A Proposal for a Recursive Object Oriented Life-Cycle

Kevin E. Carlin — EVB Software Engineering, Inc.
Dino R. Russo — TRIGON Software Engineering
Brad Balfour — EVB Software Engineering, Inc.

ABSTRACT

This paper describes the recursive object oriented life-cycle which has been developed in conjunction with the authors' Object Oriented Development (OOD) methods. The basic goals that a life-cycle must satisfy are described and the recursive life-cycle concepts are mapped back to them. A description of how this life-cycle may be implemented within the framework of existing standards such as DOD-STD-2167A is also included. This paper also summarizes the authors' experience with using this life-cycle (or elements thereof) on various projects over the last six years. Approaches to meeting the issues of formal project management within the recursive framework will also be discussed.

INTRODUCTION

Experience with the use of OOD in Ada has resulted in a serious re-examination by the authors of how software project management must be performed to realize the full benefits of the approach. So far, the industry's focus has been on adapting pre-existing methods and technology to take advantage of objects and the application of existing technologies in "object oriented" methodologies. These practices, such as the Waterfall life-cycle, which have been effectively in place since 1970 [Agresti, 1986a], arose naturally from function-oriented methods and are firmly rooted in the conditions of that time — relatively small-scale projects developed with scarce computer resources. The use of an object oriented approach negates the conditions which call for a waterfall approach to software development. For the last six years, a coherent, organic approach to object oriented project management employing the Recursive life-cycle has been emerging within the community. This life-cycle permits project management based on significant project tasks rather than software life-cycle phase.

We believe that 1) in the long term, the industry must move from a document driven development process to a process which emphasizes intermediate delivery and acceptance of *software* product by the customer; and that 2) during the transition, new approaches which facilitate intermediate evaluation and delivery will be required to provide extensive intermediate products analogous in content to the current documentation requirements associated with standards such as DOD-STD-2167A [DOD-STD-2167A, 1988]. Please note that we are *not* denouncing the proper and necessary development of requirement specifications, as-built design specifications, user's guides, etc., but rather the long chain of costly, disposable intermediate documentation produced to demonstrate the contractor's frugal adherence to good software engineering practice. We will discuss how such products fit within the life-cycle in a later section.

RECURSION AND 2167A

The 2167A software development standard is often considered an obstacle to innovations such as the Recursive life-cycle due to perceived implicit preference for the Waterfall life-cycle. The following clarification of 2167A from the Joint Logistics Command [JLC, 89] responded to government and contractor concerns that 2167A intended to restrict developers to the Waterfall life-cycle.

> "[2167A] specifies a set of activities that must take place during a software development project, but does not impose any requirements that these activities be performed sequentially or that one activity can begin before another is complete. To emphasize this approach, the standard states that the activities specified in the standard may *overlap* and may be applied *iteratively* or *recursively*." [Emphasis added.]

OBJECTIVES OF SOFTWARE PROJECT MANAGEMENT

"GOLDEN THREADS"

In the development of DoD software development standards, a general criteria arose by which to judge the quality of the result. These are referred to as the "Golden Threads" of software project management [Maibor, 1989]. These threads represent the customer's ability to monitor software projects and the customer's right to receive a quality product at a fair price. Ironically, the very measures intended to protect the customer's rights as represented by these threads serve to distract contracting engineers and the customer from the actual product with a parade of costly, disposable intermediate products.

PROJECT CONTROL POINTS

In general, control points are discrete points in a software development project when customers exercise their rights. Typically control points are defined in terms of reviews and review products[1] which are scrutinized by the client to determine if the project goals are being met. Under current practice, not only are documents reviewed, in many cases they become controlled property of the client.

DOCUMENTATION

For the customer, documentation is the tangible element in the software development life-cycle. In many cases, the customer's specifications for documentation are more well defined than their software requirements. One purpose of documentation is to serve as an anchor for project control points. Another purpose of documentation is to satisfy the need of the customer to have enforceable agreements with the developer which are more well defined than the procurement contract. Lastly, documentation must serve the practical purpose of enabling the client to use and maintain the software after deployment.

CHANGE MANAGEMENT

Recognizing the fact that change is inevitable, customers demand that change be controlled. One purpose of this control is to avoid deviating from the development path. Changes are "red flags" alerting the customer to pay attention. Another purpose, although

somewhat surreptitious, is self-protective: changes serve as the excuses for cost and schedule overruns.

ACCEPTANCE CRITERIA

Although one could argue that this area of concern is merely a sub-issue of control points and documentation, it garners enough attention to examine it as a separate point. For the customer, a software development life-cycle must provide a time and method for the specification of criteria that determines the acceptability of the products developed. The current state of the practice is that these criteria are specified as early in the life-cycle as possible. As a matter of fact, the first acceptance criteria examined are generally in the realm of documentation (e.g. specification of the format of deliverables in the proposal or the Software Development Plan). The purpose of this requirement is to provide customers with the security of knowing that the software and its associated products will meet their needs.

RECURSIVE PROCESSES

The recursive life-cycle comprises a series of steps repeated during the development process. Although the detailed steps will be determined by the process, it is assumed that the process will incorporate these basic activities: analysis, producing a software[2] requirements specification; design, producing a software architecture; implementation, expressing the design in computable form; and test, showing that the computable form does *not* meet its specification.

In most modern life-cycles, iteration among the process steps is allowed and encouraged. However, each of the phases is traditionally "completed" before the next is started. "Completed" means that the activity has achieved a level of maturity which permits phase products to be baselined, reviewed, and approved, and after which changes are much more stringently managed. Iteration is the return to a previous development process task to add new information or correct errors, propagate changes, and then return to the point in the development process from which the iteration was initiated.

figure 1: Process Sequence with Iterating Feedback.

The recursive approach applies the concept of recursion to the software development process, which is to say that, in the course of completing some specific development task, lower levels of abstraction will be identified and the full process will be performed again at that lower level. On completion of such lower levels, the current level may be completed. This is analogous to the concept of a recursive subprogram with the addition that multiple recursions may take place concurrently.

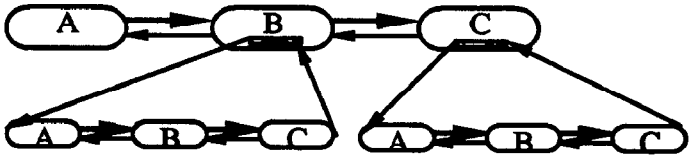

figure 2: Recursive Process with Iteration within the Process

A recursive software life-cycle employs many applications of the paradigm:

Analyze a little, Design a little, Implement a little, Test a little

In a recursive approach, the development team will apply the development methods to a single level of abstraction. Some elements of the level of abstraction will be trivial, and can be implemented directly. Other elements will be non-trivial and need further analysis and design before they may be implemented. In this case, the entire development process is repeated, recursively, on the newly discovered problem. This new branch of the problem may proceed in parallel with the implementation and test of the original, with a limited number of well-defined synchronization points.

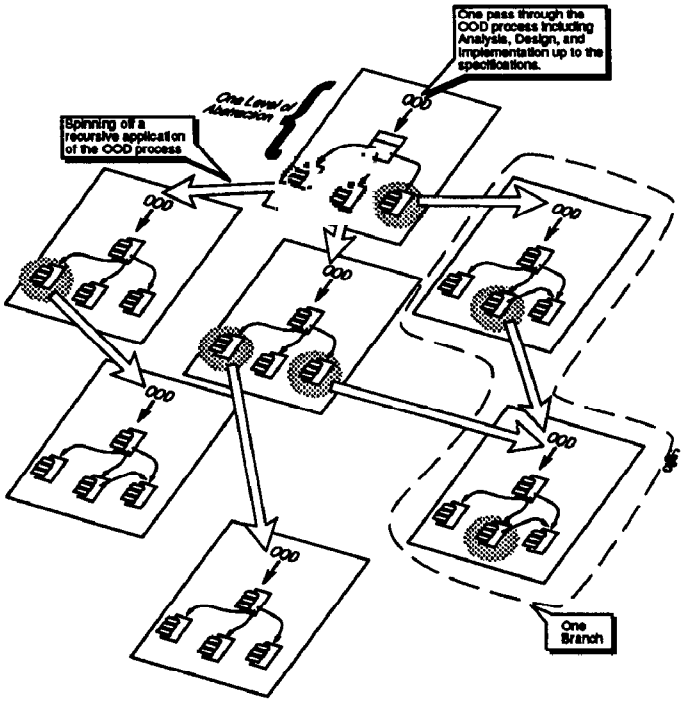

figure 3: The Recursive Life-Cycle

APPLICABILITY TO OBJECT ORIENTED APPROACHES

What makes the recursive life-cycle possible in software development is the essential character of objects as architectural building blocks. It has long been known that objects as architectural units are highly discrete, robust, autonomous entities (e.g., [Cox, 1986]). Inter-object dependencies are generally limited to the parents (inheritance) and the components (heterogeneous aggregates or "record types") of the object[3] and involve knowledge only of the external interface the object presents to the world. The dependent module has no direct access to or knowledge of the methods, variables, or internal structure of its closed-abstraction helpers.

ESSENTIAL DEFINITIONS

There are essentially four kinds of objects: classes, instances, values, and subsystems. *Classes* describe the properties of a set of objects, in the same way the concept "integer" is applied to variables of that type. A class may have its own state information and operations, usually for management of the storage or concurrency behavior of its objects. *Instances* of a class are analogous to the variables of a type. While identical in form, each instance has its own, discrete state just as each integer variable has its own value. *Values* are instances with an immutable state, which are named for their significance (e.g., Null, Closed, Empty). *Subsystems* are encapsulated aggregates of objects combined to perform useful work. A subset of component objects is presented in the subsystem interface for parameterization

purposes, while the balance of objects and the details of the process in which the objects are employed is hidden. The significant identity of a subsystem comes from the objects collected to do work, not from any specific role the subsystem (i.e., that combination of objects) might play at a specific point in an analysis or design. Unlike other forms of subsystem employed in the industry (e.g., [Booch, 1990], page 177-178), this definition 1) improves conceptual uniformity between objects and subsystems, encouraging architectural consistency, and 2) does not require extra-lingual support for implementation. Although a subsystem encapsulates the objects it uses from the perspective of the subsystem's client objects, there is no exclusivity implied in the relationship. Components of a subsystem are available to any other potential dependents in the software system as appropriate. *Operations* are the functional elements of an object, used to effect state changes and, consequently, perform useful work. Access to all objects is governed strictly by the object's interface, including usage protocols, which assures that usage is consistent with the object's abstraction.

SOFTWARE DEVELOPMENT STRATEGY

In the recursive life-cycle, the familiar life-cycle phases do not disappear, but instead shrink in size and multiply drastically in number, occurring on a per-level-of-abstraction basis. In a recursive development approach *there is no reasonable basis for saying that the project, as a whole, is in some specific development phase.* Instead, meaningful intermediate goals should be selected and, when appropriate, incremental capabilities resulting from those goals should be delivered to the customer throughout the development effort, for evaluation or actual use by the customer. Phases become a measure of work counted in man-days rather than man-years, applied to small, manageable groups of objects over a short period of time. (The Recursive life-cycle is illustrated on the next page.)

Thanks to the flexibility of recursion, it is entirely possible to emulate other approaches such as the Waterfall or Spiral [Boehm, 1986] life-cycles, but this is not suggested or encouraged. All of the strategy elements that we describe have either been employed in a contracted project or been formulated to address the issues raised in the course of those projects.

BRANCH MANAGEMENT

We refer to a given recursion level and its subordinate recursion levels, if any, as a branch (see figure 3). The key to managing the recursive life-cycle lies in the selection of which branches to prioritize for rapid development, and which to defer for later development. Branches predicted to be large, abnormally complex, or high-risk (for reasons such as incomplete system interface documentation) require special management. Branches associated with undefined application interfaces can be deferred until only those objects directly impacted by the interface remain to be developed. Note that, even employing recursion, not all interfaces can be deferred cavalierly. The choice of a particular data base system, graphical user interface, or real-time environment may have so pervasive an effect on a given application that very little profitable development may precede such decisions, even employing OOD and recursion.

RAPID DEVELOPMENT

The rapid development of branches is similar in many ways to a conventional prototype (e.g., [Boar, 1984]), except that the standard recursion products are produced and reviewed, and deliverable software components are tested and placed under configuration control. Components which are common among several branches are available from the effort of the earliest branches developed. This is *not* to say that rapid prototyping should not be employed under recursion. Rapid development of selected branches provides both a possible alternative to rapid prototyping and a natural approach for upgrading a successful prototype. Rapid prototyping may still be preferred in cases where there is a reasonable probability of prototype failure and the objects developed in the effort will not be of any other use to the overall project.

BOTTOM-UP DEVELOPMENT

Object oriented design is often suggested as a bottom-up complement to other design approaches (e.g., [Cameron, 1989, pp.293-304]). The suitability for bottom-up development does not disappear in a native object oriented development approach, and is supported by the recursive life-cycle. Not only can pre-existing "nodes," in the form of levels of abstraction, be grafted to appropriate points of the tree, but levels may even be developed prior to top-down confirmation of a need for the level. For example, a boundary object may be developed which, in a purely top-down effort, would not be reached until late in the project. In another case, early in a database-oriented project it may be obvious to developers that they want to follow a standard approach to the use of database/procedural language binding to avoid side-effects and promote uniformity. Utilities to support the approach can be rapidly developed in "isolation" from other software elements, and usually should be to assure that such utilities are sufficiently stable and mature when they are later required to support the development and testing of the many dependent elements of the software.

BRANCH MERGES

When a single level of abstraction is employed at multiple points in the tree, a merge of branches occurs. Such merges are consistent with the tree structure employed, which is in the form of a directed acyclic graph rather than the traditional hierarchy, and desirable, since they represent high-level reuse. From this it follows that, where two or more simple levels of abstraction possess considerable overlap, strategies should be considered for consolidating the two and thereby eliminating redundant design material. Merge points have a special significance in scheduling as well. For the pessimist, the merge point stands as an obstacle to the completion of multiple branches, especially in the case of a poorly-considered consolidation. Clearly, the importance of meeting schedule is greater for a node in the critical path of multiple branches and attention should be paid accordingly. For the optimist, the merge point is an opportunity to make progress on several branches very quickly.

DOMAIN ANALYSIS

Central to the management of any object oriented development project is the use of domain analysts to avoid redundant development effort, organize object libraries, specify standards to promote portability and polymorphism, and provide consultation in such matters to engineers and technical management. In contrast to project architects, whose job traditionally has been to develop a high-level software architecture for large projects to guide top-down development and determine the interfaces of major project subsystems, domain analysts are interested in those objects and operations which are common to a particular application domains. They are also concerned with the availability of compatible software elements from sources inside or outside of the developing organization. The domain analyst develops and maintains object taxonomies and consults at all stages of development to promote the most effective use of pre-existing elements as well as elements which may already be under development within the project or organization.

In a large development effort involving dozens of recursive branches simultaneously, the domain analyst's efforts become vital to reducing redundant effort and providing early identification of branch merge points and object reuse issues which only a wide perspective may provide. The domain analyst is concerned with the products of all phases, not just executable code, and not just products targeting a single implementation language. Specifications and designs for similar applications are bound to exist, and should be employed at least as a starting point where feasible.

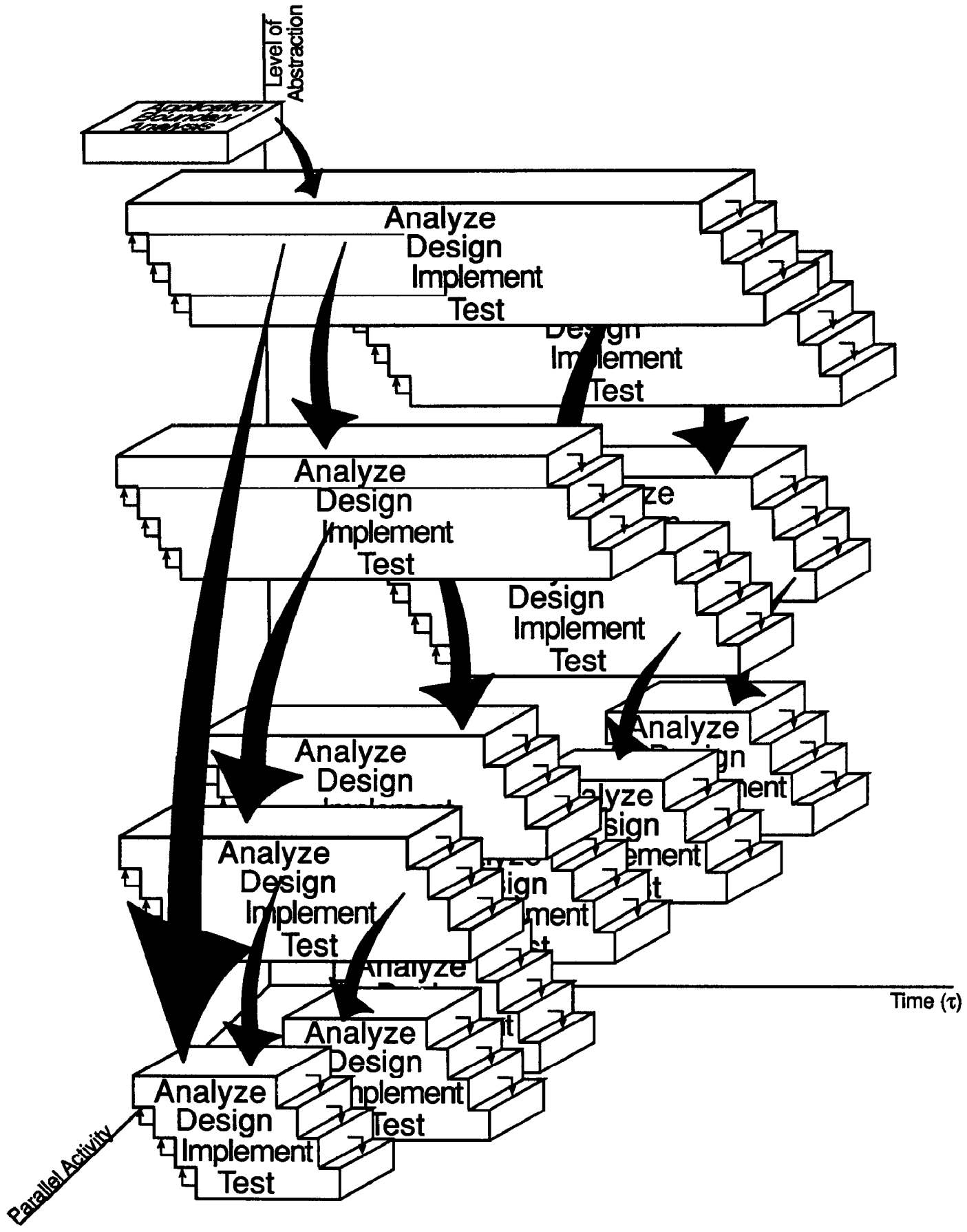

figure 4: Anatomy of a Recursive Development

The domain analyst serves as an important counterweight to the project architect. The architect is concerned with framing a solution to the specific problem at hand, while the domain analyst is concerned with how similar problems have been and will be solved. The architect is concerned with object sufficiency with regards to the application, the domain analyst with object completion as regards the abstraction's broadest potential domain.

The discipline of domain analysis is still very young, and requires cross-disciplinary skills into areas such as object classification and knowledge representation, as well as a broad and deep knowledge of the computing field. Practitioners should be innovative and have an eye to industry literature on the topic for support and new ideas. Finally, domain analysis requires the development and implementation of appropriate schema for the data base the analyst must develop, both for personal use and for browsing by development engineers. The more efficient the dissemination of domain analysis data via electronic means, the greater the return on the domain analyst's efforts and the general reuse effort, and the more efficient the cooperation between recursive branches.

PRODUCT INTEGRATION

The management and control of product integration is simplified tremendously by this approach. Since each object in the system is a complete, discrete system in its own right, the implementation and test of each component in the application becomes both unit test and system integration for that component. Testing of the level of abstraction is performed to demonstrate that the objects in question do *not* perform effectively in combination to meet the specifications for the level [Balfour, 1988]. The act of integration is implicit in the completion of each recursive level, up to and including the top-level system object.

Objects on the application boundary become a special case of integration when the corresponding external object is not available during the development or testing of a level of abstraction. In such cases, it is generally useful to develop a simulator object which, as a minimum, behaves appropriately for the necessary test cases pending replacement by the actual interface object (and related external facilities).

RISK MANAGEMENT

Risk management is a necessary part of any development process. In general, risk areas are concerned with areas of the project that are not well defined, are subject to critical constraints, or are technological problems which have not been solved. The recursive life-cycle approach recognizes that risk areas are an integral part of development and therefore provides a number of options for its management.

Areas that are involved with unknowns are not unusual in current development scenarios. In large-scale applications, concurrent development of hardware and software may cause interface problems. In the recursive approach, development of the system proceeds until that interface is reached. Since the interface is somewhat nebulous, a virtual interface may be substituted which provides the logical abstraction of the entity being interfaced to. The implementation of that object, which depends upon the actual interface, can be postponed until later.

Critical constraints and technological unknowns present similar risk problems, problems which, if unsolved, may mean project failure. In order to reduce risk, the areas of concern (branches) should have their development accelerated. Only when these areas are successfully completed should development continue. By focusing on the critical problem, cost and time risks can be reduced to an acceptable level. In other words, the amount of time and money at risk is reduced to a specific level rather than the project level.

COSTING AND STATUSING

Central to the practice of engineering is the estimation of job cost and the development of cost models to aid in project estimation and evaluation (e.g., COCOMO in [Boehm, 1981]). Such models are developed based on extensive databases containing information for dozens of projects on at least fifteen parameters. Such information is not collected and in place at this time for recursive approaches. While conventional models have not accurately predicted recursive efforts in the few cases available, these cases have done consistently better than the models predict. It is not clear at all that the assignment of a set skew factor to any conventional model will produce consistently good estimates. While experience is not yet sufficient to propose a specific model, or even suggest a conclusive set of model parameters, some general observations can be made at this time. Assuming constraints similar to mission critical software development, a level of abstraction (usually representing 400-2000 SLOC[4]) will usually take two to eight man-weeks, with complexity having a more significant impact on the schedule than simple SLOC count. SLOC estimation for a branch is usually done by "roughing out" the branch, forecasting a reasonable set of levels, assigning SLOC estimates to each one, then summing the SLOC estimates. Clearly, this requires some experience with the method and is highly subjective, but experience and judgment are always prerequisites to good estimation. Results to date have tended to be on the high side compared with delivered SLOC, though usually within 25%. For statusing, the SLOC projections are adjusted to fit the actual architecture of the branch as it emerges. When a node is completed, its actual SLOC count is added to the completed column and its latest estimated SLOC count is deducted from the to-be-developed column.

CUSTOMER SUPPORT

The following products and activities are normally used to provide the customer with a window into the project and should be tailored to meet the customers needs and interests. In an informal or non-contract development setting, these activities take very different forms, market canvasing and analysis, product testing, etc. That set of concerns and problems as it relates to recursive development projects is currently outside of our range of experience, so we will cling to the familiar ground of formal contracting in describing the customer-developer relationship.

REVIEWS

The primary mechanism for customer in-process oversight under recursion is the IDR[5] which is in many ways more similar to the DoD project *management* reviews in agenda and approach than the standard DoD technical reviews. The IDRs review project progress from the technical (rather than management) perspective. The incremental products delivered for the period in question are presented and commented on, customers and their technical representatives get direct access to project technical management, prototypes are demonstrated, etc.

IDRs are a natural consequence of the development method in that the product is developed through successive refinement of branches. Multiple branches are typically being developed concurrently and are at various stages of completion. Rather than waiting until all branches are at the same level of completion, IDRs allow for reviews to take place at a point in the development process when the review is meaningful. Typically each IDR is scheduled to coincide with a milestone that is important to the development (e.g., risk areas). Additionally, the products and scope of the next review can be determined at the current IDR.

The benefits of IDR, as opposed to more formal reviews, are obvious. First, because of the in-progress nature of reviews, there are few surprises either for the customer or developer. In other words, the potential for misunderstanding and subsequent loss of a significant

amount of time, effort, and dollars are reduced. Secondly, since the deliverable products are typically part of the review, the problem of product acceptance is virtually negated. Another significant benefit of the IDR is that it allows for delivery of products that are complete even though the project, as a whole, is not complete (e.g. a usable subsystem could be accepted prior to the acceptance of the complete system).

There is some cost incurred by the IDR concept. Multiple reviews require increased document production. Generally, the document production cycle is much less rigorous than might be the norm since the reviews are of software development documents produced by the method. IDRs which present deliverable documents (e.g. CDRLs) may be more formal. Additionally, a consequence of IDRs is that change control procedures must be adapted to the process. Experience to date has shown that the benefits of the IDR scheme of reviews far outweigh the cost incurred.

Because customers sometimes cling to 2167ish reviews, an alternative approach to the IDR process can be useful. In order to satisfy this requirement, certain IDRs take on a special status wherein the products presented are specified both in form and level of detail required. The necessity for this review "crutch" can be removed once the client has experienced the process.

MILESTONES

The strategies for selecting project milestones vary greatly depending on project conditions, developer style, and customer preferences. Perhaps the only strategy which should be ruled out *a priori* is a waterfall approach, which would hamstring the software engineers and severely limit management options for mitigating project risk.

In the recursive development approach, there are typically more milestones identified than in a non-recursive approach. These milestones are usually for smaller products and not based on project phases as much as they are based upon risk and resource management. Each milestone is generally associated with one or more branches with many independent milestones being tracked concurrently. New milestones are specified as older milestones near completion.

In a project where changes occur (almost all projects), milestones and schedules may be devastated by unexpected events. In the recursive life-cycle, since milestones are short-term, the impact of change is generally small and isolated. Additionally, since milestones are generally reviewed and/or negotiated at IDRs, the client is always aware of change impact.

There is an increase in overhead in managing a project in this fashion - there are more pieces to track. Experience, however, has shown that since the pieces being managed are smaller, lower level managers (development leads, etc.) usually have little problem managing individual branches.

DOCUMENTATION

The standard products to support an IDR are user's documentation (user's guides, operator's guides, on-line help, etc.), developer's/maintainer's documentation (an incrementally delivered product specification, incorporating object specifications, application maps, traceability information, etc.), and any separate interface specifications.

The primary development document for the recursive approach is a software product specification (SPS). The SPS is a complete document encompassing requirements specifications for objects, the design corresponding to the object specifications, the implementation of the objects' design, and lastly test products for the objects. This document is the primary review tool during IDR - showing the progression of the product throughout its development history. A particularly beneficial aspect of an SPS of this form is its ability to incorporate reuse products without any special handling. Each pre-existing object incorporated into the product under development will consist of the same basic components - requirements specifications, design, etc. For all practical purpose, there should be no difference in the form of documentation for a relatively simple object and a complex system. Each requires the same sort of information for completeness. The major distinguishing factor is the level of abstraction being considered. The ability to incorporate reusable objects in this fashion is a concept not supported by traditional phase-oriented documentation requiring separate documents for analysis, design, and so forth.

User's documentation is the least affected by these concepts except that most are currently written from a functional perspective having a more direct mapping to the functionally oriented development products. An alternative approach to user documentation is to follow what is becoming a popular paradigm in the commercial software product sector. Most modern user manuals now center their attention not on the operational characteristics of the software in the production of products, but focus on one entity at a time. Understandably, there is a high level view of the system operations, but significant attention is focused upon the *objects*, their operations, and their characteristics.

"UP FRONT" ANALYSIS

In recognition of the fact that most clients are unaccustomed to some of the ideas presented, resistance is expected. Perhaps the largest resistance area is the area of requirements. As mentioned earlier, customers (and often developers) are uncomfortable unless there are a set of requirements which attempt to specify the nature of the system to be developed to an excruciating level of detail. To the client, this requirement set is a protection mechanism designed to guarantee that the product that they receive is the product that they want. To the developer, the requirements specification also acts as a shield. The specification defines the criteria by which the product is judged. In the case of both parties, the baselined specifications serve as a safety valve for cost and schedule overruns - changes from this specification are inevitable.

In order to satisfy these needs for an up front analysis, the life-cycle is adjusted slightly. An object oriented boundary model is constructed. The boundary model is documented in an OORS[6] and is usually delivered between 1/4 and 1/3 of the way through the project's calendar schedule. Parallel activities such as analysis, design, or implementation of lower level abstractions are delayed until the boundary model and its specifications are complete. From the developer's point of view, this essentially means that useful work that could be accomplished is put on hold. To the client, the impact of this delay is cost (time and money). Of course both parties suffer from increased risk due to the nature of up front analysis. The complexity of the task and its products are much greater and subject to errors and omissions. Also, since these up front specifications are locked in, the impact of change is increased.

Up front analysis usually means the delivery of a software requirements specification as a separate product prior to beginning further development. From this point, a development document is produced. In order to maintain some level of continuity and traceability between these documents, the additional overhead of traceability maps or other information is incurred. Experienced developers and customers will generally concur that transition areas are subject to errors. The separation of requirements specification from other development products only serves to propagate the notion of distinct phases in the software life-cycle and provides little tangible benefit to either the developer or client.

It is often necessary in formal software development to specify a contractually meaningful statement of requirements to be met by the software product. This statement of requirements can be crucial to protecting both parties and stabilizing requirements in a large effort. In a functional approach, functional models of the application are developed and requirements are allocated to functional elements in the model. In performing an object oriented requirements analysis, an object oriented model is constructed. We find that an object oriented application boundary model is well suited for this purpose.

OBJECT ORIENTED BOUNDARY MODELS

A boundary model specifies the sum of system interfaces to the level of detail required for understanding and agreement between the developer and the customer. All such interface points are reducible to objects which may then be specified. These objects are referred to as boundary objects. As boundary objects are identified and specified, we must also represent how they will behave in combination to achieve higher level system processing goals, and for this purpose we add subsystems to the boundary model (though these subsystems are *not* considered to be boundary objects themselves). The resulting model specifies all critical aspects of the system in a form which will permit meaningful traceability between application specification and application design. While the appropriate decisions are left to the application designers, thanks to the nature of object oriented development, which arises from simulation development, the boundary objects offer a near one to one mapping to design objects and their target language implementations. The model's supporting subsystems will not share this advantage to the same degree, design details and efficiency considerations conspire to the contrary, but the tracing will still be far simpler than between orthogonal models (e.g., functional analysis to object oriented design) since it will be conducted between like products. While it is possible from the boundary model to demonstrate by analysis the sufficiency of producer (input) objects to satisfy consumer (output) objects, this exercise generally has a poor cost/benefit ratio when conducted without the assistance of automated tools, and should only be performed for elements of the model that will be truly immutable later.

TRACEABILITY

The tracing of requirements to their fulfillment, from analysis to design to implementation to test, is a natural consequence of an up-front analysis approach. Whenever analysis products are segregated from the corresponding products of later phases, some tracing becomes necessary to demonstrate closure in the later phases. In the boundary model we have essentially two sets of requirements, those associated with boundary objects and those associated with the subsystems completing the model. Since the problem space boundary objects will map to solution space objects in the application design, these requirements will be directly and obviously mapped to their solution space counterparts, and therefore directly to solution objects. The remainder will trace to the primary design elements, i.e., levels of abstraction, and by derivation to specific objects.

In the case where an up-front analysis has not been undertaken, it is not necessary to organize a formal boundary model. Boundary objects are identified and initial specifications are developed. These specifications are then employed and refined by levels of abstraction as needed in the course of recursive development.

In time, we expect the current traceability approaches employed in large scale contracts to be superseded by formal specification methods when high-reliability is demanded and by the ease of modifiability in an object oriented approach when proofs of reliability are not cost-effective concerns (i.e., when the cost of change is reduced an order of magnitude by the combination of greater ease of change and automation and streamlining of change control bureaucracies).

ADDITIONAL PROJECT MANAGEMENT ISSUES

The following issues may be of little interest to informal development projects, but are central to efficiently executing a recursive approach under 2167A or similar standards.

CONFIGURATION MANAGEMENT

Configuration management is significantly impacted by the Recursive life-cycle. With the high number of document submittals, change control mechanisms for the in-process product specification must be geared to the concepts of change capture and engineer accountability rather than bureaucratic configuration controls boards (which obviously still have a place in the control of formally delivered application branches).

Configuration management is concerned with the identification and control of "configuration items." Historically, when applied to software under DoD standards, such items have been design and code modules, i.e., abstract and implemented functions. These units are identified, assigned numbers, and organized into baselines.

The configuration management discipline arose when a missile program suffered a peculiar setback. In the push to meet project deadlines, engineers tried several different prototype modifications in rapid succession. The test in question was, of course, destructive. When one of the prototypes tested successfully, however, engineers were unable to identify the combination of modifications which had resulted in the successful prototype. Configuration management was introduced to assure that successfully tested systems could be reproduced

Configuration items are not rigidly defined in the standards, but their treatment is. The top level software configuration item defined by 2167A is the CSCI[7], which usually translates to an executable program, a major subsystem, or a library of common utility routines. Traditional standards such as 2167A depict configurations as hierarchical trees through which requirements are functionally decomposed from the CSCI, through one or more CSCs[8], to the lowest level requirements elements, CSUs[9]. Since software architectures delegate the satisfaction of requirements by dependency between architectural units, and since dependencies in an object oriented architecture are organized as acyclic directed graphs, we recommend that configuration management and the traceability model be adjusted accordingly, as shown below.

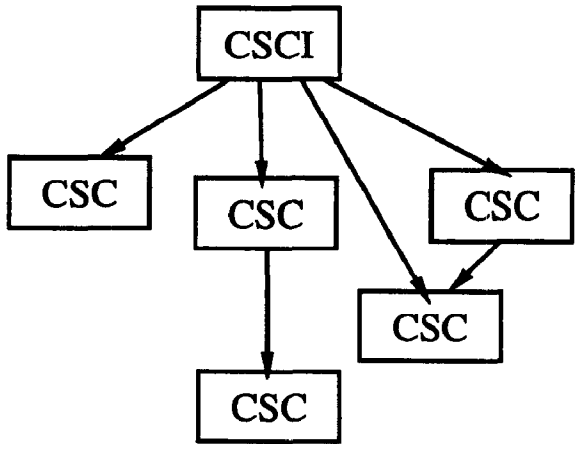

figure 5: Directed Acyclic Graph configuration

In our approach we associate CSCs with levels of abstraction. Requirements are associated with levels of abstraction , either from boundary model subsystems or, in a purely recursive analysis, from analysis performed within the recursive life-cycle. Children of those levels may derive requirements from the parent and may draw more requirements from the boundary model (unlike a decompositional approach where all requirements are levied against the top level and the remaining levels are decomposed derivations of those requirements).

We relate CSUs to objects, the significant elements in the implemented code. For example, in Ada we treat the specification *library units* [LRM, 1983, section 10.1] as objects, and therefore as CSUs. Related bodies and separate units make up the CSU's part list. This definition works well both for documentation and testing purposes. A CSC satisfies its requirements either by delegating them to lower CSCs or by satisfying them with CSUs. The resulting configuration maps CSCs to CSUs in a many to many fashion (as shown below), and is unconcerned with inter-CSU dependencies since they are subsumed in the CSC-level architecture.

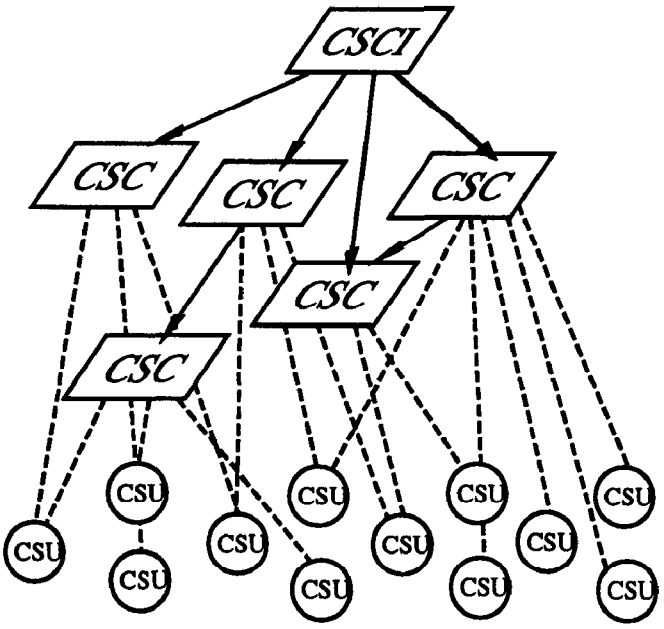

figure 6: Relation of CSCs to CSUs

In some cases, the customer may have strong expectations that the configuration tree will include only code-implemented elements based on prior experience. This will have the unfortunate effect of disowning the levels of abstraction from the requirements tracing. It is the levels of abstraction that determine how CSUs will be combined and employed to satisfy the level's requirements, making them a critical element in the requirements tracing scheme.

NOTES ON IMPLEMENTING THE RECURSIVE LIFE-CYCLE AS A SOFTWARE ENGINEERING PROCESS

Developing a software engineering process is not dissimilar from software development itself. The process consists of a set of tasks. Each task has prerequisites (inputs), a process description explaining what the engineer is to accomplish, and products (outputs). When the tasks prerequisites are available, the task may be assigned. When the tasks products have been accepted by the designated authorities, the task has been completed. The paradigmatic high-level tasks in software development are analyze, design, implement and test. The level of process definition required depends on the number of cooperating engineers, their skill levels, the criticality of the software being developed, the level of rigor demanded by the customer, and general project constraints (e.g., cost, schedule, development environment, etc.).

The Software Engineering Institute at Carnegie-Mellon in Pittsburgh has done extensive work on the evaluation of software engineering organizations focusing on the organization's own process definition, evaluation, and evolution capabilities. The related reports and studies may provide useful insight to appropriate goals in developing any new software engineering process, recursive or otherwise, within an organization.

EXPERIENCE WITH RECURSIVE LIFE-CYCLE APPROACHES

We provide six cases in which object oriented development within the recursive life-cycle model was applied to various degrees. These cases include projects using 2167A. A seventh case describes a DoD contract requiring, and structured to support, object oriented development and the recursive life-cycle.

CASE ONE

BACKGROUND

The customer, a DoD client, did not require normal DOD 2167/ 2167A standards, including specified documentation and reviews. The contract was, to some degree, a re-engineering effort in that an existing system was to be redesigned with significant enhancements. The only requirements that were formally specified were the enhancements. These requirements were specified in a two page memo and were very abstract.

PROJECT HISTORY

After obtaining training in Ada and Object Oriented Design, the contractor was convinced that they wanted to employ OOD as the development method. After an initial effort to apply an object oriented approach, it was decided to seek an outside consultant to assist in the transition to the new technology. The project was approximately six calendar months behind schedule (in an eighteen month schedule) based on the company's projections. The code that had been developed was discarded and the project essentially started fresh.

Without the restrictions of a waterfall life-cycle, the project was able to concentrate on analysis, design, coding, and testing concurrently. As simple objects were identified and specified, they were turned over to programmers for coding and testing. At the same time, a smaller team of analysts refined the abstract requirements, developed object specifications, and designed the more complex classes. As classes became less complex, they were turned over to less experienced designers for completion. A side benefit of this situation was that less experienced people were able to gain experience and contribute to the project. The better designers were easily identified and given more complex classes to design.

There were two particularly high risk areas in the project that were identified. Thanks to the nature of the life-cycle, it was possible to accelerate development of these areas. Thus, if the problems proved insurmountable, the project could be modified or cancelled at an early point with less financial or operational impact to the client.

Thanks to the nature of the product and the development method, we were able to deliver the product in phases. This enabled FQT10 to take place concurrently with development. Requirement changes, iden-

tified during FQT, were generally accommodated within a matter of hours (the longest case being two days).

Designing with reuse in mind played a significant role in the success of this project. Not only were software components reused, but also complete subsystem designs. At one point, new subsystems were being developed in two days by reusing the design and existing components. One measurement showed approximately 80% reuse achieved based on SLOC count.

OUTCOME

The project was completed approximately two weeks behind schedule. The final results of FQT uncovered few errors, errors which were addressed in one to two days each. Additionally, the slight schedule overrun was overshadowed by the partial product deliveries which enabled the customer to affect a smoother transition to the new system.

MANAGER'S PERSPECTIVE

Since the project manager had little training in the method, anxiety was high. The metrics by which the management had previously measured project milestones showed that the project would be extremely late. The management was unaccustomed to the unusually long analysis and design efforts. However, the management trusted the consultant and anxiety was relieved as partial deliveries were made. As promised, the amount of time spent in actual coding, testing, and integration was significantly reduced.

CASE SUMMARY

Despite a six month handicap on a 100,000 SLOC project, the contractor was able to mitigate customer risks and come in very close to schedule. The keys to this turnaround were the introduction of the recursive life-cycle, which made for more efficient use of project personnel , the reuse of significant amounts of design and code during development attributable to OOD, and an appropriate division of labor among project personnel.

Those people who were not directly involved in the project and had no OOD/Ada experience were quite certain that the project was going to be a disaster. Their personal metrics showed that the project would be unacceptably late, if delivered at all. Even the engineers directly involved had their doubts. It wasn't until the pieces started to come together into a cohesive whole that these engineers became convinced.

CASE TWO

BACKGROUND

The customer, a DOD organization, did specify the conventional DOD-STD-2167A standard, complete with a waterfall-based set of project deliverables. The contract specified the re-engineering of several applications. Some key technical staff members were cognizant of OOD, but had little experience applying it to a 2167A effort. Support for an object oriented approach was strong but mixed in the customer's camp, and was routinely tested by product deliveries.

PROJECT HISTORY

The proposed project management approach was to perform a conventional structured analysis to provide functional requirements and a recursive, object oriented approach to the remainder of the project. Barriers to a recursive approach were carefully tailored out of 2167A and other elements of the contract. A work breakdown structure was drafted which would not obstruct the recursive life-cycle while carefully measuring the reuse effort. It was expected that, when later contract options were exercised, experience with the recursive life-cycle would permit a fuller, more supportive tailoring of all areas of project management. This expectation was realized in case three, below. For the interim, a strategy of removing obstacles to recursion was pursued. The strategy for synchronizing with the phase-oriented products and reviews specified under the contract was to provide the correlating object-oriented products at milestones scheduled to permit the completion of the gross majority of such products under the recursive scheme.

The contractor was aware that there would be a serious problem providing meaningful traceability between detailed functional requirements and an object oriented design, but found this dilemma preferable to using any of the published OORAs[11] of the time. The customer's personnel attended the training in the object oriented approach with the contractor's personnel. The customer was persuaded in the course of the training to suggest that the contractor employ an OORA on the project, and the contractor accepted the suggestion with enthusiasm.

In the ensuing "analysis phase" the schism in the customer's camp broadened, a situation which was exacerbated by the fact that the customer was using the same contractor personnel (who were a clear and, during this period, unified faction) in program management support and IV&V[12] roles. Despite agreements to the contrary between the contractor and the customer, the support contractor attempted to use its position to effect a full return to functional analysis with a contractor-supplied tracing. As a result, this period saw all submittals rejected by the customer based on support contractor recommendation. Two events served to break this impasse. First, the contractor, faced with comments contradictory to stated customer positions, requested and got a high-level, bilateral meeting in which these issues were discussed and reconciled. Second, the first round of reviews with the customer's clients, who were located at multiple remote sites, failed to support the fears expressed by the support contractor. This review presented an object oriented analysis employing object oriented models and products, which were well received by the customer's clients in the field. While the OORA employed was not as well refined or mature as analogous functional analysis approaches, it was adequate for the applications in question and may represent the first such delivery of OORA products in a 2167A SRS.[13]

In the aftermath of those reviews the overall state of the project was recovering but not well. Engineering time intended for bottom-up and recursive development in early project phases had been lost on formulation and implementation of an OORA approach and multiple resubmittals of deliverables. The former might have been lost in any event under a functional analysis approach when traceability would be required to an architecturally orthogonal object oriented solution. Despite having met all project milestones on schedule, the head start into recursive development had been neatly sabotaged by events. Finally, the stress of learning and, essentially, beta testing an improvised OORA approach, combined with a long string of submittal rejections, had badly shaken the morale of the software engineering staff.

At this point, the project devolved into a waterfall life-cycle. With approximately three project months of planned recursive development lost and a technical management faced with meeting 2167A waterfall milestones, the process was now document driven. In addition, technical management failed to employ division of labor. All engineers were deemed equal in all development phases and used accordingly, though this obviously could not be the case.

At least one major benefit was still derived from the object oriented approach, however. The user interface portions of the application were being developed on a much larger related project by another contractor. Originally, the design and Ada specification parts for these modules were scheduled to be available over eight months prior to the project's CDR[14] milestone. This material was still unavailable when CDR was held, on schedule, for those elements of the software

that did not directly use the interface modules. At this point, development was suspended pending the availability of the necessary modules.

MANAGER'S PERSPECTIVE

Shortly prior to the completion of project requirements, there was a change in project managers for the contractor. Both managers rightly insisted that techniques be developed to determine the status and effectiveness of any software engineering process instituted. Neither was conversant in the details of an object oriented approach, but both were attitudinally supportive to the effort to develop an effective project management approach for its utilization and were willing to bear with some intermediate chaos to that end.

OUTCOME

CDR was held on schedule with the caveats mentioned, but with products that were not exceptional and at far greater expense than was necessary. The attempt to exploit combined experience in OOD and formal 2167A development produced the following conclusions:

1) The "obstacle removal" approach to contract tailoring was, in the final analysis, insufficient to assure the efficient employment of a recursive life-cycle. A more explicit and supportive structure is necessary to effectively implement a project management process. The tailored work breakdown structure showed work being diverted from recursive development throughout the analysis effort, but no effective solution was implemented.
2) Failure to employ basic management techniques, in this case division of labor, added unnecessarily to the stress and inefficiency of the software engineering process.
3) The use of OOD enabled the contractor to complete a greater portion of the design than would otherwise have been possible, and deliver it on schedule.
4) Necessary improvements in both the software engineering process and the products should be made to enhance product interrelation and truly support a recursive engineering process.
5) The customer needed independent technical support with experience in OOD to meaningfully interpret and evaluate OOD products. Despite customer enthusiasm for OOD, unfamiliarity with the approach, and a support contractor working at cross purposes effectively negated many potential benefits, including higher quality products and greater cost effectiveness.

Employment of an object oriented approach enabled the project to overcome circumstances that would normally have led to significant schedule slips and cost overruns, but much more could have been accomplished.

PERSONAL OBSERVATIONS: DINO R. RUSSO

I became associated with the project after PDR. I was asked to evaluate what had been done to determine what changes should be made in order to improve the development process. A number of things were immediately evident:

1) The requirements analysis approach had little to do with an object orientation.
2) There was no transition from requirements to design.
3) The application of OOD concepts was limited.

Because of the lack of customer conviction and due to the lack of significant practical experience applying the method to a project, many problems were encountered. The problems had a significant impact on project morale, milestones, and products. The primary problem, however, was the inability to apply the OO approach consistently across the development life-cycle.

CASE THREE

BACKGROUND

Same company and client as case two. After evaluating the problems encountered with the previous project it was decided that significant change was necessary. The SDP and 2167A DIDs were significantly modified to reflect the method in a more mature form and to apply the method consistently throughout the development. A significant amount of effort was expended involving the client in the change process. Although getting the client to accept a pure recursive life-cycle was not completely successful, the following significant changes were accomplished:

1) Documentation reflected the method. Although a requirements specification was required as a separate entity, the requirements were organized around the boundary model and its identified objects so that there was a direct transition to the SDD.[15]
2) The PDR/CDR cycle was replaced with IDRs (although the last IDR was to be called CDR). The IDRs would not be dog and pony shows but instead be a working meeting between the contractor and client to review work in progress.
3) The WBS reflected the iterative nature of OOD allowing for overlap between analysis and design allowing for more accurate measurements of effort expended.
4) Because of funding constraints, the coding and testing activities had to be postponed. High level coding (using Ada as a PDL) is permitted as required in the design process. Although not an ideal situation, the impact of this constraint is minor. The delayed activities, when funded, are simply continuations to the method and its products.

PROJECT HISTORY

The project is currently in progress. The requirements specification, which had been the largest stumbling block to previous CSCIs, was produced and accepted with few problems. The SDD[16] has been started and is ahead of schedule. The transition between the two documents has been effortless.

OUTCOME

Not yet determined.

MANAGER'S PERSPECTIVE

The project manager has acquired an understanding of the method and has been aggressively supportive throughout the process. The management methods necessary to coordinate and manage the activities involved have been modified to match the method. This was not a small undertaking. Significant effort was expended in the analysis of the method, measurable milestones, work breakdown structures, and metrics for evaluating work in progress. Management has steadfastly resisted any attempts by the contractor or support personnel to revert to older methods of development or its evaluation.

ET CETERA

Each corporate activity directly supporting the development process has expressed little less than amazement at the results to date. Requirement specifications are much more understandable and testable, QA found that the consistent application of the method made the evaluation of the products much easier, and the software engineers are ecstatic that their work is designed to contribute directly to the product and not to satisfy some artificial documentation and process criteria. Project morale has improved significantly.

PERSONAL OBSERVATIONS

Many of the problems experienced with the previous project were alleviated because the method is driving the process, rather than an artificial set of rules defined by a DoD standard. The key to this modification involved a mature understanding of the OOD approach across the entire life-cycle, and being able to transfer this understand-

ing, at least in a limited form, to the client (including the IV&V contractor). Although the pure recursive life-cycle was not considered acceptable to the client for financial and perceived contractual reasons (had to have requirements baseline), we achieved approximately 75% acceptance. The remaining 25% will not have a particularly detrimental impact overall. Though the project is in its relative infancy, confidence is high that the program will be a success and that the success will allow the contractor to reduce the client's reluctance to let go of their traditional methods and metrics.

CASE FOUR

BACKGROUND

This project was an internal development effort (named STRIKE) for the Air Force at Offutt AFB from January 1988 to March 1989.

PROJECT HISTORY

The project was conducted by a team leader and three other team members. The team members had no real training in object oriented methods and the team leaders attempts to teach them OOD had little success. The development methods devolved into a mixture of OOD and Functional Decomposition using an *ad hoc* life-cycle and approach.

OUTCOME

The project was marginally successful. Although the project was delivered on time and was marginally useful, the product was deemed unmaintainable. The development organization refused to turn the product over to the maintenance organization preferring to attempt the maintenance themselves.

CASE SUMMARY

This project led to the conclusion that training in the methods would be necessary to get the benefits of the object oriented approach and the recursive life-cycle. It also led to the conclusion that the mixing of methods should be avoided.

CASE FIVE

BACKGROUND

This project was an internal development effort (named M3online and pronounced "M-cubed" on-line) for the Air Force at Offutt AFB. The project was delivered in three builds. The first was a non-functional prototype with a team of 5 people working from January to July 1989. The second build contained partial functionality and went from July 1989 to December 1989 with a growth from 2 people to 4 people in September. The third phase went from January to May 1990 and averaged three people.

PROJECT HISTORY

This project had new management which was more open to using modern software engineering technology including object oriented approaches and a recursive life-cycle, and was also able to work with technologies that were newer and therefore of higher risk. The technical team was different on the second project. Only the team leader carried over. He and the others had now been through formal training on the methods and life-cycles. Additionally, the characteristics of the team members were different. The team leader described them as "more dynamic individuals — younger in heart and mind."

The team leader was also the project manager. A chief designer was also appointed. All team members were not immediately advocates of the object oriented methods and recursive life-cycle. However, the team leader and chief designer did most of the initial work and the others soon started to see the benefits of the methods.

This project started off with an initial analysis of the requirements. This was followed by an application of the OOD process at each level of abstraction. However, the implementation of the software was deferred longer than necessary and was completed only after most of the recursions had been performed.

OUTCOME

The system was delivered within the expected schedule and budget. and had been felt to be a good demonstration of the methods. This success lead directly to the next two project efforts (cases six and seven) and to the contract effort described below.

CASE SIX

BACKGROUND

This project was a rewrite of the STRIKE project (see case four).

PROJECT HISTORY

In May of 1990, a single individual was tasked to investigate the rewriting of the STRIKE project to take advantage of the M3online objects. While only a single individual was tasked to this effort, that proved sufficient. The analyst was able to express the STRIKE system as a new subsystem which was built using the M3online objects. This development activity took approximately 4 weeks (versus the 12 months for the initial STRIKE project). However, this smaller 4 week activity was much less formal and did not apply the life-cycle activities in strict order.

OUTCOME

A reuse level of 75% was achieved and maintainability was significantly improved as compared to the original project. Also the development time for the re-engineering process was considerably less than the original development time.

CASE SEVEN — A NEW KIND OF CONTRACT

BACKGROUND

From January 1989 to July 1989, the Air Force at Offutt AFB created a new kind of contract to specifically support a recursive life-cycle development by a contractor. The M3online system described previously is part of a larger system known as DGZ Construction.

PROJECT HISTORY

DGZ Construction was an existing FORTRAN system under maintenance by SAIC. From January to July 1989 a new contract was written for the re-engineering of the entire system (approximately 25 programs including M3online). The new system, a five year development effort, would employ Ada and OOD to re-engineer the existing subsystems and integrate them under a single user interface.

The contract was structured as a delivery order contract with multiple tasks. Each of the delivery orders must be approved by the contracting officer and the technical officer before the next one may proceed. The delivery orders specify the use of object oriented methods, the recursive life-cycle, and the Ada language.

One difference between this and typical contracts is that the delivery orders will not be structured as typical single life-cycle phases as in a waterfall approach. Rather, each delivery order will ask for the creation of all (or possibly some) products associated with a single level of abstraction. Additionally, the statement of work specifies that the contractor must reuse objects from the HQ SAC/SCWN object library and that any objects created during the development become part of that library.

Most of the problems in creating this kind of contract were not legal or procurement issues, rather the issues were related to the technical evaluation that was a prerequisite to the bidding process. Each contractor was required to submit a technical proposal so that the Air Force could evaluate their knowledge of object oriented techniques.

Many bidders submitted software architectures that were functional modules with noun names. As a result, the customer had to do much explaining to the losing bidders as to why their technical proposals were deficient.

OUTCOME

Although the contract performance has just started (award was in spring 1990), the early results are very positive. The contractor is pleased because they see lower risk since they are not required to bid on a five year contract up front, but instead can estimate costs and schedule for each recursive OOD effort on a per delivery order basis. This is especially useful since delivery orders are only issued toward the end of an existing effort so that much more information is know at the time. The customer organization is happy because they will be able to actively participate in the process, achieve a significant amount of reuse, and be able to be flexible in structuring the requirements as changes in need evolve over the five years.

There have been no protests concerning the use of Ada, Object Oriented Development or the standard reuse library. Also no award protests have been filed.

CONCLUSION

Just as the Waterfall life-cycle grew naturally from the development processes employed at the time, the Recursive life-cycle is a natural consequence of object oriented development methods. As one would expect, this life-cycle inherits many of the advantages of the development approach. Additionally, in as much as development suffers when a mixed paradigm is applied, so too do projects suffer when the life-cycle does not fit the development model. It is the hope of the authors' that the continued evolution towards the recursive model presented continues to gather support and success.

BIBLIOGRAPHY

[Arango, 1989]. Guillermo Arango, "DOMAIN ANALYSIS: From Art To Engineering Discipline," *Proceedings: Fifth International Workshop on Software Specification and Design, ACM SIGSOFT Engineering Notes,* Volume 4, Number 3, May, 1989.

[Balfour, 1988]. B. Balfour, "On 'Unit Testing' and Other Uses of the Term 'Unit,'" *MCC '88 Military Computing Conference,* Military Computing Institute, 1988, pp. 127-130.

[Boar, 1984]. Bernard H. Boar, *Application Prototyping, A Requirements Definition Strategy for the 80's,* John Wiley & Sons, 1984.

[Boehm, 1981]. B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Boehm, 1986]. B. W. Boehm, "A Spiral Model of Development and Enhancement," *Software Engineering Notes,* Vol. 11, No. 4, August, 1986.

[Booch, 1987b]. G. Booch, *Software Engineering with Ada,* Second Edition, The Benjamin/Cummings Publishing Company, Menlo Park, California, 1987.

[Booch, 1990]. Grady Booch, *Object-Oriented Design with Applications,* The Benjamin/Cummings Publishing Company, Redwood City, CA, 1990. (Note: copyright date 1991).

[Cameron, 1989]. John Cameron, *JSP & JSD: The Jackson Approach to Software Development,* Second Edition, IEEE Computer Society Press, Washington, DC, 1989.

[Cox, 1986]. Brad J. Cox, *Object Oriented Programming: An Evolutionary Approach,* Addison-Wesley Publishing Company, Reading, MA, 1986, 1987.

[DOD-STD-2167A]. DOD-STD-2167A, *Military Standard, Defense System Software Development,* Department of Defense, Washington, DC, 1988.

[JLC, 89]. Joint Logistics Commanders, JPCG-CRM, CSM, "Software Development Under DOD-STD-2167A: An Examination of Ten Key Issues," October 25, 1989.

[LRM, 1983] ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language,* United States Department of Defense, 1983.

[Maibor, 1989]. David S. Maibor, "DoD-STD-2167 Defense System Software Development," David Maibor Associates, Inc, September, 1989.

[Royce, 1970]. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings, WESCON,* August, 1970.

ENDNOTES

[1] The term "product" has an implication of completion. This supports a waterfall life-cycle but has negative impact upon more pragmatic approaches. A more liberal interpretation is in order.

[2] While there may be applicability to non-software development processes, they are not pursued in this paper.

[3] Other object oriented mechanisms for structuring objects, such as delegation and generics, are not considered here due to space considerations. When employing generics in languages such as Ada and Eiffel, we map the generic module to a partial class design element and the instantiation to completed classes, usually in a parent-child inheritance relationship.

[4] Source Lines of Code. The counting method employed, unless otherwise noted, refers to the total high-level programming language statement count, excluding comments and blank lines.

[5] Incremental Development Review.

[6] Object Oriented Requirements Specification.

[7] Computer Software Configuration Item

[8] Computer Software Component.

[9] Computer Software Unit.

[10] Functional Qualification Test.

[11] Object Oriented Requirements Analysis.

[12] Independent Verification and Validation, essentially the customer's quality evaluation function.

[13] Software Requirements Specification.

[14] Critical Design Review, the culmination of the 2167A detailed design phase.

[15] Software Design Document - actually the document is a software development document containing analysis, design, and implementation products. The SPS is merely the completed version of the SDD.