Be Careful With "Use Cases"

By Edward V. Berard
The Object Agency, Inc.

Prologue

Recently, I have had conversations with a number of people who were attempting their first object-oriented project. Most of these people claimed to be using "use cases" in their object-oriented analysis and design approaches. However, listening to them describe their projects, it became quite apparent to me that there is a great deal of confusion regarding the definition, context, and use of use cases. This confusion can (and often does) lead to poorly-designed systems.

History

In June of 1979, the Ada programming language became a reality ([SIGPLAN, 1979a], [SIGPLAN, 1979b]). The U.S. Department of Defense set up Ada training classes at West Point, the Naval Post Graduate School, Georgia Institute of Technology, the National Physical Laboratory (U.K.), and the U.S. Air Force Academy in Colorado Springs, Colorado.

At the Air Force Academy, the task of designing a week-long Ada training course fell to Major Dick Bolz and Captain Grady Booch. It was impressed upon Booch and Bolz that any Ada training course must emphasize software engineering, not merely the syntax and semantics of the language. (This point was further reinforced at an ACM symposium on Ada in late 1980 ([ACM, 1980].) Booch realized that it would be difficult merely to teach an Ada "syntax course" in a week's time. So, he sought out some simple mechanism for incorporating software engineering into the training course.

Booch found the work of Russell J. Abbott ([Abbott, 1980]) to be particularly appealing. (Abbott's work was later revised and published as [Abbott, 1983].) Abbott's approach was first to write a paragraph describing a solution to the given problem. Once the paragraph was written, it could then be examined, and the nouns, noun phrases, and pronouns would suggest candidate Ada packages, and the verbs in the paragraph would suggest candidate functions and procedures to be contained within the Ada packages.

Booch liked Abbott's approach, and combined it, along with influences from Smalltalk (e.g., [Kay, 1993]), object-oriented computer hardware (e.g., [Intel, 1980], [Intel, 1981], and [Organick, 1983]), rigor in the software development process (e.g., [Robinson and Leavitt, 1977]), and a reverence for software engineering in general (e.g., [Ross et al., 1975]), into a process he called "object-oriented design." The first versions of Booch's object-oriented design process and notations can be found in [Bolz and Booch, 1981], [Booch, 1981], [Booch, 1982a], and [Booch, 1982b].

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

[The object-oriented landscape in 1980-1981 had a number of things in common with the landscape as we know it today. Graphical user interfaces, and other object-oriented concepts had been introduced by Sutherland ([Sutherland, 1963]). Simula, what many consider to be the first object-oriented programming language, had been around for years (e.g., [Dahl and Nygaard, 1966]). Alan Kay (e.g., [Kay, 1993]) had coined the terms "object-oriented" and "object-oriented programming" around 1970. Smalltalk had already gone through several revisions (e.g., [Goldberg and Kay, 1976], [Ingalls, 1978], and [Borning and Ingalls, 1982]). There were the beginnings of (mostly academic) discussions on object-oriented databases (e.g., [Baroody and DeWitt, 1981] and [Goldstein, 1980]). Still, it was very difficult to find someone who had even heard of the term "object-oriented."]

Booch's early efforts were important because of his attempt to view object-orientation not merely in terms of a relatively informal coding practice, but rather as a (at least partial) life-cycle process. Object-oriented programming, ordinarily, was viewed as similar to structured programming (e.g., [Dijkstra, 1969]), i.e., a process that focused primarily on coding discipline. To be sure, object-oriented programming required an informal, almost entirely intuitive, means of conceptualizing the problem, but there was little in the way of formal guidance provided. Booch, through his descriptions of the object-oriented design process and his diagramming techniques, was striving for something that was closer *in form* to structured design (e.g., [Yourdon and Constantine, 1979]) than to a largely informal coding strategy.

Booch's 1980-1981 concept of object-oriented design involved the beginnings of his now famous/infamous "Booch Diagrams," and a simply-defined process ([Booch, 1981]), i.e.:

"1. Define the problem.

"2. Develop an informal strategy for the abstract world.

"3. Formalize the strategy.

"a. Define the objects and their attributes.

"b. Define the operations on the objects.

"c. Define the interfaces.

"d. Implement the operations."

It is the second step in Booch's process (i.e., the development of an informal strategy) that involves what some today would call a "use case." The "informal strategy," as Booch originally described it, was in the form of a paragraph describing a solution to the defined problem. Of course, compared with what Jacobson was to describe years later, there was little in the way of a well-defined process for creating a set of well-defined use cases. ([Berard, 1985]) was one attempt to define a systematic and repeatable process for the creation of (both individual and sets of) "informal strategies.")

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

The purpose of the "informal strategy" was twofold:

1. The informal strategy provided a means of identifying candidate objects and their attributes, e.g., nouns, pronouns, and noun phrases were used to suggest candidate objects.

2. Just as importantly, the informal strategy described the interactions and interrelationships among the objects that would effect a solution, i.e., the informal strategy had to solve the defined problem.

In the early 1980s, there was much resistance to Booch's object-oriented design approach. Software practitioners — never known for their writing skills — complained about the informality of the paragraph format. Many suggested that more rigor (and, hence, better solutions) could be accomplished through the use of graphics (e.g., data flow diagrams and state transition diagrams a la [Ward and Mellor, 1985]), or formal (mathematical) techniques (e.g., VDM, described in [Jones, 1980] and [Jones, 1986]). Booch, himself, more than once, considered ways of integrating graphical techniques into the strategy defining process, e.g., the use of data-flow diagrams in [Booch, 1986].

Today, there are increasing numbers of fairly formal techniques for expressing object-oriented designs (e.g., "object-oriented Z" ([Stepney et al., 1992])). However, there is also a decided shift towards relatively simple ways of expressing object-oriented analysis and design. It is the very (seeming) informality of use cases that makes them attractive to many object-oriented technology users. We see use cases being incorporated several object-oriented methodologies, e.g., Fusion ([Coleman et al., 1994]). If use cases become part of the so-called Unified Method ([Booch and Rumbaugh, 1995]), in a very real sense, Booch will have come full circle.

By 1986, there were already quite a few object-oriented design approaches from which to choose. At the 1986 Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) conference ([ACM, 1986]), Ivar Jacobson presented his ideas on the design of large, real-time systems ([Jacobson, 1986]). In this presentation, Jacobson discussed such things as "blocks," "services," "objects," "signals," and "messages," along with references to a formal specification language, i.e., FDL.

It was not until the 1987 OOPSLA conference ([ACM, 1987]), however, that many in the object-oriented community were introduced to both "ObjectOry" and "use cases." In his 1987 presentation ([Jacobson, 1987]), Jacobson described ObjectOry as a technique, and use cases as a tool used within that technique. The "use" in use cases was clearly intended to be from the view of a "user of the system," e.g.:

> "The system is described in ObjectOry as a black box by describing a number of aspects of the system. These different aspects, each corresponding to a behaviourally related sequence are called use cases.
>
> "The basis of ObjectOry is that it shall be designed for its users. We want to build the system for them. In order to safeguard that the users really get the system they want and need, we want to structure the system's total behaviour in aspects, where each aspect corresponds to what we can call a use case."

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

Jacobson has continued to publicize both use cases and Objectory in books (e.g., [Jacobson et al., 1992] and [Jacobson et al., 1995]), tutorials (e.g. [Christerson and Jonsson, 1995]), training seminars, and CASE tools.

Usually, those methodologists who advocate the creation of use cases, or similar items, suggest three primary motivations for use case creation:

- gaining an understanding of the problem,
- capturing an understanding of the proposed solution, and
- identifying candidate objects.

Jacobson goes further than this. He points to the use of use cases in the testing process, e.g., from [Jacobson et al., 1992]:

> "... For the first time several classes, blocks, service packages and subsystems are brought together and therefore the testing should concentrate on this. Each use case is initially tested separately. The use cases constitute an excellent tool for integration test since they explicitly interconnect several classes and blocks. When all use cases have been tested (at various levels) the system is tested in its entirety. The several use cases are executed in parallel and the system is subjected to different loads."

I would like you to take note of two interesting points:

1. People complained about the use of spoken language descriptions of systems in the Booch/Abbott approach in the early 1980s. However, today, the use of the same technique, under the name of "use case," is widely popular.

2. Some people consider use cases to somehow be uniquely object-oriented (almost a "defining characteristic"), although, as Jacobson describes them, they could be used with practically any software development approach.

In 1990, Elizabeth Gibson (then working for ParcPlace) first publicly described an object-oriented design process called "object behavior analysis" or OBA (e.g., [Gibson, 1990] and [Gibson, 1991]). In OBA, a developer would write a number of "scripts" describing how a user would interact with (use) the planned system. The scripts could then be used to gain a better understanding of the intended system, and to identify candidate objects. I also note that the scripts generated in OBA are strikingly similar to Jacobson's use cases.

I should also add that approaches such as Rebecca Wirfs-Brock's Responsibility-Driven Design (e.g., [Wirfs-Brock et al., 1990]) advocate the examination of "system requirements" with the intention of both identifying candidate objects, and establishing interactions and interrelationships among the objects. In the context of such approaches, we could use use cases as a means of describing (creating) the requirements.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

Defining and Understanding Use Cases

In [Jacobson et al., 1995], use cases are defined as follows:

> "A use case is a sequence of transactions in a system whose task is to yield a measurable value to an individual actor of the system."

To better understand this definition, we note that:

- a **use case** is "a specific flow of events through the system, that is, an instance" ([Jacobson et al., 1995]). Using the concept of a class as the set of all items which share a collection of similar characteristics, it is suggested that many similar courses of events be grouped into a "use-case class." (Note that this definition, i.e., a class is a *set* of instances, is not the same definition of class that is used in a Smalltalk, C++, or Eiffel context.)

- an **actor** is "a role that someone or something in the environment can play in relation to the business" ([Jacobson et al., 1995]). Alternatively, [Jacobson et al., 1992] defines an actor as representing "everything that needs to exchange information with the system," and [Christerson and Jonsson, 1995] defines actors as "everything that interacts with the system." An "individual actor" (sometimes referred to as a "user") is defined to be an instance of class actor. Further, the same person (or other item) can assume more than one role.

- "transactions in a system" implies that the system will make available to its actors a set of capabilities that will both allow the actors to communicate with the system and to accomplish some meaningful work (i.e., meaningful value).

- "a measurable value" implies that the performance of the task has some visible, quantifiable, and/or qualifiable impact on those things which lie outside of the system, and, in particular, the actor who initiated the task.

- a **transaction** is defined as "an atomic set of activities that are performed either fully or not at all. It is invoked by a stimulus from an actor to the system or by a point in time being reached in the system. A transaction consists of actions, decisions and transmission of stimuli to the invoking actor or to some other actor(s)." (See [Jacobson et al., 1995].)

[Jacobson et al., 1992] notes that: "the set of all use case descriptions specifies the complete functionality of the system."

[Jacobson et al., 1995] provides the following advice for use case creation, and an accompanying use case example:

> "... If we try to describe a use case that contains a great many alternative courses of events, our text can easily become difficult to understand. Therefore, it is wise to use some form of structured writing approach. In this case, our example — the use case Serving Dinner — could be as follows:

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

"Basic flow of events:

"A. The user case begins when the actor Guest enters the restaurant.

"B. The actor Guest has the possibility of leaving his/her coat in the cloakroom, after which he/she is shown to a table and given a menu.

"C. When the actor Guest has had sufficient time to make up his/her mind, he/she is asked to state his/her order. Alternatively, Guest can attract the waiter's attention so that the order can be placed.

"D. When the Guest has ordered, the kitchen is informed what food and beverages the order contains.

"E. In the kitchen, certain basic ingredients, such as sauces, rice, and potatoes, have already been prepared. Cooking therefore involves collecting together these basic ingredients, adding spices and so on and sorting out what needs to be done just before the dish is served. Also, the required beverages are fetched from the refrigerator.

"F. When the dish is ready, it is served to the actor Guest. When it has been eaten, the actor is expected to attract the waiter's attention in order to pay.

"G. Once payment has been made, Guest can fetch his/her coat from the cloakroom and leave the restaurant. The use case is then complete."

[Jacobson et al., 1995] goes on to say that, as with any use case, the above use case may be augmented with (stated) alternative courses of events. (The "structured writing approach" should not be overly complex, e.g., it should have much of the flavor of "structured English" (e.g., [DeMarco, 1979]).)

Most, if not all, references to Objectory stress that use cases should be used throughout the development part of the software life-cycle, e.g., during analysis, design, and testing. [Jacobson et al., 1995] and [Christerson and Jonsson, 1995], among others, even point out that use cases can be an important tool in business process modelling and business process reengineering as well.

Potential Problems With Use Cases

Watch out for conflicts in localization strategies. **Localization** is the process of placing items in close physical proximity to each other, usually with the connotation of having some mechanism for precisely defining the boundaries of the "area" into which the items are being gathered. Different development approaches require different localization schemes:

- Functional decomposition approaches localize information around *functions*.
- Data-driven approaches localize information around *data*.
- Object-oriented approaches localize information around *objects*.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

Given a particular abstraction scheme, we can gauge the quality of the localization within a design:

- One can assess the cohesiveness of a set of localized items (a set of items that have been placed in close physical proximity to each other. Specifically, the better the localization strategy, and the execution of that strategy, the stronger will be the logical relationships among the localized items.

- One can also assess the coupling among different localized sets of items within the same design. Specifically, if we have a good localization strategy, and a good execution of that strategy, well-localized sets of items should be loosely coupled, if at all, with other (different) well-localized sets of items within the same design.

Problems often arise in software development efforts when *differing* localization strategies are used during the *same* development effort. For example, suppose we are presented with a set of *functional* requirements, and we are told to product an *object-oriented* solution. This means that there will be a fundamental shift in localization strategies somewhere between the requirements and the creation of the final product. This shift often results in the introduction of significant errors.

Use cases, as they are most commonly described and used, are almost, if not entirely, functional in nature. This, by itself, is not a problem. However, we must be careful when using use cases within the context of object-oriented development.

Many software practitioners have a good deal of experience in functional decomposition approaches to system development. Use cases present them with an opportunity to continue their functional view of software development. I recently talked with one organization that described their "object-oriented" approach to software development as follows:

1. The development team determined the main (highest level of detail) functional capabilities of their intended application.

2. A high-level use case was written for each high-level functional capability.

3. Each high-level use case was handed to a separate team for further elaboration (decomposition).

4. Eventually, based on progressively more detail being added to the use cases, each team would implement (write code for) their particular portion of the system.

5. The efforts of each team would be integrated into the final product.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

This (almost logical sounding) approach is a recipe for disaster. The key point is to remember that objects are not functions, and functions are not objects. In the mid-1980s, I worked with more than one organization that suffered the consequences of a functional decomposition "front end," and an object-oriented "back end." Here is what happened:

- Each large functional partition had objects in common with the other functional partitions. Without a full-time dedicated effort to identify those objects that were common to two or more partitions, there was a good deal of duplication of effort.

- The functional decomposition at the beginning scattered parts of many objects across more than one functional partition. That is, to have a chance of accurately designing any given object, one would have to survey all of the partitions for information regarding that object. This resulted in major problems when it came to system integration. For example, when different parts of the system attempted to communicate with each other by passing objects among themselves, they found that each part of the system had a *different* implementation of the *same* object. This resulted in a significant amount of re-design and re-coding.

My advice is simple. It is perfectly proper and appropriate to use use cases to describe an external (user-oriented) view of the system. However, avoid the temptation to use this functional view of the system as a basis for the creation of an object-oriented architecture for that same system. As many of my clients will tell you, objects and functions do not map to each other on a one-to-one basis, and the architecture of an object-oriented system is significantly different from the architecture of a functionally decomposed system.

(Yes, I know that pathological combinations of functional architectures and object-oriented architectures are often encouraged by some of today's most popular methodologies, e.g., Objectory's so-called "control objects," Booch's so-called "manager objects" (e.g., see chapter 8 of [Booch, 1994]), Bertrand Meyer's so-called "command objects" ([Meyer, 1988]), and the countless examples of "classes" that encapsulate functions only, i.e., so-called "stateless classes." It seems that functional decomposition is still deeply entrenched in the software psyche.)

Another potential problem with use cases is the temptation to violate information hiding. **Information hiding** is the process of making certain pieces of information inaccessible. We often speak of the information as being hidden within a "black box." D.L. Parnas (e.g., [Parnas, 1972]), the man who coined the term "information hiding," advocated that the details of difficult and likely-to-change design decisions be hidden from the rest of the system. Further, the rest of the system will have access to these design decisions only through well-defined, and (to a large degree) unchanging interfaces.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

Information hiding is one of the most important aspects of object-orientation. For example, "looking inside" (viewing the underlying implementation of) an object is an activity that is reserved strictly for the creator or owner of that object. Users of an object interact with that object only through the interfaces provided by the creator or owner of the object. If a user of an object insists that he or she must have access to, or knowledge of, the underlying implementation of an object, then one of two situations is true, i.e., either:

- the user of the object is confused or ill-informed (e.g., as a result of incomplete or missing documentation), and really does not need to know anything about the underlying implementation, or
- we are dealing with a poor (substandard) object-oriented design.

When developing use cases, we must know not only the item for which we developing the use case, but also the defined/intended public interface for that item. (By "item," we mean a cohesive, well-defined system, or system component.) Developers of use cases should avoid the temptation to go beyond the public interface of an item, and attempt to describe the internal structure (design, architecture) of the item.

When we violate information hiding, we pay the consequences. For example:

- There is an increase in the level of detail with which a software practitioner must deal, and a corresponding increase the probability of errors. (As Dijkstra said of himself in [Dijkstra, 1965], "I have only a very small head and must live with it.")
- There is also a very definite increase in the probability that one part of the system will make assumptions about the underlying implementations of another part of the same system. This tightens the object coupling (see, e.g., chapter 7 of [Berard, 1993]), and, thus, makes the overall system unacceptably vulnerable to negative side effects when changes are introduced. Tightened object coupling also significantly reduces reusability.

Again, the advice is simple. When constructing a use case for an item, do not describe any details of the underlying implementation of that item. For example, when describing an Automated Teller Machine, it is perfectly proper to construct a use case that describes the sequences of activities, and the necessary input and output information, associated with, say making a deposit. It would not be proper within the same use case, for example, to provide any information on the details of the algorithm used to credit the deposit to the user's account.

As was mentioned earlier, [Jacobson et al., 1992] notes that: "the set of all use case descriptions specifies the complete functionality of the system." This means that software developers who make use of use cases, will often find themselves dealing with sets of use cases. Further, since many approaches recommend the use of use cases throughout the development and testing processes, each level of abstraction (detail) will have its own (hopefully complete and accurate) set of use cases.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

Creating, partitioning, maintaining, and otherwise using many different use cases, must be done with care:

- There must be some well-defined system for the configuration management (e.g., [Ambriola and Bendix, 1989], [IEEE, 1987], and [STSC, 1994]) of the use cases. Ideally, there is an established, well-designed, and tightly-integrated set of policies, procedures, guidelines, standards, and tools for the configuration management of all items either captured or produced during the development and testing of a product. Use cases, and their associated information, must be integrated into this system.

  I have seen more than one project where the sheer number of use cases quickly became unmanageable. It was difficult to find all the use cases, much less tell if you were dealing with the most recent ones.

- Very often, the perceived informality of use cases lulls people into a false sense of security. People become very lax regarding such things as naming conventions and other style issues. This not only increases the chances for errors, but it also decreases the chances for reuse of the use cases.

- Even if there is only one person working on the project, it is possible for two or more use cases to conflict with each other. As the number of use cases for a particular project grows, so too does the probability of contradictions among the use cases.

- I have seldom seen people take the time to test (or inspect, or walk-through, if you please) a set of use cases. As changes are introduced into the product, and as the number of use cases grows, it is very important to have some means of continually re-establishing the consistency, validity, and appropriateness of the entire set of use cases.

- There is often a great deal of confusion regarding requirements and use cases, and design and use cases. For example, is a complete set of use cases generated early on during a project the same thing as the requirements for the product? (They could be.) Are there any (product or project) requirements that are not captured in the set of use cases? Is there any aspect of the system design/architecture not captured in the set of use cases? People need clear understandings and guidelines regarding the relationships among use cases, analysis, and design.

- Coverage is a major problem with those who use use cases. It is one thing to say that that: “the set of all use case descriptions specifies the complete functionality of the system.” It is quite another thing to demonstrate that you indeed have completely captured the functionality of the system in the given set of use cases. (Software practitioners should also take care to ensure that there are no extraneous or superfluous use cases.)

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

In dealing with individual use cases, software engineers should keep the following in mind:

- There should be a published style guide for in-house use cases. Letting every software engineer "do his/her own thing" increases the number of errors, makes testing much more difficult, limits the possibility of use case reuse, and decreases the overall efficiency of the project.

- I have found that very few software practitioners have an adequate sense of the proper level of detail that should be associated with a given use case. A survey of the use cases at a given site will most likely reveal that some use cases have so little detail in them that they are unacceptably ambiguous, while others are so detailed that the slightest change to the requirements will cause them to be rewritten.

  I recently talked with one installation where the staff had broken use cases down to the level of "pushing a button," e.g., the goal of one particular use case was that a particular button in the graphical user interface be pushed. The staff had lost sight of one of the most basic concepts of use cases, i.e., a use case must be some sequence of actions that yields a measurable value to an individual actor of the system. There are very few systems where the customer would say that pushing a button, or selecting a menu item constituted a "measurable value."

- A very common comment among those who use use cases is that the use cases will be used to test the overall system. Often, however, when I ask about regular, systematic, and effective testing (reviews, inspections, walk-throughs) of use cases, I get blank stares. What "testing" I have seen for use cases is often too informal to be effective.

Final Comments

Use cases are a new variation on an old theme, i.e., describe the system in terms of how the user will see it, and in terms of delivered measurable value to the user. (See, for example, the discussions of process descriptions ("mini specs") in [DeMarco, 1979].) Even within the object-oriented community, there are other very similar approaches.

It is the very simplicity of use cases that makes them so powerful. Yet, this simplicity can lead to problems if the creation, integration, and maintenance of use cases is not carefully controlled. The larger and/or more critical the end product, the greater will be the need for rigor in the creation and handling of use cases.

Lastly, in using use cases, object-oriented software engineers should be on guard for problems of conflicts in localization strategies, violations of information hiding, and sloppiness in the creation and configuration management of use cases.

Bibliography

[Abbott, 1980]. R.J. Abbott, "Report on Teaching Ada," *Technical Report SAI-81-313-WA*, Revised December 1980, Research Report for DARPA Order No. 3456, Contract No. MDA 903-80-C-0188, Science Applications, Inc. McClean, Virginia, 1980.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

[Abbott, 1983]. R. J. Abbott, "Program Design by Informal English Descriptions," *Communications of the ACM*, Vol. 26, No. 11, November 1983, pp. 882 - 894.

[ACM, 1980]. Association for Computing Machinery, *Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language*, Boston, Massachusetts, December 9-11, 1980, Association for Computing Machinery, New York, New York, ISBN 0-89791-030-3, 242 pages, 1980.

[ACM, 1986]. Association for Computing Machinery, *OOPSLA '86 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 21, No. 11, November 1986.

[ACM, 1987]. Association for Computing Machinery, *OOPSLA '87 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 22, No. 12, December 1987.

[Ambriola and Bendix, 1989]. V. Ambriola and L. Bendix, "Object-Oriented Configuration Control," *Proceedings of the 2nd International Workshop on Software Configuration Management*, special issue of Software Engineering Notes, Vol. 17, No. 7, November 1989, pp. 133 - 136.

[Baroody and DeWitt, 1981]. A.J. Baroody and D.J. DeWit, "An Object-Oriented Approach to Database System Implementation," *ACM Transactions on Database Systems*, Vol. 6, No. 4, December 1981, pp. 576 - 601.

[Berard, 1985]. E.V. Berard, *An Object-Oriented Design Handbook for Ada Software*, EVB Software Engineering, Inc., Frederick, Maryland, 1985.

[Berard, 1993]. E.V. Berard, *Essays on Object-Oriented Software Engineering, Volume 1*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[Bolz and Booch, 1981]. D. Bolz and G. Booch, *Software Engineering With Ada*, Department of Astronautics and Computer Science, United States Air Force Academy, Colorado Springs, Colorado, 1981.

[Borning and Ingalls, 1982]. A.H. Borning and D.H.H. Ingalls, "Multiple Inheritance in Smalltalk-80," *Proceedings of the National Conference on Artificial Intelligence*, August 1982, pp. 234 - 237.

[Booch, 1981]. G. Booch, "Describing Software Design in Ada," *SIGPLAN Notices*, Vol. 16, No. 9, September 1981, pp. 42 - 47.

[Booch, 1982a]. G. Booch, "Object Oriented Design," *Ada Letters*, Vol. I, No. 3, March-April 1982, pp. 64 - 76.

[Booch, 1982b]. G. Booch, "Solve Process-Control Problems With Ada's Special Capabilities," *EDN*, June 23, 1982, pp. 143 - 152.

[Booch, 1983a]. G. Booch, *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, California, 1983.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

[Booch, 1983b]. G. Booch, "Object Oriented Design," *IEEE Tutorial on Software Design Techniques*, Fourth Edition, P. Freeman and A.I. Wasserman, Editors, IEEE Computer Society Press, IEEE Catalog No. EHO205-5, IEEE-CS Order No. 514, pp. 420 - 436.

[Booch, 1986]. G. Booch, "Object Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211 - 221.

[Booch, 1994]. G. Booch, *Object-Oriented Analysis and Design With Applications*, Second Edition, Benjamin/Cummings, Menlo Park, California, 1991.

[Booch and Rumbaugh, 1995]. G. Booch and J. Rumbaugh, *Unified Method: User Guide, Version 0.8*, Rational Software Corporation, Santa Clara, California, 1995.

[Christerson and Jonsson, 1995]. M. Christerson and P. Jonsson, *Tutorial 1: Object-Oriented Software Engineering*, Tutorial Notes, Presented at OOPSLA '95, October 15-19, 1995, Austin, Texas, Association for Computing Machinery, New York, New York, 1995.

[Coleman et al., 1994]. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Dahl and Nygaard, 1966]. O.J. Dahl and K. Nygaard, "SIMULA — an ALGOL-Based Simulation Language," *Communications of the ACM*, Vol. 9, No. 9, September 1966, pp. 671 - 678.

[DeMarco, 1979]. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, New York, 1979.

[Dijkstra, 1965]. E. Dijkstra, "Programming Considered as a Human Activity," Proceedings of the 1965 IFIP Congress, Amsterdam, The Netherlands, North Holland Publishing Company, 1965, pp. 213 - 217, Reprinted in *Classics in Software Engineering*, E.N. Yourdon, Editor, Yourdon Press, New York, New York, 1979, pp. 3 - 9.

[Dijkstra, 1969]. E.W. Dijkstra, "Structured Programming," originally appeared in a report on a conference sponsored by the NATO Science Committee, Rome, Italy, October 1969, reprinted in *Classics in Software Engineering*, Edited by Edward N. Yourdon, Yourdon Press, New York, New York, 1979, pp. 43 - 48.

[Gibson, 1990]. E. Gibson, "Objects — Born and Bred," *Byte*, Vol. 15, No. 10, October 1990, pp. 245 - 246, 248, 250, 252, 254.

[Gibson, 1991]. E. Gibson, "Flattening the Learning Curve: Educating Object-Oriented Developers," *Journal of Object-Oriented Programming*, Vol. 3, No. 6, February 1991, pp. 24 - 29.

[Goldberg and Kay, 1976]. A. Goldberg and A. Kay, Editors, *Smalltalk-72 Instructional Manual*, Technical Report SSL-76-6, Xerox PARC, Palo Alto, California, March 1976.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

[Goldstein, 1980]. I. Goldstein, "Integrating a Network-Structured Database Into an Object-Oriented Programming Language," *Proceedings of the Workshop on Data Abstraction Database and Conceptual Modelling*, M.L. Brodie and S.N. Zilles, Editors, Pingree Park, Colorado, June 23-26, 1980, pp. 124-125.

[IEEE, 1987]. Institute for Electrical and Electronics Engineers, *An American National Standard: IEEE Guide to Software Configuration Management, ANSI/IEEE Std 1042-1987*, The Institute of Electrical and Electronics Engineers, Inc., New York, New York, 1987.

[Ingalls, 1978]. D.H.H. Ingalls, "The Smalltalk-76 Programming System Design and Implementation," *Fifth Annual ACM Symposium on the Principles of Programming Languages*, January 1978, pp. 9 - 15.

[Intel, 1980]. Intel Corporation, *iAPX 432 Object Primer*, Manual 171858-001, Revision B, Intel Corporation, Aloha, Oregon, 1980.

[Intel, 1981]. Intel Corporation, *Engineering Specifications of the iAPX Extensions to Ada*, Manual 171871-001, Intel Corporation, Aloha, Oregon, January 1981.

[Jacobson, 1986]. I. Jacobson, "Language Support for Changeable Large Real Time Systems," *OOPSLA '86 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 21, No. 11, November 1986, pp. 377 - 384.

[Jacobson, 1987]. I. Jacobson, "Object-Oriented Development In an Industrial Environment," *OOPSLA '87 Conference Proceedings*, special issue of *SIGPLAN Notices*, Vol. 22, No. 12, December 1987, pp. 183 - 191.

[Jacobson et al., 1992]. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, 1992.

[Jacobson et al., 1995]. I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage: Business Process Reengineering With Object Technologyt*, Addison-Wesley, Reading, Massachusetts, 1995.

[Jones, 1980]. C.B. Jones, *Software Development: A Rigorous Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[Jones, 1986]. C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[Kay, 1993]. A.C. Kay, "The Early History of Smalltalk," *SIGPLAN Notices*, Vol. 28, No. 3, March 1993, pp. 69 - 95.

[Meyer, 1988]. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Organick, 1983]. E. Organick, *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, New York,1983.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090

[Parnas, 1972]. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems Into Modules," *Communications of the ACM*, Vol. 5, No. 12, December 1972, pp. 1053-1058.

[Robinson and Leavitt, 1977]. L. Robinson and K. Leavitt, "Proof Techniques for Hierarchically Structured Programs," in *Current Trends in Programming Methodologies, Volume 2*, Raymond T. Yeh, Editor, Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[Ross *et al.*, 1975]. D.T. Ross, J.B. Goodenough, C.A. Irvine, "Software Engineering: Process, Principles, and Goals," *IEEE Computer*, Vol. 8, No. 5, May 1975, pp. 17 - 27.

[Rumbaugh et al., 1991]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[SIGPLAN, 1979a]. Anonymous, "Preliminary Ada Reference Manual," SIGPLAN Notices, Vol. 14, No. 6A, June 1979.

[SIGPLAN, 1979b]. Anonymous, "Rational for the Design of the Ada Programming Language," SIGPLAN Notices, Vol. 14, No. 6B, June 1979.

[Stepney et al., 1992]. S. Stepney, R. Barden, and D. Cooper, Editors, *Object-Orientation in Z*, Springer-Verlag, London, United Kingdom, 1992.

[STSC, 1994]. Software Technology Support Center, *Software Configuration Management Technology Report*, September 1994, Software Technology Support Center, Hill Air Force Base, Utah, 1994.

[Sutherland, 1963]. I. Sutherland, *Sketchpad, A Man-Machine Graphical Communication System*, Ph. D. Thesis, Massachusetts Institute of Technology, January 1963.

[Ward and Mellor, 1985]. P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems, Volumes 1-3*, Yourdon Press, New York, New York, 1985.

[Wirfs-Brock et al., 1990]. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Yourdon and Constantine, 1979]. E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1979.

101 Lakeforest Blvd., Suite 380
Gaithersburg, Maryland 20877
(301) 548-9090