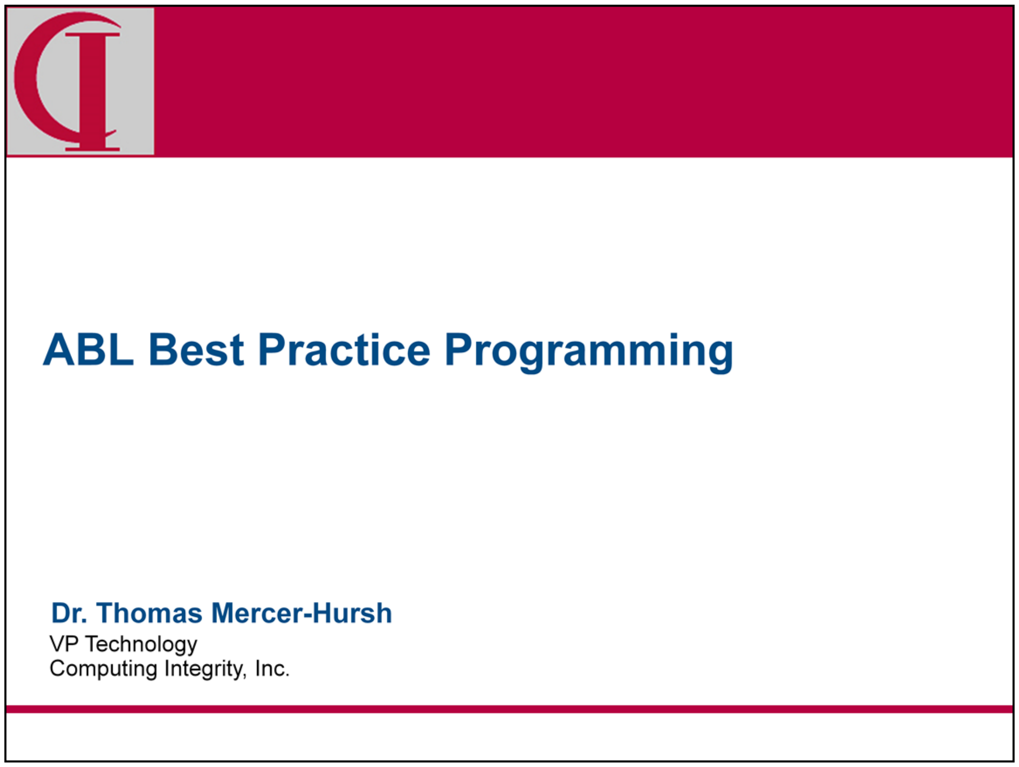
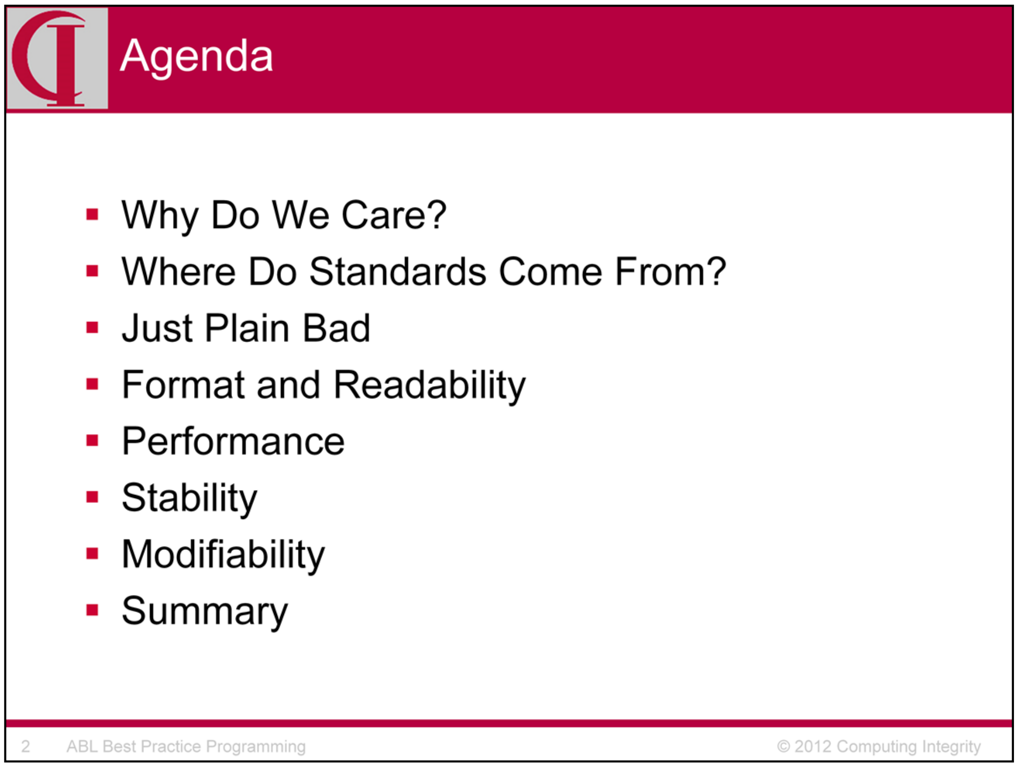


ABL Best Practice Programming

Dr. Thomas Mercer-Hursh
VP Technology
Computing Integrity, Inc.

Let me begin by introducing myself. I have been a Progress Application Partner since 1986 and for many years I was the architect and chief developer for our ERP application. In recent years, I have refocused on the problems of transforming and modernizing legacy ABL applications. I have been personally responsible as architect, tool maker, and programmer for producing on the order of 2 million lines of ABL code.

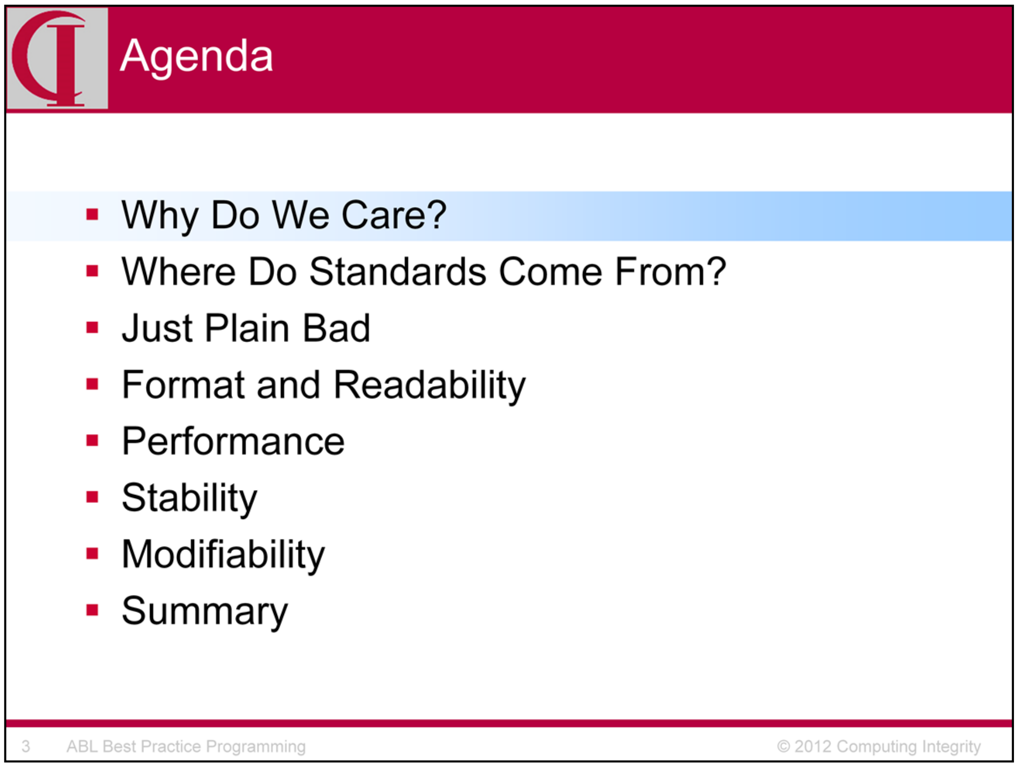


Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

2 ABL Best Practice Programming © 2012 Computing Integrity

Here is our agenda for the day. We are going to first talk about a little background and then review a number of best practice standards in a number of categories.

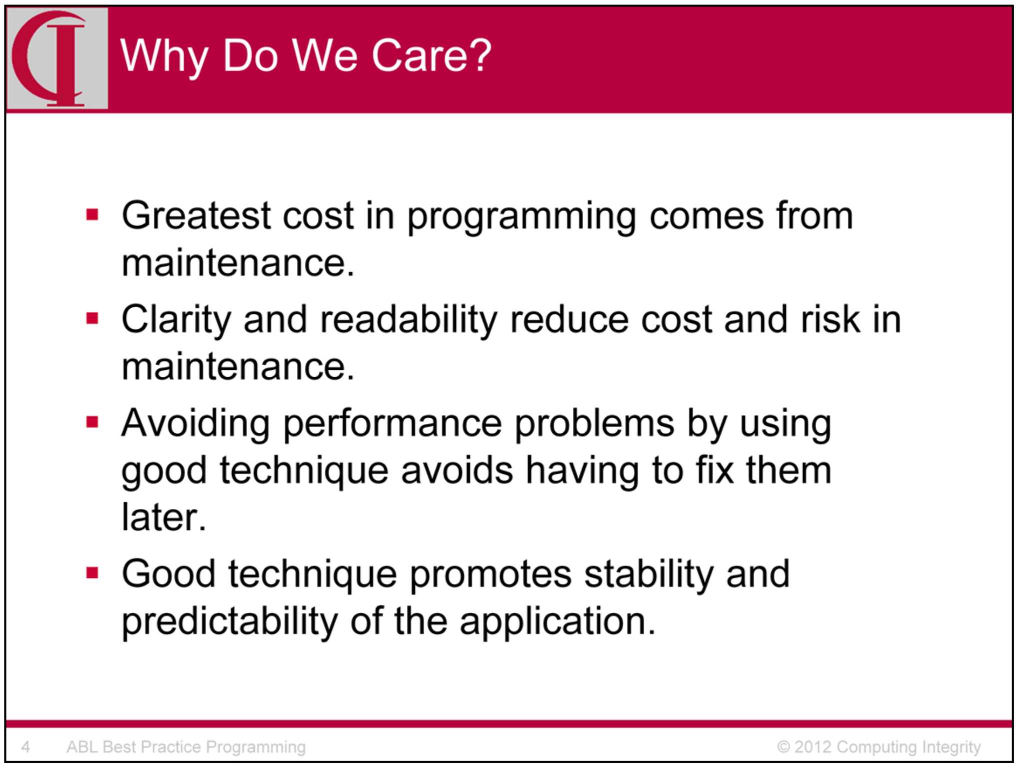


Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

3 ABL Best Practice Programming © 2012 Computing Integrity

First, let’s ask ourselves why we even care about best practice standards.

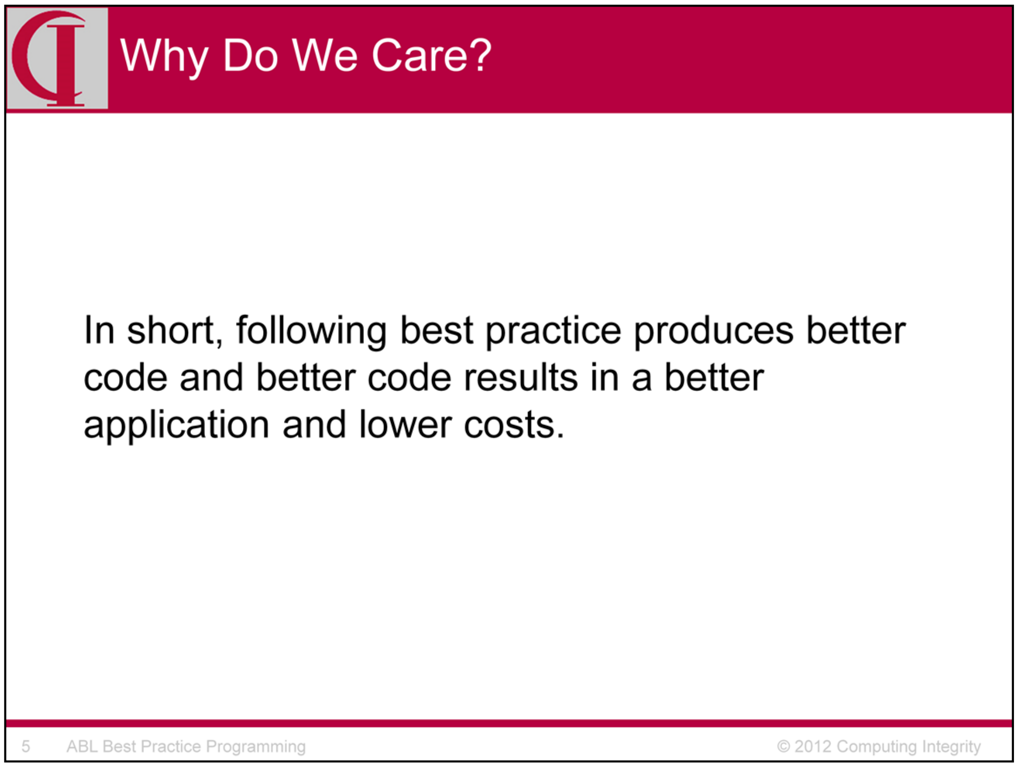


Why Do We Care?

- Greatest cost in programming comes from maintenance.
- Clarity and readability reduce cost and risk in maintenance.
- Avoiding performance problems by using good technique avoids having to fix them later.
- Good technique promotes stability and predictability of the application.

4 ABL Best Practice Programming © 2012 Computing Integrity

If one wrote code and never looked at it again, one might not care what the code looked like … although I would still argue that many of the standards we will talk about today also help in getting the code to work right in the first place. But, the ugly truth is that the cost of developing software in the first place, especially business software, is only a small fraction of the total cost of ownership of that software over its life. Business needs keep changing … and at an ever increasing pace, as Progress keeps reminding us. Small data sets become big data sets become huge data sets. So, anything that helps reduce the cost of maintenance is going to pay significant dividends. A key element is how easily one can find the part that needs changing, and then understand how that part works, so one is able to see how the code has to change. Another key element to reducing costs of modification lies in avoiding future changes, e.g., for performance problems, because one created the right code in the first place. And, of course, stable code which does what it is supposed to only has to be changed when requirements change, but buggy software needs changing all the time.

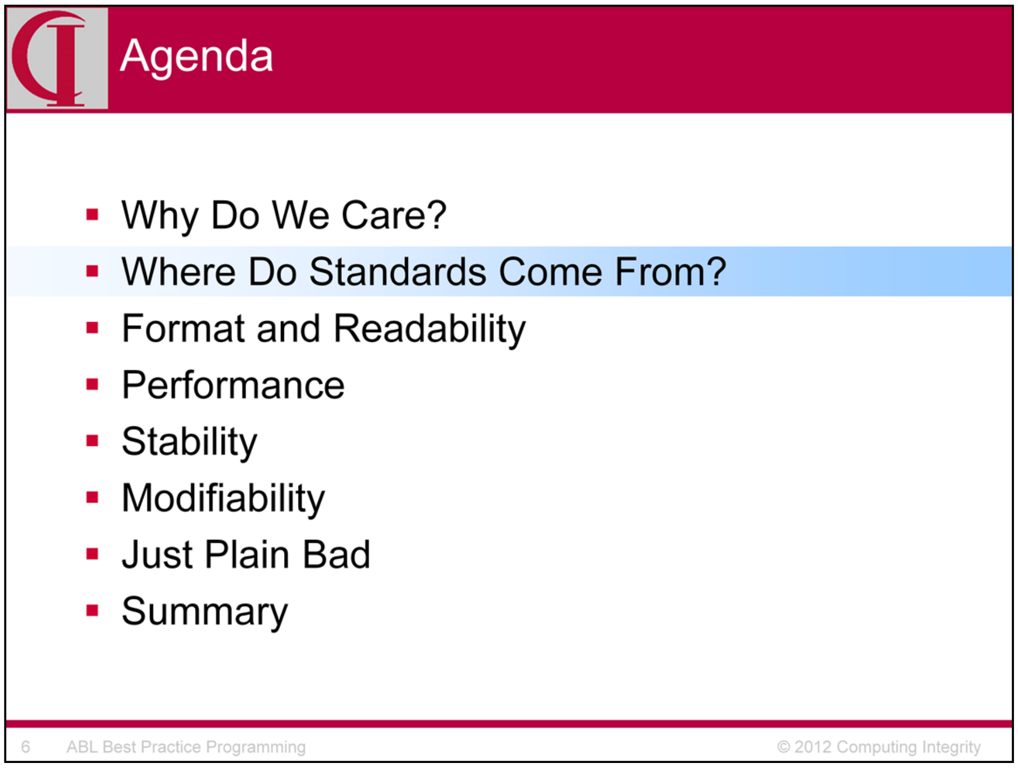


Why Do We Care?

In short, following best practice produces better code and better code results in a better application and lower costs.

5 ABL Best Practice Programming © 2012 Computing Integrity

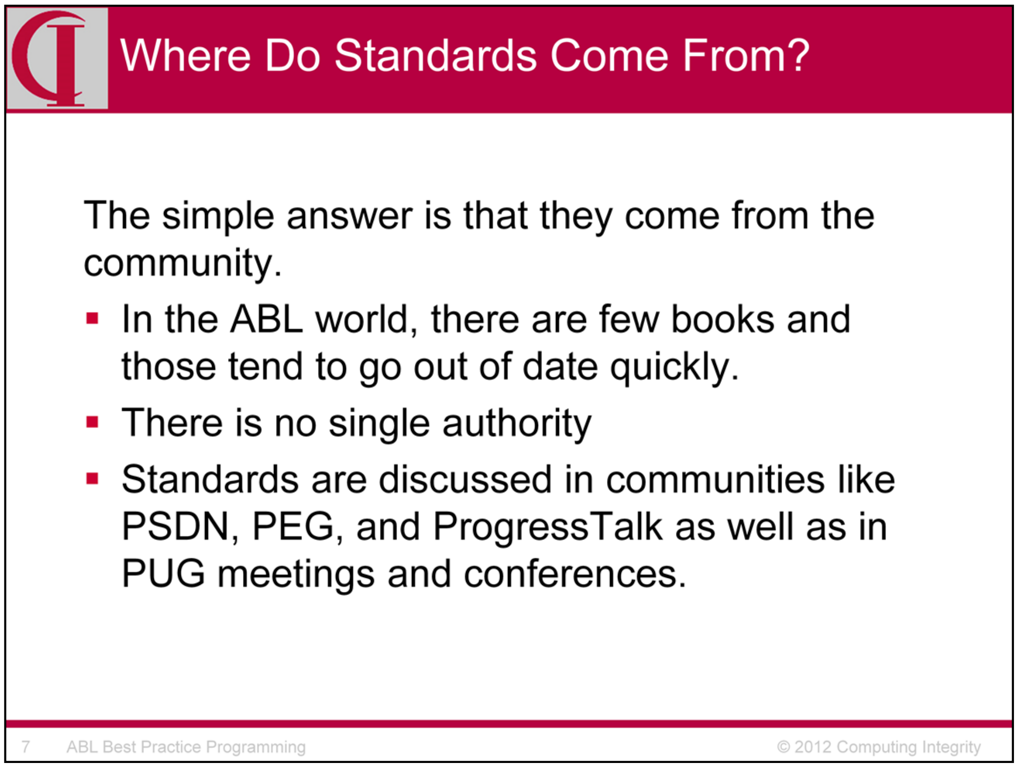
In short, following best practice produces better code and better code results in a better application and lower costs.



Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Format and Readability
- Performance
- Stability
- Modifiability
- Just Plain Bad
- Summary

Next, let’s consider briefly where best practice standards come from.

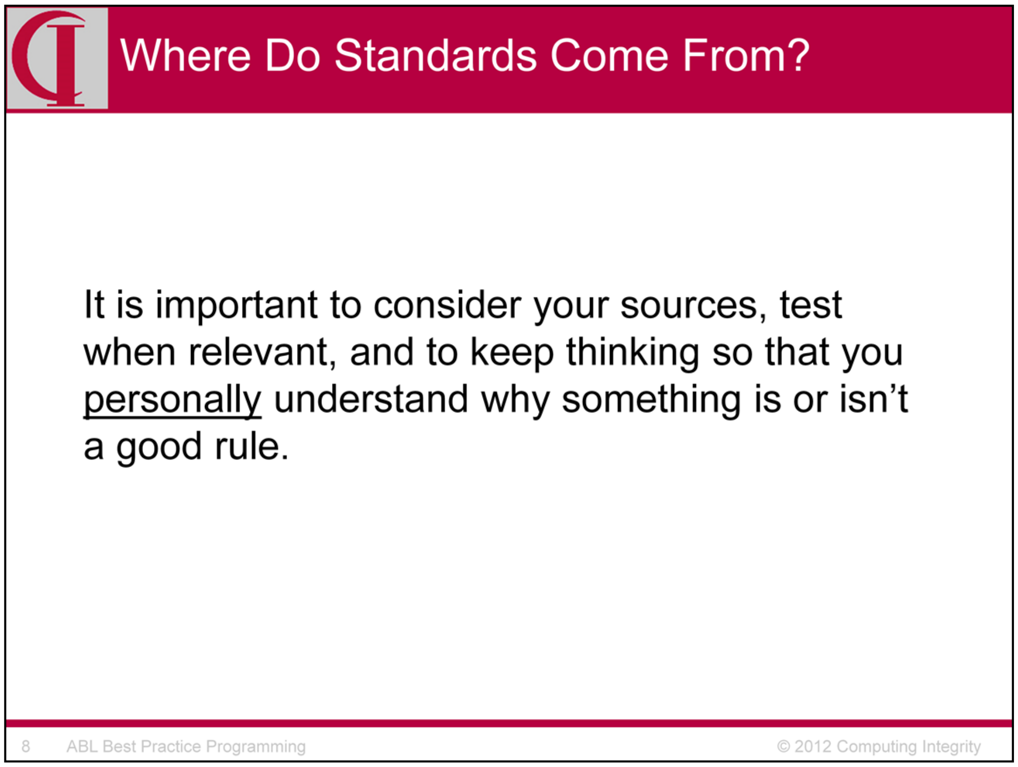


Where Do Standards Come From?

The simple answer is that they come from the community.

- In the ABL world, there are few books and those tend to go out of date quickly.
- There is no single authority
- Standards are discussed in communities like PSDN, PEG, and ProgressTalk as well as in PUG meetings and conferences.

The simple answer is that they arise from the community. In other languages, we might have books and websites to point to as standards, but in ABL those are few and often go out of date. There is no one authority we can safely look to, not even PSC since their documentation is filled with examples that many would say do not illustrate best practice. Standards are talked about in the on-line communities, often in reaction to their being violated.

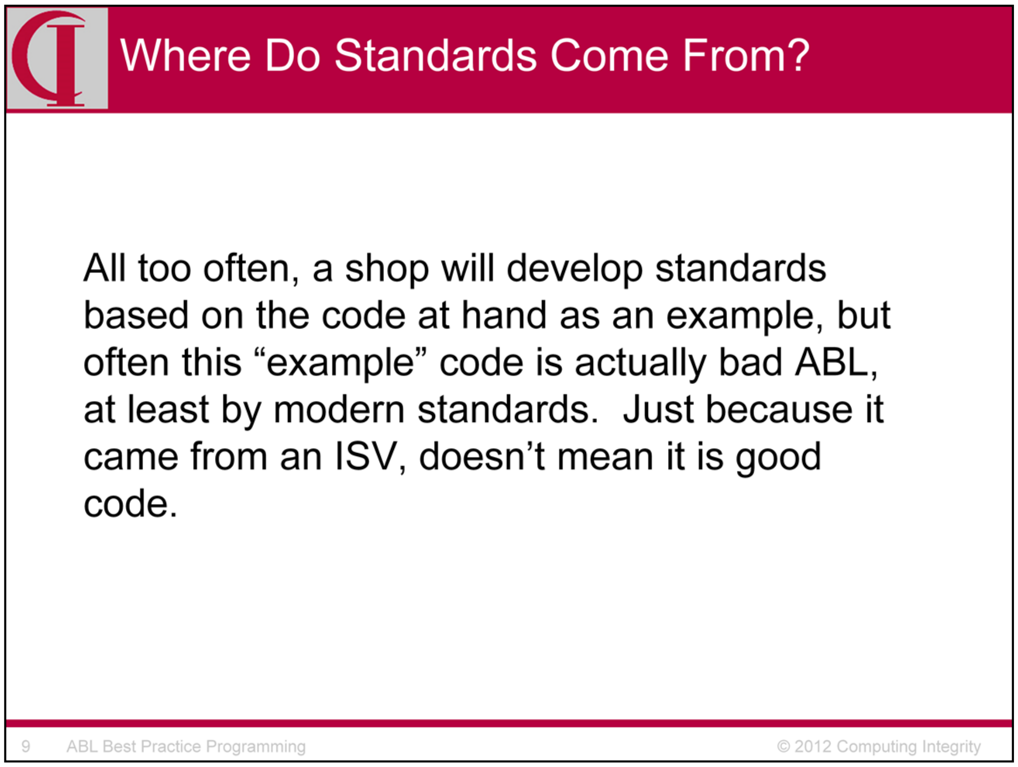


Where Do Standards Come From?

It is important to consider your sources, test when relevant, and to keep thinking so that you personally understand why something is or isn't a good rule.

8 ABL Best Practice Programming © 2012 Computing Integrity

One has to learn what sources to trust and which to question when looking for standards. The best guideline is to not unquestionably accept a standard from any source, but to think about why the standard has been advanced and what impact it will have. When you understand the reasons, then you have the basis to make your own judgments. Of course, if the standard comes from your boss, you may have to follow it regardless, but that shouldn't keep you from making your own judgments.

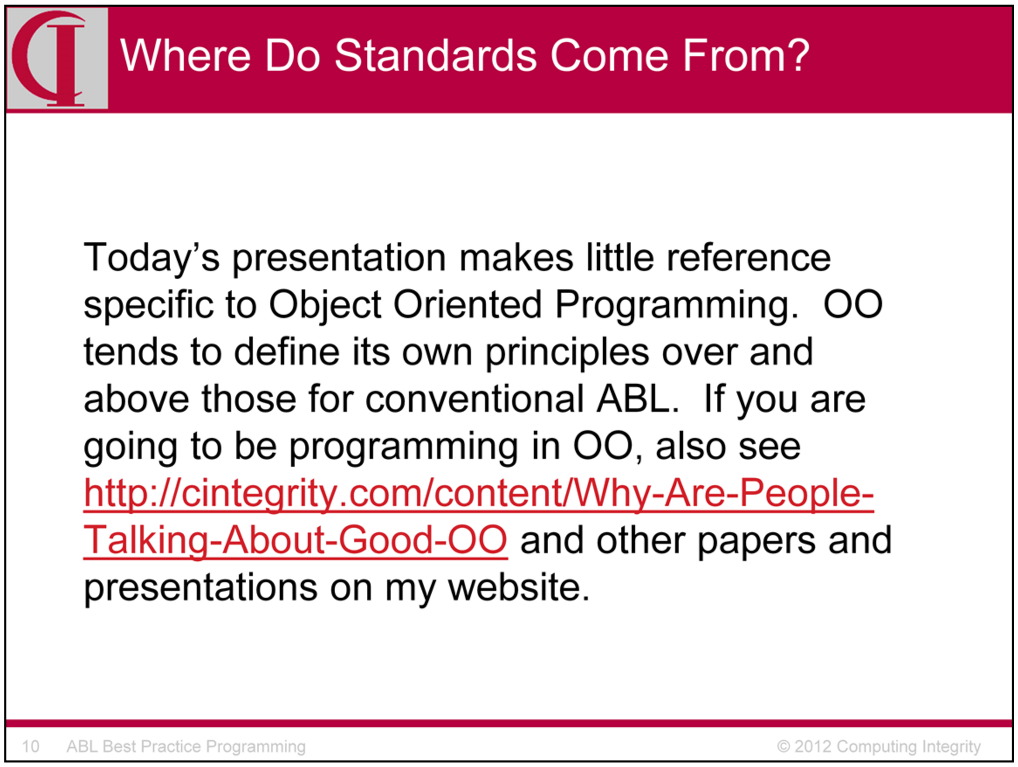


Where Do Standards Come From?

All too often, a shop will develop standards based on the code at hand as an example, but often this “example” code is actually bad ABL, at least by modern standards. Just because it came from an ISV, doesn’t mean it is good code.

9 ABL Best Practice Programming © 2012 Computing Integrity

Many shops adopt the style and techniques of the code which they have in their own shop already as their own best practice standards.   Often, the rationale is that much of the code came from an ISV and someone selling their software must know more about Progress than the locals do. This is often a bad mistake since there is a lot of code from ISVs that reflects 20 year old standards, all too often standards that weren’t even very good 20 years ago, and they are poor models for modern code.

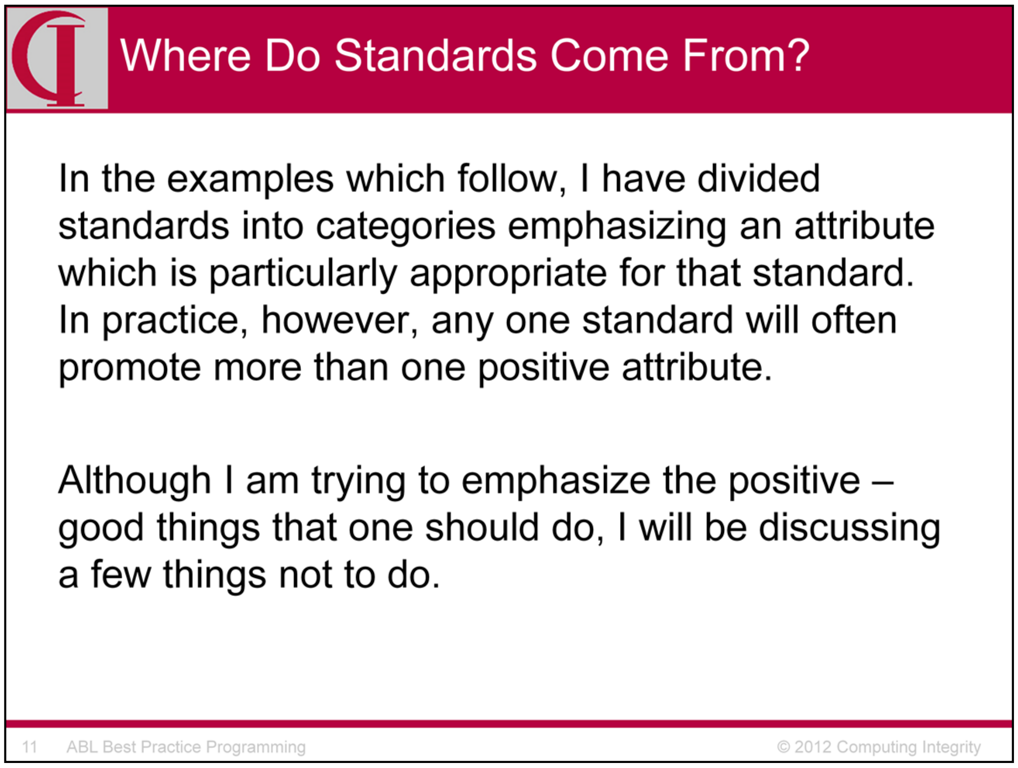


Where Do Standards Come From?

Today's presentation makes little reference specific to Object Oriented Programming. OO tends to define its own principles over and above those for conventional ABL. If you are going to be programming in OO, also see http://cintegrity.com/content/Why-Are-People-Talking-About-Good-OO and other papers and presentations on my website.

10 ABL Best Practice Programming © 2012 Computing Integrity

For those who have been used to my rôle in recent years as an evangelist for object oriented programming, I will note that I am going to make little reference to OO today. Many of the principles here also apply to OO, but OO also tends to define its own principles which I have discussed elsewhere.


Where Do Standards Come From?

In the examples which follow, I have divided standards into categories emphasizing an attribute which is particularly appropriate for that standard. In practice, however, any one standard will often promote more than one positive attribute.

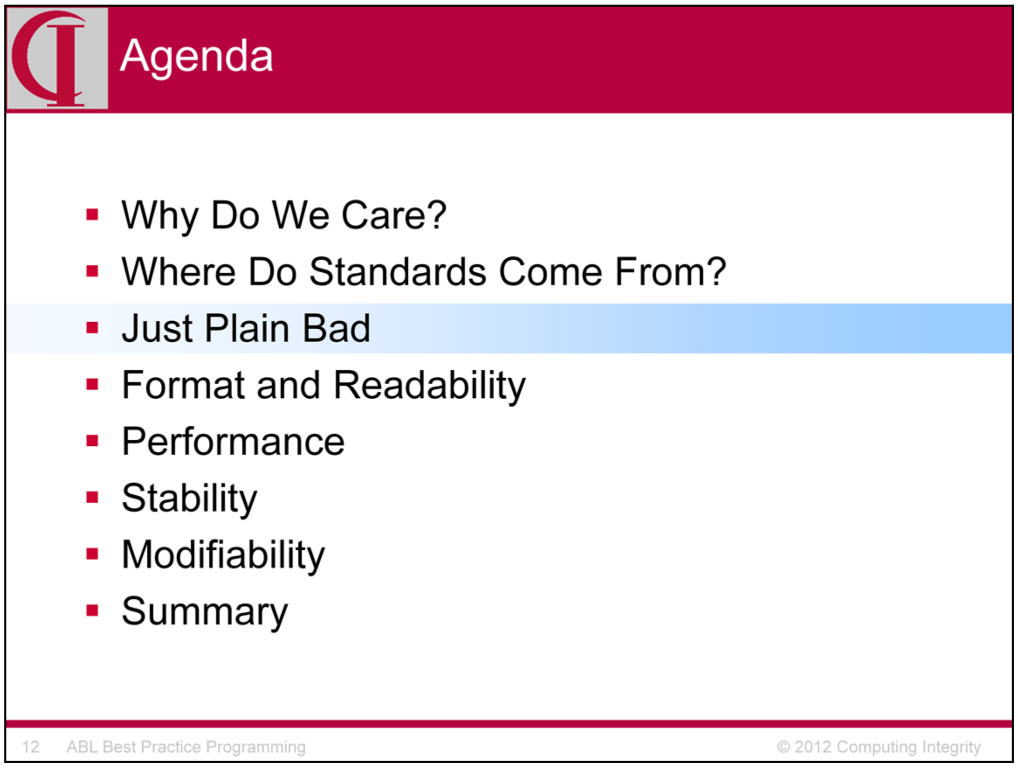
Although I am trying to emphasize the positive – good things that one should do, I will be discussing a few things not to do.

11 ABL Best Practice Programming © 2012 Computing Integrity

In the review of standards which follows, I have divided standards into categories based on what seemed like their principle impact, but many, perhaps most standards often have more than one impact or benefit.

While the emphasis is on what one should do, one can't discuss this area without also saying what not to do.

The list I present today is necessarily incomplete, but hopefully it will get you thinking and make it easier for you to make your own judgments.

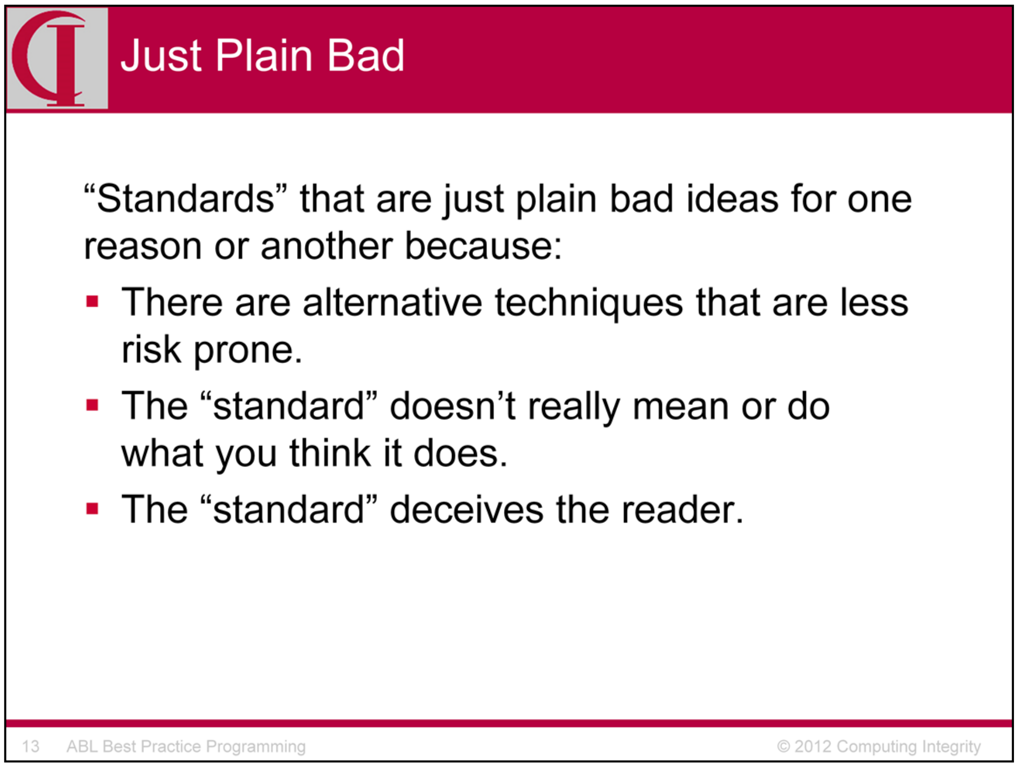


Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

12 ABL Best Practice Programming © 2012 Computing Integrity

First, let’s discuss a couple just plain bad anti-standards.



Just Plain Bad

"Standards" that are just plain bad ideas for one reason or another because:

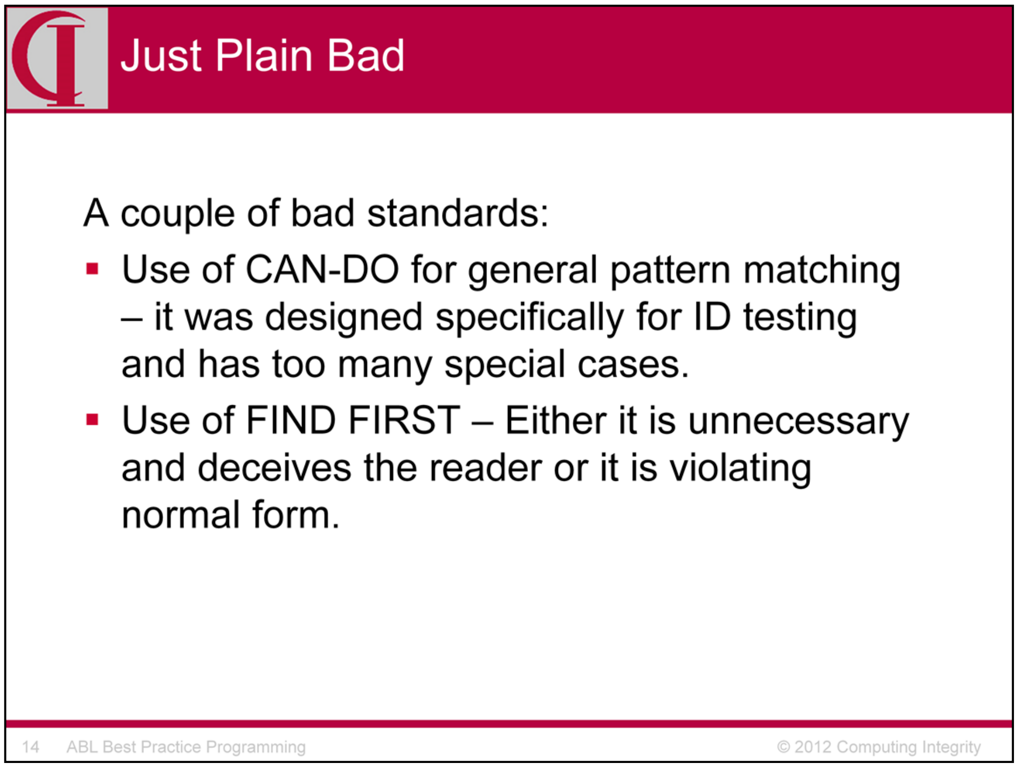
- There are alternative techniques that are less risk prone.
- The "standard" doesn't really mean or do what you think it does.
- The "standard" deceives the reader.

13 ABL Best Practice Programming © 2012 Computing Integrity

While I hate to start off on the negative, let's first consider "standards" that are just plain bad. They are bad because:

- There are alternative techniques that are less risk prone; or
- The "standard" doesn't really mean or do what you think it does; or
- The "standard" deceives the reader.

Or all three!



Just Plain Bad

A couple of bad standards:

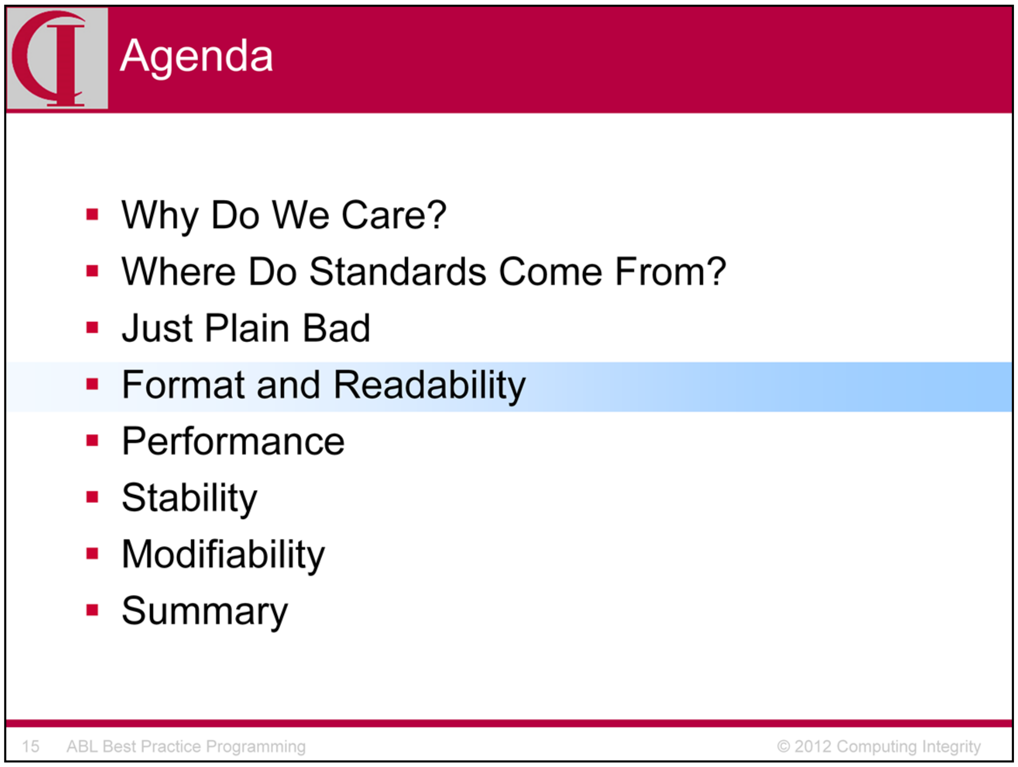
- Use of CAN-DO for general pattern matching – it was designed specifically for ID testing and has too many special cases.
- Use of FIND FIRST – Either it is unnecessary and deceives the reader or it is violating normal form.

14 ABL Best Practice Programming © 2012 Computing Integrity

There are many people who use the CAN-DO verb for general pattern matching. This is a mistake because it was specifically designed for testing IDs and has many special cases that can provide puzzling effects. There was recently someone on the PEG who used CAN-DO extensively for pattern matching in their application and was suddenly facing massive rework because of the recent addition of @ as a new special character for separating an ID into name and domain components and he had @ in his patterns. Ouch.

If you say FIND FIRST then you imply that you are finding the first of a group which has the same keys. If the keys actually specify a unique record, then you are deceiving the reader. If you are actually finding the first in a group and treating it differently, then you are violating normal form. Moreover, the supposed performance advantage which some claim as the reason for doing this doesn't exist.

Clearly, there are lots of these and I could probably do an entire talk on what not to do, but today I want to emphasize the positive, so I have just thrown out these two because they have come up recently.

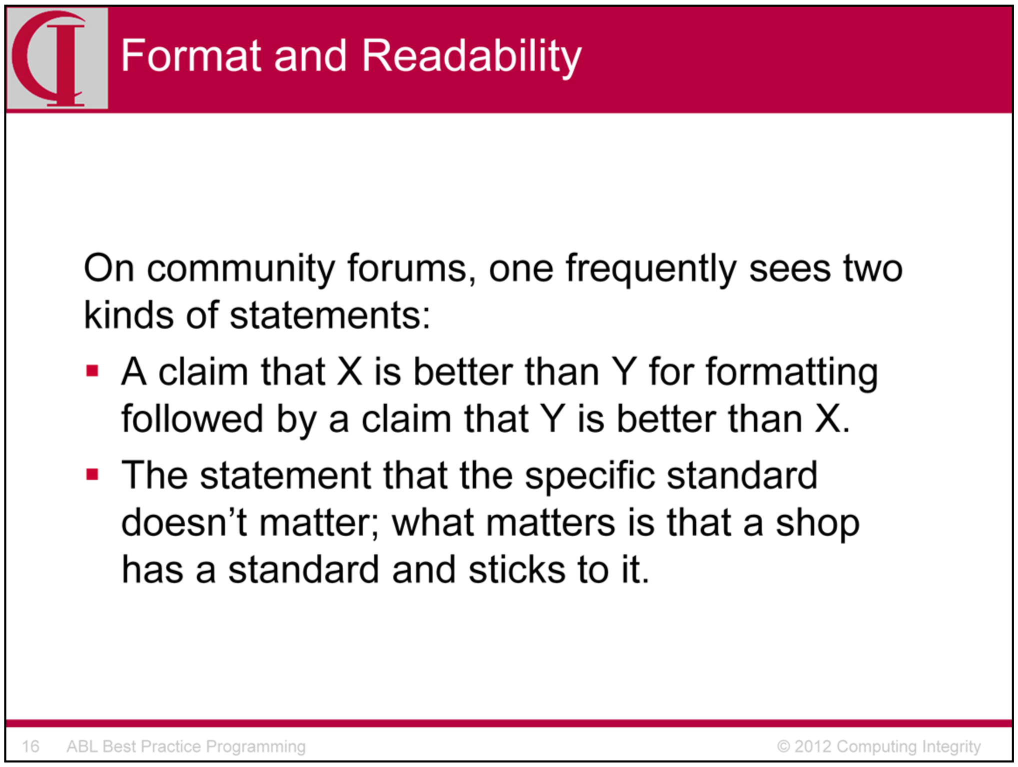


Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

15 ABL Best Practice Programming © 2012 Computing Integrity

OK, let’s get to the standards, starting with those relating primarily to format and readability.



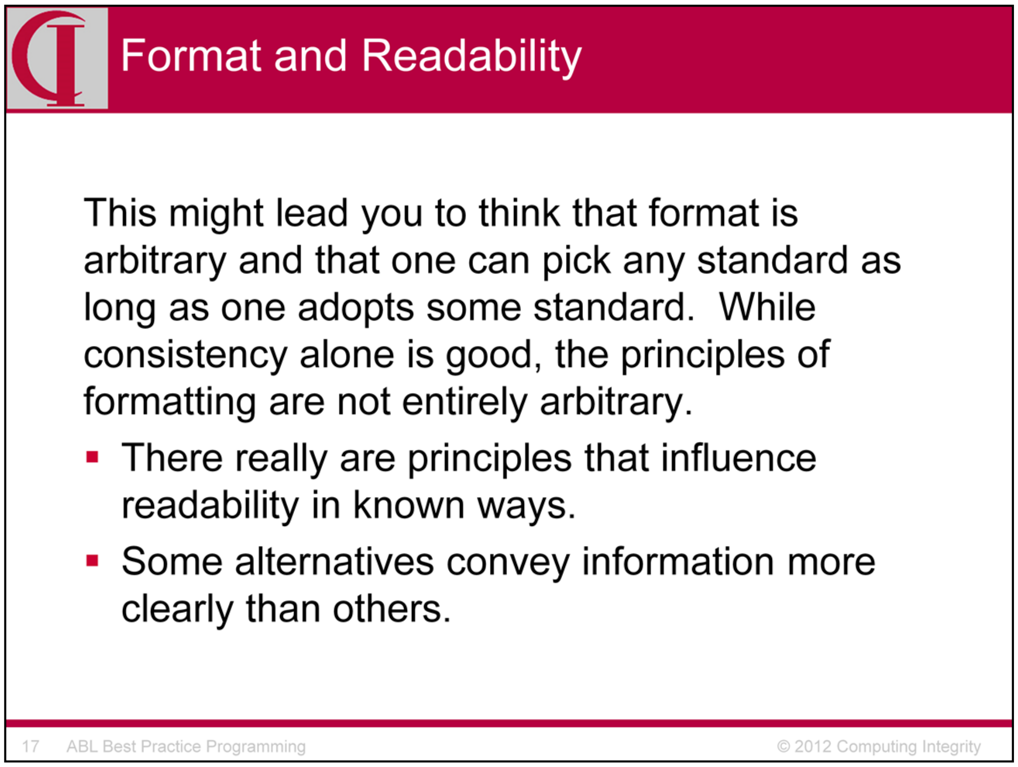
Format and Readability

On community forums, one frequently sees two kinds of statements:

- A claim that X is better than Y for formatting followed by a claim that Y is better than X.
- The statement that the specific standard doesn't matter; what matters is that a shop has a standard and sticks to it.

16 ABL Best Practice Programming © 2012 Computing Integrity

Just to make things confusing, there are two common ways in which standards, particularly format related ones, are talked about in on-line communities. One is that someone will advocate a standard and someone will then advocate nearly the opposite. The other is that someone will say that the details of a standard don't matter, just that one has one and sticks with it.



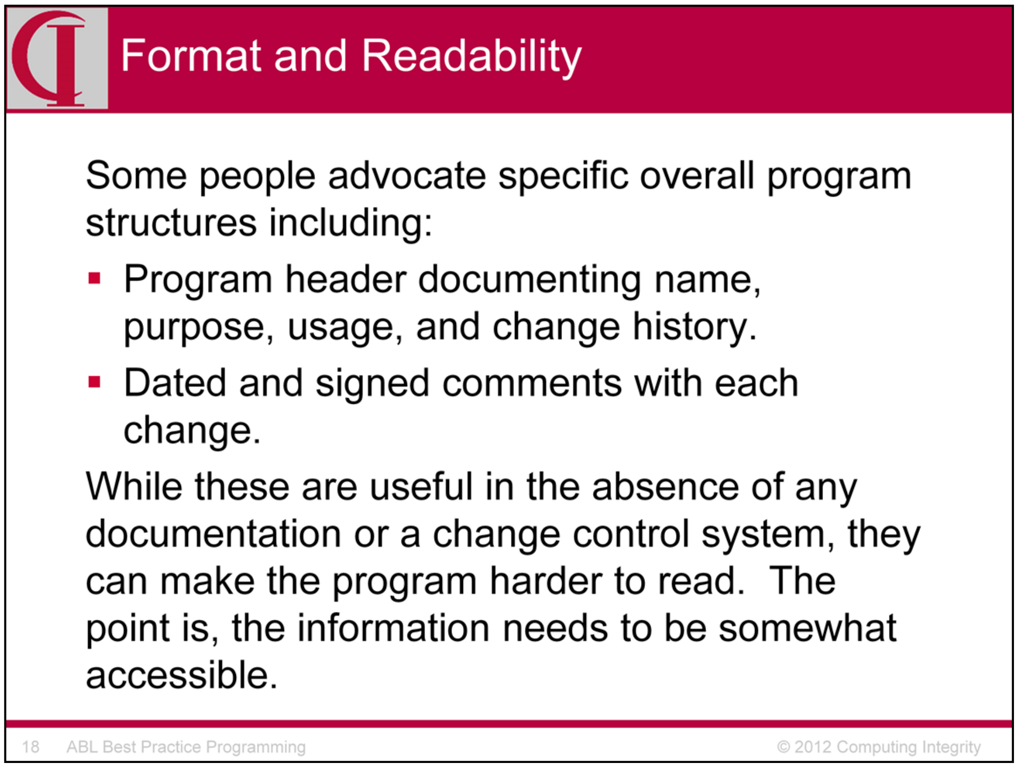
Format and Readability

This might lead you to think that format is arbitrary and that one can pick any standard as long as one adopts some standard. While consistency alone is good, the principles of formatting are not entirely arbitrary.

- There really are principles that influence readability in known ways.
- Some alternatives convey information more clearly than others.

17   ABL Best Practice Programming   © 2012 Computing Integrity

While it is certainly true that any standard is … usually … better than no standard, this doesn't mean that the choice of formatting standards is entirely arbitrary. In a set of standards, picking one choice or another may not make a huge difference, i.e., the big contrast is between following a standard and not. But, that doesn't mean that there aren't sound bases for the choices one makes. Readability, for example, is something that we know about from a much wider context base than just ABL code.



Format and Readability

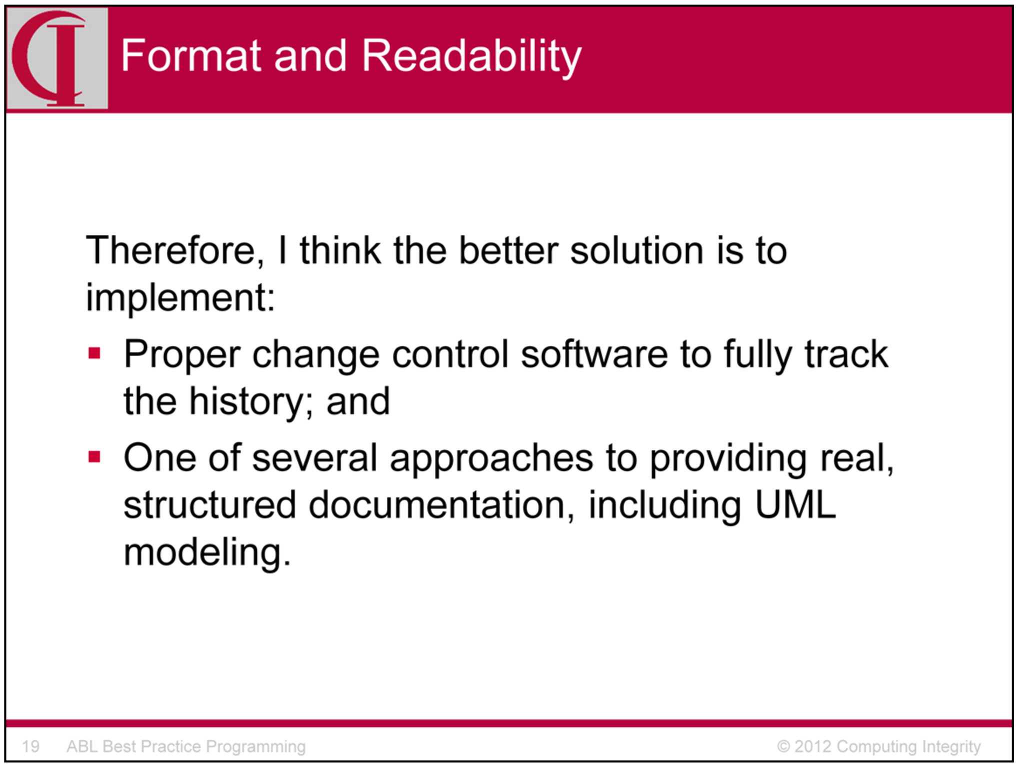
Some people advocate specific overall program structures including:

- Program header documenting name, purpose, usage, and change history.
- Dated and signed comments with each change.

While these are useful in the absence of any documentation or a change control system, they can make the program harder to read. The point is, the information needs to be somewhat accessible.

18 ABL Best Practice Programming © 2012 Computing Integrity

First, let's consider standards which are offered for the overall structure of the program. Typically, these don't have to do with readability, but rather with documentation either in the form of compact overall documentation at the top of the program and/or comments within the program, specifically documenting change. These are something of a mixed bag. If the choice is between following one of these standards and doing no documentation at all, then clearly following the standard is a plus. If one is editing code in vi on character terminals, maybe this is the best one can hope for. But, putting material into the program makes the actual code longer, possibly a lot longer, and the in-line aspects make the code harder to read because they break up the flow of the code with information which is not related to understanding what the code is doing. Moreover, one can't really tell the whole story of the change history with this simple mechanism since any restructuring … a frequent need … will destroy the continuity.



Format and Readability

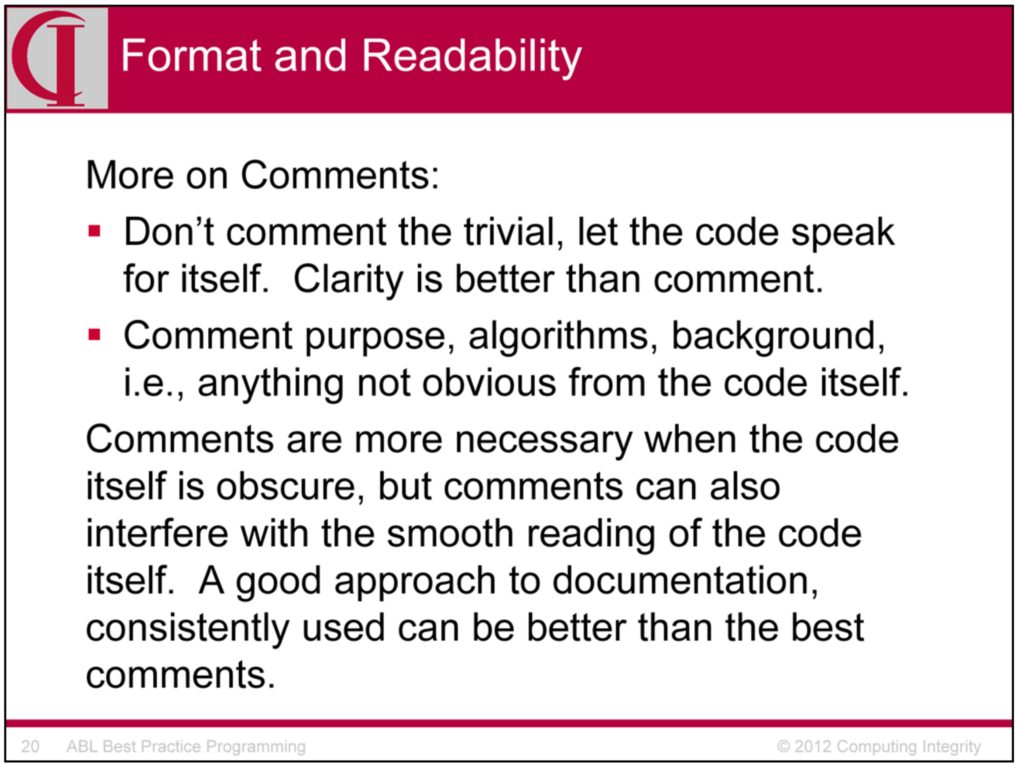
Therefore, I think the better solution is to implement:

- Proper change control software to fully track the history; and
- One of several approaches to providing real, structured documentation, including UML modeling.

19 ABL Best Practice Programming © 2012 Computing Integrity

Therefore, personally, I think the better solution is to implement:

- Proper change control software to fully track the history; and
- One of several approaches to providing real, structured documentation, including UML modeling.



Format and Readability

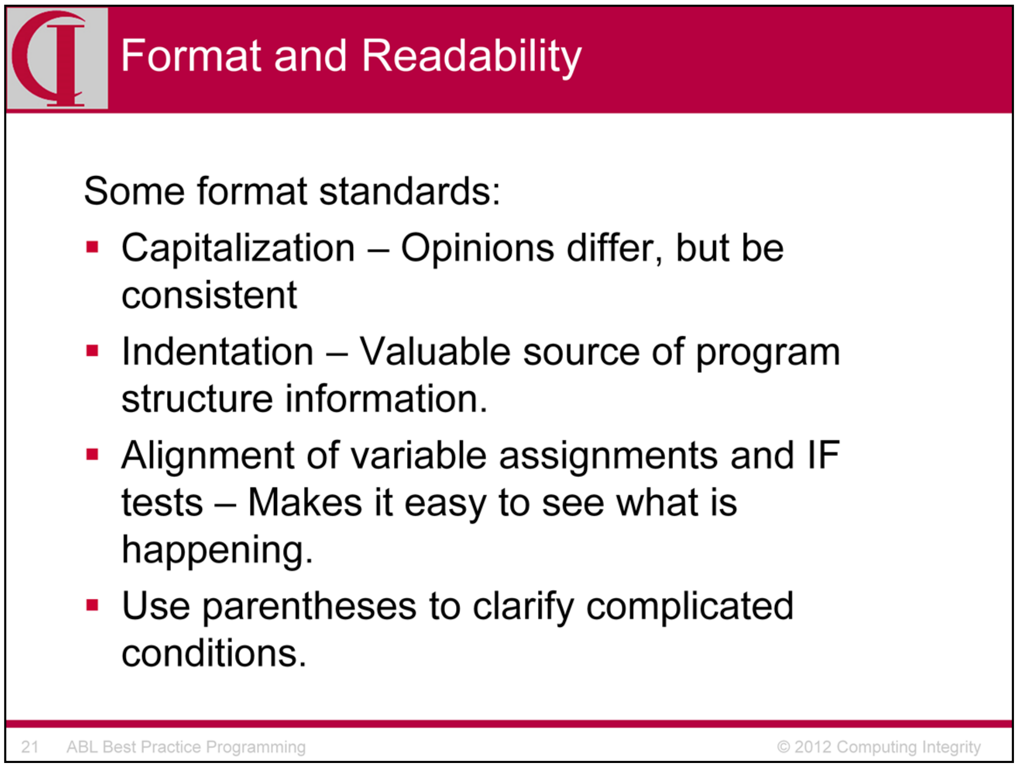
More on Comments:

- Don't comment the trivial, let the code speak for itself. Clarity is better than comment.
- Comment purpose, algorithms, background, i.e., anything not obvious from the code itself.

Comments are more necessary when the code itself is obscure, but comments can also interfere with the smooth reading of the code itself. A good approach to documentation, consistently used can be better than the best comments.

20 ABL Best Practice Programming © 2012 Computing Integrity

Similarly, there is broad support for comments among those who advocate standards. Here, I think one should separate the purpose from the mechanism. Clearly, documenting the trivial gets in the way of understanding and wastes effort. But, at the same time, documenting the purpose some code is supposed to fulfill, the algorithm which the code implements, any background one might need to understand the code, alternate algorithms, i.e., anything about the code that is not obvious from reading the code itself, is highly valuable, possibly even fundamental. Consider whether the best way to do this is using in-line comments or some other form of documentation.



Format and Readability

Some format standards:

- Capitalization – Opinions differ, but be consistent
- Indentation – Valuable source of program structure information.
- Alignment of variable assignments and IF tests – Makes it easy to see what is happening.
- Use parentheses to clarify complicated conditions.

21 ABL Best Practice Programming © 2012 Computing Integrity

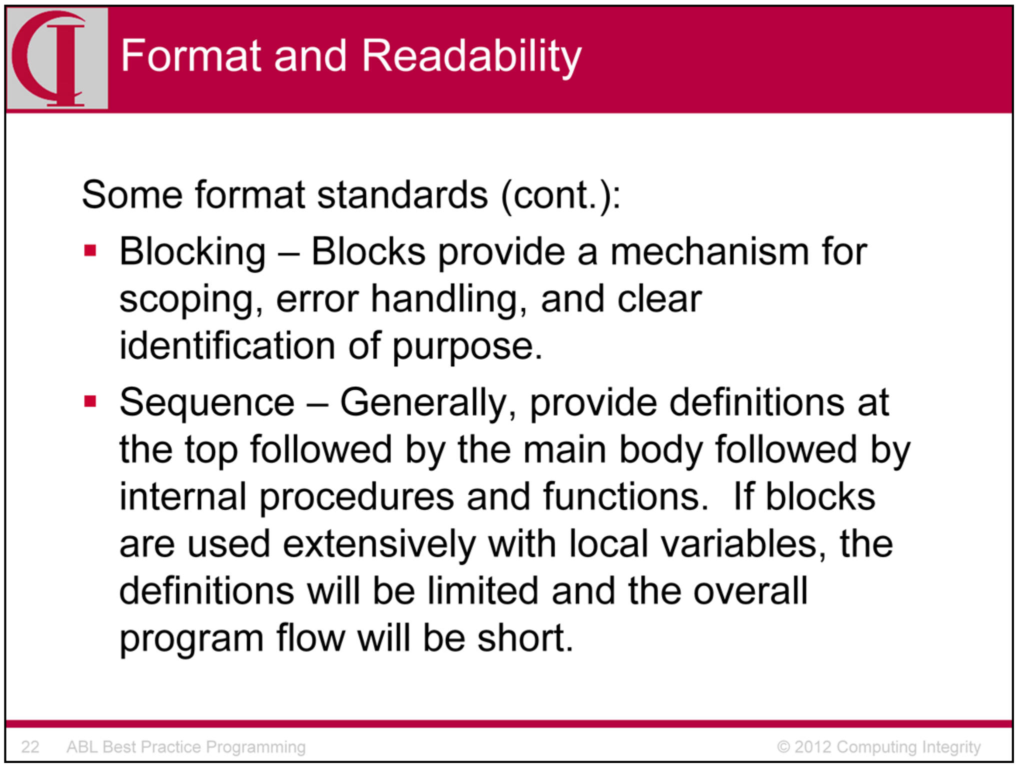
So let's consider some specifics.

Some advocate capitalizing keywords, as PSC does, and others advocate lower case key words. Consistency is clearly important to optimize readability, but I have always felt that capitalization was emphasis and that by capitalizing the keyword one was placing the emphasis in the wrong place. In a for each, for example, it is the table name which is most important, not the verb. It is also an old principle of readability that not only are capitals harder to read, but mixed case is preferable to all lower case. To me, this provides a strong argument in favor of mixed case table, field, and variable names and lower case keywords. Obviously, opinions vary!

I don't know anyone who is opposed to indentation, but there are certainly arguments for many different indentation standards. I have my personal preferences, of course, but I think this is an area where having a consistent standard is far more important than exactly what that standard is. The goal is for the indentation to convey the structure of the program to the reader and make it easy to understand that structure without having to parse all the details.

Similarly, aligning the parts of a block assignment or IF test can make it easier for the reader to recognize what is happening - clear structure and clear associations.

When necessary to clarify complex structures, the use of parentheses is far preferable to depending on the reader understanding rules of precedence.



Format and Readability

Some format standards (cont.):

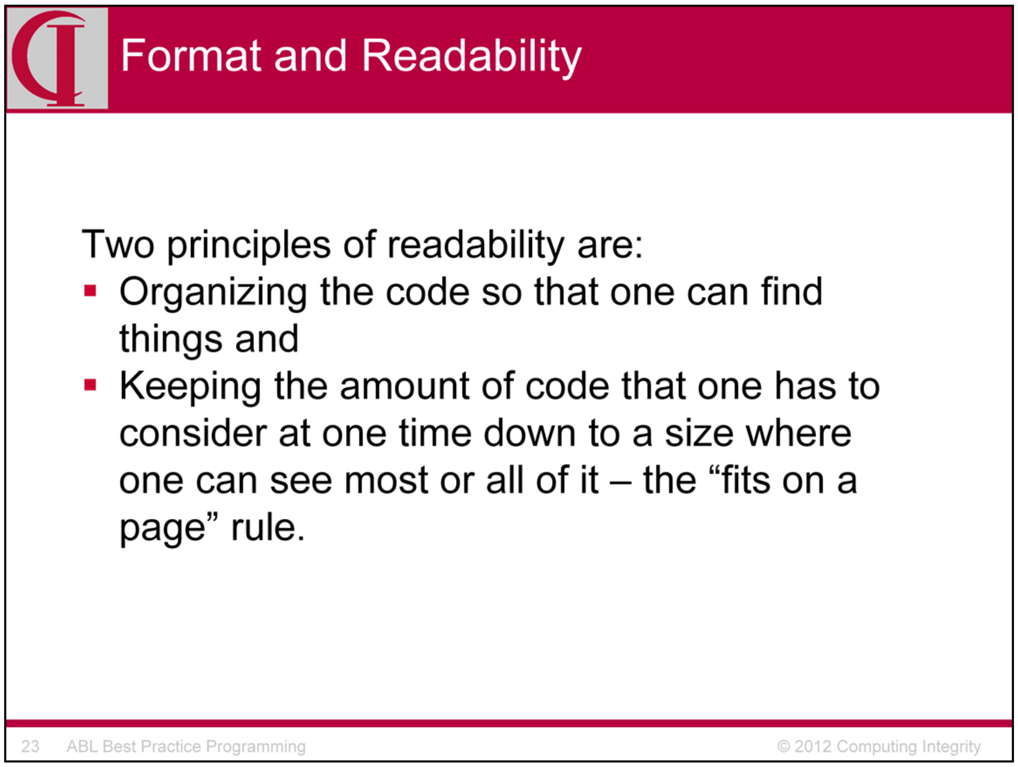
- Blocking – Blocks provide a mechanism for scoping, error handling, and clear identification of purpose.
- Sequence – Generally, provide definitions at the top followed by the main body followed by internal procedures and functions. If blocks are used extensively with local variables, the definitions will be limited and the overall program flow will be short.

22 ABL Best Practice Programming © 2012 Computing Integrity

One of the most important structuring elements of an ABL program, both in terms of formatting for understanding and in terms of controlling functionality are the various block types. They are important for scoping of buffers and transactions, iteration, error handling, and defining units of work. As such, they also help define the logical structure of the program which aids understanding. If you see an ABL program that is little more than a long, linear sequence of code, you know the programmer did not think about decomposing the problem into functional units.

One common, but unfortunately not universal practice is to organize all definitions of variables, buffers, temp-tables, etc. at the top of the program, preferably in labeled sections by type. If one has used blocks extensively in the program and properly defined local variables for the blocks for those things used within the block, i.e., values largely flow in and out via parameters, then the number of definitions at the top of a program should be limited … not the 5 pages one sees all too often.

Those definitions should be followed by the program's main body, clearly demarked, and preferably short so that one can easily see the overall flow. Below that, clearly marked, should be the internal procedures and functions which implement the detailed logic.

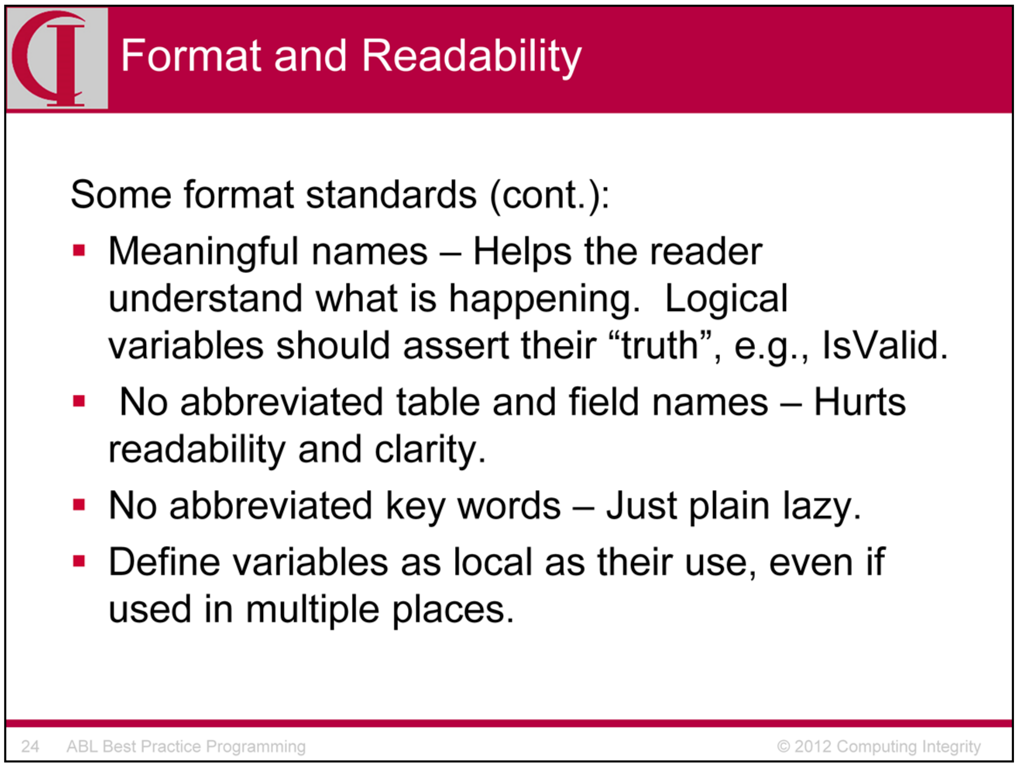


Format and Readability

Two principles of readability are:
- Organizing the code so that one can find things and
- Keeping the amount of code that one has to consider at one time down to a size where one can see most or all of it – the “fits on a page” rule.

23 ABL Best Practice Programming © 2012 Computing Integrity

Two principles of readability are:
- Organizing the code so that one can find things and
- Keeping the amount of code that one has to consider at one time down to a size where one can see most or all of it – the “fits on a page” rule.



Format and Readability

Some format standards (cont.):

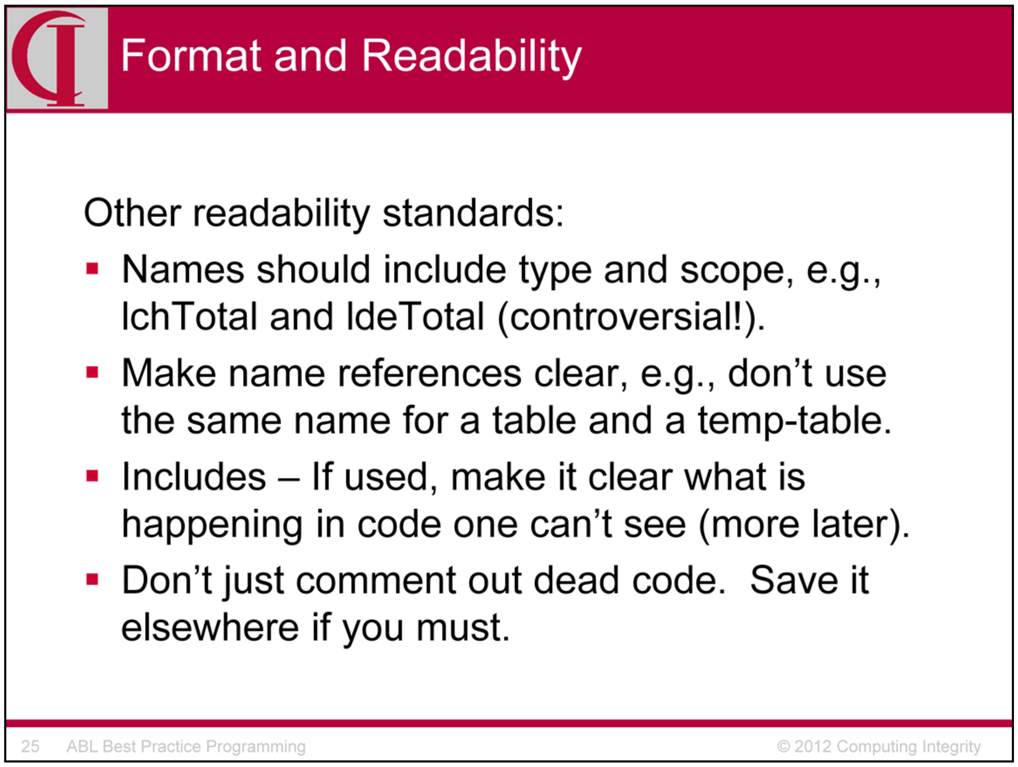
- Meaningful names – Helps the reader understand what is happening. Logical variables should assert their "truth", e.g., IsValid.
- No abbreviated table and field names – Hurts readability and clarity.
- No abbreviated key words – Just plain lazy.
- Define variables as local as their use, even if used in multiple places.

24 ABL Best Practice Programming © 2012 Computing Integrity

Meaningful names for variables, buffers, temp-tables, tables, fields, etc. is one of the simplest and yet most effective tools we have for making code readable and easily understandable. The slight effort of typing a few more characters will pay great dividends when someone has to read the code and understand what it is doing … even the person who wrote the code in the first place. Cryptic names need comments or documentation to tell us what they mean; good names convey meaning without additional documentation. In particular, logical variables should tell you what true and false mean, e.g., IsValid conveys intent in a way that Flag does not. Even Valid is less clear.

You can tell that I hate abbreviations … abbreviating table and field names not only hurts readability, but it is a great way to make future trouble for yourself. Abbreviating key words is just lazy and turns easily read ABL into something cryptic.

As suggested earlier, variables should be defined as local as the scope of their use. Even if you use a variable with the same meaning in three different blocks, if there is no connection between the use, define it local to each block so that you make it clear that the scope of its use is that block.


Format and Readability

Other readability standards:

- Names should include type and scope, e.g., lchTotal and ldeTotal (controversial!).
- Make name references clear, e.g., don't use the same name for a table and a temp-table.
- Includes – If used, make it clear what is happening in code one can't see (more later).
- Don't just comment out dead code. Save it elsewhere if you must.

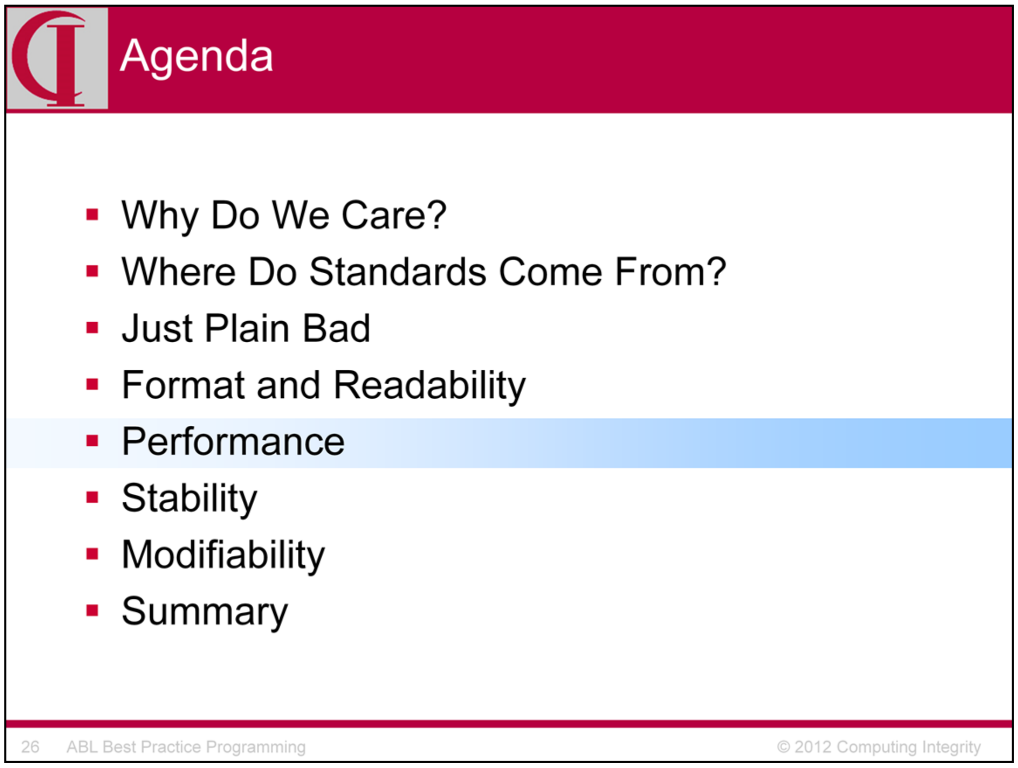
25 ABL Best Practice Programming © 2012 Computing Integrity

This next point is more controversial than I think it should be. I advocate using a form of what is called Hungarian notation for variable naming in which a prefix provides both type and scope information. Thus chTotal is the Total in character form and deTotal is the amount in decimal form. Similarly, lchTotal shows that the variable is scoped local to the internal procedure. There seem to be a lot of people who disagree with this idea as well as a fair number that will use only the type prefix, but I'm not sure why there is as much resistance as there is. See http://www.oehive.org/Hungarian for more discussion.

I believe that name references should clearly identify the object to which they refer, thus, one shouldn't use the same name for a buffer or temp-table as the database table because it isn't clear to the reader which you are referring to, even if it is clear to the compiler.

Later, I am going to comment more about includes, but if you have to use them, make sure that the reference makes it clear what is happening in the code that the person can't see without looking at a separate piece of code.

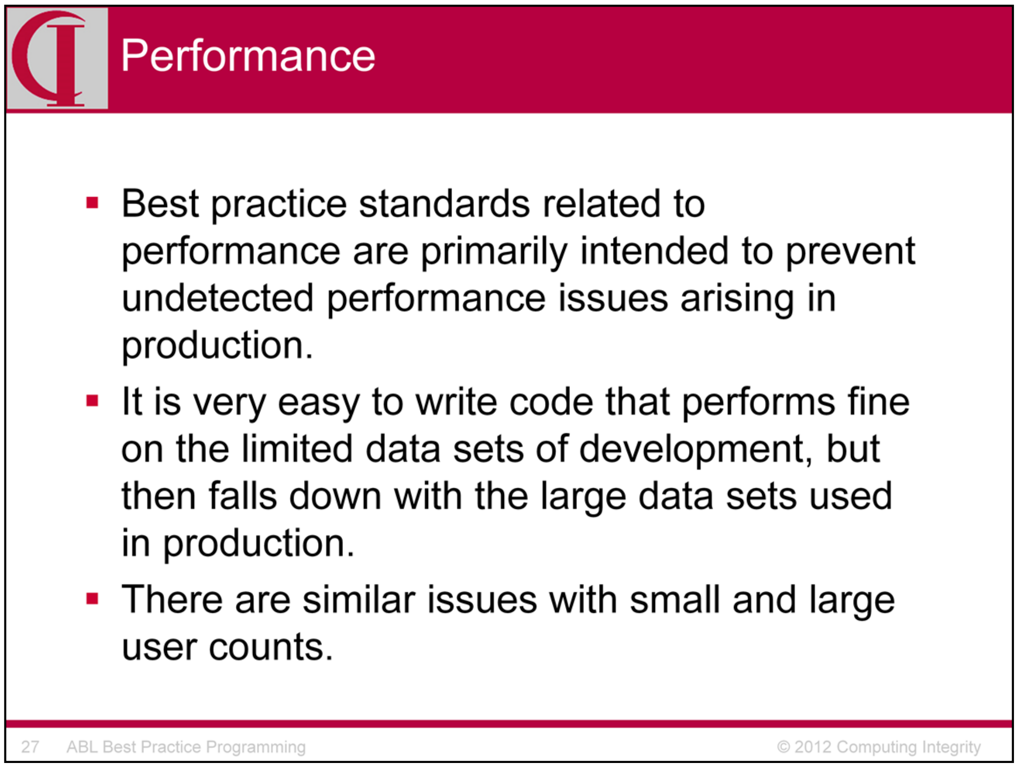
There is a horrible practice of commenting out code which is no longer used, but leaving it in place. Apparently, this was done to make the change set minimal since the only thing changing were the two lines with the open and close comment. This is terribly confusing and is very annoying when searching. Make your code clean.



Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

Now lets look at some standards related primarily to performance … good and bad.

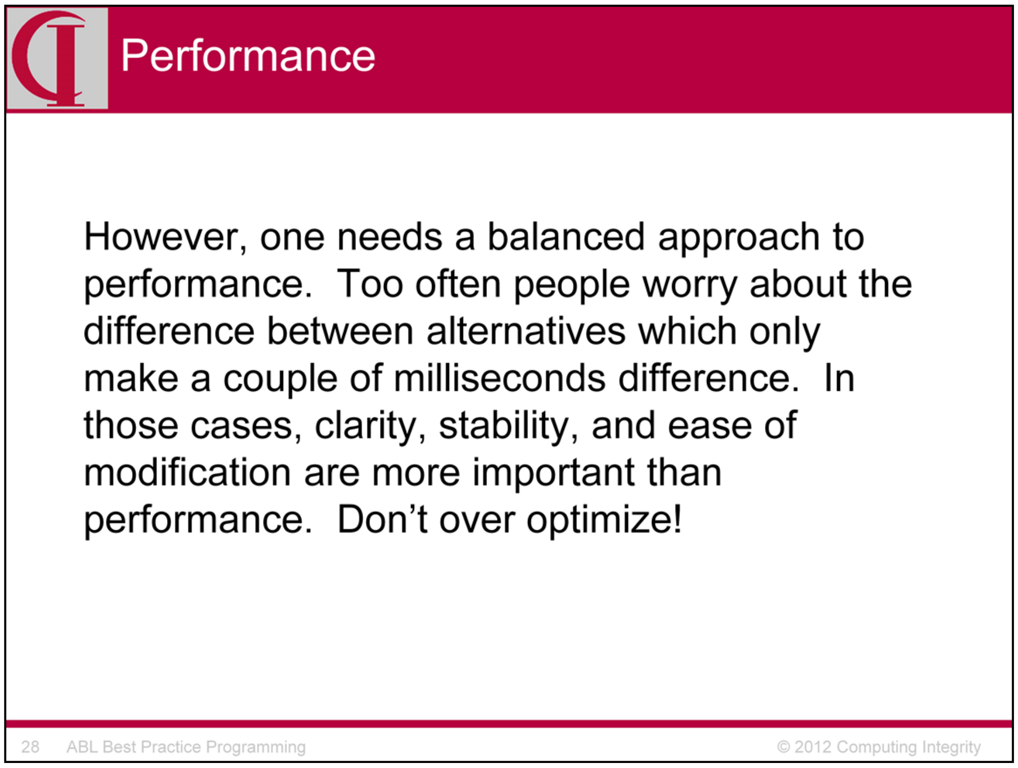


Performance

- Best practice standards related to performance are primarily intended to prevent undetected performance issues arising in production.
- It is very easy to write code that performs fine on the limited data sets of development, but then falls down with the large data sets used in production.
- There are similar issues with small and large user counts.

27 ABL Best Practice Programming © 2012 Computing Integrity

The primary target of performance related standards is reducing the risk that code which performed well in development against limited data sets and with a limited number of users will perform poorly in production with large data sets and user counts.



Performance

However, one needs a balanced approach to performance. Too often people worry about the difference between alternatives which only make a couple of milliseconds difference. In those cases, clarity, stability, and ease of modification are more important than performance. Don’t over optimize!

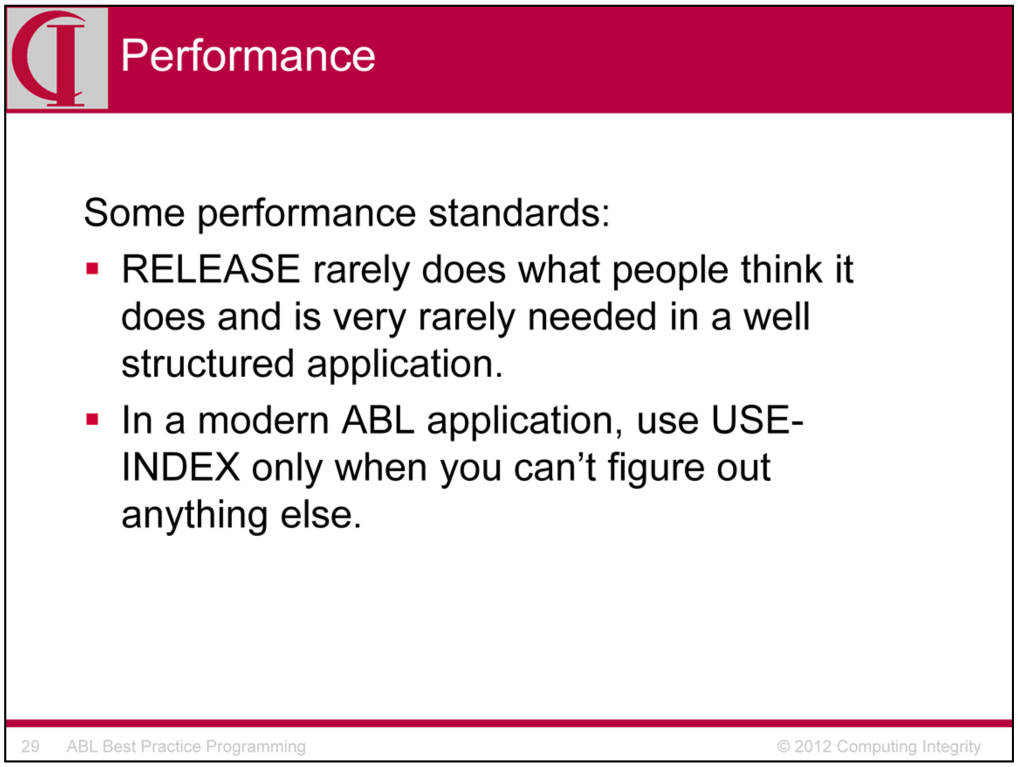
28 ABL Best Practice Programming © 2012 Computing Integrity

But, one also needs to be sensible about how much one cares about performance. There are frequent inquiries on on-line forums asking which of two or more constructs is faster. Even when there is a predictable difference, unless one is performing the operation in a loop being iterated a million times, who cares if there is a few milliseconds difference?

In most cases, the judge of performance is the user. There is an old principle in user interface design that the user will tolerate delay in relation to the perceived value of what is happening. Thus, a delay in character echo quickly becomes intolerable since the only value is confirming what one has just typed, but a few seconds delay when one has posted an order is easily tolerated because meaningful work is being done and there is a natural break in the workflow.

Moreover, one needs to keep things in perspective. If an operation requires retrieving information from the disk, for example, the time required to do the retrieval is so long compared to the computational steps around it, that using an extra fraction of a millisecond in the associated computation isn’t meaningful in the larger picture.

In most cases, clarity, stability, and ease of modification are more important than these minor performance differences.


Performance

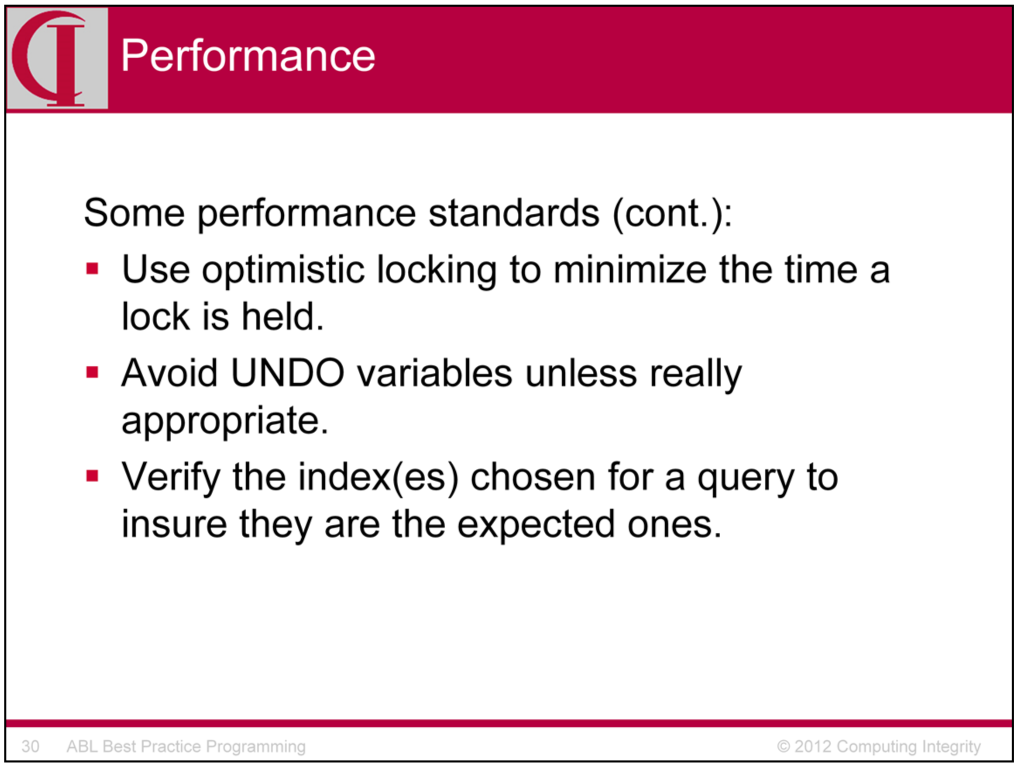
Some performance standards:

- RELEASE rarely does what people think it does and is very rarely needed in a well structured application.
- In a modern ABL application, use USE-INDEX only when you can't figure out anything else.

29 ABL Best Practice Programming © 2012 Computing Integrity

There are various ways in which some shops recommend the regular use of RELEASE on the theory that it is doing something good. In point of fact, it rarely has any effect because the transaction and buffer scope is larger than where the RELEASE is used. Moreover, if one is properly using blocks to control transaction and buffer scope, there is almost no reason to ever use it.

In the old days, it was considered good practice to specify USE-INDEX to make sure that the compiler picked the index the developer knew was best. But, times change. In particular, there are places where the compiler will use multiple indexes to resolve a query … but not if you specify USE-INDEX. So, best practice now is to leave it out because almost all of the time the compiler will make the right decision. Of course, it doesn't hurt to check up on the compiler with COMPILE LISTING.



Performance

Some performance standards (cont.):

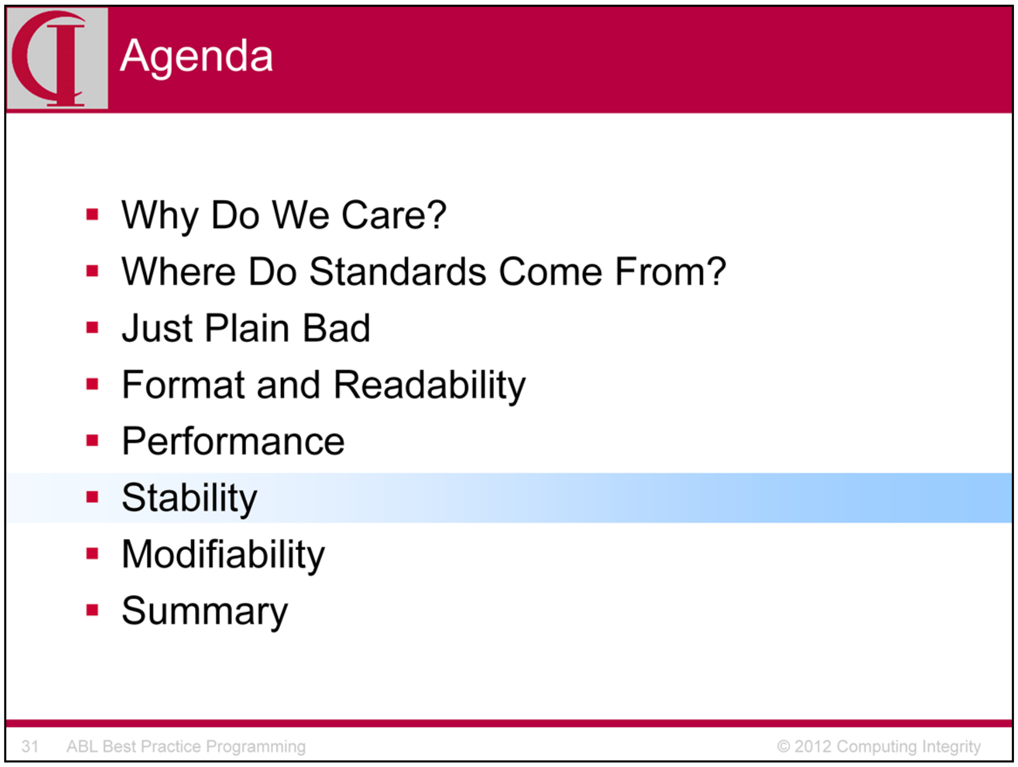
- Use optimistic locking to minimize the time a lock is held.
- Avoid UNDO variables unless really appropriate.
- Verify the index(es) chosen for a query to insure they are the expected ones.

30 ABL Best Practice Programming © 2012 Computing Integrity

In the old days it was common to do updates directly against database buffers, thereby locking those buffers throughout all the user interaction. We have long since realized that leaving all those locks increases contention and thus impacts performance and the friendliness of the application. Now, one should be doing updates into local variables and temp-tables and, when all the information is ready to be persisted, then open a very short transaction to do all updates. If the data originated in the database, this means reading the data with no lock, then checking before persisting the modified data to make sure someone else hasn't changed the data in the meantime … which, with reasonable design, is highly unlikely.

It is unfortunate that the default for variables is not no-undo, but once upon a time I guess someone at Progress thought it was good idea for variables to have undo processing. But, it is surprising how much overhead one can add to an application by having it keep track of before and after values for every variable. The first time I experienced this, almost 25 years ago now, I was amazed at how much difference we were able to make in performance by moving to no-undo.

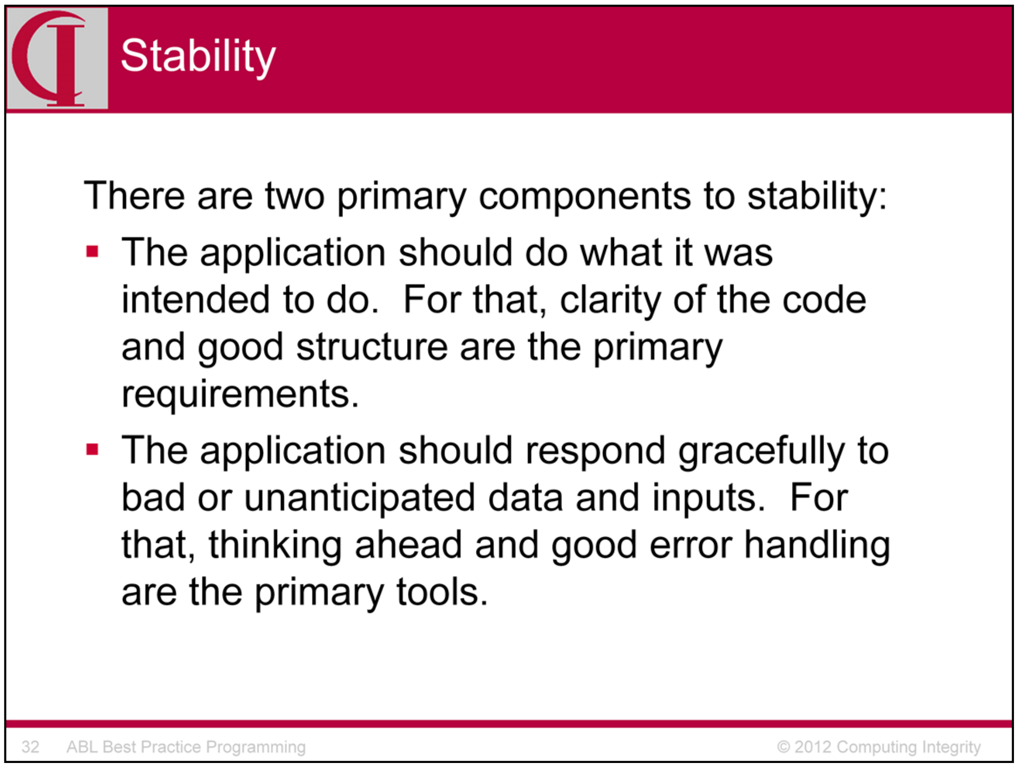
And, as indicated in the reference to USE-INDEX, trust but verify to make sure that the compiler is picking the index it should.



Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

On to some standards for stability.



Stability

There are two primary components to stability:

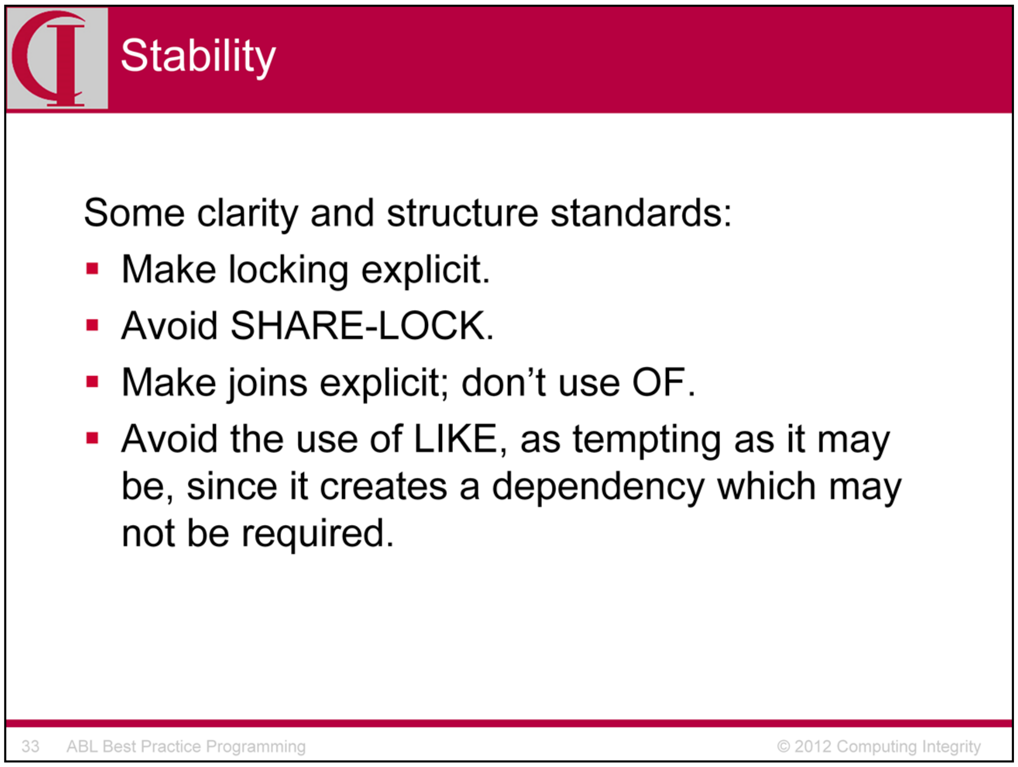
- The application should do what it was intended to do. For that, clarity of the code and good structure are the primary requirements.
- The application should respond gracefully to bad or unanticipated data and inputs. For that, thinking ahead and good error handling are the primary tools.

32 ABL Best Practice Programming © 2012 Computing Integrity

There are two aspects to application stability.

One aspect is that one wants the application to do what it was intended to do … no strange quirks, no needing special tricks, no surprises. To achieve that our primary tools are having clear code and good structure so that when we read the code we can understand clearly what it is going to do. We still need the user to be clear about what the program should do, of course, but having that we should be able to produce clear, unambiguous code that implements the design requirements and we should be able to easily modify that behavior as requirements change.

The other aspect is that we want the application to respond gracefully when something unanticipated happens such as bad data or incorrect inputs. I say “unanticipated”, but in fact we can anticipate some of these types of errors and I will talk more about that in a moment. To handle these situations, we need to think ahead about what might happen and provide good error handling for when the unexpected happens.



Stability

Some clarity and structure standards:

- Make locking explicit.
- Avoid SHARE-LOCK.
- Make joins explicit; don't use OF.
- Avoid the use of LIKE, as tempting as it may be, since it creates a dependency which may not be required.

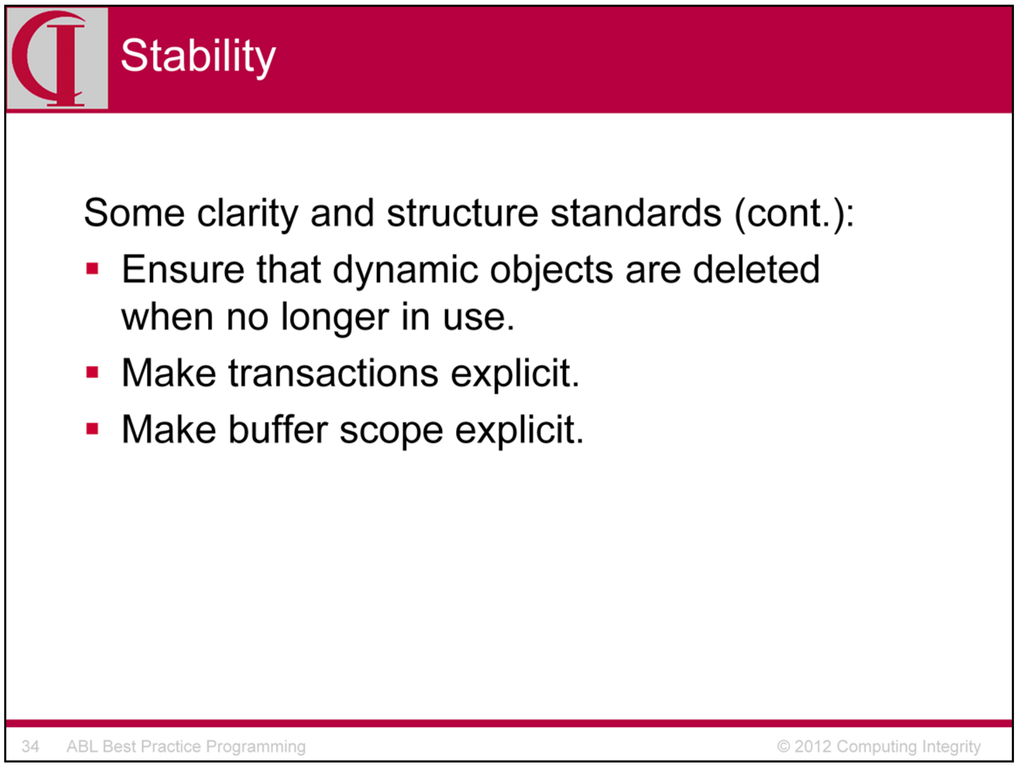
33 ABL Best Practice Programming © 2012 Computing Integrity

For standards related to clarity and structure we have:
Make locking explicit. There is never a reason to let the compiler decide which locking to use.

Avoid SHARE-LOCK altogether. There are only a few special circumstances where there is a reason for a share-lock and even those are questionable. No-lock unless you are going to change something and exclusive-lock for the shortest possible interval.

In queries, don't use OF to make joins and depend on the compiler to pick the right fields. State the join explicitly so that a reader will know the exact relationship between the tables.

I recommend against the use of LIKE in defining temp-tables and variables. It is tempting, both to save typing and to ensure that any changes to the database will get reflected in the code, but it creates a database dependency that might not otherwise exist. In V6 style programming where user interaction, business logic, and database updates are all mushed together in the same code, this issue might not seem significant, but when one is dividing these areas of responsibility into layers, one wants only the code that actually touches the database to have the database dependency.


Stability

Some clarity and structure standards (cont.):

- Ensure that dynamic objects are deleted when no longer in use.
- Make transactions explicit.
- Make buffer scope explicit.

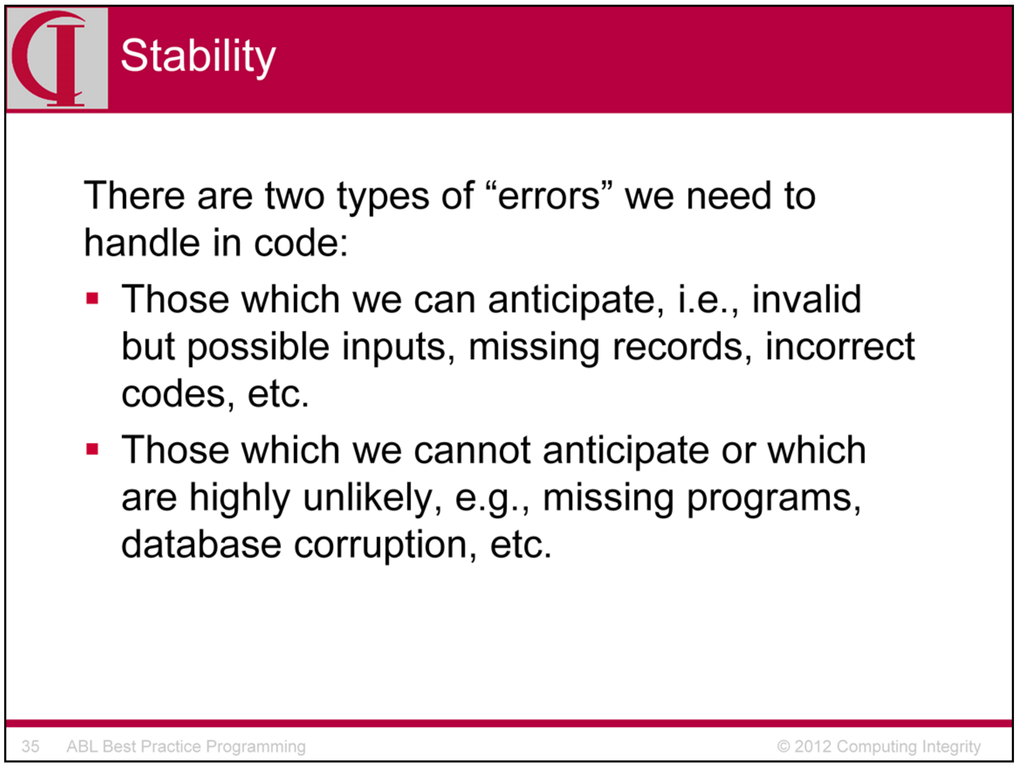
34 ABL Best Practice Programming © 2012 Computing Integrity

When using dynamic objects of any kind, make sure to delete them when they are no longer in use. The usual rule is “if you create it; you delete it.” There are cases of “one way” objects where this is not practical, but there should always be a way to know when one is done and to clean up. Failure to do so is likely to lead to unpleasant surprises.

Incidentally, I also apply this policy to objects in OO code. While there is garbage collection for OO, I think it is better for the programmer to assert that one is done and do the cleanup. Among other things, this makes it clear to a later reader when one is done.

While the compiler does a very good job at automatic scoping of transactions, it is far better to use the TRANSACTION keyword to explicitly state where you want the transaction scope. If nothing else, this will detect scoping errors when you specify a transaction within an existing transaction. Many errors discovered in production can be traced to errors in transaction scope.

Similarly, make buffer scopes explicit by using the FOR <filename> construct and defining local buffers in internal procedures. Another large class of production errors comes from buffer scope being wider than expected or simply not thinking about buffer scope.



Stability

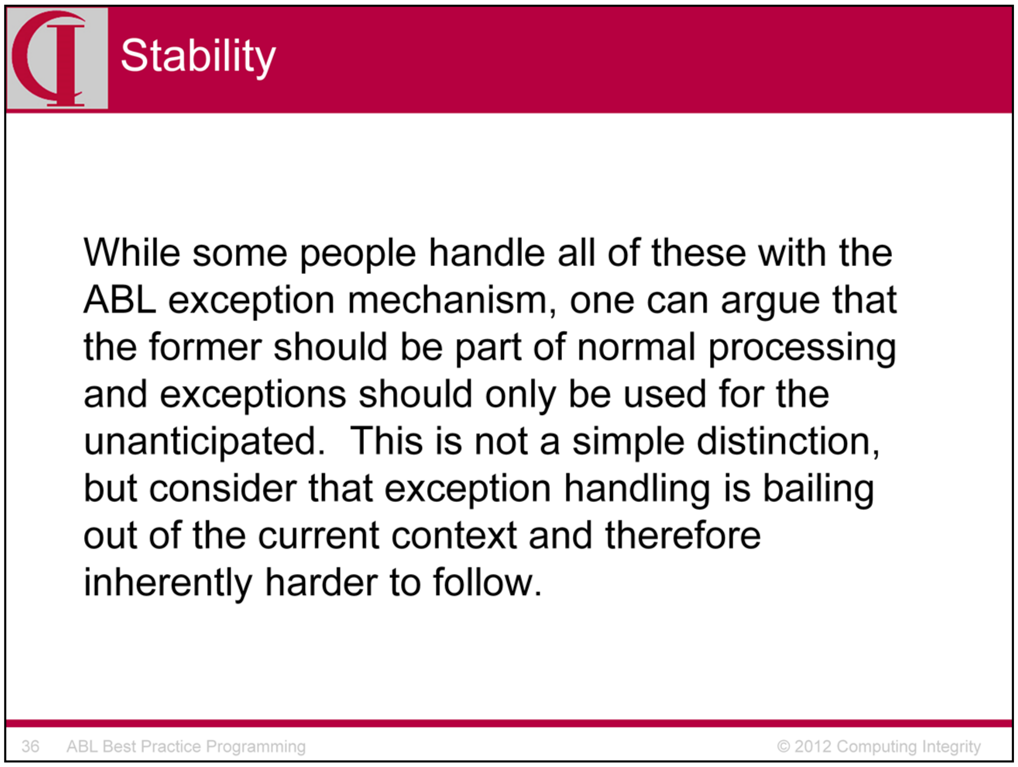
There are two types of "errors" we need to handle in code:

- Those which we can anticipate, i.e., invalid but possible inputs, missing records, incorrect codes, etc.
- Those which we cannot anticipate or which are highly unlikely, e.g., missing programs, database corruption, etc.

35 ABL Best Practice Programming © 2012 Computing Integrity

As I mentioned before, there are two different types of things which can happen other that those things that are supposed to happen – those we can anticipate and those we can't. When I say we can't anticipate them, I mostly mean that, whether or not we can anticipate the possibility, we can't associated it with a particular place in the code. Thus, if we go to look up a customer record based on a code stored in an order, we can anticipate that the customer record may not be found. Whereas, while we can anticipate that a database might become corrupt, we can't associate that possibility with any particular part of the application.

There is more of a sliding scale here than an absolute distinction. For example, when we run a program, we could anticipate that the program would not be found. But, this should be highly unlikely and there is typically nothing about one run statement or another which makes it more or less likely in that instance. So, I am inclined to regard missing programs as being of the second type unless it is a situation where one is actually generating the program to run or something similar.

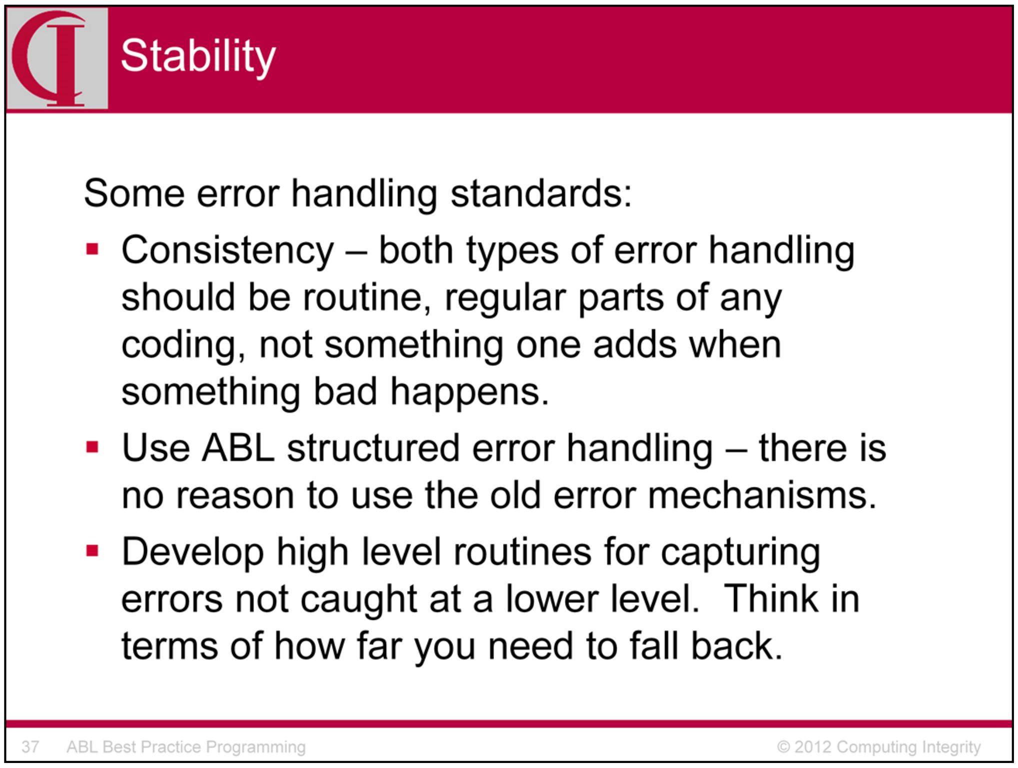


Stability

While some people handle all of these with the ABL exception mechanism, one can argue that the former should be part of normal processing and exceptions should only be used for the unanticipated. This is not a simple distinction, but consider that exception handling is bailing out of the current context and therefore inherently harder to follow.

36 ABL Best Practice Programming © 2012 Computing Integrity

One can handle both types of “error” with the ABL exception mechanism, but there is a point of view which suggests this is not best practice. The issue is that the ABL exception handling structure, in common with most languages, leaps one out of the flow of control. While it is certainly determinable where one goes … and, in fact, necessary to get out of the flow of control in some circumstances … this is not something that is easy to read.   Therefore, the recommendation is that anticipated “errors” be handled by regular code and the exception handling structures be reserved for unanticipated errors, notably those for which there is no simple recovery possible.



Stability

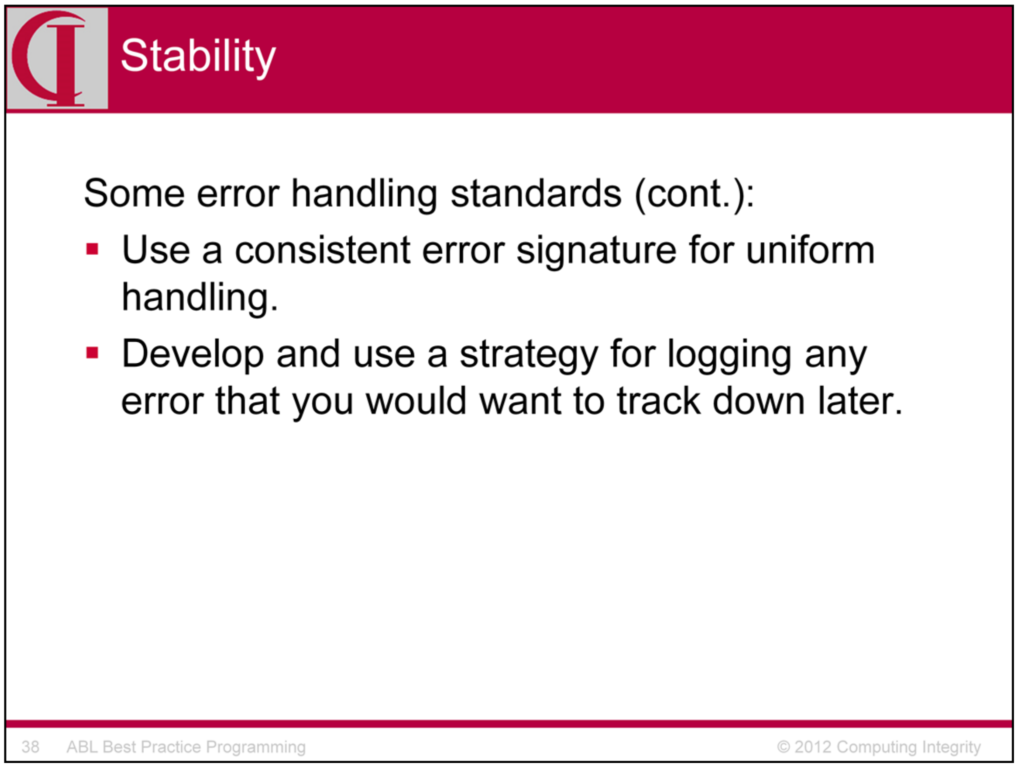
Some error handling standards:

- Consistency – both types of error handling should be routine, regular parts of any coding, not something one adds when something bad happens.
- Use ABL structured error handling – there is no reason to use the old error mechanisms.
- Develop high level routines for capturing errors not caught at a lower level. Think in terms of how far you need to fall back.

37 ABL Best Practice Programming © 2012 Computing Integrity

Some error handling standards:

- Consistency – both types of error handling should be routine, regular parts of any coding, not something one adds when something bad happens.
- Use ABL structured error handling – there is no reason to use the old error mechanisms.
- Develop high level routines for capturing errors not caught at a lower level. Think in terms of how far you need to fall back. The ultimate of these should be a routine at the top level which catches any error thrown by a lower level and not handled before then.



Stability

Some error handling standards (cont.):

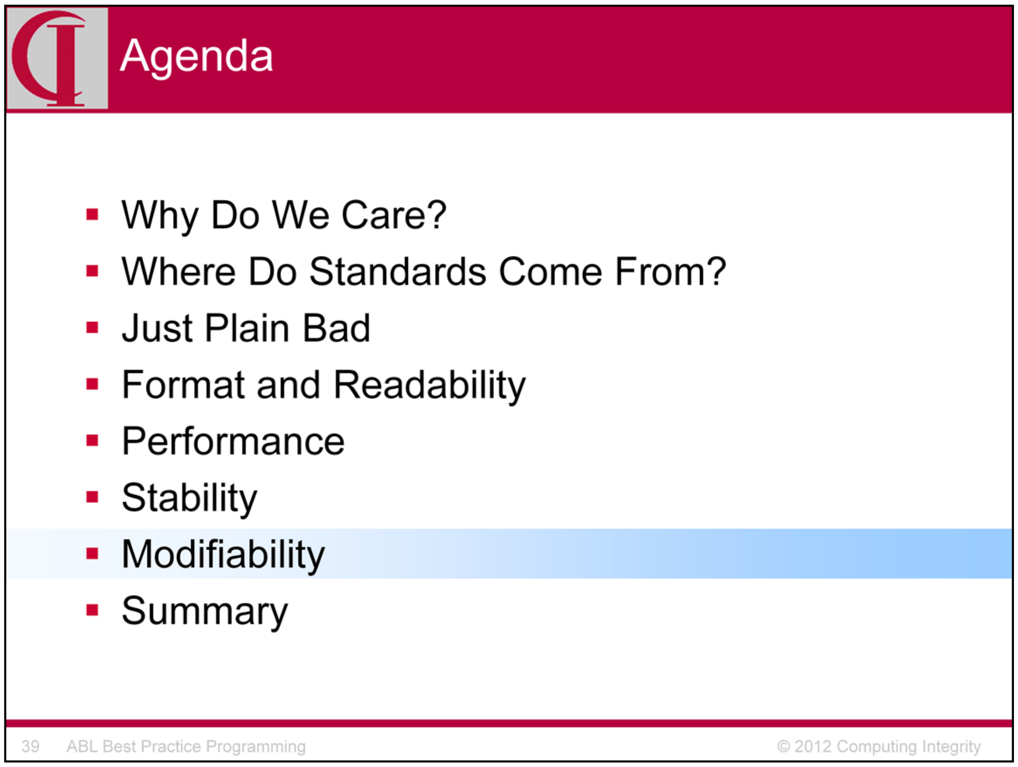
- Use a consistent error signature for uniform handling.
- Develop and use a strategy for logging any error that you would want to track down later.

38 ABL Best Practice Programming © 2012 Computing Integrity

These two are about creating a common strategy that you can then easily apply everywhere.

If you create a consistent error signature, then you can pass error objects up the call stack and handle them in a uniform way.

Similarly, create a strategy for consistently logging any error which you might want to know about. That includes all of the unanticipated errors and might include some of the anticipated ones, particularly security violations.

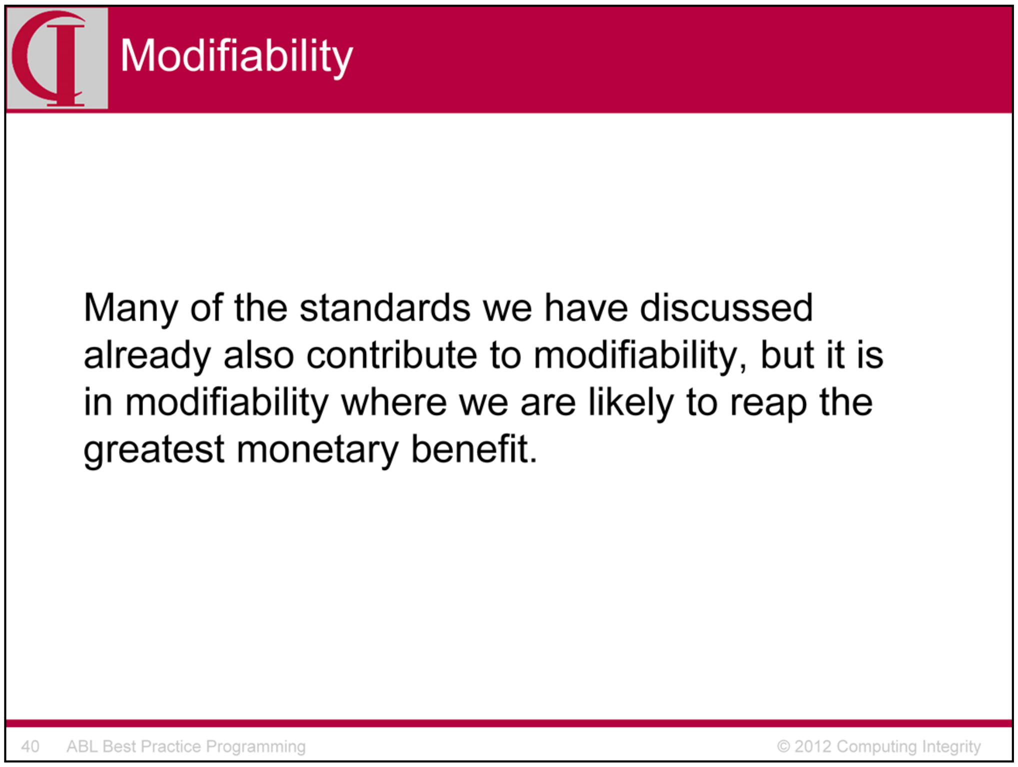


Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

39 ABL Best Practice Programming © 2012 Computing Integrity

Now, let’s talk a little about standards that promote modifiability.

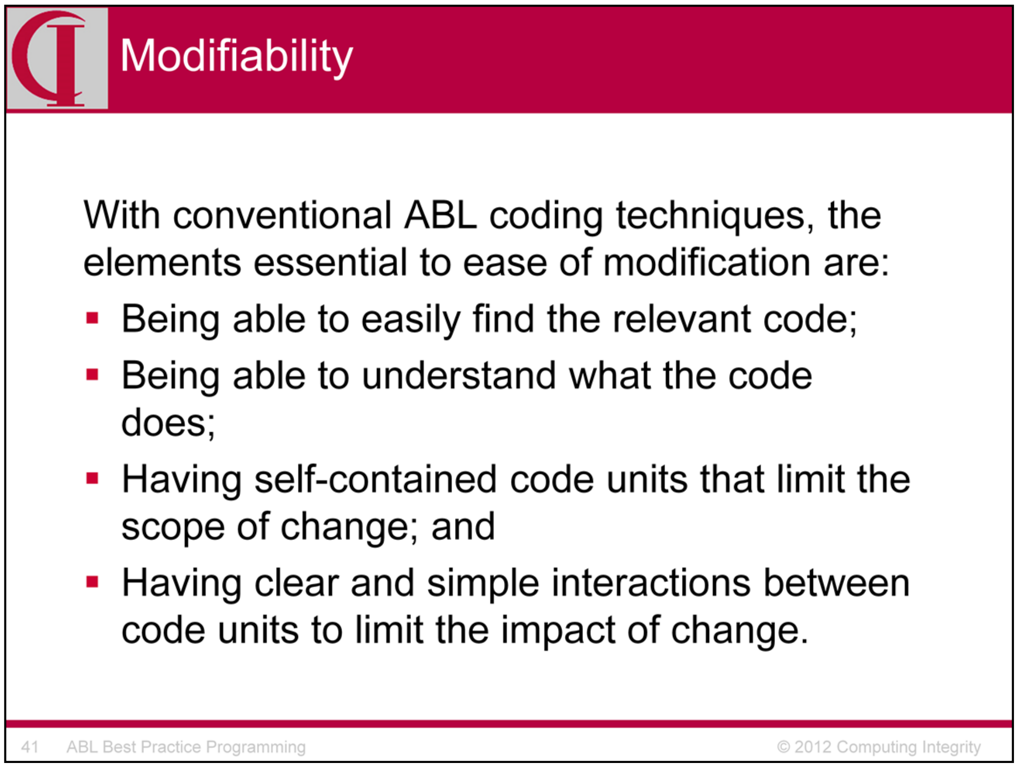


Modifiability

Many of the standards we have discussed already also contribute to modifiability, but it is in modifiability where we are likely to reap the greatest monetary benefit.

40 ABL Best Practice Programming © 2012 Computing Integrity

As I have said, the largest part of the cost of software over its lifetime is modification. Many of the standards I have talked about today also contribute to modifiability and thus contribute to cost reduction.


Modifiability

With conventional ABL coding techniques, the elements essential to ease of modification are:

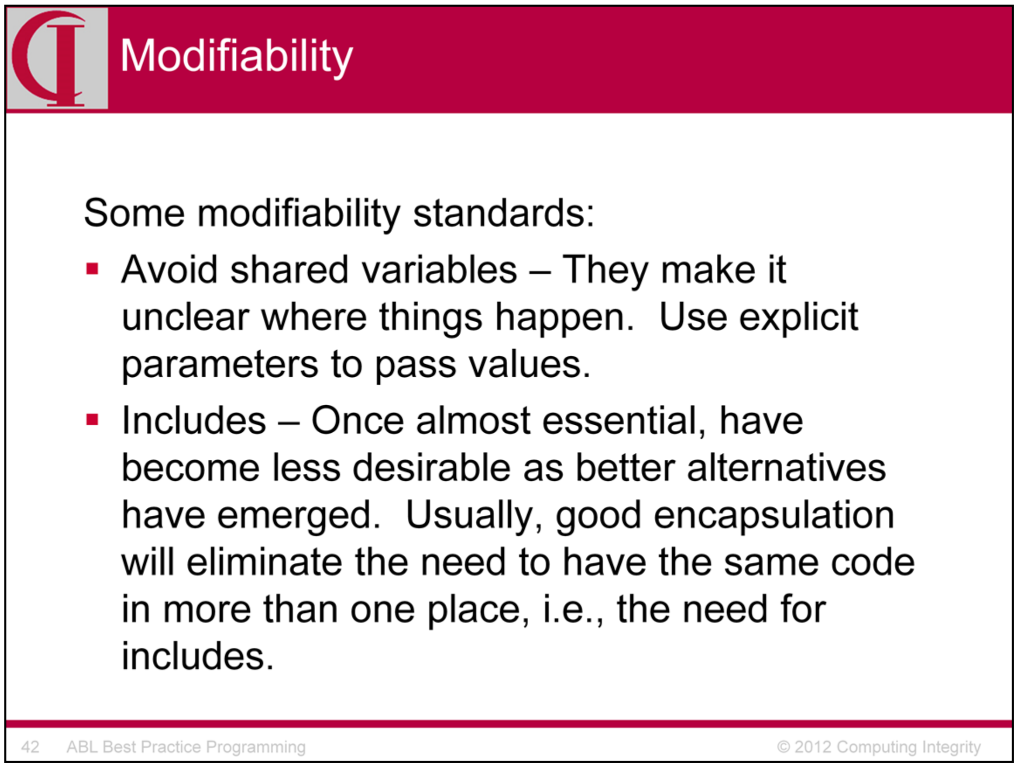
- Being able to easily find the relevant code;
- Being able to understand what the code does;
- Having self-contained code units that limit the scope of change; and
- Having clear and simple interactions between code units to limit the impact of change.

41 ABL Best Practice Programming © 2012 Computing Integrity

There are four elements which contribute to ease of modification in conventional ABL code:

- Easily finding the code that needs modification;
- Easily understanding what that code does and how it works;
- Having small, self-contained units of code so that there is only a small amount of code that needs to change for any one change in requirements; and
- Having clear and simple interactions between the units of code, again so that the scope of change is limited.

Some of you may recognize these as some of the primary motivations behind Object-Orientation.



Modifiability

Some modifiability standards:

- Avoid shared variables – They make it unclear where things happen. Use explicit parameters to pass values.
- Includes – Once almost essential, have become less desirable as better alternatives have emerged. Usually, good encapsulation will eliminate the need to have the same code in more than one place, i.e., the need for includes.

42 ABL Best Practice Programming © 2012 Computing Integrity

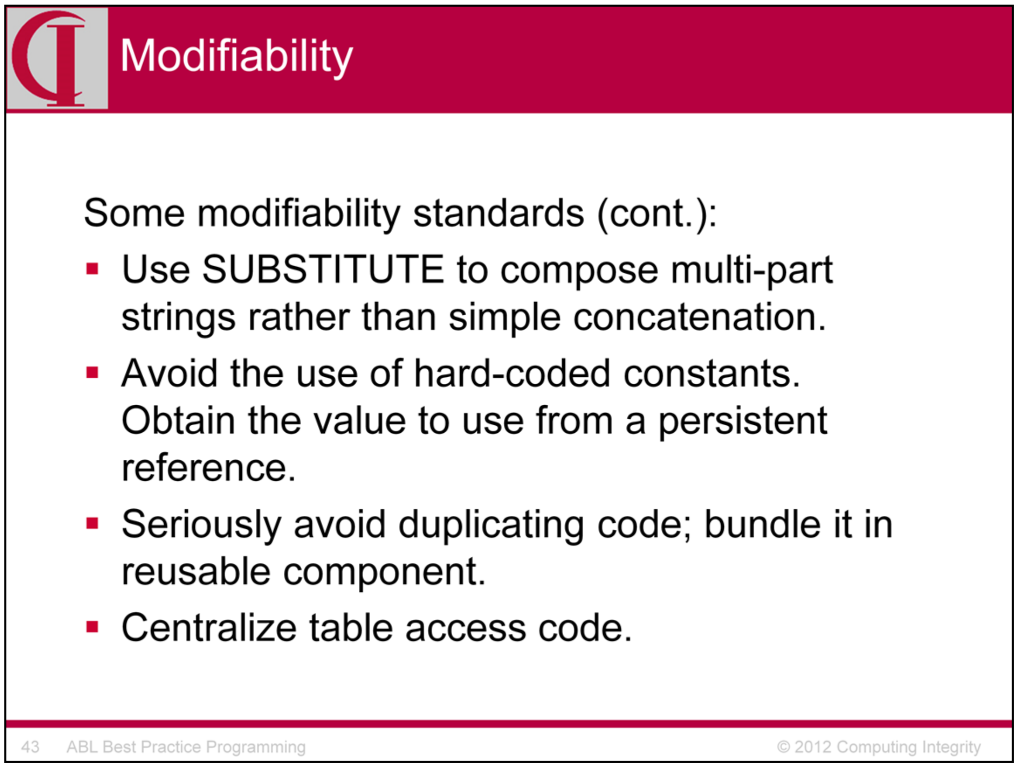
On to specific standards …

Don't use shared variables. Period. There is no case in which they are justified. They make it very hard to tell where values are modified and used. Use explicit parameters to pass values. And, I should add, by containing a function to a single code unit, there is less need to pass values other than inputs and results.

I am likely to get some push back on my next standard. In the old days, include files were the mark of good practice because they provided a way to put common code in a single place and reuse it wherever needed. Of course, there were people who abused the concept, like putting 100 variable definitions in one include and then referencing it in all programs for a function. But, we now have much better techniques for encapsulation including persistent and super procedures and classes. If you ever find yourself thinking of using an include, ask yourself why that same code needs to be in more than one place? Why can't you encapsulate it so that it only appears in one place. A classic example is using an include for a temp-table definition. Why isn't all of the code that processes the temp-table encapsulated in one code unit with inputs and outputs for those that need to use this functionality?

One big problem with includes is that one can't see the code in the include when looking at the place it is used.

One probably acceptable use of includes is language extensions, i.e., where one really wishes that ABL did something that it does not so one encapsulates that functionality in an include and uses it in-line as if it were part of the language.



Modifiability

Some modifiability standards (cont.):

- Use SUBSTITUTE to compose multi-part strings rather than simple concatenation.
- Avoid the use of hard-coded constants. Obtain the value to use from a persistent reference.
- Seriously avoid duplicating code; bundle it in reusable component.
- Centralize table access code.

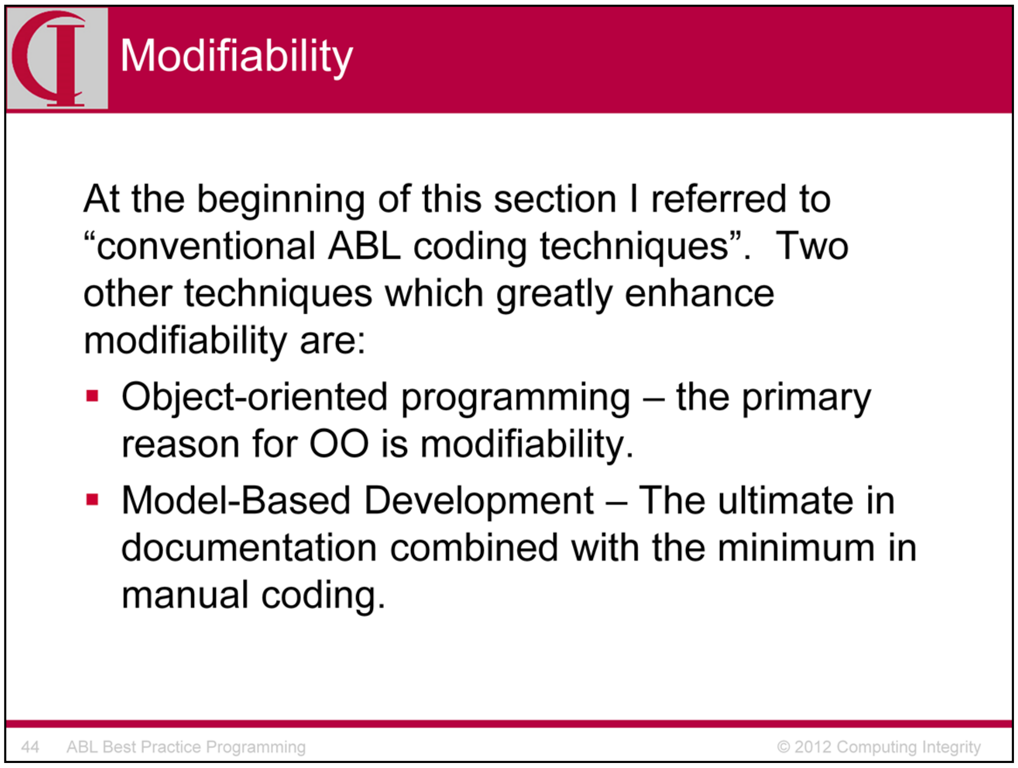
43 ABL Best Practice Programming © 2012 Computing Integrity

When you need to compose a multi-part string, e.g., for output, always use SUBSTITUTE and fill in the computed parts rather than simply concatenating the pieces of the string. This is much easier to modify and to translate.

Avoid hard coded constants (magic numbers) since, if you ever need to change them, you will need to change them manually everywhere they are used. Much better is to obtain any such value from a persistent source.

Duplicate or nearly duplicate code is a maintenance nightmare. It is very easy for the copies to get out of sync and have slightly or dramatically different behavior. Instead, bundle that code in a reusable component which can be referenced wherever needed.

Whether or not you are going to work toward a fully layered, OERA-type, architecture, consider encapsulating any table access code in a reusable component. Then, when something changes, there is only one place that needs to change instead of having to search through the entire code base for references to the table.



Modifiability

At the beginning of this section I referred to "conventional ABL coding techniques". Two other techniques which greatly enhance modifiability are:

- Object-oriented programming – the primary reason for OO is modifiability.
- Model-Based Development – The ultimate in documentation combined with the minimum in manual coding.

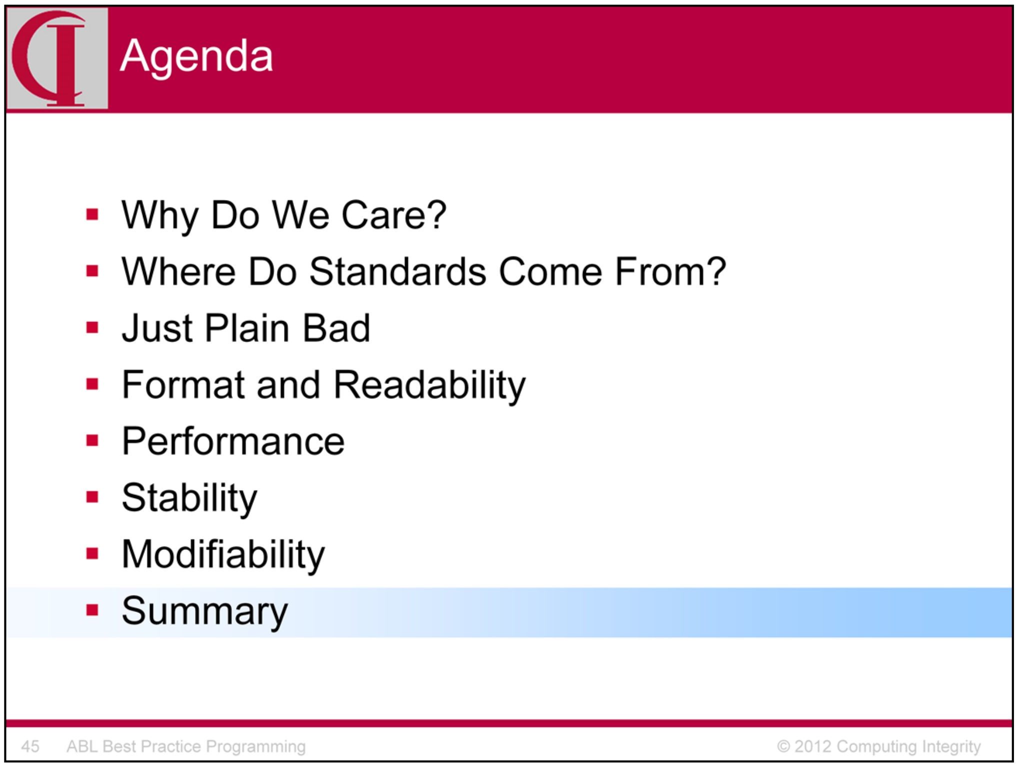
44 ABL Best Practice Programming © 2012 Computing Integrity

At the beginning of this section I referred to "conventional ABL coding techniques". Two other techniques which greatly enhance modifiability are:

Object-oriented programming – the primary reason for OO is modifiability. OO code is more easily modified both because of:

- The strong encapsulation means changes are typically confined to a single code unit and
- The more natural relationship between the code units and the elements of the problem space means that changes to one thing in the problem space tend to point to only one object.

Model-Based Development – The ultimate in documentation combined with the minimum in manual coding. This is, of course, a whole different approach which you can read more about on my website.

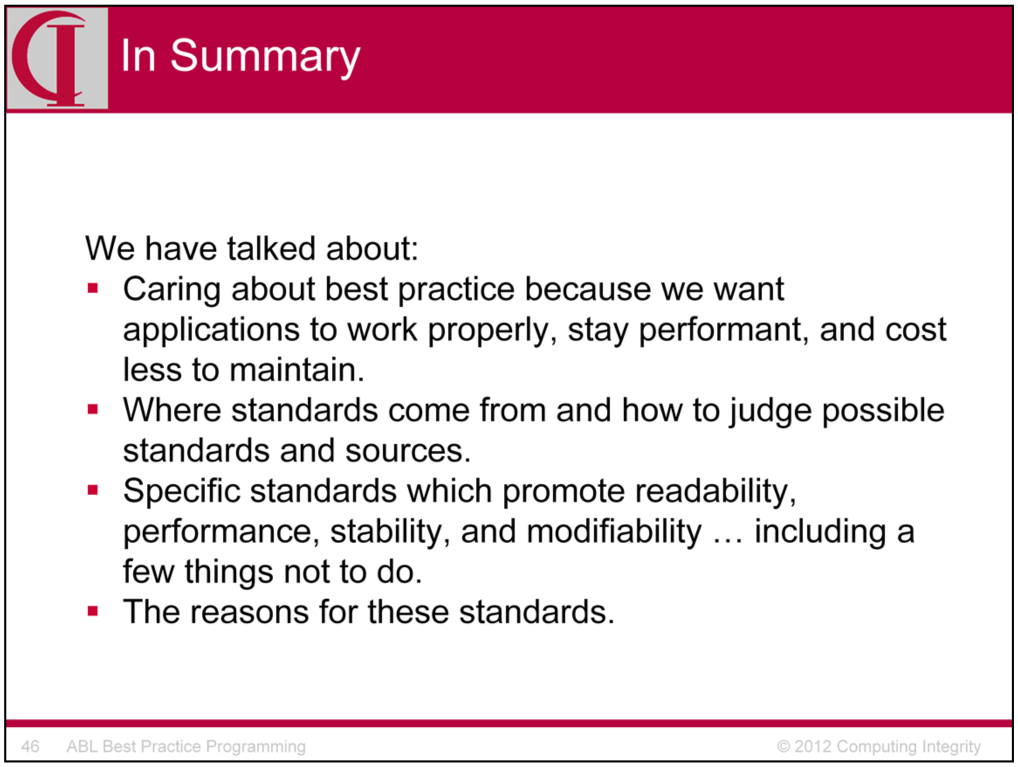


Agenda

- Why Do We Care?
- Where Do Standards Come From?
- Just Plain Bad
- Format and Readability
- Performance
- Stability
- Modifiability
- Summary

45 ABL Best Practice Programming © 2012 Computing Integrity

So, to summarize …



In Summary

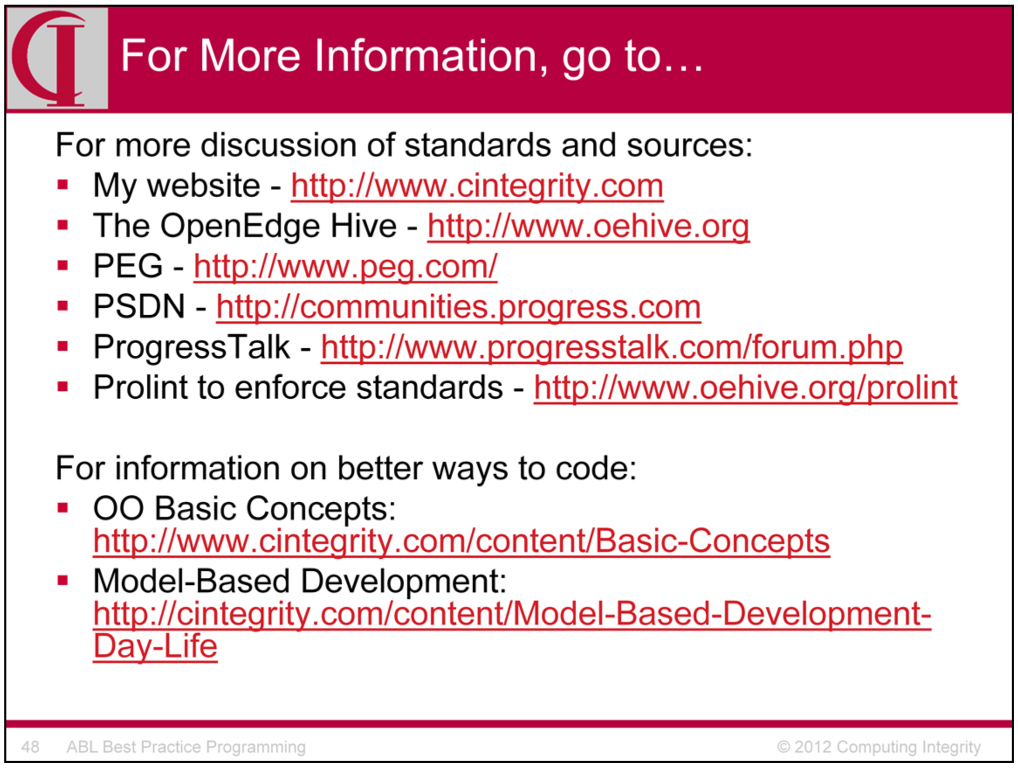
We have talked about:

- Caring about best practice because we want applications to work properly, stay performant, and cost less to maintain.
- Where standards come from and how to judge possible standards and sources.
- Specific standards which promote readability, performance, stability, and modifiability … including a few things not to do.
- The reasons for these standards.

In Summary

Not everyone will agree with everything I have said, but consider why I said it before dismissing it. You may come up with something arguably as good, but if you just ignore the idea, then your code won't be as good … and you will be costing someone money.

Keep thinking, so that you personally understand why a rule is or is not a good rule … and make sure to test any performance claims.



For More Information, go to…

For more discussion of standards and sources:

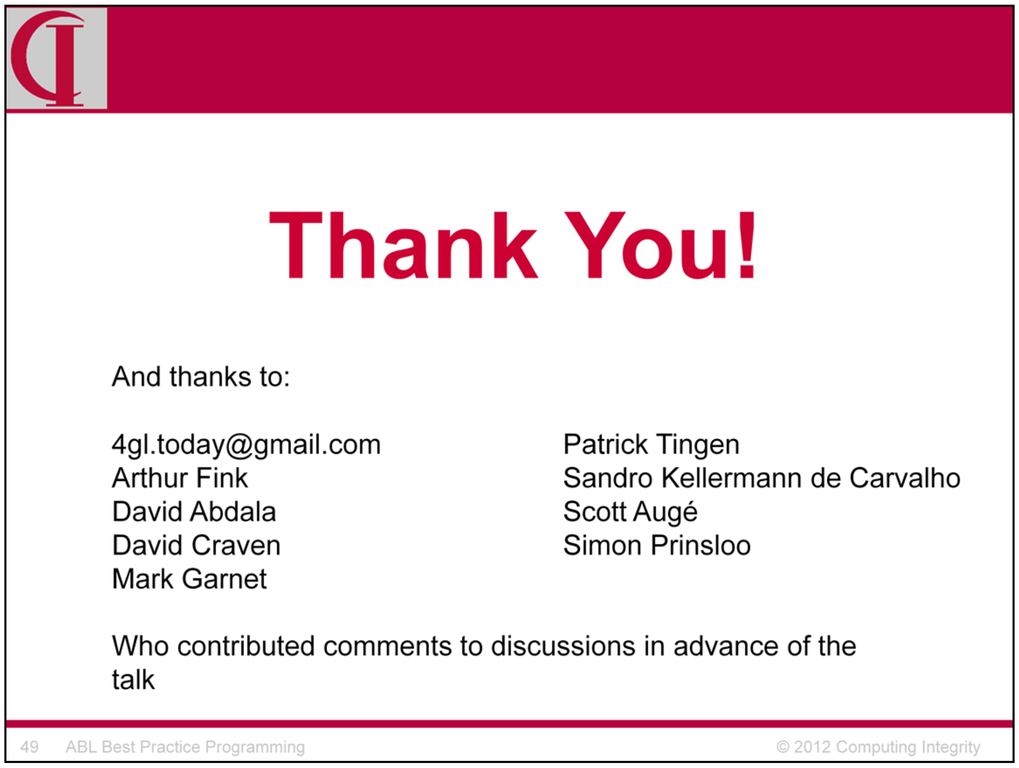
- My website - http://www.cintegrity.com
- The OpenEdge Hive - http://www.oehive.org
- PEG - http://www.peg.com/
- PSDN - http://communities.progress.com
- ProgressTalk - http://www.progresstalk.com/forum.php
- Prolint to enforce standards - http://www.oehive.org/prolint

For information on better ways to code:

- OO Basic Concepts: http://www.cintegrity.com/content/Basic-Concepts
- Model-Based Development: http://cintegrity.com/content/Model-Based-Development-Day-Life

48 ABL Best Practice Programming © 2012 Computing Integrity

Here are some links for more information.



Thank You!

And thanks to:

4gl.today@gmail.com	Patrick Tingen
Arthur Fink	Sandro Kellermann de Carvalho
David Abdala	Scott Augé
David Craven	Simon Prinsloo
Mark Garnet	

Who contributed comments to discussions in advance of the talk

Thank you.

Questions ?

For more information:

http://www.cintegrity.com

thomas@cintegrity.com

217-355-4469

50 ABL Best Practice Programming © 2012 Computing Integrity

And now for questions.