The Ada 2005 Language Design Process

S. Tucker Taft
SofCheck, Inc.

The Ada 2005 language design process was significantly different from that used by Ada 83 and Ada 95 in that it was based on an essentially volunteer committee rather than a full-time design team. Design-by-committee has a well-deserved reputation as being a sure way to create an awkward and unpleasant collection of disjointed compromises. Nevertheless, the result of the Ada 2005 process produced a language that is even better integrated and consistent than its predecessors. Producing this result depended on having a strong, shared, language design philosophy driving the design decisions. This article discusses this language design philosophy, contrasts it with the philosophy behind various other programming languages, and shows how the philosophy helped to ensure a successful, integrated, and consistent result.

Ada is now entering its third standard incarnation, currently known as Ada 2005. Its earlier incarnations were Ada 83, which was designed in the late 1970s and early 1980s by a team led by Jean Ichbiah, and Ada 95, which was designed in the early 1990s by a team led by this author. In contrast to the earlier incarnations, Ada 2005 was designed by a largely volunteer committee, led by Pascal Leroy. The only member of the committee actually on the *payroll* was Randy Brukardt, who was supported by AdaEurope and the Ada Resource Association in his official role as editor of the new standard.

The lack of a full-time design team to drive and shape the design process created misgivings among some members of the committee who felt it could impede the process. The design-by-committee process has a well-deserved reputation for producing awkward and unpleasant collections of disjointed compromises. The question was whether the Ada 2005 process could sidestep these pitfalls.

With a full-time design team and a clear team leader, the Ada 95 revision benefited from a cohesive vision that kept the design from becoming a scattered combination of ideas. With a volunteer committee, there was a danger that the need to create consensus without the hierarchy present in a design team would result in inconsistencies, that each committee member would be mollified by being given their own *pet* feature, and the language would descend into a balkanized conglomerate of sublanguages.

Ada 2005 seems to have escaped the notorious design-by-committee problems. The proposed changes seem to have brought the language into a state where it is, if anything, more integrated and more consistent. How was this accomplished? In retrospect, the key factor in achieving this desired goal has been a strong, shared, language-design philosophy driving design decisions. This kind of shared philosophy might not have been possible in earlier Ada standardization activities, as the language and the community of users were still relatively new. For the Ada 2005 process, we had a set of committee members with many years of experience both as users and implementors of Ada and a shared vision of what makes Ada powerful and productive, namely its unique combination of safety, flexibility, efficiency, and its real-time support. Our goal in Ada 2005 was to preserve and enhance these strengths while reducing any impediments to productive use.

"In retrospect, the key factor in achieving this desired goal [designing Ada 2005] has been a shared language-design philosophy driving design decisions."

The two *anchors* in the shared vision were safety and efficiency, with safety given more weight – though never absolute precedence – when there was a conflict. Ada's focus on safety is in strong contrast with certain other languages, where the attitude might be expressed as *give programmers very sharp tools and then get out of their way*, although this latter attitude sounds great for *real programmers*. In fact, even the best programmers make mistakes. Part of the Ada philosophy is that *by appropriate human engineering, you can produce a language that is in the end more productive*. The design of the language allows the compiler and the run-time to catch typical programmer errors before they become tedious debugging problems.

To illustrate how this shared philosophy interacted with the Ada 2005 design process, it is useful to study the evolution of a particular Ada 2005 feature as it moved from a perceived language problem – through the debates over the multiple ways to solve it and finally to the ultimate consensus around one particular solution.

Solving the Mutual Dependence Problem

One of the first challenges for the Ada 2005 revision was allowing for mutual dependence among types that were *not* all declared in the same package. As an example, one might have a type representing employees and another representing departments where a department record would include a pointer to the employee who is the manager, and the employee record would include a pointer to their department. In Ada 83 and Ada 95, by using an incomplete type declaration, a software engineer was able to define such mutually dependent types, but only if they were all in the same package. This limitation led to large, unwieldy packages, particularly in the context of object-oriented programming.

Although this mutual dependence problem was one of the first identified, it was one of the last problems solved during the revision process. The problem proved extraordinarily difficult to solve in a way that satisfied the various criteria inherent in our shared design philosophy. In the end, seven different approaches were considered, six of which were considered viable:

1. A new kind of incomplete type called a *separate incomplete type* whose completion is given in a separate library unit, rather than in the same library unit that contains the incomplete type declaration.
2. A variant of the *separate incomplete type* called a *type stub* where the type stub identifies the particular library unit where its completion will be found.
3. A new kind of incomplete type declaration that specifies that the completion will occur in a particular child or nested package of the package containing the incomplete type declaration.
4. A new kind of *with* clause called a *with type* clause to specify that a particular type will exist in a separate package, without requiring the package itself to be compiled prior to the referencing unit.
5. A new kind of compilation unit called a *package abstract* to contain incomplete type declarations that are to be usable (via a *with abstract* clause) as part of a mutual dependence that crosses package boundaries.
6. A new kind of *with* clause called *a limited with* clause that gives visibility on an implicitly created *limited view* of a second package, where the limited view contains incomplete versions of the (non-incomplete) types declared in the second package; *limited with* clauses are allowed to create circular dependencies between packages.

Although each of these proposals had its particular merits, only one ultimately emerged as the best when judged against all of the design criteria. One of the most important criteria was that the feature should preserve the ability to identify all of the inter-compilation unit dependencies by only looking at the name and the *context clause* of a compilation unit (the *context clause* is the set of *with* and *use* clauses that immediately precede a compilation unit). The *separate type* and *type stub* proposals lacked this. Alternative three, allowing a type to be completed in a child, also lacked an indication in the context clause that a dependence on the child existed, though it was given some credit for keeping the unannounced dependence to a unit in the same package hierarchy.

After eliminating the proposals that introduced unannounced dependencies, we were left with the *with type, package abstract,* and *limited with* proposals. The *with type* proposal was abandoned because it did not really solve the whole problem since it did not provide any visibility on an access type, and a mutual dependence between types necessarily involves an access type. Furthermore, it created a namespace with holes in it, where visibility was granted by a *with type P.T*, on a single declaration within package P, without providing visibility on the rest of the visible part of P. This was unprecedented and was inconsistent with a choice made in Ada 95 when designing *with* clauses that mention child units where the entire parent package visible part was included rather than just the named child. An important criteria in our design philosophy has been to try to make consistent choices so that the programmer's intuition about how the language works is reinforced as they learn more of the language rather than being forced to learn new rules in each corner of the language.

The *package abstract* proposal was abandoned primarily based on the criteria of simplicity of use and implementation. Adding a new kind of compilation unit, a package abstract, would be a significant disruption to all existing Ada tools. Forcing the user to decide which types of the package to include as incomplete types in the package abstract felt somewhat arbitrary, and the decision might change repeatedly as the system grew. Although this proposal was abandoned, its heritage can be seen in the simpler-to-use *limited with* proposal. Here, the implementation creates the equivalent of the package abstract implicitly, creating incomplete type definitions for *all* of the types in the original package, while relieving the user of having to perform the potentially error-prone copy-and-paste process manually. Furthermore, implementability concerns were lower because many non-compiler tools could largely ignore these implicitly created limited views.

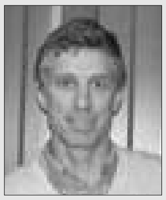
In the end, there was agreement that the *limited with* proposal was clearly superior to the other five alternatives. But the process of reaching this point was long and arduous, with many person-months of effort invested in several of the other proposals, including relatively detailed analyses of implementation effort, realistic examples of use, and vigorous debates of the pros and cons. The fact that a consensus was eventually reached depended in a large part on the shared fundamental design philosophy, both at the high level, such as simplifying the work for the user, reducing the need for arbitrary decisions, and remaining consistent with other analogous choices to the lower level such as ensuring that inter-unit dependencies are fully captured in the name and context clause of a compilation unit.

Conclusion

Although the mutual dependence problem was probably the most difficult design problem we faced, there were many other problems where a number of alternative approaches were proposed as possible solutions. In each case, we debated the alternatives vigorously, but ultimately a consensus emerged, shaped by the criteria provided by our strong, shared design philosophy. Safety, clarity, consistency, ease of use, and efficiency of implementation provided strong criteria that allowed us to select among the competing proposals with a feeling of satisfaction in the end that we had chosen a clearly superior approach rather than just settling arbitrarily for one of many equivalent alternatives.

Although it is conventional wisdom that a design-by-committee generally produces a set of compromises that leave everyone somewhat unhappy, the Ada 2005 design process left its design committee with an unusual level of satisfaction and sense of accomplishment. It seems clear that this outcome was largely a result of the long history behind the committee. This history enabled us to debate vigorously, but we all feel very good about the final result. We had been able to tie our decisions to criteria that we all shared and which we agreed were the key to the unique safety and productivity of the Ada language.◆

About the Author



S. Tucker Taft is the founder of SofCheck, Inc., which develops tools for automating software quality improvement. From 1990 to 1995, Taft served as the lead designer of the Ada 95 programming language. In 2000/2001, he led the development of the J2EE-based <Mass.gov> portal for the Commonwealth of Massachusetts. Since 2001, Taft has been a member of the International Organization for Standardization Rapporteur Group developing Ada 2005.

11 Cypress DR
Burlington, MA 01803
Phone: (781) 856-3344
Fax: (781) 750-8064
E-mail: tucker.taft@sofcheck.com