly our social sense is developed along patterns experienced in our early childhood. For some, a picture or drawing simplifies and explains everything (for them the phrase "a picture is worth a thousand words" is true), but some of us cannot understand graphical representations at all. Think of a mathematics teacher trying to help pupils to imagine a three-dimensional geometrical problem.

This leads to the question of what is intuitive and so what is the best way to represent and access a computer system. When application and system development was done by only the mathematically gifted, the world was in balance. These persons share common patterns. The problem occurs when more and more people with other abstraction templates are confronted with computers in their daily life. This is now the case. Graphical user interfaces, multimedia, and the first examples of "virtual reality" allow more senses to be addressed.

To specify and build applications that include all these different features, the language to express the requirements becomes less formal, at least less strictly mathematical, and fuzzier. Human nature in its whole breadth of diversity demands reflection in the user interface as well as in the way we capture the needs and wishes of the user. This is certainly true for consumer-oriented software (games, for example). It is also true for commercial applications, from business process modeling used by management consultants and financial applications to the cashier systems of a department store or in any other enterprise.

All of these requirements must (still) be transformed to computer-understandable commands, in the end to a formal textual language with a Western linguistic pattern. This leads to conflicts and misinterpretations. In order not to broaden the scope of this short essay too far I will focus on commercial application systems. Here business analysts, systems analysts, and programmers must find a common language, some way to communicate without generating too many misunderstandings.

With the recent wave of business process and object models, and the influence of languages, techniques, and semantics from object-oriented analysis and design approaches, we encounter the search for communication between the business and the software development worlds, using and adapting the same terms and similar representations. But there is different understanding of words and a noncongruity in the two areas. The words "process" and "object" mean something different for a programmer than for a business analyst, and for a chemist still another meaning applies. Yes, the term "object" is misleading and confusing.

Is there a common understanding between application developers and users?

Process for the programmer is the transformation of input data, following a defined algorithm, to provide output data, which are the same each time when executed under the same conditions. An object is then a set of data values and attached algorithms, executed when requested. Objects were introduced to most programmers through object-oriented programming languages, where the emphasis was primarily on the notions of class and inheritance. These are the facets perceived as the most beneficial for the task of writing applications. The most often used object-oriented programming languages and also some object-oriented design techniques reflect exactly this. No wonder, because object concepts were often introduced as additional abstractions (the next generation of programming languages) and as the anticipated formalization of good programming practices—copying often-used program patterns, routines, and data structures became easier by referring to an appropriate superclass.

But what does a business analyst understand of an object and a process? In business analysis, an object (called a business object) is an abstraction, an image or stereotype of a real-world thing. It might be a clerk with a certain task, a customer, a contract, an account, a specific technical device, etc., mapped in the mind according to a given pattern. Business processes are the rules and transformations, the specific tasks that are needed to maintain a concrete business. They are executed by and on business objects. The predominant notion here is that tasks are carried out obeying given rules in a specified order with these objects. Inheritance or the concept of classes in the object-oriented sense has no immediate meaning. A business analyst does not show, and in most cases does not need, the deep mathematical

analysis of similarities and patterns of relationships important to computer systems analysts.

In contrast to this, notions of collaboration and containment are meaningful to a business person. Contracts are manifested by a collaboration between at least two different business objects. Containment can be easily translated to and from business terms: an invoice almost always contains items such as customer identification, parts and their amounts and unit prices, etc. There is still a gap between the business analyst and the systems analyst, however. Although contracts and interaction between the objects involved in a given business can be seen as collaborations, the business objects may not map directly to a class structure. The invoice, a piece of paper containing certain items, is naturally perceived as an entity, not as a relationship between other entities. You might recall similar difficulties with entity-relationship modeling.

The latest trend in software development and business modeling shows the two worlds approaching each other. This is not yet in mainstream thinking, but more and more methods appear on the market that deal with both business process models and object models. Also in recent public discussions (e.g., at the Object Management Group [OMG][15]), we find the desire to close the gap between the technical, mathematical definition of computer instructions and the rather pragmatic approach used by business analysts and end users.

From another perspective, for some time we have conceived that application systems could be built from prefabricated parts like integrated circuits (ICs), mounted together in accordance with plans, an architecture, and design drawings produced by the systems analyst. The discussion of whether we can find standards for business objects, or at least a common classification scheme for components, is public, and the OMG (founded as a group to define standards for object-oriented programming) now has several industry-specific task forces dealing with such questions.[16]

IBM's Visual Age* family of application development tools,[17] as well as the workflow-based application development approach,[18] make use of this new paradigm of constructing software from parts. The methods to specify the build plan will improve over time as an increasing number of providers deliver standard and special components. This marks the entrance to an era of "industrial-strength" software engineering. The recent efforts to define standards for business objects, common facilities, and the way to describe, use, and represent objects highlight this upheaval.

The mapping of the specified business items (objects and processes) to the existing generation of object-oriented programming languages (OOPLs) will become more and more automated. This is similar to the situation that existed when compilers became reliable and we finally learned to trust them. The cases where an assembler language programmer could improve the generated instructions in a timely manner, or even detect a compiler error, were diminished to nearly zero. A compiler is accepted now, as is the operating system or a disk drive, to be simply performing its task.

Nevertheless, the skills needed to specify an application using these new concepts are beyond those of an average programmer today. An application developer needs to understand much more of the business and the requirements on the computer system today than in the past. In answer to the question posed for this section, the application developer and the user, represented by the business analyst as requirements provider, need to come closer together, and their concepts and semantics have to converge in order to ease understanding.

Can software engineering really happen?

As in any other industry, in computer science there are some things that are understood by the layperson. There are also secrets—specific languages, tools, and processes, that are mastered only by the professional. From first Babylonian, Greek, and Roman cultures, then from medieval guilds to modern Europe, we have the concept of apprenticeship, a more or less formal way to become educated in a certain profession. An apprentice has a contract to gain well-defined knowledge and skills in his or her *métier* of choice. In this case a master (Meister, maître), a senior qualified person, transfers his or her knowledge and skills to the apprentice, while performing customary tasks.

Today we find this path to becoming a professional in "trainee" programs and "learning-on-the-job" situations. These terms, introduced from the United States, have made their way into European everyday usage and sometimes replace the more traditional expressions. Having gone through the phase of apprenticeship, a professional is accepted as qual-

ified, differentiating him or herself from the amateur.

In the field of software, the range of knowledge and skills that define the profession of application developer should include more than merely knowing and mastering certain programming languages. Yet today we encounter in nearly all information systems organizations programmers who were employed because of the high demand for programming skills.

Most of us do not trust an unknown thing when it first appears; we want to see that it works.

Many of these programmers originally intended to enter other professions, and were trained for them. But a programming job often meant a higher income or simply a safer position. This phenomenon is characteristic of a young, developing industry. We may compare it to the time of the Industrial Revolution, when unskilled and poorly educated people, mostly small farmers from underdeveloped areas, were hired to work in mines and factories in Europe and North America during the last century. None of these was considered an engineer.

But in both situations we find that soon a group of people emerged who were capable of controlling the development process, building an architecture, and designing particular parts in enough detail to be constructed by others. As discussed in the introduction to this essay, in our situation methods, tools, and techniques were developed to automate certain programming tasks (e.g., the higher-level programming languages and data storage systems). There are also methods, methodologies, techniques, and tools that were invented and, over time, refined to improve the software development process itself. The first category of methods and tools is analogous to machines and engines, the latter covers engineering aspects including software architecture, quality assurance, requirements analysis, and other aspects.

The approaches that use languages and symbols similar to those of systems analysts to describe a business situation demonstrate that we have passed a milestone along our path toward software engineering. There is also reason for hope that bridges over the gap between the requirements providers and the software developers will be in place soon. Techniques that reflect the customer requirements in a way that is closer to the language and abstraction patterns of the business will be even more helpful when they are automated, or at least capable of being transformed (see compilers) to executable systems without introducing errors. We find already a number of tools on the market providing animation, execution of models, and simulation capabilities. These tools are based on virtual machines, some capitalizing on expert system concepts. Here the artificial intelligence wave finds its renaissance in new clothes.

Must we always find new solutions, or can we use existing solutions?

You may think this is an unfair question. We have always reused software; in the era of the punched card we copied control cards. (To copy is one of the first things a programmer learns.) Many algorithms are no longer coded and invented by every programmer. Dynamic link libraries (DLLs) allow the common use of general functions; data storage patterns for standard situations are used without question. Yet there are discussions on reuse and complaints that we do not take advantage of other people's work and ideas.

It seems to be part of human nature that most of us do not trust an unknown thing when it first appears. People need to become familiar with a new thing; we want to see that it works before accepting and using it. Most of us hesitate; only a few are courageous enough to fearlessly try everything they encounter. The latter find their own path to the new thing and may soon demand improvements and new "toys"; the former need guidance before they can welcome new approaches.

Experience has shown that certain approaches to software systems development have been very successful in keeping to the schedule, or in producing extremely low error rates. As we examine these approaches, we find that they were intensively developed over a long time by an intimate team. The process and the techniques used were improved and refined again and again, and became very comprehensive, but also less simple and therefore harder to teach and to adopt.

We can find analogies to this in many fields. For example, a skilled cabinetmaker, either alone or with a team of skilled workers, can produce a custom-built kitchen that is beautifully made and exactly meets all requirements of the homeowner. In contrast, a contractor can be hired who purchases factory-made cabinets and, with his or her team, fits them into the kitchen, adding and adjusting pieces to adequately meet the requirements. This solution, although not precisely what the customer imagined, can be done for a much lower price.

We would like to find the industrial, or factory, solution for building software. Although the industrial approach may not produce the quality of the custom approach, especially in the beginning, it is the only way for our profession to truly become an engineering discipline and to satisfy the accelerating demand for software solutions.

To achieve a high volume of custom-made solutions, we would need many more highly skilled teams than we can produce. We cannot teach all the skills needed for perfect craftsmanship to so many people in a reasonable amount of time. Therefore we need to concentrate on the key ingredients of an industry, i.e., the sophisticated parts and assembly-line production.

To successfully sell software components or parts, precise descriptions of their interfaces are needed, but not information about their insides. A standard description format helps, as well. It reduces the amount that must be learned. Look at what engineering means in other industries, e.g., manufacturing. We need tools to develop parts. Such tools have been used successfully to produce custom-built software systems. To reach our goal of software engineering, we also need tools that are appropriate for the assembly of factory-produced software parts.

We can take from the papers in this theme ways to rearrange and compose existing software systems, or parts of them, so that they can become components and be assembled into new applications. The paper on workflow-based application development[18] shows how new systems can be built taking advantage of the results of business process models. Here the processes can be defined to fit the workflow, and wrapped components can be inserted. There are still drawbacks caused by the way software is currently constructed. Not every monolithic program can be wrapped so elegantly that it is of use for this kind of software development.

This and many of the recently published approaches[19] allow application development to more closely resemble the way other industries engineer their products. In other words, we are in the middle of the industrial revolution of the software industry. The vision of CASE gets another assist from the use of formal descriptions to solve business issues. Sophisticated metamodels supporting these approaches allow automation and bring us a step beyond the use of graphical editors instead of paper and pencil to draw nice diagrams.

As with its historic model in manufacturing, the Industrial Revolution, we will encounter powerful changes in the software business and in the professions in this field. It will proceed, however, as a fast-paced evolution rather than as a radical change in the development process. To this, all the approaches contribute that combine modern ideas with traditional procedures or adopt newer concepts to be used in older systems. You will find a few of them here, collected for this theme on application development.

Cited references and notes

1. F. Dyson, *Disturbing the Universe*, Harper & Row, New York (1979).
2. M. Pagnol, *Critique des Critiques*, Nagel, Paris (1949).
3. *American Heritage Dictionary of the English Language*, second edition.
4. T. E. Potok and M. A. Vouk, "The Effects of the Business Model on Object-Oriented Software Development Productivity," *IBM Systems Journal* **36**, No. 1, 140–161 (1997, this issue).
5. P. J. Guinan, J. G. Cooprider, and S. Sawyer, "The Effective Use of Automated Application Development Tools," *IBM Systems Journal* **36**, No. 1, 124–139 (1997, this issue).
6. First-generation languages are at the level of machine instructions; second-generation languages have a higher level of abstraction (FORTRAN, COBOL, and ALGOL). Third-generation languages add strong procedural and data structuring capabilities (Pascal, Modula-2, C, and Ada), and fourth-generation languages raise the level of abstraction still further. Program generators, decision-support languages, prototyping languages, and the formal specification languages that produce executable code are considered to be 4GLs. (See R. S. Pressman, *Software Engineering: A Practitioner's Approach*, second edition, McGraw-Hill, New York [1987].)
7. C. J. Date, *Guide to the SQL Standard*, third edition, Addison-Wesley Publishing Co., Reading, MA (1993).
8. P. P. Chen, "The Entity-Relationship Model—Towards a Unified View of Data," *ACM Transactions on Database Systems* **1**, No. 1, 9–36 (March 1976).
9. "Data mining" is a way to search through a large amount of

information, using user-supplied parameters, in an attempt to find meaningful and useful relationships between variables. It is up to the user to determine whether or not the correlations thus found are meaningful or simply coincidental.
10. Lt. Cmdr. (Mr.) Data, a character on the 1987–1994 science fiction television series "Star Trek: The Next Generation," was a state-of-the-art android with encyclopedic knowledge. As operations manager of the starship "Enterprise," he represented the voice of logic.
11. V. Srinivasin and D. T. Chang, "Object Persistence in Object-Oriented Applications," *IBM Systems Journal* **36**, No. 1, 66–87 (1997, this issue).
12. B. L. Whorf, *Language, Thought and Reality*, MIT Press, Cambridge, MA (1963).
13. Ibid., see second chapter, "The Linguistic Relativity Principle."
14. F. Vester, *Denken, Lernen, Vergessen* (Thinking, Learning, Forgetting), Deutsche Verlagsanstalt, Stuttgart, Germany (1975).
15. The Object Management Group is a software development consortium with more than 600 members, formed to create a component-based software marketplace. Its charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. See http://www.omg.org/bacgrnd.htm for more information.
16. From the OMG home page at http://www.omg.org, use the internal search capability to find more on these topics.
17. A. H. Lindsey and P. R. Hoffman, "Bridging Traditional and Object Technologies: Creating Transitional Applications," *IBM Systems Journal* **36**, No. 1, 32–48 (1997, this issue).
18. F. Leymann and D. Roller, "Workflow-based Applications," *IBM Systems Journal* **36**, No. 1, 102–123 (1997, this issue).
19. For example, those proposed in I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*, Addison-Wesley Publishing Co., Reading, MA (1995) and D. Taylor, *Business Engineering with Object Technology*, John Wiley & Sons, Inc., New York (1996).

Accepted for publication November 11, 1996.

Norbert Bieberstein *IBM Arthur K. Watson International Education Center, Chausee de Bruxelles, 135, 1310 La Hulpe, Belgium (electronic mail: bsn@vnet.ibm.com).* Mr. Bieberstein is a German-born software architect, developer, development project coach, and teacher. He received master's degrees in mathematics and geography from Aachen University of Technology, Aachen, Germany. Before joining IBM in 1989 he taught high school (mathematics, computer science, and geography) and developed software for technical and commercial application systems. He joined IBM's Software Development Laboratory in Hannover, Germany, as a consultant on software engineering. He later consulted on customer projects and taught courses at IBM's International Education Center in La Hulpe, Belgium. These experiences are reflected in his book, *CASE Tools: Auswahl, Bewertung, Einsatz* (CASE Tools: Selection, Evaluation, Usage), published by Hanser-Verlag, Munich, in 1993. From 1994 to 1996 he was on the staff of the technical director for application development at IBM's software headquarters in Somers, New York. In August, 1996, he accepted his current job as relationship manager for IBM's software products to the banking, finance, and insurance solution units of IBM in La Hulpe. During 1996 he also contributed to IBM's OMG activities in the areas of object-oriented analysis and design, meta-object facilities, and business object facilities related to proposals for standards to be adopted in 1997.

Reprint Order No. G321-5631.