MIXING APPLES AND ORANGES
or
what is an Ada Line of Code anyway?

Donald G. Firesmith
Magnavox Electronic Systems Company
M/S 10-C-3 Dept. 566
1313 Production Road
Fort Wayne, IN 46808
(219) 429-4327
firesmit@ajpo.sei.cmu.edu

The Ada Community has long recognized that the full benefits of the Ada language can only be achieved through the application of sound, modern software engineering principles and methods. Numerous examples of this recognition include: the name of the ASEET Team (i.e., Ada Software Engineering Education and Training), CREASE (i.e., Catalogue of Resources for Education in Ada and Software Engineering), Department of Defense Directive DODD 3405.2 which states "Software engineering principles that facilitate the use of the Ada language ... shall be fully exploited...", and books such as Grady Booch's SOFTWARE ENGINEERING WITH ADA.

Although engineering is based on mathematics, it is definitely not engineering when every engineer uses his or her own private mathematics. Unless there is a standard approach to measurement, developers are reduced to talking apples and oranges -- a sure irritant to the Ada person weaned on Ada's strong typing and support for data abstraction. And as Tom DeMarco pointed out in his book, CONTROLLING SOFTWARE PROJECTS, "You can't control what you can't measure."

Yet this is exactly the current state of the Ada Community with regard to the question of "What is an Ada line of code?" There is no standard definition, and most Ada projects have developed their own unique approach.

Current major definitions of an Ada Source Line of Code include counting:

1) Physical lines (i.e., carriage returns or line feeds).
2) Non-comment, non-blank physical lines.
3) Semicolons.

   Note that this ignores many non-comment, non-blank physical lines including such important lines as:

```
separate (EXAMPLE_SUBUNIT)
function EXAMPLE(EXAMPLE_OBJECT : EXAMPLE_TYPE) return EXAMPLE_TYPE is
type EXAMPLE_RECORD is record
```

4) Terminal semicolons (i.e., all semicolons except for those in character strings and comments). This includes pragmas (LRM 2.8), declarations (LRM 3), statements (LRM 5), and formal parameters (LRM 6.1).
5) Limited terminal semicolons (i.e., all terminal semicolons except those in formal parameter lists.
6) Statements.

   Because some languages consider declarations to be non-executable statements, some people incorrectly believe that counting terminal semicolons is equivalent to counting statements.

Although all Ada statements are by definition executable (LRM 5), they are sometimes redundantly referred to as "executable statements" because of the above misunderstanding.

Some people even mix and match from the above. For example, some projects count non-comment, non-blank physical lines in specifications and terminal semicolons in bodies.

And if this were not bad enough, there is the issue of how to count generic instantiations, reused software, and modified software. Some projects count the generic unit and only one line for each generic instantiation (i.e., the single line where the instance of the generic unit is declared -- LRM 12.3); other projects count the generic unit and only the lines of the first generic instance; and still other projects count both the generic unit and all the lines of all the generic instances. With reuse, the solutions are just as diverse. Some projects count each reused unit only once; some projects count each use; and some projects count only those lines actually used (e.g., only those subprograms or declarations in an imported package actually used). Some projects make a distinction between reused code developed external to and internal to the project or company. Note that because generics and reuse should increase productivity and because productivity is most often calculated as the number of source lines of code per unit time, all generic instances and reused code should probably be counted. With modification, the choices include not counting modified units, counting only the modified lines, or counting all lines of modified units. Generics, reuse, and modification are not independent issues because total source code size may eventually decrease due to the removal of redundant code. Projects must also decide how this should affect code size and productivity.

The above definitions can produce radically different line counts. For example, the Advanced Field Artillery Tactical Data System (AFATDS) project primarily used non-comment, non-blank physical lines as its definition, but also counted physical lines and limited terminal semicolons for purposes of comparison. Generic instances were not counted, and reused software was counted only once. All lines of modified units were counted. For Release 4.04, these definitions produced the following three line counts:

1) Physical lines = 2,517,594
2) Non-comment, non-blank = 1,175,498
3) Limited terminal semicolons = 409,982.

Ease of counting often seems more important than the logic behind the counting. It is far easier to count each unit once (e.g., by counting all files in a configuration management library) than to count each unit each time it is used (or reused). Similarly, it is easier to count all lines of each unit than to count only certain relevant lines. Finally, tool support is absolutely critical, and it is easier to count using a pre-existing tool (regardless of quality or sophistication) than it is to decide on a counting strategy and develop (or acquire) the associated tool.

Each of the definitions of an Ada source line of code above depends on coding style, development method, and coding standards. While some of this dependency is unavoidable, at the very least a single coding standard and standard code formatter or pretty printer should be used PRIOR to counting.

Published figures concerning code size and productivity typically do not come with adequate documentation. For example, the AdaIC Ada Usage data base of current Ada projects leaves the definition of an Ada source line of code totally up to the projects supplying the information and does not even store the definition used. Because different definitions of an Ada source line of code produce significantly different size and productivity values, these figures are nearly meaningless. Even in those rare

cases where adequate definitions are published, the lack of a standard makes the job of comparison extremely difficult.

Although there are many good technical reasons for the various definitions, non-technical reasons are often more important. The (sometimes heated) arguments as to whether or not to count certain lines depend on the purpose of the code count (e.g., sizing, costing, or productivity). Thus, the question may not be simply "to count or not to count." For example, if costing is the driver, one would want to count both newly developed lines and reused or instantiated lines, and one would also want to count them separately.

The need for an industry standard definition of an Ada source line of code is obvious, and one WILL come. The question is whether it will come slowly as a de facto industry consensus after years of wasted effort and expense, or whether the Ada Joint Program Office will supply one (or possibly more depending on why the code is to be counted). I therefore strongly urge the AJPO to survey the Ada Board, the SEI, and relevant industry groups (e.g., the AdaJUG Ada Software Cost Estimation Working Group) for input and then publish their official position. Only a formal AJPO definition (or small set of different definitions for different purposes) will put an end to the confusion, permit the development of standardized (and sophisticated) counting tools, and provide proper input for cost and schedule models.

Meanwhile, in order to make meaningful comparisons and to calibrate one's cost and schedule models, companies should adopt the following approach:

1) Research the industry definitions of an Ada source line of code.
2) Mandate a company standard definition based on both technical and non-technical (e.g., availability of tools) grounds.
3) Acquire a company standard line counter based on this mandated definition.
4) Develop a company coding standard and standardized pretty printer for use prior to counting.
5) Collect data from all relevant projects.
6) Modify this approach as necessary.

Although this problem may be difficult to solve for both technical and non-technical reasons, we should accept as fact that the government will compare Ada projects on an Ada source line of code basis whether or not the definition is standardized. Thus, even a single poor standard would be better than none at all.