Technical Report

CMU/SEI-89-TR-025
ESD-89-TR-033

Classifying Software Design Methods

John P. Long

William G. Wood

David P. Wood

August 1989

Classifying Software Design Methods

John P. Long

SYSCON Corporation,
Resident Affiliate, SEI

William G. Wood

David P. Wood

Specification and Design Methods
and Tools Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

Table of Contents

List of Figures

Classifying Software Design Methods

Abstract. A few mature and popular methods are currently being used to specify and design real-time embedded systems software, and these methods are the basis for a large number of tools automating the process. Unfortunately, some of the tools support only parts of a method, while others support a mixture of different methods. Because of the large number of tools involved, companies selecting tools for their particular needs are faced with a significant problem. As a result, the choice of tools often depends on the best salesperson rather than on the most appropriate method, leading to disappointment on the part of end users of the tools. The Software Engineering Institute (SEI) has had a project underway for some time that provides a basis for selecting methods and tools. This paper describes some of the results of this effort with respect to classifying design methods for Ada-based software.

1. Introduction

There are many organizations of various shapes and sizes engaged in evaluating software development methods, and tools supporting those methods, for use in developing applications software for embedded systems. Current "popular" methods have been developing over the past 15 years, and have recently increased in popularity due to the development of bit-mapped graphical workstations that support automation of those methods. There have been numerous forms of publications interrelating design methods: special issues [TSE 86]; surveys [Yau 86]; classifications [Hesse 84, Kelly 87]; tutorials [Freeman 83, Bergland 81]; and comparisons [DoD 82]. All of these publications are useful for understanding, classifying, and evaluating design methods; but none of them leads to a process for effectively comparing methods with the intent of selecting a method for a software development project. The Software Engineering Institute (SEI) has had a project underway for some time that provides a basis for selecting methods and tools. This paper describes some of the results of this effort with respect to classifying design methods for Ada-based software. Some of the principal project ideas that differentiate this classification from the previously noted classifications are enumerated below.

1. Separate the classification and evaluation of methods from supporting tools. This is necessary to distinguish between a good method with poor tool support and a poor method with good tool support.

2. Separate the classification of methods (describing what a method does) from the evaluation of a method's capabilities (describing how well it does it).

3. Use a simple and concise classification scheme, even if evaluation criteria are complex and numerous.

4. Use a scheme that is independent of the process that produces the software. A successful approach does not consider whether the process uses a waterfall or spiral model, or whether it proceeds from a bottom-up, top-down, or middle-out point of view.

The remainder of the paper describes the classification scheme. In Appendix A, a classification of some of the currently popular design methods for Ada-based software is presented.

2. Classification Scheme

At all stages in a development process, a representation of the system under development must be created, and it is important to understand how this representation is created. Following [Hesse 84], each method has three considerations:

- What is the form of representation of the artifacts?
- How are these representations derived?
- How are the representations examined?

The representations may include graphical and textual information, and such considerations as how they can be partitioned for separate development and integrated smoothly when necessary. The derivation of the representations can involve some automatic transformations, but consists largely of guidelines, rules, and procedures to successfully derive the representations. The examination of representations can be analytic (and hence can be automated) or can involve manual techniques such as walkthroughs and inspections.

2.1. Views of the System

The classification scheme described in this paper uses only the different forms of representation to classify the methods, and following [Harel 86], considers three distinct views as described below.

1. The **functional** view shows the system as a set of entities performing relevant tasks. This includes a description of the task performed by each entity and the way the entity interacts with other entities and with the environment. Ideally, the functional view should complement the behavioral view (described below). Each transaction in the behavioral view should be traceable through the system from the initial input through the interfaces and functional units to the final output. The functional view is the normal starting point for the design process because it is commonly the way the system is decomposed into smaller and simpler parts.

2. The **structural** view shows how the system is put together—the components, the interfaces, and the flow between the components through the interfaces. It also shows the environment, and the interfaces and information flows between it and the system. Ideally, the structural view should be an elaboration of the functional view. Each entity in the functional view is decomposed into a set of primitive software components that can be implemented separately and then combined to build the entity. The design process therefore generally converts a functional view into a structural view. However, the structure of a system is influenced by resource constraints that prevent the use of arbitrarily many or arbitrarily large components. The structure is also influenced by certain implementation constraints, which require that specific types of components (e.g., MIL-STD-1750a processors) be used, or that components be connected in a specific manner (e.g., by MIL-STD-1553 buses).

3. The **behavioral** view shows the way the system will respond to specific inputs: what states it will adopt, what outputs it will produce, what boundary conditions exist on the validity of inputs and states. This includes a description of the environment that is producing the inputs and consuming the outputs. It also includes constraints on performance that are imposed by the environment and function of the system. Real-time systems especially have performance requirements as an essential part of their correct behavior.

2.2. Software Development Stages

The different development stages used in the classification scheme are enumerated below. Each development stage is characterized by what it represents and the way it represents it.

1. The first stage is to take an ambiguous, incomplete, and inconsistent requirement and turn it into a flawless **specification**. This is probably not yet possible with current technology, but there are many reasonable ways of proceeding that give a serviceable specification. The specification describes what the software is to do and the constraints to be imposed on the designers. Although the design process is not the primary consideration in this paper, it is worthwhile noting that production of the specification is not limited to a front-end activity, but will change throughout the life cycle of the system.

2. The **design** representation describes how the system is structured to satisfy the specification. It describes the system in a large-grained manner. It defines the breakup of the system into major tasks, describes persistent data objects and their access mechanisms, the important abstract data types and their encapsulation in the heavyweight tasks, and the message structures between the tasks. There must also be some consideration for how the resources are to be allocated and the performance requirements satisfied.

3. The final development stage is **implementation** with source code, object code, resource usage, and initialized data structures. This is the level at which algorithms are developed and represented explicitly.

We chose not to include the **requirement**, since it is a description of what end-user audiences view as their needs, is often a rather eclectic description, and usually covers the needs of each audience in the end-user community in a very uneven manner.

2.3. High-Level Classification Scheme

The scheme for the classification of software development methods is presented in [Firth 87]. The matrix categorizes methods into three **development phases** (specification, design, and implementation) and three **forms of representation** (functional, structural, and behavioral) as shown in Figure 1. This scheme is considered useful for general classifications to initially reduce the number of methods to consider. The classification scheme presented in the following section represents an enhancement of the original matrix that facilitates the side-by-side, in-depth comparison of multiple design methods.

3. Classification of Design Methods for Ada Software

This section presents classifications of four methods that have been proposed for the design of Ada-based software systems. The derivation of the classification scheme is discussed, along with details of the classification itself.

3.1. Derivation of the Classification Scheme

The high-level classification matrix (Figure 1) describes a single method using the vertical matrix column to show development phases and the horizontal rows to show representational forms. The detailed scheme (shown in Figures A-1 through A-8) provides the capacity to describe several methods on one page, where each vertical column describes one method and the horizontal rows classify specific aspects of each method, simplifying direct comparison of features.

Figure 1: The Classification Matrix

The detailed scheme can comprise one or several charts. At a minimum, one chart must be provided that contains the same information provided on the high-level matrix, namely, the development phase(s) covered by the method and the representation forms used. In addition, rows are also provided in this first chart to describe high-level classifications of notations, derivation techniques, and forms of examination available with each method.

Since the methods to be classified are all Ada-based and use graphical representations, there is more depth in the charts on both **Ada** and **graphical representations**. During the development of the classification charts, two or three of the methods were quickly charted to test the relevance of the criteria. Not all criteria for classification of design methods was intended to be included in the charts. Our classification charts were devised to illuminate the Ada-based and graphical nature of the methods classified. We did not include criteria which would produce empty rows even though the criteria may be of importance to designs of particular systems. Additional charts could be devised to classify further items of importance or to provide a greater level of detail as necessary.

We feel that a clear distinction between classifying methods (describing what methods do) and evaluating methods (saying how well they do it) is important. The goals of the classification scheme are to keep the classification simple, but meaningful. The classification scheme is directed toward technical considerations, while administrative and economic considerations are left to evaluation.

3.2. Background on the Design Methods

The classification scheme discussed above has been used to describe four methods for software design which we will refer to as: Booch, Buhr, PAMELA, and SYSCON. The background on each of these methods follows.

In his book entitled *Software Engineering in Ada*, Grady Booch "introduced an object-oriented design methodology that exploits the power of Ada and, in addition, helps us to manage the complexity of large software solutions." An additional Booch book, *Software Components with Ada*, concentrates on reusable software components, Ada programming style, object-oriented techniques, data structures and algorithms. The classification is based the software development principles and the object-oriented design methodology embodied in these books, [Booch 83] and [Booch 87].

The objectives of R. J. A. Buhr's book entitled *System Design with Ada* were to provide a design-oriented introduction to Ada, to present and to illustrate a graphical design notation, and "to arm the novice system designer with philosophies, strategies, tactics, techniques and insights into ways of effectively carrying out the design process". The classification is based the "object-oriented structured design" approach presented in the book, [Buhr 84].

PAMELA, Process Abstraction Method for Embedded Large Applications, and PAMELA-2, Pictorial Ada Method for Every Large Applications, were created by George W. Cherry. PAMELA was documented in the *PAMELA Designer's Handbook* as an Ada specific software development method that is based on a high-level, graphical program design and description language. PAMELA-2 retains PAMELA's features for supporting process-oriented design, and was enhance to add support for object-oriented design and behavioral specifications. The graphical representations were adapted from [Booch 83] and [Hoare 85]. The PAMELA method is directly supported by the software tool AdaGRAPH. This classification was based on both *PAMELA Designer's Handbook*, [AdaGraph 86a] and [AdaGraph 86b], and *PAMELA-2: An Ada-Based Object-Oriented Design Method*, [Cherry 88].

SYSCON's methodology manual describes the design method developed during their work on Ada contracts and Ada research and development projects. The method and the accompanying graphical representations were influenced by the work of [Booch 83] and [Buhr 84]. SYSCON's development of a suite of Ada design and development tools illustrate the method. Therefore, SYSCON's methodology manual and users manuals from the suite of tools were referenced for the classification. Currently, SYSCON's methodology manual and the suite of tools are company proprietary.

4. Conclusion

There are some mature software development methods and many tools supporting those methods (or parts of them) available in an expanding commercial marketplace. Those methods and tools can assist experienced engineers and managers to produce a high quality, maintainable software product. But they should be purchased with the realization that they are not a panacea for removing all software development problems. They all have weaknesses and limitations, and it is important to determine how to work around those weaknesses effectively.

Classifications such as these are intended only to categorize methods to help simplify the selection process. The classifications tell us very little about the *quality* of a particular method, its *suitability* to a given application, or the level or quality of *automation* in terms of tool support. Such matters are certainly important in the final selection of methods, but are beyond the scope of classifications such as these.

Reasonable choices and decisions can be made by examining the information in this paper and by applying the classification scheme to other candidate methods. Armed with classifications of candidate methods, an organization can pursue the detailed evaluation of these and other important characteristics of methods prior to final selection.

5. References

[AdaGraph 86a]   TASC (The Analytic Sciences Corporation).
                  *Pamela Designer's Handbook, Volume 1: Commentary and Ada PDL and
                      Code*.
                  June, 1986

[AdaGraph 86b]   TASC (The Analytic Sciences Corporation).
                  Pamela Designer's Handbook, Volume 2: Figures and Graphs.
                  June, 1986

[Bergland 81]   Bergland, Glenn D., and Gordon, Ronald D.
                  Tutorial: Software Design Strategies, 2nd Edition.
                  IEEE Computer Society Press, Los Angeles, 1981.

[Booch 83]   Booch, Grady.
                  Software Engineering with Ada.
                  Benjamin/Cummings, Menlo Park, CA, 1983.

[Booch 87]   Booch, Grady.
                  Software Components With Ada—Structures, Tools, and Subsystems.
                  Benjamin/Cummings, Menlo Park, CA, 1987.

[Buhr 84]   Buhr, R.J.A.
                  System Design with Ada.
                  Prentice-Hall, Englewood Cliffs, N.J., 1984.

[Cherry 88]   Cherry, George W. Cherry.
                  PAMELA 2: An Ada-Based Object-Oriented Design Method.
                  Conference paper.
                  February, 1988

[DoD 82]   Ada Joint Program Office.
                  Ada Methodologies: Concepts and Requirements.
                  Technical Report, Department of Defense, November, 1982.

[Firth 87]   Firth, R., Wood, B., Pethia, R., Roberts Gold, L, Mosley, V., and Dolce, T.
                  A Classification Scheme for Software Development Methods.
                  Technical Report CMU/SEI-87-TR-41, ESD-TR-87-204, Carnegie-Mellon Uni-
                      versity, Software Engineering Institute, November, 1987.

[Freeman 83]   Berra, Bruce P.; Carroll, Bill, D.; Cotton, Jack; Cox, Jerome, R . Jr.; Nahouraii,
                  Ez; and Wu, Chuanlin (editors).
                  Tutorial on Software Design Techniques, 4th Edition.
                  IEEE Computer Society Press, Los Angeles, 1983.

[Harel 86]   Harel, David.
                  Statecharts: A Visual Approach to Complex Systems.
                  Concurrent Systems , February, 1986.

[Hesse 84]   Hesse, Wolfgang.
                  A Systematics of Software Engineering: Structure, Terminology and Classifi-
                      cation of Techniques.
                  NATO ASI Series: Program Transformation and Programming Environments,
                      Vol F8.
                  1984

[Hoare 85]   Hoare, C. A. R.
                 Communicating Sequential Processes.
                 Prentice/Hall International, London, 1985.

[Kelly 87]   Kelly, John C.
                 A Comparison of Four Design Methods for Real-Time Systems.
                 Research paper supported by a NASA/ASEE fellowship at JPL, CA.
                 1987

[TSE 86]   The Institute of Electrical and Electronics Engineers, Inc.
                 Special Issue.
                 IEEE Transactions on Software Engineering SE-12(2), February, 1986.

[Yau 86]   Yau, Stephen S., and Tsai, Jeffery J. P.
                 A Survey of Software Design Techniques.
                 IEEE Transactions on Software Engineering SE-12(6):713-721, June, 1986.

Appendix A: Discussion of the Classification

A successful classification of design methods would highlight the differences among the chosen methods in a straightforward manner, yet still allow for the classification of other methods as needed. A clear classification should also facilitate detailed assessment of the methods, allowing the evaluator to concentrate on investigating the particular differences among the methods rather than the similarities. A good classification scheme should also eliminate the ambiguity of conflicting terminology that is prevalent in the field of software development methods. We have developed the classification charts and their accompanying descriptions to satisfy the desired characteristics of a classification.

As described previously, Figure A-1 provides a high-level outline of the characteristics of each of the four methods. One can determine quickly where the similarities lie (e.g., formats of representation) as well as the differences (e.g., behavioral views). This allows the evaluator to focus energy on assessing those areas that are most likely to be discriminating factors in the selection of a method. In that light, we have further classified the four methods to an additional level of detail: Figure A-2 classifies graphical representations and is elaborated by the presentation of the icons for each method in Figures A-3 through A-6, Figure A-7 classifies the use of Ada features, and Figure A-8 classifies application environment considerations.

Depending on your concerns, extension of the classification scheme may be appropriate, but we recommend that the classification remain a simple scheme with a technical focus. Below is a limited list of additional technical criteria that may be relevant to classifying other design methods.

multi-program interfaces	communicating between separate programs
persistent objects	describing characteristics (e.g., sequential or direct file access)
periodic/aperiodic events	managing response times for both regular processing and "burst" processing
hard deadline scheduling	satisfying critical jobs for real-time systems
system redundancy	providing standby systems with nearly instantaneous switchover
fault tolerance	minimizing downtime due to failure conditions
data extraction	debugging in the target environment
formal representations	deciding program completeness and consistency
programming languages	suiting the design method to the implementation constraints

A.1. High Level Outline

Figure A-1: A High-Level Outline

Object-oriented development is founded upon the view that a program implements a model of reality that can be identified as a set of objects that interact with each other. The major steps in Booch's method to object-oriented development are to:

1. Identify the objects and their attributes.
2. Identify the operations suffered by and required of each object.
3. Establish the visibility of each object in relation to the other objects.
4. Establish the interface of each object.
5. Implement each object.

Buhr proposes an informal design methodology that is based on the data-flow-driven structured design strategy. The major steps are to:

1. Identify the major obvious subsystem modules.
2. Sketch the data flow external and internal to the major modules.
3. Refine the data flows and modules internals.
4. Assign functions in detail to modules.
5. Develop one or more structure graphs defining candidate architectures in terms of packages and tasks.
6. Define the system interfaces in detail.

The major steps of the PAMELA-2 method to software design are to:

1. Develop the external objects graph.
2. Develop the top-level library graph.
3. Develop the specification for each module in the top-level library graph.
4. Develop the top-level processing graph.
5. Write internal logic description for each simple unit identified in steps 2-4.
6. Repeat steps 2-4 for each component and nonsimple unit until all components have library graphs, specifications, and processing graphs.

SYSCONs method splits the design into two major phases, the top-level design and detailed level design, and advocates the use of compilable program design language (PDL) to augment and clarify data structures and processing throughout the design process. The major steps to the method are to:

1. Perform the top-level design to:

   a. Identify the virtual package (subsystem) components.
   b. Identify the dependencies among the virtual package components.
   c. Identify the library units.
   d. Identify all exported (visible) package units.
   e. Identify compilation unit dependencies.
   f. Identify the internal (hidden) package units.
   g. Identify nested program components.
   h. Establish the data flow logic by specifying formal parameters, and the associated visible types, for exported tasks, subprograms, and generic declarations.
   i. Establish the major control flow logic between, and within, complex visible compilation units.
   j. Identify the exported exceptions.

2. Perform the detailed-level design to:

   a. Complete the specification of all components of the visible and private portions of all library units.
   b. Refine the nested program components by identifying nonvisible Ada tasks.
   c. Complete the identification of local types and variables within the program units.
   d. Refine the major control flow within complex visible program units.
   e. Establish the major control flow logic within all program units.
   f. Identify exception handlers.

A.2. Graphical Representations

Figure A-2: Graphical Representations

The graphical representations that are used to **show software program structure** are presented in Figures A-2 through A-6. Each method extended the representations beyond the scope of Ada by including an icon to provide an abstraction that has no direct Ada syntactic identity. Booch uses a subsystem icon and PAMELA uses a super component to group logically related library units. Additionally, SYSCON's virtual package icon and Buhr's uncommitted module icon also allow the designer freedom by not forcing commitment to a particular Ada construct early in the design process.

The differently named graphical representations showing the **dependency structure** are described below to highlight their usage. Booch's dependency/data line can be used to represent an Ada context clause or to denote data flow. Buhr uses two representations to show data access and data flow. PAMELA uses import links to denote an Ada context clause and annotated data flow links to show data flow. SYSCON uses the visibility lines to represent an Ada use clause.

Buhr's graphical representations include a feature to **illustrate communication to tasks** not exposed in the classification chart or Figure A-4. Buhr provides a means for grouping entries of tasks. The grouping feature provides a closer representation to the intended design of the select statement within the task body.

Because all of the methods contain graphical representations for packages, subprograms, and tasks (although the icon shape may be different), we chose to concentrate on the package speci-fication. The **package specification** (the interface) criterion is used to describe how the method represents the exported items of an Ada package. Booch has one icon that represents both Ada objects and Ada types, and one icon that represents both procedures and functions. Buhr de-scribes the interface in terms of nonprocedural and procedural sockets, where nonprocedural sockets refer to objects and types while procedural sockets include procedures, functions, and task entries. PAMELA does not export entities by placing icons on the edge of the parent entity; instead the entry, procedure, and function interface is actually defined by the call link attached to the package. SYSCON uses unique icons for each of the listed entities that can be declared within the package specification.

Figure A-3: Booch's Graphical Representations

Figure A-4: Buhr's Graphical Representations

Figure A-5: PAMELA's Graphical Representations

Figure A-6: SYSCON's Graphical Representations

A.3. Use of Ada Features

Figure A-7: Use of Ada Features

The **use of packages** to embrace the software engineering principles of modularity, localization, abstraction, and information hiding is fairly consistent across the methods. The only difference in the classification chart is due to PAMELA's process orientation, which adapted the term abstract process for a multi-threaded package.

The classification information for the **use of tasks** needs further elaboration because the depth in which the methods describe the use of this Ada feature is so varied.

Booch divides the application of tasks into four general areas:

- concurrent actions
- routing messages
- managing shared resources
- interrupt handling

The four general areas are elaborated into specific examples that expand the four application areas into the apparently broader range given by Buhr and PAMELA.

Buhr lists eight canonical, structured **system parts** that may be composed of tasks:

- slave
- server
- scheduler

- buffer
- secretary
- agent
- transporter or messenger
- users or managers

These parts are classified and discussed so that the appropriate system part is selected for a particular circumstance.

PAMELA defines thirteen task **idioms** that identify "sensible" units to implement with Ada tasks:

- binary semaphore
- bounded buffer
- bounded pushdown automaton
- cyclic activity
- device driver
- forwarder
- hybrid
- interrupt handler
- monitor
- pump
- unbounded buffer
- unbounded pushdown automaton
- state machine

The method considers the thirteen task idioms as the basic building blocks for real-time software design.

SYSCON's method follows along the same approach of Booch. The designer is provided **unlimited** use of the task construct. The method does not restrict the use of tasks to a particular system component and does not define explicit task primitives.

The **use of generics** is consistent across the methods due to the nature of this Ada feature. Generics provide the ability to create code templates, which are parameterized or not, from which corresponding subprograms or packages can be instantiated. The phase of the development cycle in which this feature is exploited exposes a slight difference in the methods. Booch, PAMELA, and SYSCON discuss the use of generics from initial design through implementation to use and develop reusable components. Buhr's philosophy is that the use of generics is more closely related to implementation than system design.

The design mechanism for dealing with errors or other exceptional situations that arise during Ada program execution is provided by the **use of exceptions**. Booch states that exceptions should be used to plan for the resolution of possible error states of the objects and algorithms, and not to provide some sort of implicit *goto* facility. Buhr feels that exceptions should be relegated to the level of internal details within modules, with errors at module interfaces handled by parameters of calls. PAMELA implements exceptions for error conditions within callable processes and subprograms and requires all callers to have an exception handler when an exception can be propagated. SYSCON uses exceptions and localized exception handlers for truly abnormal error or exceptional circumstances, and not to effect normal changes in control flow.

A.4. Application Environment Considerations

Figure A-8: Application Environment Considerations

Although our classification has concentrated on Ada and graphical representations, this chart would be applicable for classifying design methods that are not Ada-based or do not contain graphical representations. The emphasis of this chart is to show how the methods support the designer with regard to application environment considerations.

The design methods classified provide very little direction in enforcing **performance and resource constraints** imposed by the system specification. Real-time programs must provide confidence about timing correctness at the tasking level of abstraction and must manage resources such as CPU, I/O drivers, and memory. Therefore, design descriptions must include a means of representing, or directly referencing, these performance and resource usage requirements as presented in the specification. The design should sufficiently direct the implementation to enforce the requirements. By incorporating performance and resource requirements into the design descriptions, the program structure is more likely to persevere through the implementation phase.

The exception construct within Ada allows one to **manage error conditions** at the lowest level by alternative courses of action (as stated in [Booch 83]). Namely, these alternatives are:

- Abandon the execution of the unit.
- Try the operation again.
- Use an alternative approach.

• Repair the cause of the error.

Expanding on the area of managing error conditions, Buhr introduces intertask protocols and reliability units for controlling aberrant behavior:

intertask protocols   Formalized dialogue between tasks for the purposes of reliable communication in the face of possible failures of either the entities or the communications medium between them, but not failure of the protocol.

reliability units   One or more tasks with an associated communication package that is isolated from other reliability units but which provides for reliable communication via intertask protocols with other reliability units.

Booch doesn't **develop test strategies** but suggests a "design a little, code a little, test a little" approach. Buhr explains the design of instrumented testbeds for complete systems in operational form to detect, record, and isolate bugs. PAMELA suggests supporting the testing of the executable units by developing simulations of the external objects graph. SYSCON uses code instrumentation techniques to generate test log files, which are then used to animate the graphical representations.