N91-10616

A Modernized PDL Approach for Ada Software Development

Paul Usavage Jr.
(215) 354-3165

M&DSO / Ada Core Team
Valley Forge, PA

ABSTRACT

The desire to integrate newly available, graphically—oriented CASE (Computer Aided Software Engineering) tools with existing software design approaches is changing the way PDL is used for large system development. In the approach documented here, Software Engineers use graphics tools to model the problem and to describe high level software design in diagrams. An Ada—based PDL is used to document low level design. Some results are provided along with an analysis for each of three smaller GE Ada development projects that utilized variations on this approach. Finally some considerations are identified for larger scale implementation.

BACKGROUND

In 1987, the Ada Core Team was formed within GE's Military & Data Systems Operation to apply advanced technologies including the Ada language to the development of large satellite ground systems that form our business base. GE M&DSO has been producing real time satellite ground stations for 15 years with a strong, established methodology. The addition of graphics workstations and graphics tools to this methodology is just a natural evolution of these methods. The techniques proposed here have grown out of GE's methodology and been refined through use on various Ada projects and IR&D work. The information in this paper is based primarily on the results of these efforts.

INTRODUCTION

The availability of automated graphic tools supporting structured analysis and structured design techniques, and the need for major improvements in productivity and quality are causing software organizations to rethink their software engineering methodologies. PDL (Program Design Language or Process Description Language) is the most commonly used design tool in many organizations. As a result there is a wide base of experience in PDL as a descriptive medium.

Yet, when an organization wants to add CASE (Computer Aided Software Engineering) tools to their existing methodology, it often is unclear what role PDL should play. Are PDL and graphic CASE tools redundant, or can they both contribute to modern software design practices? And what about the practice of coding some Ada constructs (notably package specifications) during detailed an even preliminary design? Does this narrow the scope of PDL's usefulness?

This paper is intended to document our analysis of the most effective tools for each portion of the software design cycle. Each tool, graphics, PDL, and Ada source code, has characteristics that make it useful to apply to part of the design problem. PDL has been used in the past for the representation of many design aspects. Today there are areas where PDL is best suited, and areas where other tools are better suited than PDL.

By way of further introduction, let us examine the traditional design approach and use of PDL.

TRADITIONAL APPROACH TO SOFTWARE DESIGN

Traditional documentation of a program with PDL involves two parts. The primary part is the process description, which is a description of the implementation or algorithm used in a program, subprogram, process, function or procedure. The second part is the prologue, which is usually present to support the process description by explaining input/output data items and local variables. The prologue often provides references to the design or requirements documentation, and usually includes information and format necessary to an automated PDL processor. Sometimes the term PDL is used to refer to just the process description, and others it is used to refer to the prologue as well. In this paper PDL will be used to refer either to the process description and to the language used for process description.

Software Design Phases

The evolution of a software design occurs in distinct steps over several project phases. During the **Software Requirements Analysis** phase, a software system is partitioned into Computer Software Configuration Items (CSCIs), and all software system requirements are allocated among these CSCIs.

During the **Preliminary Design** phase, a high level design is conceived for each CSCI sufficient to satisfy its allocated requirements. This design is described in English in a continuous, flowing, 'easy to read' paragraph format. Software hierarchy charts are usually prepared next for the design review. Database and file format designs are initiated during this phase to reflect attributes of the preliminary design.

The software design process continues during the **Detailed Design** phase with the generation of preliminary software source modules for each design component. The method to be used in these modules is described using a PDL process description. The first 'cut' at this description would likely be at a high level of abstraction (showing fewer details). Iterative refinements are then made of the PDL process description, assisted somewhat by the use of structure charts. The design is refined by adding more detail on how the module's functionality is to be provided. This lengthens the process description, and separate, subordinate modules are then created to break out cohesive elements of this process description. A PDL processor is used during this activity to check for syntax errors and to create calling trees and object/variable cross-references for analysis use.

The end of the PDL refinement process is reached when two criteria are felt to be satisfied. The first requires that the process descriptions should be detailed enough that the module can be coded by someone familiar with the technology but unfamiliar with the design. The second criteria requires that process descriptions must be of a suitable length (between 1 and 2 printed pages) to result in reasonably sized code modules. Consistency and quality are encouraged by the establishment of PDL standards, by the informal sharing of sample PDL, and by peer review or structured walk—through of the PDL processor printed output.

The **Coding** phase implements the design. The source code is written into the same modules already containing the prologues and PDL process descriptions. In some cases the source code is

interspersed throughout the PDL in a style that explains a step of conceptual processing with a block of PDL, then implements it with a block of source code. In other cases the entire process description is kept intact at the beginning of the module, followed by the entire source code. The former makes it easier to match PDL to source code, while the latter allows the PDL (and the source code) to be better seen and understood in whole.

Benefits Of Traditional Approach

Our Software Development section has enjoyed steady productivity gains since this PDL methodology was adopted. PDL usage has resulted in higher quality and greater productivity than previous development methods (which made use of, among other things, English prose descriptions and flowcharts). Of course, many factors are at work in increasing productivity including the availability of more and better hardware, but at least some of this improvement can be attributed to the use of a vigorous, robust, well-known and well-followed methodology. The use of PDL contributes to quality and productivity in the following ways:

1) Creation and maintenance of documentation is easier when employing the same tools (*e.g.*, computer terminals, editors) used in writing the source code.

2) Design descriptions are more complete, rigorous, detailed, and more standardized.

3) Design walkthroughs may be used more readily to reduce the number of design errors.

4) Some aspects of the design (*e.g.*, syntax, keyword balancing, call trees, indexing of references) may be checked automatically.

5) Deliverable documentation may be produced automatically from source code containing PDL.

6) Fewer errors are made when representing actual software implementation due to the proximity of PDL and source code.

7) Less effort must be spent on explanatory comments when the PDL is located with the source code.

Disadvantages Of Traditional Approach

Usage of this approach has also shown some disadvantages. Some of these are:

1) The 'easy to read' English prose used in preliminary design documentation is hard to write in a way that is free from ambiguity.

2) The PDL documentation for a large system is copious and very low—level in detail; it can be very difficult to find the PDL associated with a given aspect of system behavior.

3) PDL does not support well the more formalized *structured* approaches to partitioning (*e.g.*, analyzing coupling and cohesion) and automated checking, especially when experts try to review the partitioning decisions of others or when automated tools are used to *verify* the design.

4) PDL approaches traditionally have neglected the data part of a design

Advantages of Newer Graphic Tools

CASE tools now available automate graphically—oriented regimens in system analysis and software design. These tools include support for such approaches as Data Flow and Control Flow Diagrams, Structure Charts, Entity Relationship Diagrams, Object Dependency Diagrams, Object Interrelationship Diagrams, Data Dictionaries and integrated tool databases. GE has used the **team***work*® tool from Cadre Technologies, Inc. for the studies described in this paper.

The automated graphic tool approach to Structured Analysis and Structured Design has many commonly recognized benefits:

1) Communication via graphics seems to occur at a much higher information bandwidth, using visible relationships and psychological cues to more quickly attain a high level of reader understanding.

2) Graphics seem to provide better support in decomposing or partitioning a software problem or design, and in examining alternatives and reviewing the results.

3) Production of graphics for formal presentations and reviews is automated.

4) Tools can often assist in the storage, control of and access to information by design teams.

5) Tools can provide higher levels of automated balance and consistency checking by including a data dictionary, and in some cases can automate design verification.

6) Graphic tools seem to better represent system level behavior, interface design, and data design.

Disadvantages of Graphical Tools

Graphics CASE tools also have their disadvantages, including:

1) Graphics are generally less effective than PDL when dealing with larger quantities of low level details (for example, flow charts become considerably less attractive when used to document low level details of very large programs)

2) Newer, more complicated approaches may require much more extensive tool and methodology training to be successful.

3) Graphics CASE tools can involve a substantial additional investment in both hardware and software.

4) Development schedules must be adjusted to reflect additional time spent on the front-end design.

5) It is very difficult to prove (*e.g.*, to customer or business management) that the additional time and money spent up front results in cost savings later.

6) Human nature sometimes leads people to believe that the tool will do the work for you; really it just helps to represent work *you* do yourself.

PROPOSED METHODOLOGY

The following methodology, documented in our Software Development Plan, has been synthesised from our existing methodology and from proposals by many authorities. It has been adapted to complement our existing approach and is recommend by our group for GE's large development contracts. The phases here are much the same as in other approaches, including the classical waterfall approach and the default cycle documented in DoD—STD—2167A. Familiar activities occur during the phases but more effective tools, refinement techniques and documentation media are used.

The basic approach uses graphics at the higher levels of abstraction and PDL at lower levels. This documented approachsupports the use of the Ada language well. A non-Ada version of the Software Development Plan is planned to properly exploit this same methodology on non-Ada projects. The current Plan version makes use of object—oriented terms and methods. However, it is intended to support either object—oriented or functional decomposition of a system, or an approach that hybridizes the two.

Approach By Phase

The Software Requirements Analysis activity uses a basic Structured Analysis approach (as described by Yourdon & DeMarco, McMenamin & Palmer, Ward & Meller, Hatley & Pirbhai, and others) including the use of Data Flow and Control Flow Diagrams and a Data Dictionary for Essential and Incarnation models (see the references). The purpose of this is to model the problem in more detail in order to understand it. This is done first in a way that removes the consideration of technology from the statement of the problem solution, and then adds it back into consideration. The results of this analysis, in the form of Data Flow Diagrams, are input to the next phase of software development.

Preliminary Design involves the identification of Configuration Software Components (CSCs) from the Data Flow Diagrams. These may be high—level objects and operations identified in an Object—Oriented approach. Object Dependency Diagrams are produced for the identified objects. Interfaces between CSCs (and CSCIs if not done during Requirements Analysis) are defined, then depicted using package specifications. The package specifications are coded in Ada, showing the Ada declaration of each resource (mostly types and subprograms) exported from the package specification, along with Ada *with* clauses showing necessary dependencies. Compiling these interface specifications checks for consistency and makes a firmer foundation for further breakdown of development work. High—Level executive CSCs are described with PDL at this stage to show the major elements of control. The PDL for the executives would include the creation of their declarations in package specifications or as stand-alone subprograms, along with Ada with

clauses for their dependencies. The PDL consists of structured language process descriptions based on the Ada executable statements for iteration, loops, and conditionals. No attempt is made to compile the executives at this point, the purpose is to describe control dependencies inherent in the design. This PDL may in fact be contained solely within the CASE tool and not within a source code member at all. This makes it instantly accessible when documenting and refining later stages of the design.

The design process continues during the **Detailed Design** phase as structure charts are generated for each CSC. These show the architectural details involved in implementing the CSC. Computer Software Units (CSUs) are identified. These may be lower level objects in an object—oriented system. The implementation of individual CSUs are described in PDL process descriptions within the CASE tool graphics environment. This gives the programmer a better sense of partitioning and of the overall system structure than does writing the PDL into a disconnected source file. No compilation is attempted of these process descriptions. They are based on the Ada language syntax for universality of understanding, not for compilability at this stage. However, new interfaces derived at this detailed level of design (*i.e.*, more package specifications) are coded in Ada and checked with the compiler. These package specifications declare all types and data structures necessary to components external to the package specification. Also, within the package bodies, internal types and major internal data structures are coded in Ada and compiled. This helps to firm the data design and package dependencies. This is a major design component that is best described and checked with the Ada language and compiler itself.

The **Coding** phase that follows detailed design involves transfering the PDL from the CASE tool into existing and new Ada source modules, then writing Ada code for the design represented in the PDL process descriptions.

TRIAL PROJECTS

A number of GE Ada projects have been undertaken using variations on the traditional and proposed methodologies. The following projects have been selected to present some variety in approaches to PDL. No hard metrics are available for these projects to give insight into the contribution of methodology components, such as the number of errors created and found during a phase, or even created but not discovered. Instead, project team members were interviewed about problems, rework and errors that occurred. Their comments were then analyzed for apparent relation to the choice of methodology.

The projects described here are IR&D projects that have occurred over the last two years at GE. They appear here in chronological order, and in fact show an evolution in methodology over this time period. Methodology refinement was not the primary intention of these IR&Ds, each one was instead performed with what seemed the best approach to those directing the efforts at the time. Methodologies of later projects were of course tuned to benefit from the lessons of the earlier ones. Most participants were first time Ada programmers, although each project (after the first) had at least one person assisting during coding that had benefitted from some experience on a previous phase. The experienced people were not usually available during the design phase, however.

Project 1

One study in Ada software development involved the redesign and re-implementation of a predictive mathematical simulator. The project resulted in approximately 8000 compiled Ada statements (counted by semicolons, not including blank or commented lines). Automated CASE tools were not available during the study. Diagrams were produced using a PC-based general-purpose drawing tool. The Ada compiler itself was used to check the PDL for syntax. PDL consisted of coded and compiled Ada block constructs (e.g. loops, conditionals), compiled type and variable declarations, and Ada comments instead of procedural (sequential) statements.

During Preliminary Design, narrative English specifications were produced according to more traditional development methodology. Object/ Package Dependency Diagrams and Control Flow Diagrams were drawn. These were presented during the Preliminary Design Review (held at the end of the Preliminary Design Phase), but effort was not spent to maintain these diagrams for use during Detailed Design. High-level objects and procedures were identified and package specifications coded (but not compiled—the development environment was not available at the time).

During Detailed Design, the Ada package specifications were entered and compiled. Any interface errors detected then were corrected. Package bodies, subprograms and most types and

variables were declared in compiled Ada within the code modules.

In the Coding phase, the unimplemented (commented) portions of the compiled PDL bodies were coded and the components integrated and debugged.

The study was a quite a success as far as Ada software development was concerned. However, an analysis is possible of problems that arose during the study for possible effects of the choice of methodology. For instance, there was a wide variation among the six programmers participating in the study in the style and composition of the compiled Ada PDL. Some felt very comfortable during Detailed Design writing almost complete Ada code and very few PDL comments. Some felt very uncomfortable with the Ada syntax and compiler and wrote mostly comments and few compiled types/objects/block constructs. This sometimes resulted in inconsistent levels of abstraction of the PDL design description.

In general, the project tended to achieve different levels of abstraction and maturity at different times. It took longer for a programmer to write PDL that was mostly code. It took less time to write PDL that was mostly comments, but more time to write the source code in the next phase. Management misunderstandings resulted from this when attempting to assess the progress of the effort at a given point in time.

The problem with different styles of PDL and different PDL/code contents appears to be more common with projects that use an Ada compiler to check PDL. This also seems to occur more frequently when there is less experience with Ada and the PDL approach. One remedy for this is more and better training. Another is *not* to use the Ada compiler to check PDL syntax—and the problem goes away if a PDL processor is used which has a more forgiving syntax, or if only a visual check is performed on the PDL. The visual check is appropriate only if module sizes are kept small. After all, PDL syntax errors are only damaging if they cause ambiguity or incorrect interpretation in the design.

The problem with inconsistent levels of PDL abstraction that showed up on this project is common to many different approaches and processors. This is bad because it is confusing, it makes the design less understandable and less easily checked by others. Abstraction is useful because it hides those details unnecessary to this portion of the problem solution. The more localized the scope of detail, the less affected the system will be if it changes. Each person (or component of software) has to be an expert in fewer areas, and is free to concentrate and come up with a better, more pure solution in his/her/its own area. Removing unnecessary detail makes a system design more understandable, modifiable and robust.

The consistency problem decreases with programmer experience. Levels of abstraction can also be checked for consistency during peer review or structured walkthrough, giving feedback to the programmer and allowing the descriptions to be corrected. The best level of abstraction for a PDL process description of a given module is somewhere above (less detailed than) the level at which the source code for that module would need to be written.

Despite the apparent problems the team was able, however, to bring all portions of the system to completion by the end of the test phase. The productivity of the total effort was only very slightly lower (a few percent) than that of the more traditional projects. This was probably affected by a variety of factors including less effective training, lack of tools and technical difficulties with the platforms used, but also that slightly less documentation was produced than is normal.

Project 2

The second project for analysis was a 1988 IR&D effort to design and implement a platform-independent Ada binding for a Man-Machine Interface. Portions of the project made use of the graphic CASE tool when it was available. It used an Ada based, uncompiled PDL but no PDL processor. This project resulted in a larger design than was implemented, with about 2000 lines of compiled Ada code (again by semicolons, not including blank or commented lines) being produced.

During Requirements Analysis, Data Flow Diagrams were constructed to describe physical, logical, and incarnation models. The resultant diagrams were used during Preliminary Design to help identify high-level objects and to partition the system. Ada package specifications and their bodies were written (with subprograms deferred) and compiled to document the interfaces. Object Dependency Diagrams were drawn to show the object relationships.

During Detailed Design, extensive use was made of the Ada compiler. Drivers were identified and coded in Ada. Important type and object declarations were coded within the package bodies. A

key routine in each of the major objects/packages was coded and tested to ensure the feasibility of the design. A *key routine* was some subprogram that, when demonstrated, would validate most of the design decisions for the rest of the subprograms in an Ada package. Other, non-key subprogram bodies were designed and documented only in PDL within the source modules. This PDL used Ada syntax but was commented and not compiled. Some type and data declarations were coded compiled. Some structured design diagrams were constructed but not many. The burden of design documentation and analysis and refinement was performed using compiled package specifications, compiled *key routines*, and PDLed subprograms. The CASE tool was not continually available during this phase due activities involving the tool evaluation and purchasing mechanism.

During the Coding phase the subprograms already expressed in PDL were expanded to code. The coded portion of the system was integrated, tested and demonstrated.

Again, the overall project was successful but some useful methodological refinements may be suggested from observation. One such observation is that because the graphic CASE tool was not always available during the project, a graphics approach was not taken during much of the preliminary and detailed design stages. Instead, emphasis was placed very early on representing the design with coding package specifications and bodies. Much rework was involved as new alternative designs were identified, coded in Ada package specifications and bodies, reviewed, then modified. The normally constructive and necessarily iterative process of conceiving a solution, expressing it, evaluating it, and suggesting other alternatives suddenly seemed to involve too much effort and be too destructive to the participants.

One possible approach to this difficulty of rework involves exploring the design in more detail, using graphics and PDL within the CASE tool, before package specifications are coded. The tool has fairly good support for this. Balancing is checked, and creation and modification of graphics is made easy within a window—and—mouse oriented environment. The tool checks balancing and graphic relationship rules for the resulting diagrams. Then, when the Ada package specifications are coded and compiled, they are built on a foundation of previous work which has already involved consideration of many of the possible alternatives. There should be less need for generating alternatives.

Overall, the productivity of this project met that of other projects in our organization's past.

Project 3

The third project was the most recent and the most closely matched to the proposed methodology. The late—1988 project completed the coding and testing phase during the writing of this paper. It redesigned and coded two CSCs (functions) of a prototype real-time distributed ground system in Ada. Over 7000 lines of Ada code (measured by the same criteria as in the other projects) were written. Extensive use of the graphic CASE tool was made throughout the entire design effort. Again, an automated PDL processor was not used.

During the Software Requirements Analysis phase, the system was modeled in Data Flow Diagrams. During Preliminary Design, these DFDs were used to generate Objects and Operations, and Object Interrelationship Diagrams were drawn using the CASE tool. Major objects were coded as Ada package specifications, with their operations being the subprograms exported from the package specification.

During Detailed Design, Structure Charts were drawn showing the interrelationships of each objects operations in performing some component of the system's purpose. Each operation was described with Ada—based PDL within the confines of the CASE tool. Refinement was performed by editing the PDL to increase the detail, then breaking out pieces of this new detail into new software components and creating new modules for them in the structure chart. When analysis and review of the structure charts and PDL met with satisfactory results, matching Ada package specs were created. Each specification was coded to show the exported resource (mostly types and subprograms) and the procedures stubbed out. PDL prologues were placed in the Ada modules, but no PDL. The PDL remained within the CASE tool database retrievable through the structure charts.

During the Coding phase, the subprograms were written in Ada either from the PDL printed from the CASE tool, or from the same PDL cut and pasted into the modules through the window and mouse-oriented workstation environment. The design information remained available within the CASE tool database (and would be delivered that way, in a soft copy documentation scheme for deliverable software).

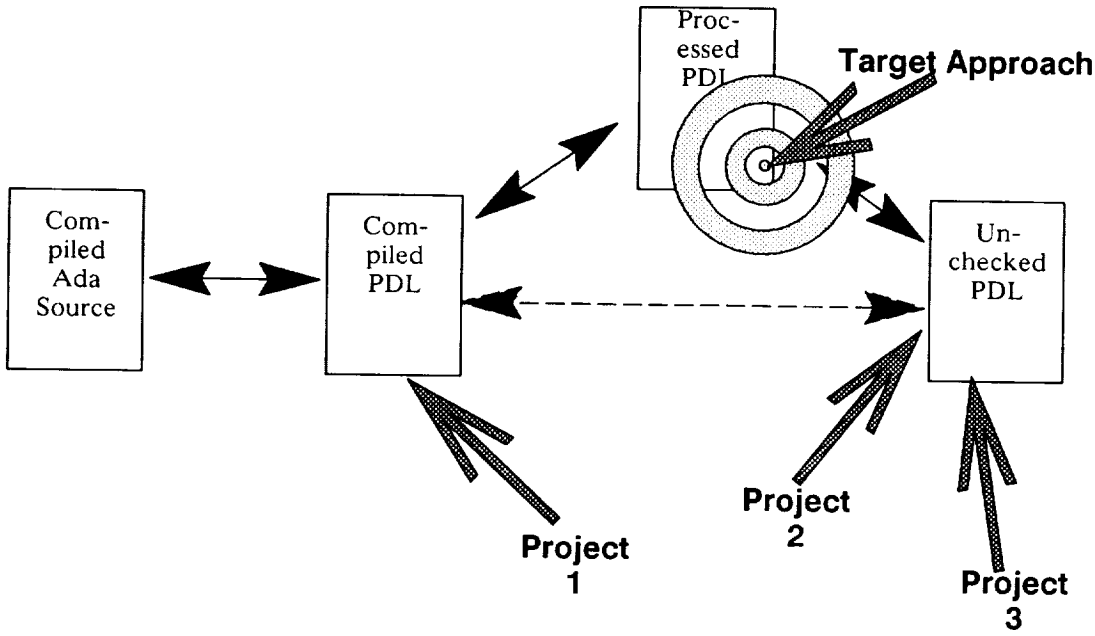
This approach seems to have paid off in a number of ways. Partitioning seems to have been so fully explored using the CASE tool that little rework of

compiled Ada package specifications was necessary. Design alternatives were efficiently analyzed within the CASE tool, where graphic and PDL information combined to give a good view of the system at several different levels of abstraction.

Module sizes were judged to be excellent: a half page maximum of PDL. Quite a few modules tested correctly when first compiled, even when coded from PDL by a first—time Ada programmer. This was attributed to the simplicity of the modules and the clarity of the PDL, which in itself might be attributed to the quality of partitioning.

The quality of the PDL seemed to be enhanced by its proximity to the graphic representation of the overall hierarchy, and the relative ease of traversal from PDL description to PDL description throughout the hierarchy. This ease of use contributed to good partitioning showing good coupling and cohesion characteristics.

The productivity on this project seems to be well ahead of that established for traditional projects (in the ball park of a 10–20% improvement for a first Ada project).



A view of PDL alternatives and our target approach
Figure 1

CONCLUSIONS AND SUGGESTIONS

Choice of Representation

One general theme in the methodology is to explore a design fully given the tool appropriate to the level of abstraction. The choice of tool should efficiently allow representation of that level of abstraction, and allow review, generation of alternatives, and easy representation of the final choice. Alternatives should be explored fully and adequately at the design stage under consideration, with the tool that does so in a most efficient (and reliable) manner.

Graphics seem to be a useful, powerful, and efficient tool for upper to middle level design. They also, with the proper tool, serve as an outstanding mechanism for indexing or gaining access to the low level of design. A graphical tree structure with a system breakdown is more easily understandible and more efficient a representation when searching for a given piece of a system than anything that we've seen before.

Quality and Testing

The alternatives and final choice of design from a phase should be subjected to *some form of testing*, that is, analysis, review, compilation, balance checking, or whatever else can be done to find as many errors as possible and to demonstrate as much quality as can be demonstrated. This provides a firmer foundation for the work that follows in development. As everyone knows, latent (un-

discovered) errors output from a phase are much more expensive to fix in later stages.

Scaling Up to Large Systems

The methodology was designed from experience in large systems—for application on large systems. The one place where scaling will change emphasis is on the choice of and number of tools. No PDL processor was used at all for any of the examined projects. This was due to the size of the projects versus the cost of procuring a tool. This approach should be re-examined for a larger projects.

On larger projects with more people it is more difficult and more important to have consistent, quality PDL. A PDL processor can contribute toward this goal. It certainly doesn't hurt to automatically check PDL for syntax and balancing errors, as long as the correction of errors does not detract from the creativity of design as sometimes happens with a strict Ada compiled PDL. No PDL processor is currently available that is integrated with the chosen CASE tool, but alternatives are being evaluated.

THE VIEWGRAPH MATERIALS

FOR THE

P. USAVAGE, JR PRESENTATION FOLLOW

A Modernized PDL Approach for Ada Software Development

Paul Usavage, Jr.
Ada Core Team

Ground Systems Department
GE/Military & Data Systems Operations

PAGE 12 INTENTIONALLY BLANK

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Agenda

Introduction

Traditional Approach

Proposed Approach

Three Study Projects:

- Methodology
- Analysis

Conclusions and Suggestions

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Introduction

The Problem:

Upgrade existing strong, large–system methodology to:

- add the benefits of new graphics–based design tools and methods
- produce DoD—STD—2167A design documentation
- incorporate the benefits of the Ada language

The Investigation:

- graphic CASE tools
- graphical methods for Structured Analysis/Structured Design
- automated document production
- high–performance workstations
- Ada–based PDL language
- compiled Ada package specifications

The *New* Problem:

Which tool *best fits* which part of the design process?

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Proposed Improvements

Requirements Analysis Phase:

- Software design based on accumulated results of Structured Analysis
- Requirements depicted using Data Flow Diagrams and other graphical representations
  - ☞ helps to understand the problem when decomposing system

Preliminary and Detailed Design Phases:

- Design represented with integrated graphics and PDL
- Edit PDL from CASE tool windows within graphic structure chart
- Interfaces represented in compiled Ada code
- Iterative refinement performed against graphics and PDL together
- Preliminary & As-Built design documents automatically produced
- Graphics high level design diagram serves as index to PDL
  - ☞ ties together small PDL descriptions into *big design picture*
- Tool maintains design database for shared use
- Innovative approach with necessary stability for large systems

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

First Study Project – Background

Project:

- 8000 Ada statements – Mathematical Simulator

Methods:

- High–level design and requirements in English prose
- Structure charts used for presentation
- Object–oriented approach
- Coded high–level interfaces as Ada package specifications
- Used compiled Ada PDL
- Compiled block constructs—loops, conditionals
- Ada comments for sequential statements

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 1 — Analysis

Limited use of graphics, mostly for presentation purposes

- Graphics constructed with PC–type drawing tool, separately from system where code was developed
- Graphics were not accessible:
  - ☞ not effective partitioning analysis tool

Partitioning analysis performed with compiled PDL

- Compiler detection of syntax errors detracted somewhat from effort spent on design representation
- Extra effort involved for some people to represent design in compiled Ada
- Made it more difficult to represent alternative designs and decide between them
- Lower level design representation could obscure higher level tradeoffs

Design alternatives took more time to work out than if quicker representation were available

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 2 – Background

Project:

- 2000 Ada statements
- Designed and implemented Ada Binding for Man–Machine Interface

Methods:

- Structured Analysis generated Data Flow Diagrams (DFDs)
- DFDs used to partition system into components
- Object–Oriented design used for system partitioning
- Ada package specifications used to show object interfaces
- Object Dependency Diagrams [Booch] used to analyze objects
- Structure Charts and Buhr Diagrams used for presentations but not for partitioning decisions
- PDL stored with Ada source code
- PDL processor not used

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 2 – Analysis

Analysis:

- High–level partitioning performed based on Data Flow Diagrams
- Implementation–level partitioning performed using PDL and compiled Ada package specifications
- Structure Charts used for presentations, not for partitioning analysis
- Partitioning with code and PDL took longer to evaluate alternatives, and more work to incorporate changes
- PDL stored with Ada source code
- CASE tool was not available to store PDL within graphic notation
- It was somewhat tedious to find a given passage of PDL if you were not intimately familiar with the design
- PDL processor not used – no automated index production

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 3 — Background

SOFTWARE DEVELOPMENT

Project:

- 7000 Ada statements
- Redesigned and implemented in Ada 2 functions of Real–Time Distributed Ground Station prototype

Methods:

- Structured Analysis generated Data Flow Diagrams
- DFDs used to partition system into components
- Object–Oriented design used for system partitioning
- Ada package specifications used to show object interfaces
- Object Dependency Diagrams [Booch] used to analyze object partitioning
- PDL stored within each process box in Structure Chart
- Structure Charts and PDL used to repeatedly analyze and refine software partitioning
- PDL processor not used

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Project 3 — Analysis

Analysis:

- High level partitioning begun using Data Flow Diagrams
- Object–Oriented approach used to go from Data Flow Diagrams to an Object—Oriented Design
- Structure Charts and PDL used to repeatedly analyze and refine software partitioning
  - ☞ Partitioning and analysis with structure charts and PDL within the same tool allowed more thorough analysis and refinement of software partitioning
- Graphics seem to be very effective for analyses of system structure

A MODERNIZED PDL APPROACH TO Ada SOFTWARE DEVELOPMENT

Conclusions and Suggestions

SOFTWARE DEVELOPMENT

Representation:

- Explore design fully with the best tool for that level of abstraction
  - ☞ Graphics are best for upper and middle level design
  - ☞ Graphical tree–shaped system breakdown for entire system (top to PDL) is most effective
- Examine partitioning alternatives, review, decide on the best approach before going on to next phase/abstraction/representation

Quality and Testing:

- All results and choices should be tested/analyzed/reviewed/checked for errors before building next layer of design

Scaling up:

- PDL processor (somewhat forgiving of syntax) is helpful to find true design errors
  - ☞ should integrate with graphics CASE tool environment