Structured Analysis and Object-Oriented Development are not Compatible

Donald Firesmith, President
Advanced Software Technology Specialists
17124 Lutz Road
Ossian, Indiana, USA 46777
(219) 639-6305

Since its introduction in 1978, traditional Structured Analysis has been an industry standard method for software requirements analysis that is supported by numerous CASE tools. Since their introduction in the early 1980s, various forms of Object-Oriented Development (OOD) have also become the preferred approach for the design and coding of Ada software. More recently, OOD has included various forms of Object-Oriented Requirements Analysis. OOD has therefore come into direct competition with Structured Analysis. While some methodologists have advocated retaining Structured Analysis and have worked to merge the two paradigms, others have pointed out significant disadvantages of combining them and urge the use of a unified object-oriented paradigm throughout all development activities. A recent article published in Ada Letters by Ken Shumate [SH 1991] is illustrative of this controversy and has prompted this reply.

It is the thesis of this paper that traditional Structured Analysis, even when modified for real-time systems, is a technically obsolete way to specify software requirements, and that Structured Analysis and Object-Oriented Development are fundamentally incompatible and unnecessarily difficult for software engineers to combine effectively.

1) Background

Structured Analysis was developed primarily by Tom DeMarco and was published in the now classic book *Structured Analysis and System Specification* [DM 1978]. Like its predecessor, Structured Design [YC 1975], it is a hierarchical functional decomposition method. Object-Oriented Development (a.k.a. Object-Oriented Design) was first introduced by Russell Abbott [AB 1983] and popularized by Grady Booch in his classic book *Software Engineering with Ada* [BO 1983,87]. Because the original OOD methods did not address software requirements analysis, many projects (e.g., the Advanced Field Artillery Tactical Data System) preceded OOD with functional decomposition analysis methods (e.g., Structured Analysis), typically with questionable results. Significant problems due to fundamental incompatibilities between the two paradigms accelerated the development of the numerous object-oriented analysis methods (e.g., [BA 1989], [BR 1991], [CB 1989], [CM TBD], [CY 1989], [FG 1991], [RB 1991], [SM 1988]).

2) Overview of Structured Analysis

As stated in the previous section, traditional Structured Analysis is a hierarchical, functional decomposition software requirements analysis method. Originally developed for Automatic Data Processing (ADP) applications when most programs were written in procedural languages (e.g., COBOL, CMS-2, FORTRAN, JOVIAL, Pascal), it was updated for real-time embedded applications by Paul Ward and Steven Mellor [WM 1985] and by Derek Hatley and Imtiaz Pirbhai [HP 1987]. These updates, however, were primarily based upon control flow with some concurrency issues and process abstraction; object-orientation was largely ignored.

The basic notational tool of Structured Analysis is the Data Flow Diagram (DFD). Each DFD depicts the relevant data stores, transform bubbles, and the data (and possibly the control) flows between them. See Figure 1: Example Functional Decomposition DFD. As part of the functional decomposition paradigm, each major functional abstraction (represented by a transform bubble) was decomposed into lower-level transforms on a child DFD.

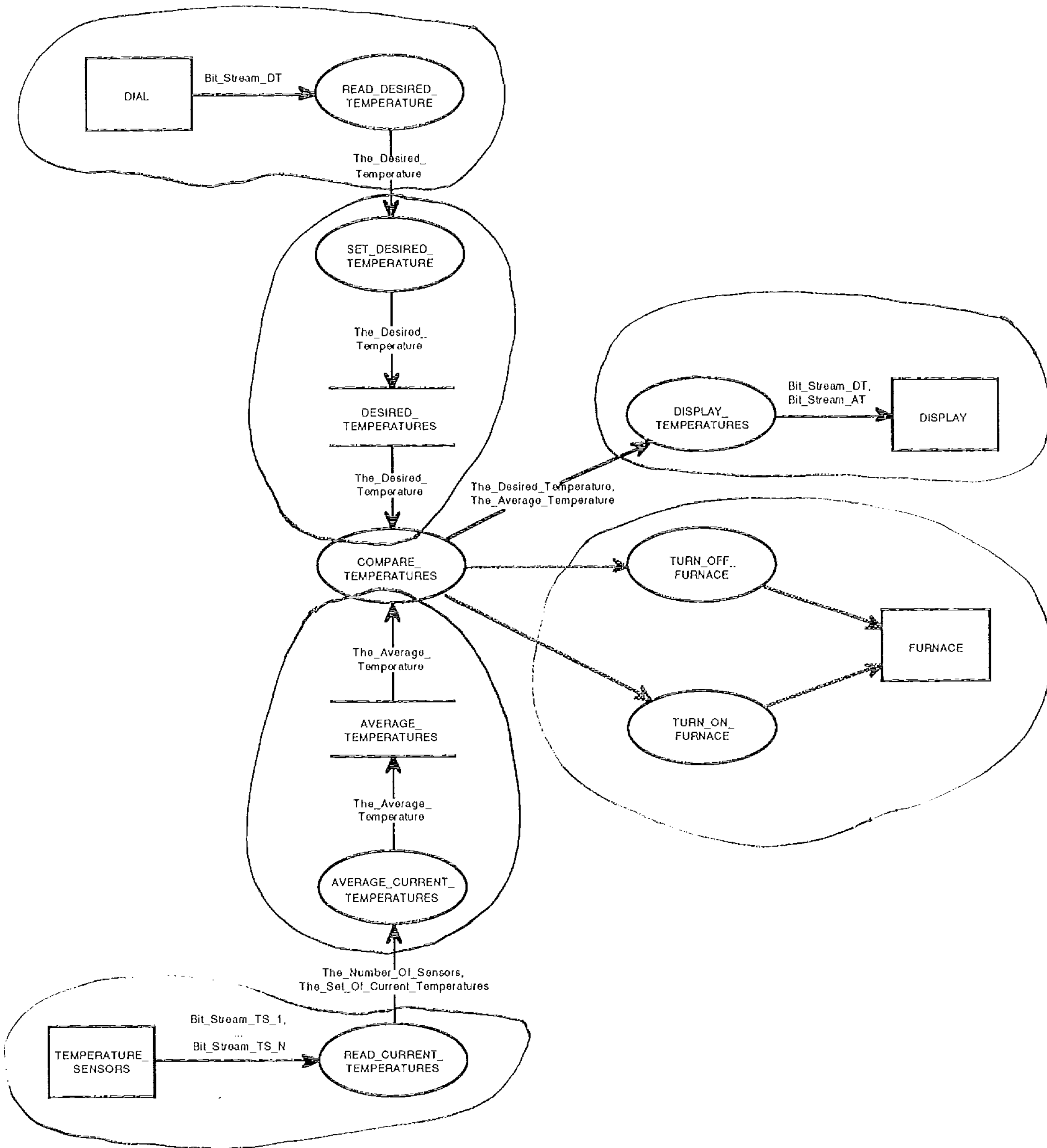

Figure 1: Example Functional Decomposition DFD

3) Overview of Object-Oriented Development

Object-Oriented Development (OOD) is a class of related, typically globally recursive software development methods based upon object, rather than functional, abstraction as the primary localization paradigm. Originally restricted to design and coding, many OOD methods now include requirements analysis, testing, and integration to produce a truely "analyze a little, design a little, code a little, and test and integrate a little" approach to software development. Most methods currently use multiple models to analyze and design the software. In typical order of development, these include the:

1) Semantic Model consisting of one or more semantic nets, dependency diagrams, classification diagrams, and composition diagrams. See Figure 2: Example Semantic Net.
2) Object-Interaction Model. See Figure 3: Example Object-Interaction Diagram.
3) State Model consisting of multiple state transition diagrams, state transition tables, or Harrell state charts.

4) Control Model consisting of one or more *object-oriented* control flow diagrams and/or data flow diagrams. See Figure 4: Example Object-Oriented Control Flow Diagram and Figure 5: Example Data Flow Diagram.

Semantic Nets show the objects (i.e., the concurrent object DIAL and the sequential objects DISPLAY and FURNACE) and classes (i.e., the concurrent class TEMPERATURE_SENSORS) in an assembly or subassembly (i.e., THERMOSTAT_SYSTEM), the terminators (i.e., the hardware devices DIAL, DISPLAY, FURNACE, and TEMPERATURE_SENSORS), and the important relationships between them.

Object-Interaction Diagrams show the objects and classes in an assembly or subassembly, the terminators, and the exported creator, modifier [M], preserver [P], and destroyer operations they require of one another to meet their requirements.

Object-oriented Control Flow Diagrams (CFDs) show the objects and classes in an assembly or subassembly, the terminators, and operations (both visible and hidden), and the control flows between them.

Object-oriented Data Flow Diagrams (DFDs) show the objects and classes in an assembly or subassembly, the terminators, operations (both visible and hidden), the data stores, and the data flows between them.

4) Proposed Transition Approaches

The three primary approaches typically recommended for the transition from Structured Analysis to OOD are to identify an object or class for each:

1) Terminator on a context diagram.

2) Data store and its associated transform bubbles on a DFD.

3) Complex data flow on a DFD.

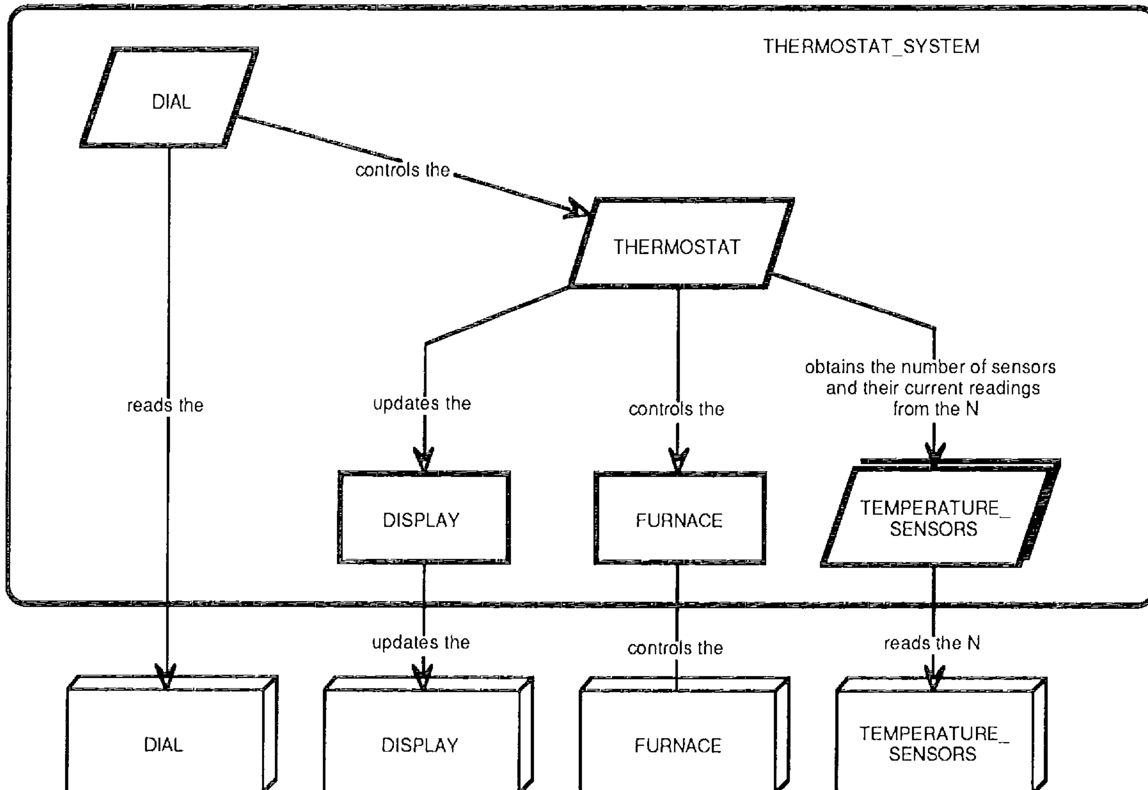

Figure 2: Example Semantic Net

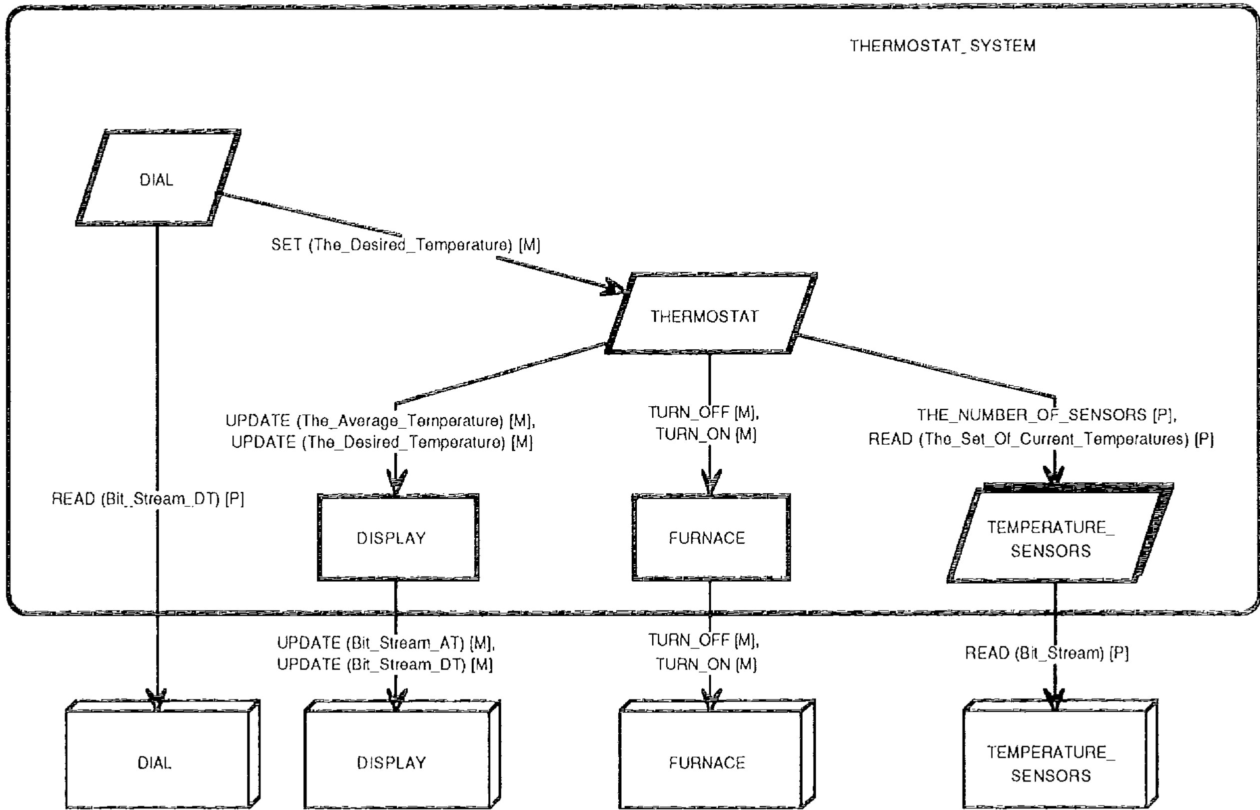

Figure 3: Example Object-Interaction Diagram

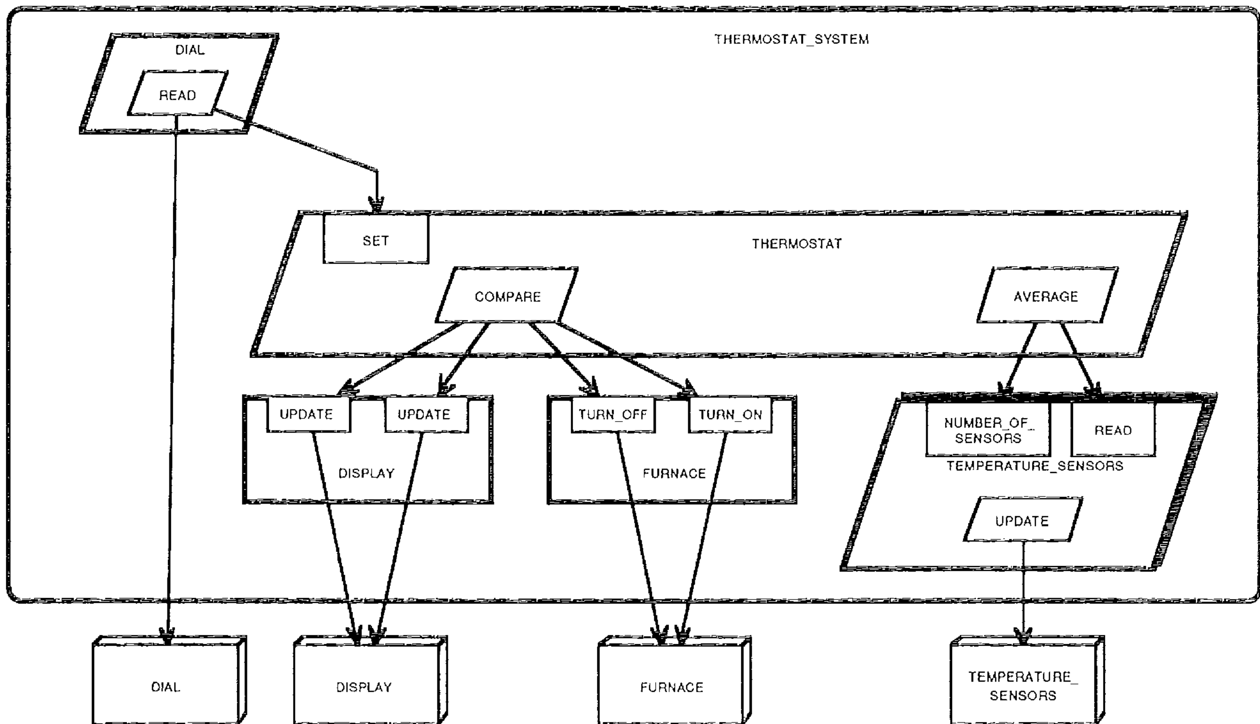

Figure 4: Example Object-Oriented Control Flow Diagram

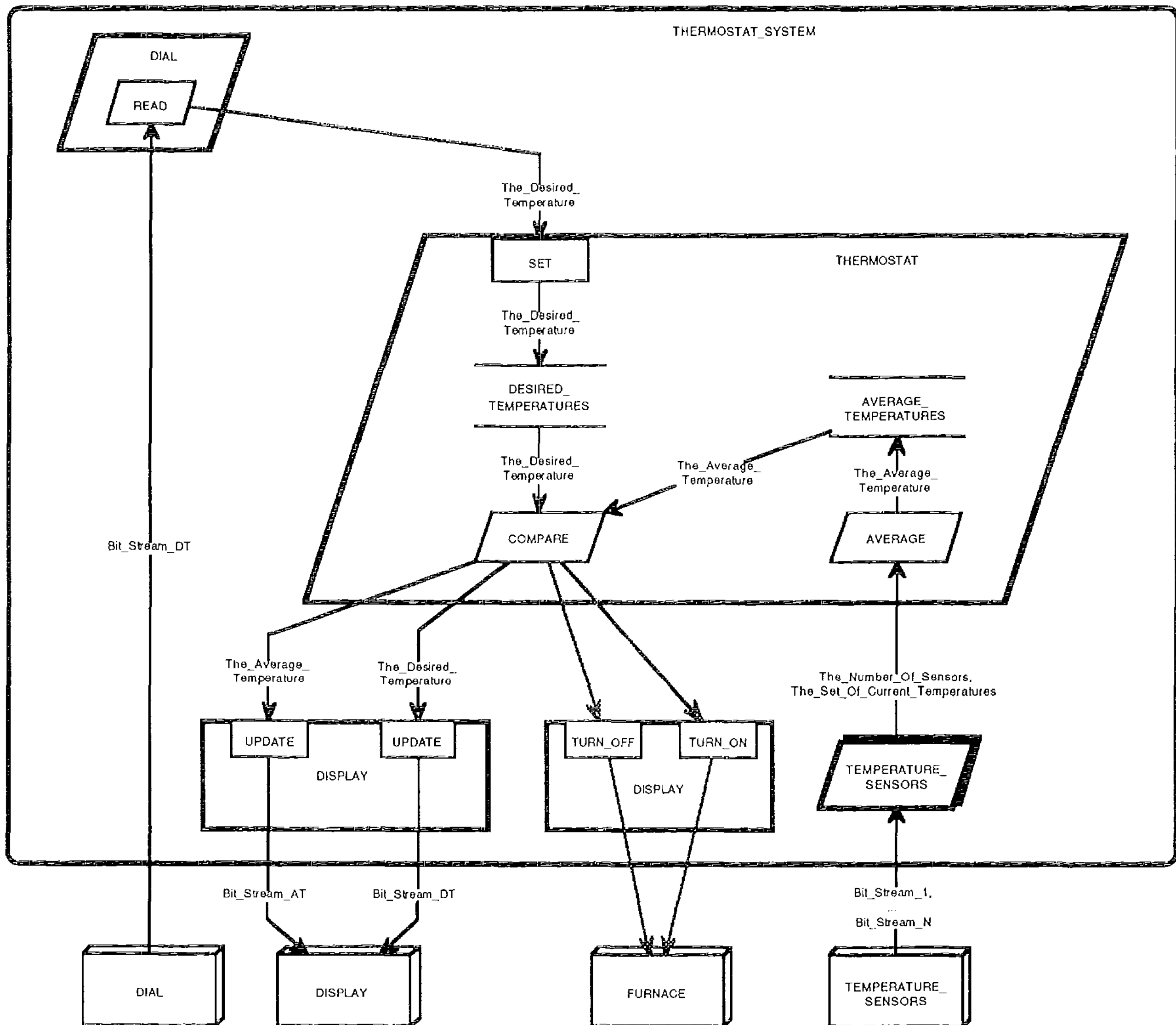

Figure 5: Example Object-Oriented Data Flow Diagram.

The first approach uses terminators on context diagrams to identify an object or class. Figure 1 shows four terminators (i.e., DIAL, DISPLAY, FURNACE, and TEMPERATURE_SENSORS) which can be correctly mapped to three objects and one class, respectively. This approach is easy to use, highly reliable with very few false positive identifications, and is based upon context diagrams that are often available early in the project.

The second approach uses data stores on DFDs to identify associated objects or classes and uses all (or part) of the associated transform bubbles to identify the associated operations. See Figure 5: Use of Data Stores. Each object or class thus encapsulates the corresponding data and operations on that data. This approach was the result of a "Search for the holy Grail" of many managers in that it was a way to save traditional functional decomposition requirements analysis as a front end for object-oriented design, and therefore make use of the recently acquired expensive Structured Analysis tools. Several methodologists independently and simultaneously invented this approach (obviously an idea whose time had come). It was very popular between 1986 and 1988. Benefits of this approach included:

- the fact that many managers, analysts, and software engineers have been trained in DFDs.
- significant, production-quality tool support exists to support DFDs.

- Systems analysts often provide DFDs.

The third approach identifies an object or class with each complex data flow on a DFD that may require operations to manipulate that data.

5) Basic Incompatibilities

It is possible to successfully mix Structured Analysis and OOD on the same project if one:

- defines success solely in terms of meeting project requirements on schedule and within budget.
- has adequate schedule and budget.
- is aware of the associated risks and takes significant steps to mitigate them.

The probability of success, however, decreases if one defines success in terms of meeting project requirements and maximizing the achievement of the goals of software engineering (e.g., maintainability, reusability, etc.) while minimizing lifecycle costs as well as project schedule and budget. Similarly, most projects run into significant problems because they have difficult schedules and budgets and do not take all necessary steps to minimize the incompatibilities between Structured Analysis and OOD.

The incompatibilities between the two paradigms cause significant problems in the areas of:

- quality.
- productivity.
- cost.

These incompatibilities result from the differences in:

- total abstraction use and emphasis.
- order of abstraction use.
- localization (i.e., functional versus object abstraction).
- total model use.
- order of model use.
- development cycle used.
- developer expertise with OOD which requires significant training, time, experience, and expense.

Structured Analysis relies primarily upon functional abstraction with minimal data abstraction. The significant use of process abstraction as part of Structured Analysis occurs only when using a real-time version of Structured Analysis (e.g., [HP 1987] and [WM 1985]). Analysts using Sturctured Analysis rarely consider object and exception abstraction. On the other hand, proper OOD methods must use all five types of abstraction to adequately model real world entities. OOD methods use:

- object abstraction to identify objects and classes.
- data abstraction to identify and model attributes.
- functional abstraction to model sequential operations.
- process abstraction to model the concurrent resources (e.g., operations) of concurrent objects and classes.
- exception abstraction to model the ways objects can fail and manage failure.

Structured Analysis uses functional abstraction first and only deals with data abstraction second. Updated versions of Structured Analysis often properly consider process abstraction first [SH 1991] when used on multiprocessor, multitasking, real-time projects. OOD methods use object-abstraction first to identify objects and classes, data abstraction second (especially when in conjunction with languages that support strong typing) to model attributes, functional and process abstraction next to model operations, and exception abstraction last since most exceptions are associated with operations and not objects and classes as a whole. Some OOD methods that were developed for real-time projects (e.g., [CB 1989], [FG 1991]) also consider concurrency issues at the beginning of the analysis, but do so only in terms of object abstraction (i.e., they initially consider objects and classes to be either concurrent or sequential since this has a major impact on their operations and how they

interact). OOD methods therefore not only use more types of abstraction than Structured Analysis, they also use them in a fundamentally different order.

Structured Analysis localizes (and decomposes) according to functional abstraction whereas OOD methods localize (and either decompose or compose) according to object abstraction. What the one paradigm groups together, the other tends to scatter to the four winds on a project. For example, when a function is decomposed into subfunctions, these subfunctions are not typically operations belonging to the same object or class nor are all attributes and operations belonging to the same object or class localized in the functional decomposition structure (e.g., processes and data stores on the same DFD). Similarly, since Structured Analysis has no requirement that functions correspond to operations on individual objects or classes, functions are often compound operations that operate on more than one object or class.

Structured Analysis is primarily based upon the Control and State Models with emphasis on the data flow parts of the Control Model. On the other hand, the most powerful OOD methods are based on the Semantic Model, the Object-Interaction Model, the State Model, the Control Model, and the Timing Model with emphasis on the Semantic and Object-Interaction Models. Thus, powerful OOD methods use more models and emphasize different models than Structured Analysis.

Structured Analysis uses the Control Model first and follows it with the State Model. On the other hand, the most powerful OOD methods typically use the following models in the following order: Semantic Model, Object-Interaction Model, State Model, Control Model, and Timing Model. Thus, powerful OOD methods use the models in a different order than does Structured Analysis.

Finally, Structured Analysis is a locally recursive method used during the Software Requirements Analysis Phase of the traditional "Waterfall" development cycle. Conversely, OOD methods are typically globally recursive methods that require a recursive lifecycle in which the software is developed in subassemblies (i.e., small increments) consisting of no more than the Miller Limit [ML 1956] of seven plus or minus two objects and classes. This use of global recursion was made practical by the combination of localization by object abstraction, the arrival of languages (e.g., Ada, Modula-2) that support modularity above the subroutine level, the separation of specification and body, and strong typing, all of which supported the creation of software "firewalls" that limit the spread of the impact of analysis and design errors. Global recursion also added the ability to incrementally verify and validate the analysis and design as one recursively developed the software. Thus, the two paradigms typically use different development cycles with corresponding differences in formal reviews, times between the making and finding of errors, and resulting costs of fixing errors [BM 1981].

6) Risks associated with Transition Approaches

6.1) Risks associated with Using Terminators on Context Diagrams

Terminators on context diagrams only identify a small number of the most obvious objects and classes and misses most nontrivial (e.g., conceptual) ones. The objects and classes are identified early, often as part of the initial subassembly, and yet often belong at different levels of abstraction. This often causes the initial subassembly to increase beyond the Miller Limit causing problems in quality and productivity. Often, this approach is only useful for identifying objects or classes corresponding to terminators called by the software to be developed. Such interface objects and classes provide a logical interface to these callee terminators that allow the rest of the software to be independent of changes to these terminators. On the other hand, new objects and classes corresponding to terminators that call the software to be developed may not be required because these caller terminators may be able to call operations of other objects and classes. If always used for both callee and caller terminators, this approach may produce unnecessary and partially redundant objects and classes for some caller terminators.

The wrong context diagram may be used:

- Context diagrams are often the result of the functional decomposition of the system into functional CSCIs or assemblies.
- Object-Interaction Diagrams used as context diagrams provide a better view of the terminators (i.e., in terms of operations required instead of data flows).

Although this approach is still worth doing in spite of risks, one should concentrate on other, more modern and effective approaches (e.g., the use of semantic nets, object-interaction diagrams, object-oriented state transition diagrams).

6.2) Risks associated with Using Data Stores on DFDs

This approach is also based more on data abstraction than object abstraction and is therefore indirect and quite subject to incorrect mappings.

Objects and classes identified may be incomplete:

- Many abstract objects and classes contain more than one data store.
- Because DFDs are often functionally decomposed, pieces of an object or class may be scattered, both horizontally and vertically, across several DFDs assigned to different analysts. See Figure 1 for an example of a functional DFD in which parts (e.g., the operation COMPARE and the data stores AVERAGE_TEMPERATURES and DESIRED_TEMPERATURES) of the single object (i.e., THERMOSTAT) would be allocated to two different misidentified "classes" (i.e., AVERAGE_TERMPERATURES and DESIRED_TEMPERATURES).
- Because most DFDs are functionally decomposed, transforms were not required to be operations on objects. Transforms are therefore often compound and are not allocatable to abstract objects or classes. Conversely, parts of a single operation may be allocated to multiple "objects".

Thus DFDs, which are often in software requirements specifications that are under software configuration management, usually require redrawing after the initial partitioning fails. It is not cost-effective to do inadequate and inappropriate DFDs and then have to fix them.

Traditional functional DFDs have the wrong scope:

- DFDs usually contain more than one data store and thus more than one abstract object or class.
- Because they would violate the Hrair limit by containing too many nodes (especially transforms) to be understandable, DFDs usually do not document an entire subassembly.
- Pieces (i.e., operations and data stores) of the same abstract object or class are often on several DFDs.

Analysts can largely avoid these risks by only drawing object-oriented DFDs. See Figure 5 for an example of an object-oriented DFD. Such DFDs typically have the following properties:

- All nodes can be allocated to objects or classes.
- All parts of an object or class are on the same DFD.
- The scope of most object-oriented DFDs is a single abstract object or class, and possibly its terminators.

Many analysts and software engineers are so used to developing functional DFDs that it is difficult for them to produce object-oriented DFDs.

Because of the many risks listed, some methodologists (e.g., Ed Berard) advise not using DFDs at all. Others, such as myself, recommend that one should;

- be aware of the many risks associated with functional DFDs.
- develop object-oriented (instead of functional) DFDs.
- develop object-oriented DFDs only where cost-effective and useful. For example, Many objects contain so many (often interacting) operations that even object-oriented DFDs are not always useful.
- use functional DFDs only if:
  - they already exist and can provide useful information, and
  - they are required (e.g., for contractual or political reasons).

Developers should concentrate on other, more modern and effective, approaches (e.g., semantic nets, object-interaction diagrams, object-oriented state transition diagrams).

6.3) Risks associated with Complex Data Flows

This approach is also based more on data abstraction than object abstraction and is therefore indirect and subject to incorrect mappings. Not only does this approach have suffer from most of the risks mentioned in the previous section, it also is often based on improperly decomposed functional DFDs. Data flows with the necessary structural complexity:

- are rare so only a very small percentage of the abstract objects and classes are this way.
- often imply that the original DFD is incomplete and should be fixed.

6.4) General Risks

Additional general risks include:

- increased difficulty with requirements traceability.
- psychological risks due to changing paradigms in the middle of a project.

If the requirements are developed with and documented according to Structured Analysis, they will have a hierarchical functional decomposition structure. If the design is developed using OOD, it will have a significantly different structure (especially at levels containing few essential tangible objects and classes) that is more graphical than hierarchical. The mapping from functionally sorted requirements to an object-oriented design will not even approach a one-to-one mapping. This is why DOD-STD-2167A no longer contains the requirement to use a requirements traceability matrix. This lack of one-to-one mapping (that approximately holds if one consistently uses either a functional or object-oriented paradigm) makes verification and testing more difficult as well as decreasing the understandability of the associated documentation.

If one uses the classic "waterfall" development cycle, one will spend a significant amount of time at the beginning of the project reinforcing a functional decomposition mindset. It is unrealistic to expect designers to be able to make the abrupt transition to an object-oriented paradigm quickly, easily, and correctly. This is a major reason many Ada projects that officially use OOD contain significant amounts of software that is more functionally decomposed than object-oriented. Using the same paradigm throughout supports the principle of uniformity and the goal of understandability. This is also an improvement over the combination of Structured Analysis and Structured Design that uses significantly different techniques (i.e., DFDs and Structure Charts) for analysis and design causing a major transition at the boundary.

Recommended Approach

Since modern OOD methods provide all of Structured Analysis's strengths (by incorporating its major concepts and techniques) without the significant disadvantages mentioned in the previous sections, I recommend that they be used to replace Structured Analysis, especially on projects using object-oriented or object-based languages such as Ada. This requires training and significant change in the way software development is done, but the benefits are well-worth the effort as can be attested by those who have developed software both ways and who have suffered by trying to combine the disparate paradigms.

Conclusion

In conclusion, although Structured Analysis and similar hierarchical functional decomposition software requirements analysis methods represented great steps forward and at one time represented clearly the best available paradigm for software requirements analysis, they have since been obsoleted by improvements in methods (i.e., OOD) and language (e.g., Ada, Modula-2, C++) development. The fact that functional decomposition is still widely used for systems and software analysis is no justification to continue using the approach after it has outlived its preeminence. Such excuses were once used to retain Structured Design in the face of Object-Oriented Design and are no more relevant now than then. Although all methods have risks and state-of-the-art methods have their own unique risks (e.g., those assoicated with training, transitioning, and limited usage), it is nevertheless the responsibility of analysts to use the best current methods for the application and to stay abreast of today's fast paced technology in order to make the right choice. Although difficult to do, anything less is a violation of our duty to our clients and our profession.

Acknowledgment

All graphics used in this paper were generated using version 1.8 of the Objectmaker CASE tool developed by Mark V. Systems Limited. The author is also indebted to Ken Shumate who provided a prepublication draft of his new book that expanded upon the topic of his *Ada Letters* article that prompted this response.

References

[AB 1983] Abbott, Russell J. "Program Design By Informal English Descriptions," *Communications of the ACM,* Volume 26, Number 11, November 1983, pages 882-895.

[BA 1989] Bailin, Sidney C. "An Object-Oriented Requirements Specification Method," *Communications of the ACM*, Volume 32, Number 5, May 1989, pages 608-632.

[BM 1981] Boehm, Barry W., *Software Engineering Economics*, Prentice Hall, 1981.

[BO 1983] Booch, Grady, *Software Engineering with Ada*, Benjamin/Cummings Publishing Company, 1983.

[BO 1987] Booch, Grady, *Software Engineering with Ada*, *Second Edition,* Benjamin/Cummings Publishing Company, 1987.

[BR 1991] Berard, Edward V., "Object-Oriented Requirements Analysis," Berard Software Engineering, Inc., (301) 353-9652, 10 February 1991.

[BU 1987] Bulman, Dave, "Model-Based Object-Oriented Design for Ada," Pragmatics, Inc., (808) 883-9011, 14 July 1987

[CB 1989] Colbert, Edward, "The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development," *Proceedings of TRI-Ada'89 -- Ada Technology In Context: Application, Development, and Deployment, 23-26 October 1989,* pages 400-415.

[CM TBD] Comer, Ed, *Ada Box Structures Methodology Handbook,* Software Productivity Solutions, (407) 984-3370, 31 July 1989.

[CY 1989] Coad, Peter and Yourdon, Ed, *OORA - Object-Oriented Requirements Analysis,* Prentice Hall, 1989.

[DM 1978] Demarco, Tom, *Structured Analysis and System Specification.* Yourdon Press/Prentice Hall, 1978.

[FG 1991] Firesmith, Donald and Gaumer, Dale, *The ASTS Development Method 2 (ADM_2) User's Manual,* ASTS, Fort Wayne, Indiana, (219) 639-6305, 1991.

[HP 1987] Hatley, Derek J. and Pirbhai, Imtiaz A. *Strategies for Real-Time System Specification*, Dorset House Publishing Company, 1987.

[ML 1956] Miller, G. A., "The Magical Number Seven, Plus or Minus Two," *The Psychological Review*, Volume 63, Number 2, March 1956.

[RB 1991] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick and Lorensen, William. *Object-Oriented Modeling and Design,* Prentice Hall, 1991.

[SH 1991] Shumate, Ken, "Structured Analysis and Object-Oriented Design are Compatible," *Ada Letters*, Volume XI, Number 4, May/June 1991, pages 78-90.

[SM 1988] Shlaer, Sally and Mellor, Stephen J., *Object-Oriented Systems Analysis, Modeling the World in Data,* Yourdon Press, 1988.

[WM 1985] Ward, Paul and Mellor, Stephen J., *Structured Development for Real-Time Systems, Volume 1: Introduction and Tools, Volume 2: Essential Modelling Techniques, Volume 3: Implementation Modelling Techniques*, Yourdon Press, 1985.

[YC 1975] Yourdon, Edward and Constantine, Larry L., Structured Design. Yourdon Press, 1975.