A comparison of industrial coding rules

J-P. Rosen
Adalog, 19-21 rue du 8 mai 1945, 94110 ARCUEIL, France; email: rosen@adalog.fr

Abstract

AdaControl [1] is a (free) tool whose purpose is to enforce coding standards and programming rules in Ada programs. As AdaControl is more and more widely used in the industry, we had to review many industrial coding standards, in order to write the corresponding AdaControl rules.

This paper presents our experience with rules of various origins, analyzes the rules commonly encountered, and provides some lessons-learned about good and bad programming rules.

Introduction

With the raising of the use of its AdaControl tool, Adalog has developed a growing activity in consulting and services related to the checking of programming rules. This includes helping QA people to define rules, improving AdaControl to support new rules, and performing code reviews (both automatically and manually).

This activity has lead us to reviewing coding standards from many origins, but mainly from safety critical domains: air-traffic management, avionics, railway control… One could think that rules from these domains should be, more or less, the same. If there is effectively a core of generally accepted rules, there are also differences, for good and sometimes bad reasons. In this paper, we first present a classification of commonly encountered rules, then we discuss the importance of automatically checking the rules, and finally present some lessons learned.

Classification of rules

This "classification" is not intended as a formal taxonomy, but rather as an experimental categorization of the programming rules, intended to show the strengths, but also the difficulties and sometimes the weaknesses of many rules.

General useful rules

Some rules are of general interest, have clearly only benefits. and are therefore commonly found. For example, most projects require "only one statement/declaration per line", "no single array declarations", "unit name must be repeated after **end**"…

As another example, a simple and common rule is to require that every use of an identifier uses the same casing as in its declaration.

Some rules are very useful but extremely difficult to enforce by manual inspection. For example, the "no local hiding" rules forbids a local name from hiding an identical name in an outer scope; it prevents confusion of variables that depend on visibility rules.

Many projects do not use certain features of the language, like tasking or tagged types. This results in general from a design decision, made at the very beginning of the project. It is then a good practice to explicitly forbid the use of the corresponding language features.

The rule that prevents use of the `'Address` attribute is also commonly found, and is an important one, but for a special reason. Although there are very legitimate uses of addresses, experience shows that very often, use of `'Address` results from insufficient knowledge of the possibilities of Ada by people who come from other languages with insufficient training. The goal of this rule is thus not to prevent all usage of `'Address`, but to make sure that any use of it is justified and pair-reviewed.

Trivial rules

Some commonly found rules are of minimal value, simply because they are always obeyed in practice. We call these rules "trivial" because they might well be the only rules that we never found violated in any project we had to review!

For example, almost every coding standard forbids using the **goto** statement. Although the reasons are obvious, it is, in practice, extremely rare to find violations.

Another example is a rule that forbids declaring identifiers with the same names as entities defined in Standard. Of course, violating this rule could cause horrible confusion, but in practice, few programmers even *know* that they are allowed to declare identifiers that hide the ones from Standard!

It is also common to have a "rule" that forbids the use of TAB characters in programs. Although there are of course good reasons for it, it is hardly a rule; most editors have features to eliminate tabs, so they go away without the programmer being even aware of it. And otherwise, it is very easy to write a simple clean-up program.

Redundant rules

It is very common to find rules that repeat other rules, in a slightly different way, because they appear in a different context or were defined for a different purpose.

For example, a rule may explicitly require that, when assigning fields of records, there be only one field assignment per line. This rule is obviously redundant with the more general "one statement per line" rule. Another example is a general rule that states that "a package spec should export only entities that are used by other units",

and then have a rule that states that "if a type is declared in a package specification and used only in the body, it shall be moved to the body".

Such redundancies are annoying, because they are useless and increase artificially the number of rules. Moreover, a violation can (must?) be traced to several rules, thus making reporting more difficult.

Layout and comments rules

Some guidelines go into deep details about the number of characters that should be used for indentation, maximum length of a line and how long lines should be folded, how aggregates should be aligned, etc. A uniform presentation is an important issue as far as understandability and uniformity are concerned, however checking these rules manually is almost impossible, and writing a tool to check them automatically is roughly equivalent to writing the corresponding reformatter. It is therefore better to require the use of a reformatter (which is now included in every syntactic editor) and go with whatever layout the reformatter does, than to require a presentation that does not correspond to any tool. Uniformity is important, exact details of layout are not.

Various rules deal with comments. The easiest ones are those that require a standard header for every compilation unit. Automatic checking shows that this kind of rule is harder to enforce than one may think. Although the headers *look* conformant, there are very often small differences, like extra comment lines, missing separators, incorrect number of spaces at various places…

Header comments of subprograms are more difficult to check, since they are expected to describe the purpose of the subprogram and the semantics of the parameters – something that can be checked only manually.

Sometimes, there is a requirement that certain declarations (types, variables) be commented. Once again, a manual check is required for this kind of rule, but systematic checking requires inspecting *all* the code – something that cannot be performed routinely. There is therefore a high risk that such a rule stays as "recommended practice" without systematic checking.

Finally, some projects require a density of comments in the code (like "there must be 20% of comment lines"). In one project, the rule document failed to define how the lines are counted, which raises a number of issues: are blank lines counted? Are header comments counted?

Rules that are not coding/programming rules

Many guides include rules that are more *design* or *good-practice* rules than coding/programming rules. For example, a rule that requires that "different types shall be used to represent data from different domains". Although such rules have value, they should be kept separate from programming rules, because they cannot generally be verified automatically. Typically, they should be checked by pair-review, rather than by code inspection.

Controversial rules

Some rules are controversial, in the sense that various projects take opposite decisions., either about whether to allow some constructs, or in the way the rule should be applied. Note that this is not surprising: a life-critical project may impose rules that ensure maximum safety, even at the cost of readability and maintainability, while a less critical application may choose different trade-offs.

For example, almost every project imposes naming conventions for various elements. But some projects impose separating words in an identifier by the use of capitalization and forbid underscores (like in `LineLength`), while others prohibit that style, and require words to be separated by underscores (like in `Line_Length`). Some projects require type names to start with "`T_`", or end with "`_Type`". Renamings are an interesting issue, as far as naming convention is concerned: should renamings have their own naming convention to show that they are aliases, or should they follow the rule for the renamed entity?

Using the use clause is another controversial issue: some projects disallow it altogether, other allow it only if restricted to the innermost scope where it is useful, and some place no restriction to it.

Some rules require systematic initialization of all variables at the point of declaration. Although it may seem useful to make sure that every variable receives a proper value before being used, this is an interesting case of a rule that may have adverse effects. The rule may induce people into assigning a "default" value to variables (that may not be appropriate) just to pass the check; this may in turn result in more subtle bugs than those caused by a plain non-initialized variable. For this reason, some rules forbid systematic initialization (especially when the initialization value is known to be overridden later on).

Insufficient rules

Some rules are intended for a certain purpose, but if they are not properly formulated and/or explained, they can fail to achieve their intended goal. For example, it is common to disallow the use of predefined numeric types. This is intended to promote the definition of higher level, more abstract numeric types. However, in a project, this resulted in the definition of types like "Int_8", "Int_16", and "Int_32" that were used everywhere. There was some benefit to it, as it made the program independent of the size of the predefined integer types, but did not bring the benefits expected from strong typing of numeric values.

Often, the rule does not assert all the consequences. For example, there can be a rule that says "no package shall be declared in a procedure". Such a rule is generally intended to limit the complexity of subprograms, but does it also apply to instantiations of generic packages? They are formally local packages, but the rule would prevent, for example, instantiating Integer_IO inside of an IO routine – a very legitimate construct actually.

Sometimes, rules are written with a very narrow perspective. We encountered a rule that said that "when an

array is assigned in full, all components of the aggregates should be named". But of course, assigning an array in full does not necessarily use an aggregate; and what about aggregates that appear in a context other than as the right hand side of an assignment? Should the rule apply to record aggregates? Clearly, the person who wrote the rule had used aggregates only in very limited contexts, and wrote the rule according to that usage.

Inappropriate rules

Sometimes, rules are clearly a legacy from other languages, or simply show ignorance about Ada. For example, a project required an order for declarations: constants, then types, then variables (and failed to define an order for Ada entities that had no Pascal equivalent, like packages and exceptions!). This was clearly a remaining from the Pascal philosophy, but prevented for example the grouping of declarations that were logically related.

In another case, a rule required the presence of an "else" part for every "if", leading to many "else null;" in the program. This rule was derived from Misra-C, where it is intended to prevent the "dangling **else**" problem in C. The Ada syntax (which requires "**end if**") does not have this problem, but the rule was reconducted anyway.

Another (funny) example is "rules" that forbid constructs that are actually not legal Ada; we have encountered a project that banned the use of anonymous array types as record components, or default initialization of array components … Such rules are harmless by themselves, but create suspicion about the validity of other rules.

A special kind of dangerous rules are those that are justified by efficiency considerations. Rules sometimes require or forbid the use of some constructs for efficiency reasons. Although this may seem justified in time-constrained software, experience shows that actual measures of the run-time cost of such structures have only very rarely been performed; often, the rule just expresses the "intimate belief" of those who wrote the rules, without the backing of hard figures. Very often, these rules are not justified at all, and may even force using less efficient constructs. Even when such rules are justified, it must be remembered that "inefficient" constructs may become very efficient with the next version of the compiler.

A special (and even worse) case of the above is rules that are intended to work around compiler bugs. Such rules tend to stay forever, years after the bug has been fixed…

Note that it is often the *motivation* of the rule which is wrong, not the rule by itself. For example, a project required short circuit forms (**and then** and **or else**) rather than plain **and** or **or**, on the ground that they were more efficient. Such a general statement is highly likely to be plain wrong – at least in some cases, and the gain in micro-efficiency does not justify the rule. On the other hand, another project had the same rule, but on the ground that it would simplify unit testing, because each logical operation would require only three tests instead of four with the regular operators. This reason was perfectly acceptable.

Good rules that are harder to enforce than they seem

Some rules are apparently well motivated, but very hard to apply in practice, or (almost) impossible to check. For example, several projects wanted to prevent the use of "magic numbers", i.e. numerical values that appear directly in the program text; instead, every such value should be given a name, as a constant or named number. Obviously, this rule cannot apply to literals used precisely in the definition of constants and named numbers. But there are many other cases where numeric literals cannot be avoided, like in representation clauses for example. And in X**2, it would be stupid to forbid the use of "2"… If taken too literally, this rule would force people to declare constants like Number_2, which would bring no benefit at all.

It is also common to find rules that prevent assignment to fields of records, in favour of whole assignments with aggregates. This is an important rule for maintainability, since the addition of a component to a record will result in illegal code everywhere the corresponding modification has been omitted. But sometimes, you just want to assign a value to one component: should you force a full aggregate assignment in this case? Let us assume for a start that an aggregate is required if every component is changed, and that single assignment to a component is allowed if no other component is changed. Where should the limit when aggregate assignment is required be placed? If more than XX components are changed? If less than YY components are not changed? If more than ZZ% of the components are affected? Making a rule which achieves the desired goal and is still practical is far from obvious.

Rules not checkable by nature

Finally, some rules are, by nature, impossible to enforce automatically, generally because they involve some value judgement. This includes rules like "parentheses should be used to improve readability", "elements should be grouped in a package according to the logical structure", and of course "identifiers should have meaningful names".

The checking of this kind of rule must be done manually. In some cases, a tool can be of help by identifying automatically the constructs that must be reviewed manually; in other cases, checking the rule requires a detailed reading of the whole source.

Actually, this kind of "rule" should really be guidelines, and separated from the true coding rules.

The value of a tool for checking rules

In the previous chapter, we repeatedly addressed the issue of the checkability of the rules. It is nice to issue rules, but a rule is meant to be enforced; counting on programmers' discipline simply does not work.

It must therefore be stressed that rules are of little value, unless there is a tool to enforce them. No manual inspection can approach the level of scrutiny provided by a tool; actually, *all* of our clients were greatly surprised when we ran AdaControl on their carefully reviewed code,

sometimes finding *thousands* of violations that had escaped manual inspection.

Moreover, manual inspection is a lengthy and costly process. It can be performed once for every major release of the product, for example at the time of formal certification for safety-critical software[1], but can certainly not be done routinely.

There are several such tools on the market: in addition to Adalog's AdaControl, popular tools include AdaCore's Gnatcheck, GrammaTech's Ada-Assured, LDRA's Testbed, Logiscope's Rulechecker, and RainCode's Adarc. Moreover, many compilers include options to enforce coding rules at compile time. Some rules can even be enforced by the language with the use of **pragma** restriction.

An important issue when choosing a tool is ease of use in day-to-day development. When rules checking is performed late in the development process, one discovers generally a huge amount of violations, and fixing them requires a tremendous effort; it is sometimes extremely difficult to do when the software has already gone through various validation phases that would be ruined by massive corrections. When the tool is integrated into the development environment, programmers can run it routinely each time they develop new modules or modify existing ones, ideally by simply clicking a button in their favourite IDE. The sooner checking is performed in the development process, the better.

From this point of view, it could seem useful to have rules checked directly by the compiler. But compilers do not have such sophisticated and parameterizable rules like dedicated tools have. Unlike language rules, programming rules depend heavily on the kind and constraints of the project; parameterization is therefore absolutely necessary. Moreover, rules checking must also be performed by quality assurance people, at the time of integration. Having some rules checked by the compiler while other still require the use of another tool would force QA people to run two tools as part of the process, with different outputs that are hard to merge. Therefore, even if the compiler does some checks, it is important that the rule checking tool be able to enforce also rules checked by the compiler.

Lessons learned

How to define "good rules"

Providing a good set of programming rules is not easy. Sometimes, it seems that rules are there just for the sake of having rules; occasionally, rules may have an effect opposite to their intent.

It is therefore important that every rule be motivated and justified. Some of the questions that need be answered to check the value of a rule are:

- What is the problem that this rule will prevent/minimize?
- Is this rule really necessary?
- What are the possible adverse or perverse effects of the rule?
- Is this rule automatically checkable?
- What are the cases where the rule should *not* be obeyed?

Of course, it does not make sense to reinvent the wheel every time. A programming rules document should start from some existing and recognized document, like the famous "Ada Quality and Style Guide"[2], which is actually a generic template intended precisely to serve as the basis for coding standards. It was surprising that, among the documents we reviewed, many of them didn't even quote the Ada Q&S Guide, although they often referred to coding standards from other languages… Another valuable source of inspiration is the NASA coding standard for the Goddard Dynamic Simulator, which is freely available on the internet [3].

Coding rules should really be coding rules. They should be defined separately from design rules, and also from guidelines, which are common sense recommendations that cannot be specified – and even less checked – formally.

Rules should be proposed by QA people, but should be reviewed and discussed with programmers and language experts. Otherwise, there is a risk that the cost of a rule, even a perfectly reasonable one, be higher than its benefits, for reasons linked to the technical details of the project.

It should be also understood that developing a good set of rules is an iterative process; experience shows that some rules are useless, some have an adverse effect, and some are missing. There should be a process for getting feedback from the developers and improving the rules document.

Derogations

When a rule is proposed, it is very important to be aware that there *will* be cases where the rule should *not* be obeyed. Derogations to a rule are normal; however, derogations should only be granted by QA, after review and justification.

Failing to recognize the need for derogations can lead to two equally bad effects:

- Either force application of the rule in any case, often resulting in twisting the code to match the rule with a very poor result as far as quality is concerned
- Or simply abandon the rule, on the ground that it cannot *always* be applied.

[1] but at that time, it is generally too late to correct massive violations, and the project ends up with a document to justify why the violations are not safety-critical, rather than fixing them.

Therefore, every coding standard should include a process for requesting a derogation, and tools should provide a way to ignore violations at indicated places. The process for granting a derogation when appropriate should not be too heavy; otherwise, it may appear simpler to the programmer to obey by the rule, even where not appropriate, rather than to request a derogation.

Form of the document

The coding rules document should ideally specify, for each rule:

- The statement of the rule
- The motivation for the rule
- An example where the rule is obeyed
- An example where the rule is not obeyed
- Cases where the rule is *not* applicable
- Whether and how the rule can be checked by automatic tools

The goal of this is to make sure that the programmers understand the rule, understand and accept the motivation of the rule, know how to check it, and know how to ask for a justified derogation.

Since such a document can become rapidly quite thick, having a quick summary of the rules with pointers to the full explanation can make the document much more usable.

Communication

Coding standard should be perceived by programmers as a help rather than a burden. It is of course important to have clear and easily accessible documents to describe the rules, but organizing team meetings, where the rules are presented and their motivations explained, can be very effective. Such general presentations bring several benefit:

- they provide feed-back from the base to the QA people, often resulting in improvements to the rules;
- they make acceptance of the rules easier; people have no problem following rules when they understand their purpose
- with sufficient tool support, it will help making the checking of the rule a routine, therefore catching violations early in the development process and avoiding massive rewritings.

Conclusion

A good set of programming rules is one which really contributes to the quality of the code without putting unnecessary burden on the programmer, is precisely defined and well understood by all users, and easily enforceable by automated tools. Defining such a set of is far from easy: some rules are general, but others depend on the particular context of the application.

It must be acknowledged that programming rules have to be refined iteratively, and that good communications between QA people and users is a key to achieving a set of rules that really improves the quality of the project.

References

[1] http://www.adalog.fr/adacontrol2.htm

[2] Software Productivity Consortium, "Ada 95 Quality and Style Guide".

[3] Stephen Leake, "Goddard Dynamic Simulator, Ada Coding Standard", http://fsw.gsfc.nasa.gov/gds/code_standards_ada.pdf.