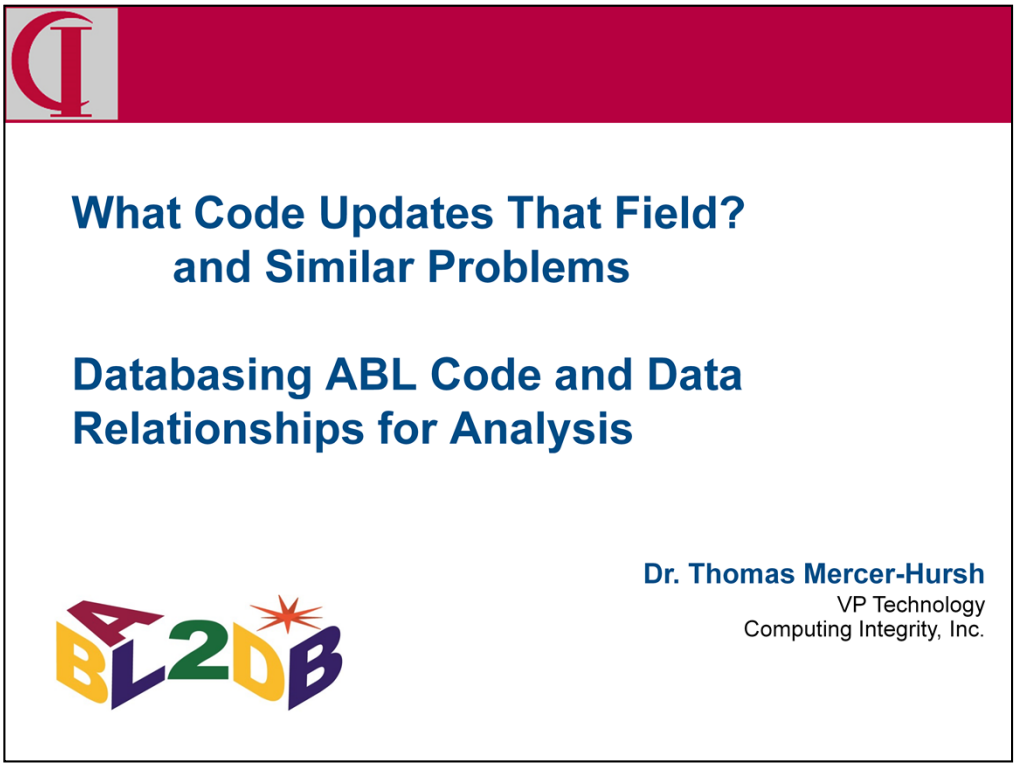
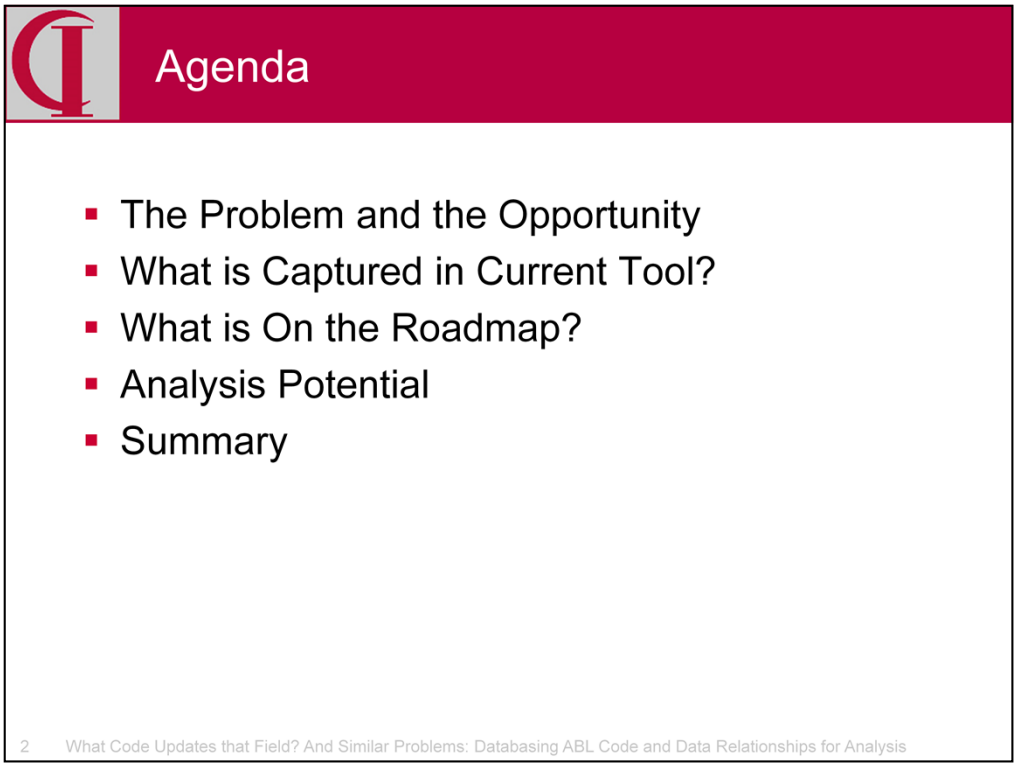
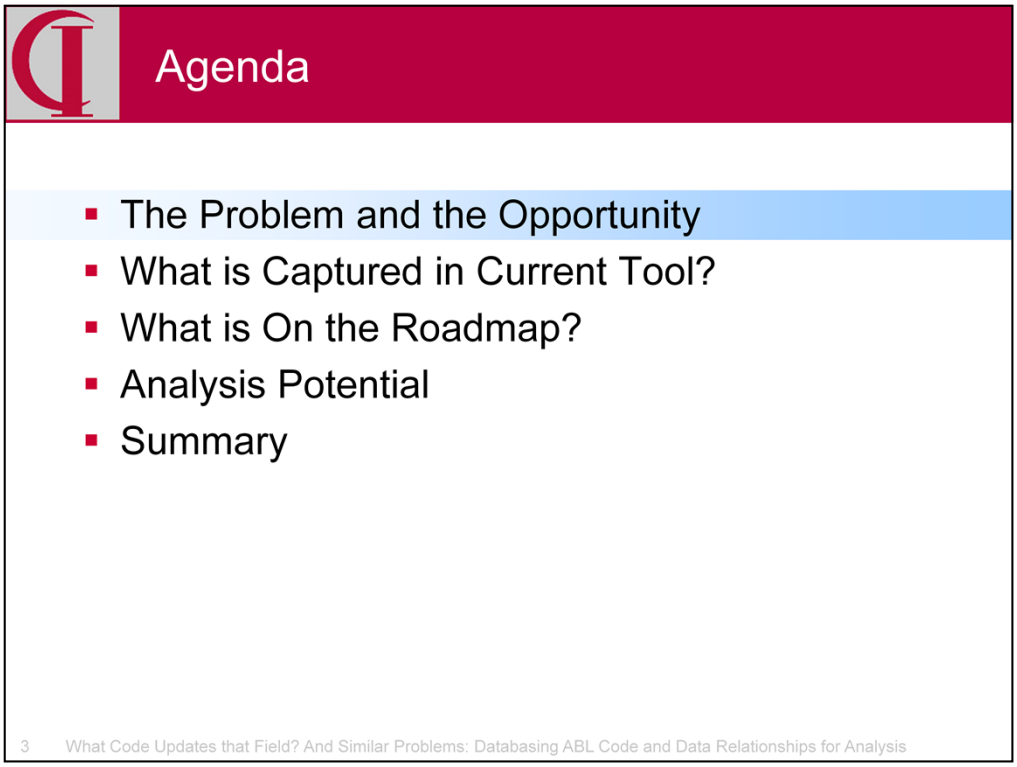


Let me begin by introducing myself. I have been a Progress Application Partner since 1986 and for many years I was the architect and chief developer for our ERP application. In recent years I have refocused on the problems of transforming and modernizing legacy ABL applications. This implies needing to understand what the existing legacy application is doing so that we can change or replace it with confidence. Today I am going to tell you about an open source tool I have been building.



Here is our agenda for today. First we are going to talk a little bit about the background, then about what is in the current version of the tool, the future additions to the tool, and finally the potential for analysis.



First, let's talk a bit about the nature of the problem and what opportunities there are for addressing the problem.

The Problem and the Opportunity

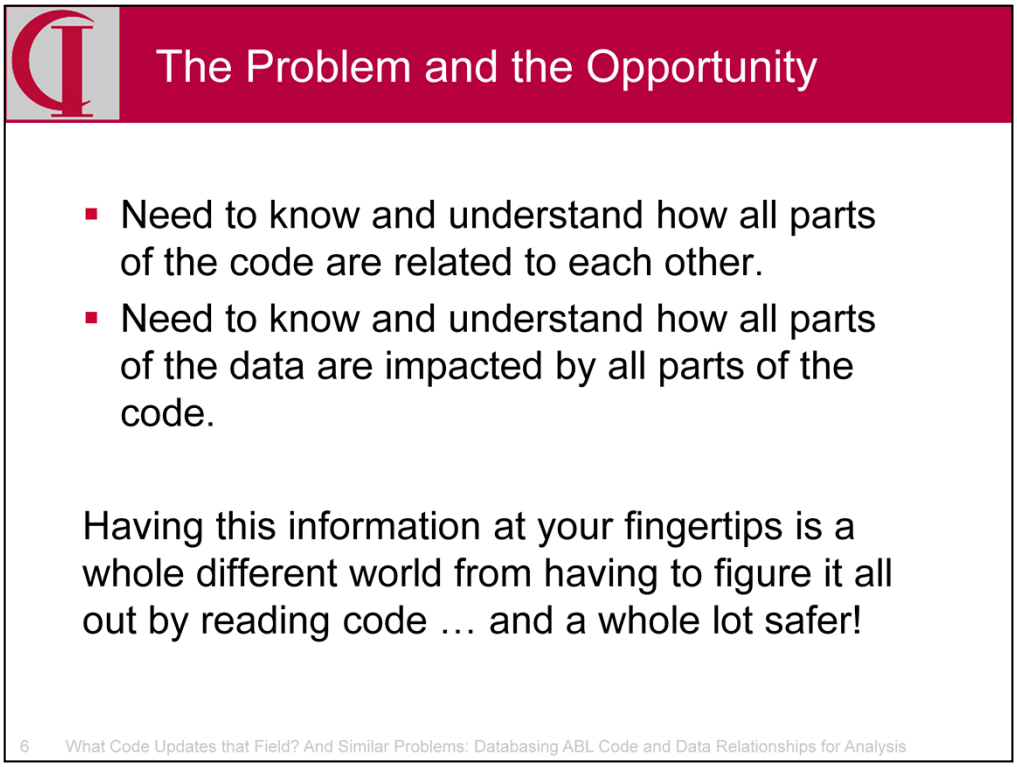
- There is lots and lots of legacy ABL code out there.
- Very little of it is meaningfully documented.
- Knowledge is very often based only on experience, which can be incomplete, unavailable, or even gone.

I'm sure that most of you who have been around for a while can identify the problem … large legacy ABL systems, often millions of lines of code, documentation which could be written on three napkins, and typically a reliance on some old hand who has been around for years to help everyone else figure out where things are and how they work … until he or she isn't there any more.

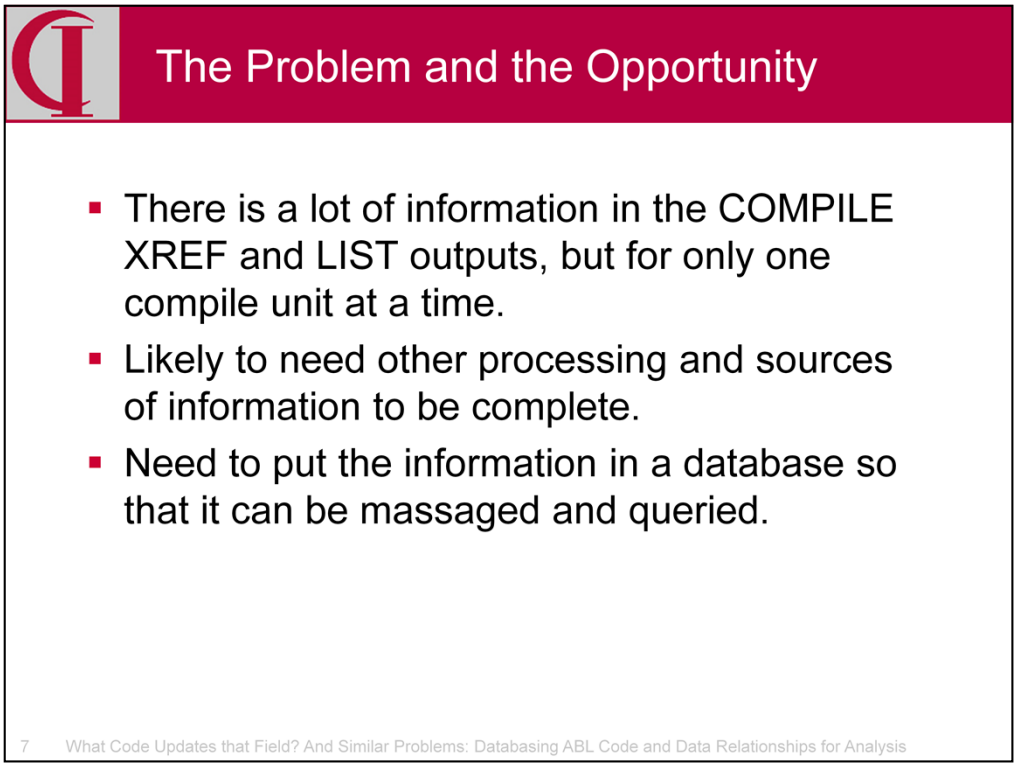
The Problem and the Opportunity

- Making changes safely requires solid knowledge of impact.
- Mistakes can be very costly and fixing them can be even more expensive than the original work.
- Worst cases can impact the integrity of the data.
- Business impact often exceeds the technical cost.

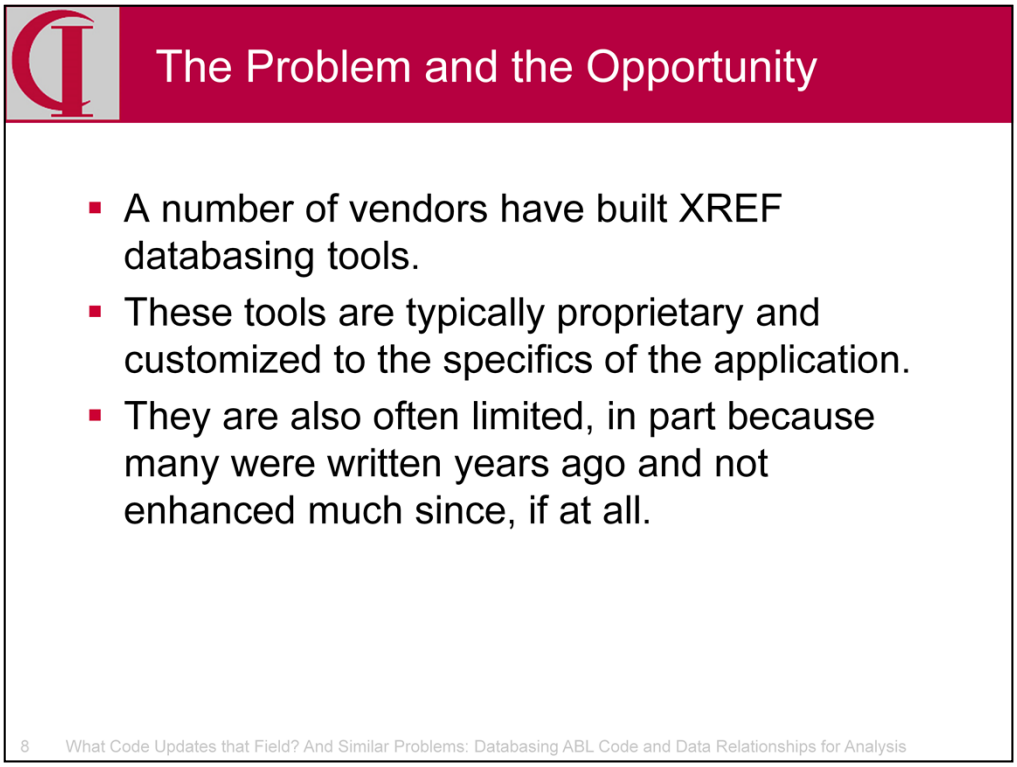
And we all know that when we make changes, we should understand what we are doing before we start … what will be impacted by any given change. But, without documentation, there isn't really any way to do that and as a result we make mistakes. Those mistakes often have a big impact, even on running the business. Not knowing the impact, we don't really even know what to test — so the problem can be in some area unrelated to where we did the work. In the worst cases, the cost impact to business operations exceeds all of the technical expense invested in making the original change and in repairing the damage.



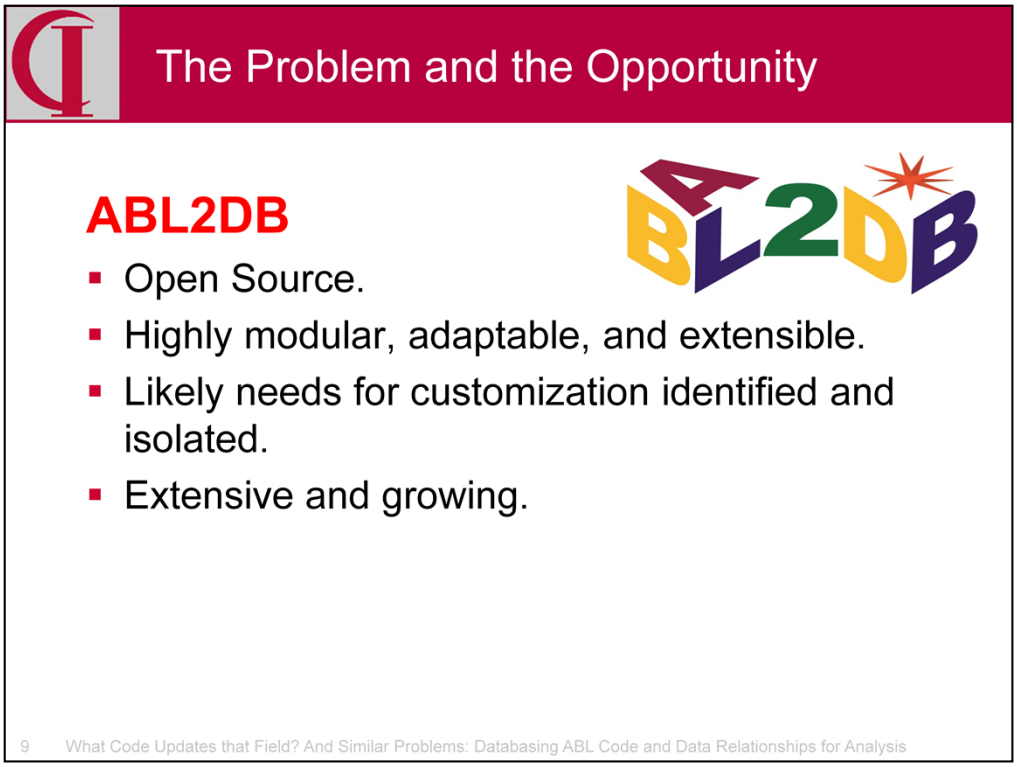
To make changes safely we need to understand how all parts of the code interact with other parts of the code and how all parts of the data are impacted by all parts of the code. We don't necessarily need to know everything all the time, but we do need to know anything relevant to the current project, so somewhere we need to have that knowledge. Having this information at your fingertips is a whole different world from having to figure it all out by reading code.



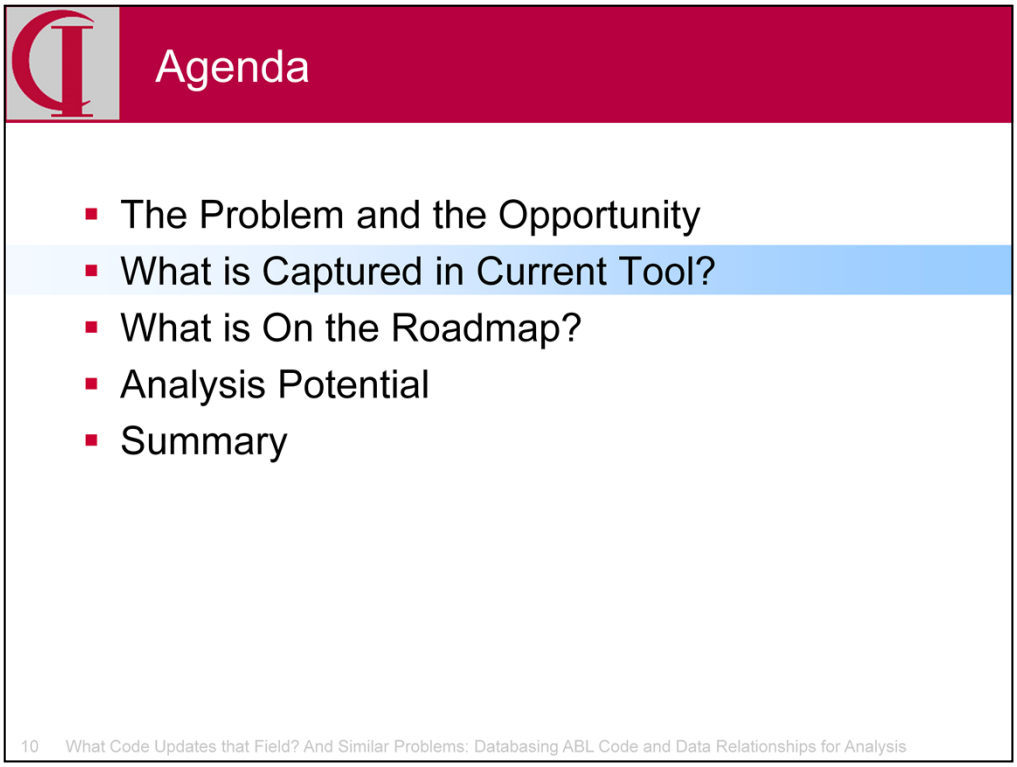
One of the sources of information accessible to all of us is the output of COMPILE XREF and COMPILE LIST. Both contain useful information, though not always in the most accessible form. And, the information is about only one compile unit at a time. Moreover, clearly, all the information we want is not there, so we need access to other sources. And, we need to capture that information in a database so that we can get at any and all of it efficiently and quickly.



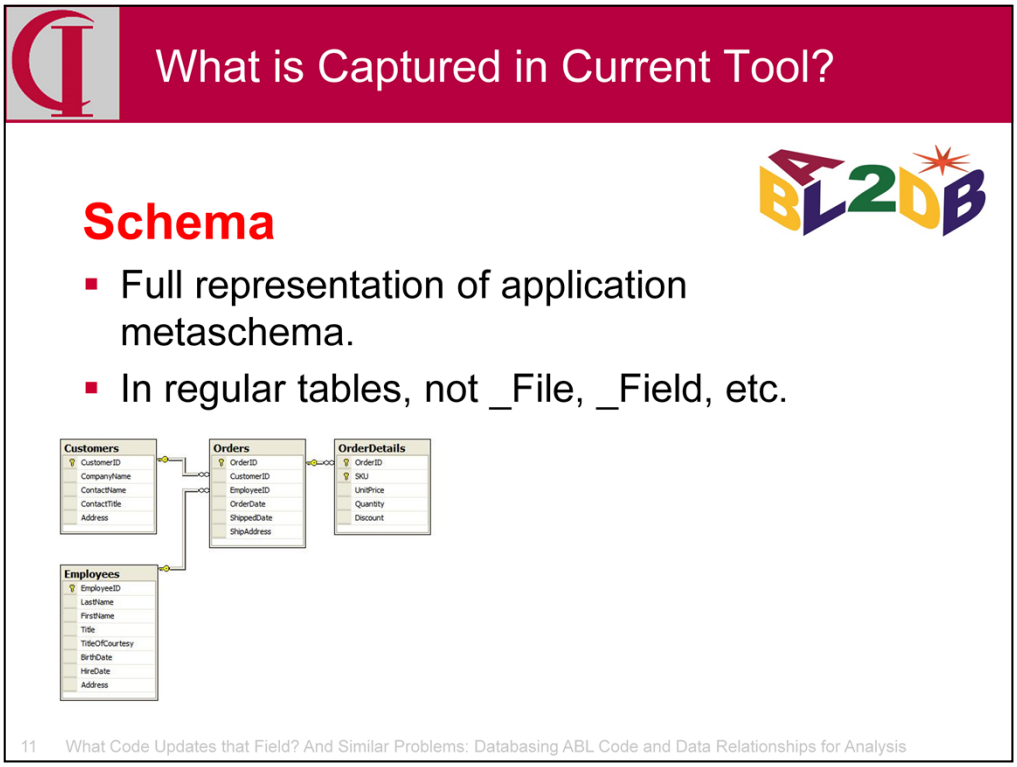
Of course, this is not a new idea. Many vendors have built some form of engine to capture and store XREF data, but these are proprietary systems not accessible to the general community and are often quite tailored to the specifics of the application for which they were defined. And, many are quite limited in the information they gather, in part because that is all the designers tackled and in part because some date back to the 1980's when there was less information available.



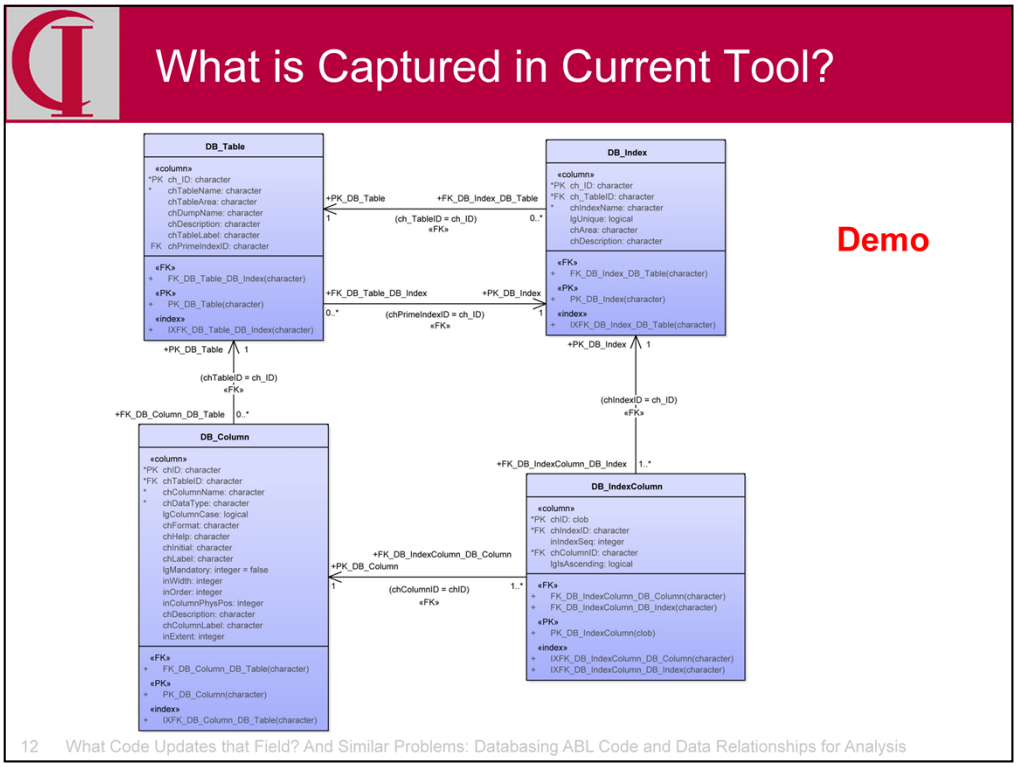
Which brings me to ABL2DB, my own offering for tackling this problem. It is open source and has been designed to be highly modular, highly adaptable, and easily extended. There are some components which, by necessity, need to be tailored to the individual code base, but these have been isolated, identified, and packaged for easy customization. As you will see, the current version is already collecting a substantial amount of information and the roadmap is in place for extensions soon.



So, let's take a look at what is in the code today.



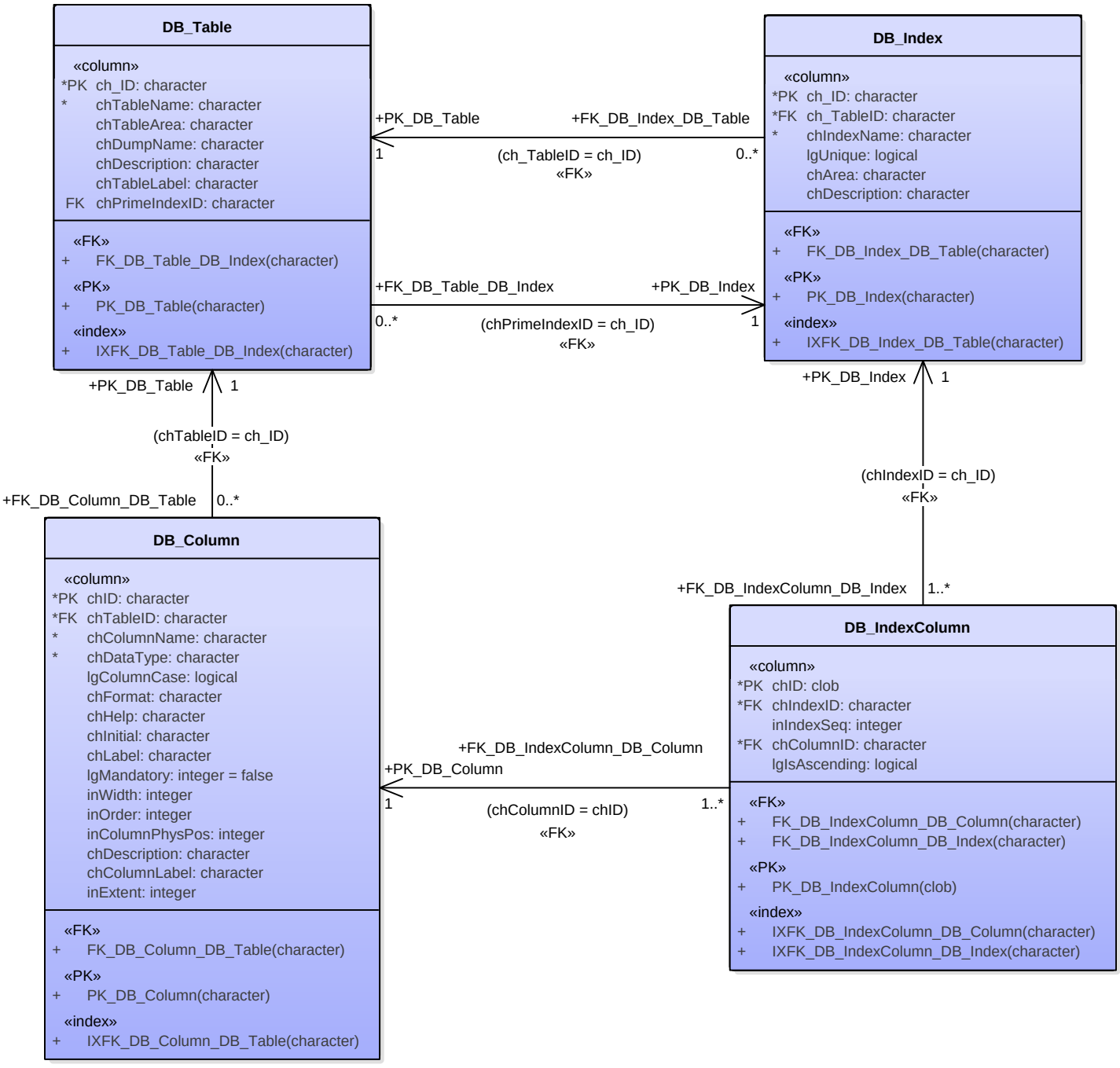
I created a set of database tables to mirror _File, _Field, etc. because those files will have the analysis schema in our target database. The fields provided are very complete and a program is provided to load these from a .df. The load program doesn't yet cover every possible option because I don't have a .df sample with everything to work from, but the structure is very easily expanded to any new dump syntax.

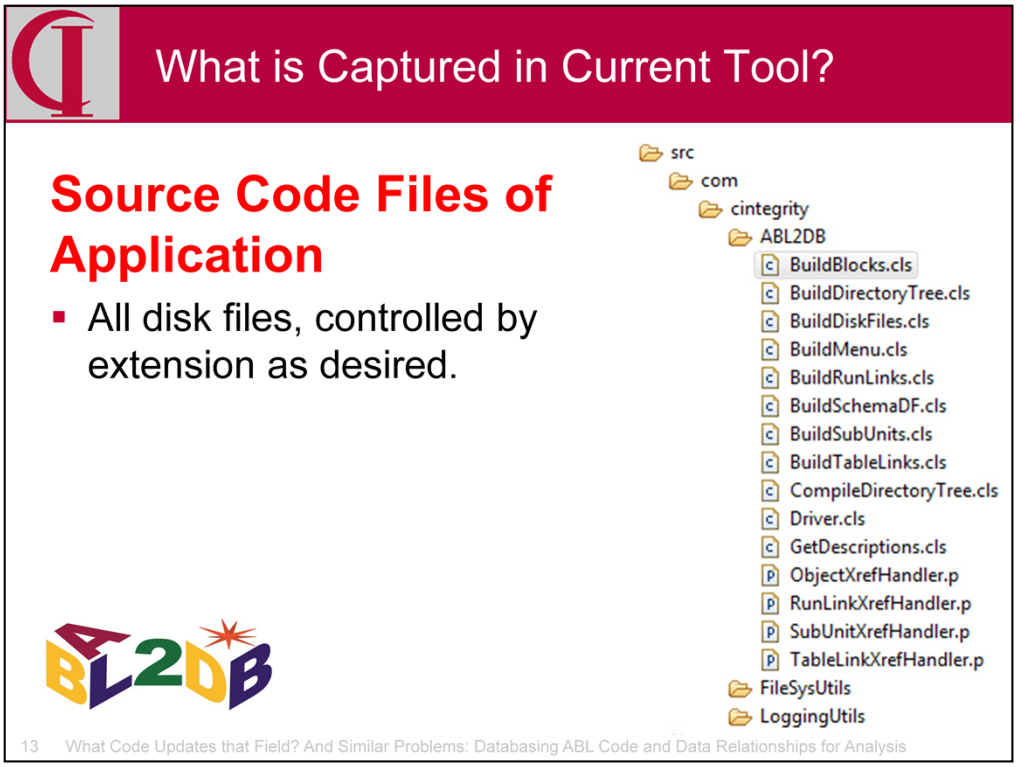


DEMO: So, let's take a quick look in Enterprise Architect at the schema of these tables, which should be familiar to most of you from the parallel to metaschema tables. This diagram does not show all possible fields, only those which are the most likely to be relevant to the typical application. In this and the following diagrams I am not going to go over all of the details since that would be both boring and time consuming, but the details will be in the published presentation.

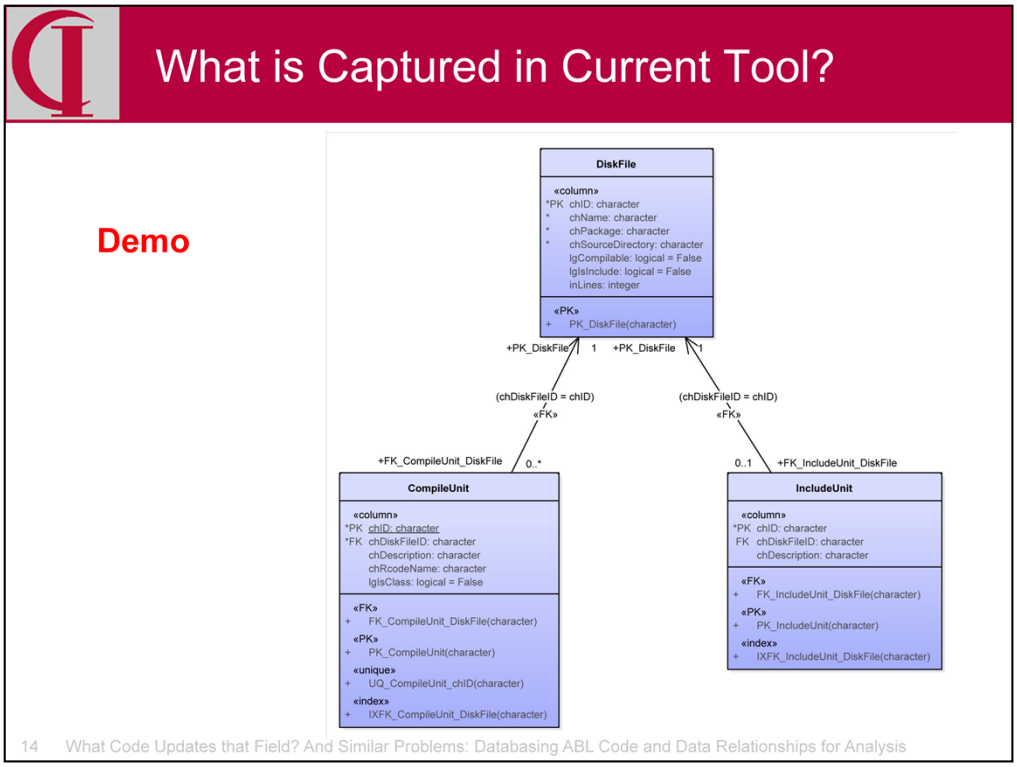
In the upper right we have DB_Table which holds the information from _File. I took the opportunity to use more modern nomenclature and make some of the field names less cryptic. Connected to it is DB_Column in the lower left. Then, in the upper right there is DB_Index which connects to DB_Table and below it DB_IndexColumn which points to the columns used in the index. Note that I am creating one additional column in every table for the RECID so that any joins by RECID can be recorded.

Performed as a demo so that it was possible to zoom in on the individual tables and read at least some of the text.

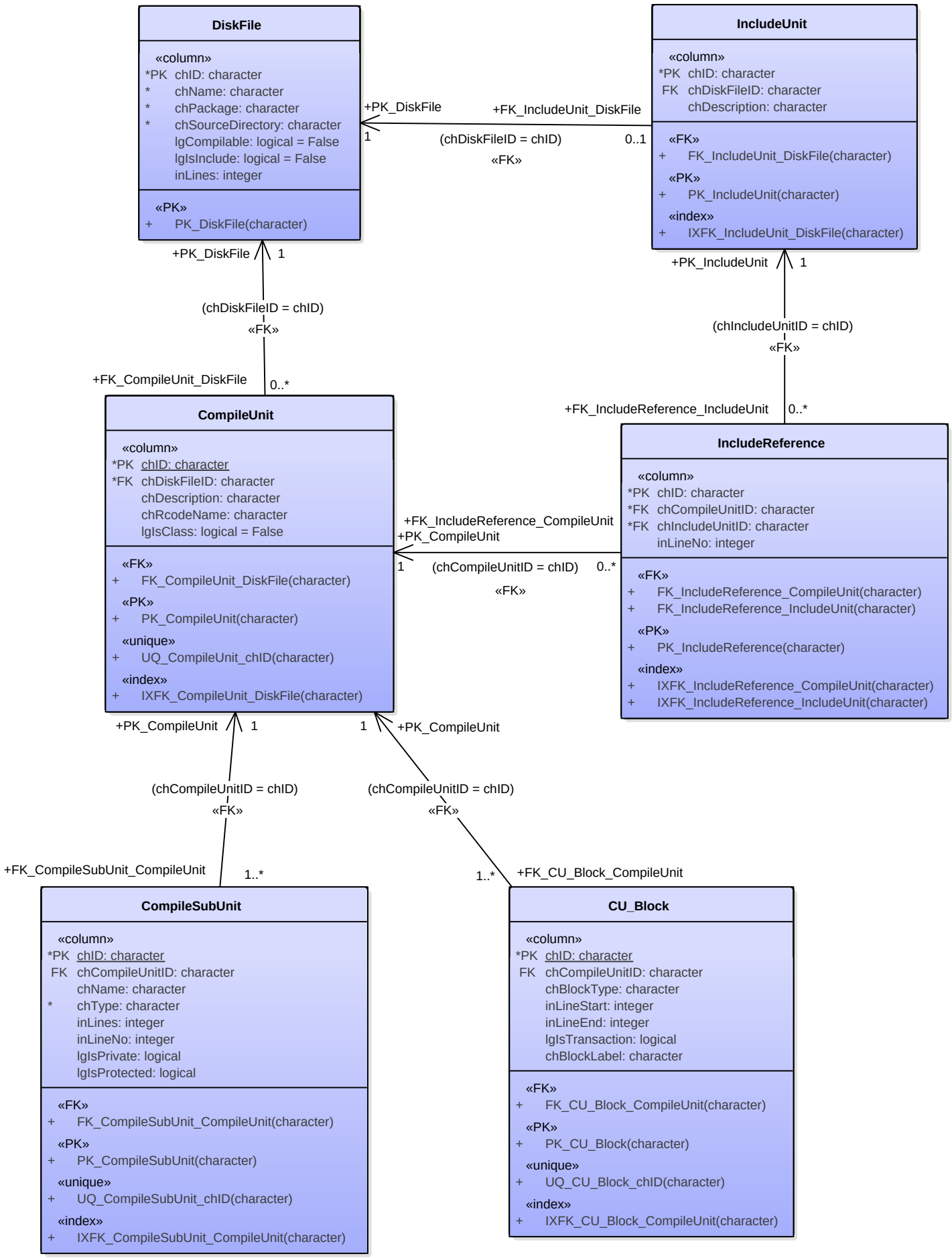

DB_Table
«column»
*PK ch_ID: character
* chTableName: character
chTableArea: character
chDumpName: character
chDescription: character
chTableLabel: character
FK chPrimeIndexID: character
«FK»
+ FK_DB_Table_DB_Index(character)
«PK»
+ PK_DB_Table(character)
«index»
+ IXFK_DB_Table_DB_Index(character)
DB_Index
«column»
*PK ch_ID: character
*FK ch_TableID: character
* chIndexName: character
lgUnique: logical
chArea: character
chDescription: character
«FK»
+ FK_DB_Index_DB_Table(character)
«PK»
+ PK_DB_Index(character)
«index»
+ IXFK_DB_Index_DB_Table(character)
+PK_DB_Table
1
+FK_DB_Index_DB_Table
0..*
(ch_TableID = ch_ID)
«FK»
+FK_DB_Table_DB_Index
0..*
+PK_DB_Index
1
(chPrimeIndexID = ch_ID)
«FK»
+PK_DB_Table
1
(chTableID = ch_ID)
«FK»
+FK_DB_Column_DB_Table
0..*
+PK_DB_Index
1
(chIndexID = ch_ID)
«FK»
DB_Column
«column»
*PK chID: character
*FK chTableID: character
* chColumnName: character
* chDataType: character
lgColumnCase: logical
chFormat: character
chHelp: character
chInitial: character
chLabel: character
lgMandatory: integer = false
inWidth: integer
inOrder: integer
inColumnPhysPos: integer
chDescription: character
chColumnLabel: character
inExtent: integer
«FK»
+ FK_DB_Column_DB_Table(character)
«PK»
+ PK_DB_Column(character)
«index»
+ IXFK_DB_Column_DB_Table(character)
+FK_DB_IndexColumn_DB_Index
1..*
DB_IndexColumn
«column»
*PK chID: clob
*FK chIndexID: character
inIndexSeq: integer
*FK chColumnID: character
lgIsAscending: logical
«FK»
+ FK_DB_IndexColumn_DB_Column(character)
+ FK_DB_IndexColumn_DB_Index(character)
«PK»
+ PK_DB_IndexColumn(clob)
«index»
+ IXFK_DB_IndexColumn_DB_Column(character)
+ IXFK_DB_IndexColumn_DB_Index(character)
+FK_DB_IndexColumn_DB_Column
+PK_DB_Column
1
(chColumnID = chID)
«FK»
1..*

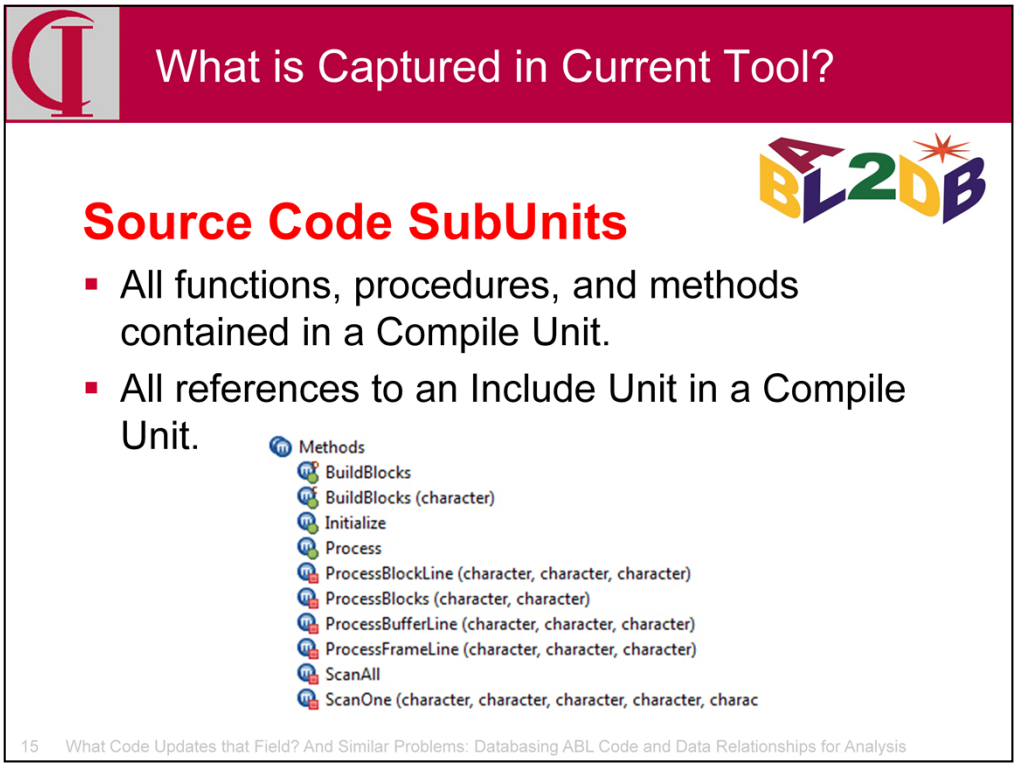


Second, we are capturing basic information about all source code files in the application. Filters are provided so that you can define what is and isn't source in the same directory tree in case you have non-source files mixed in with the source. It even handles old Varnet systems where there is source with no extension.

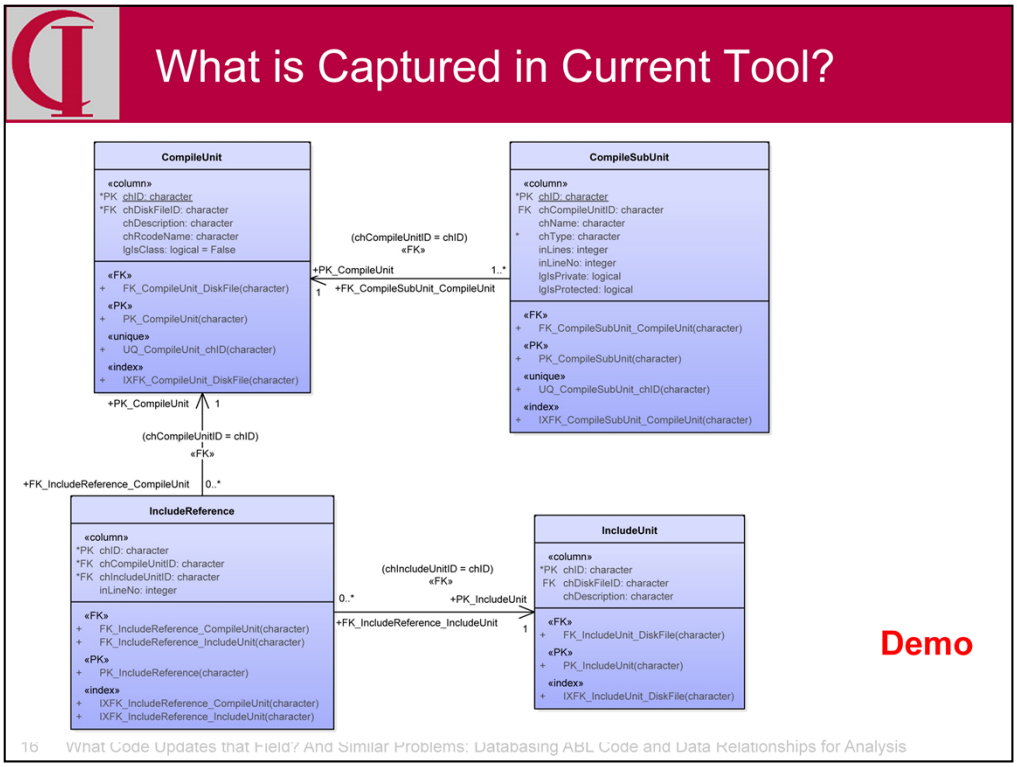


DEMO: Every distinct source code file on disk is identified as a DiskFile. This list is built after a full compile so that the test of whether a DiskFile is compilable is whether or not there is a corresponding .r. Include files are likewise empirically identified in a later pass. Compile units are identified as being classes on the basis of the extension .cls.

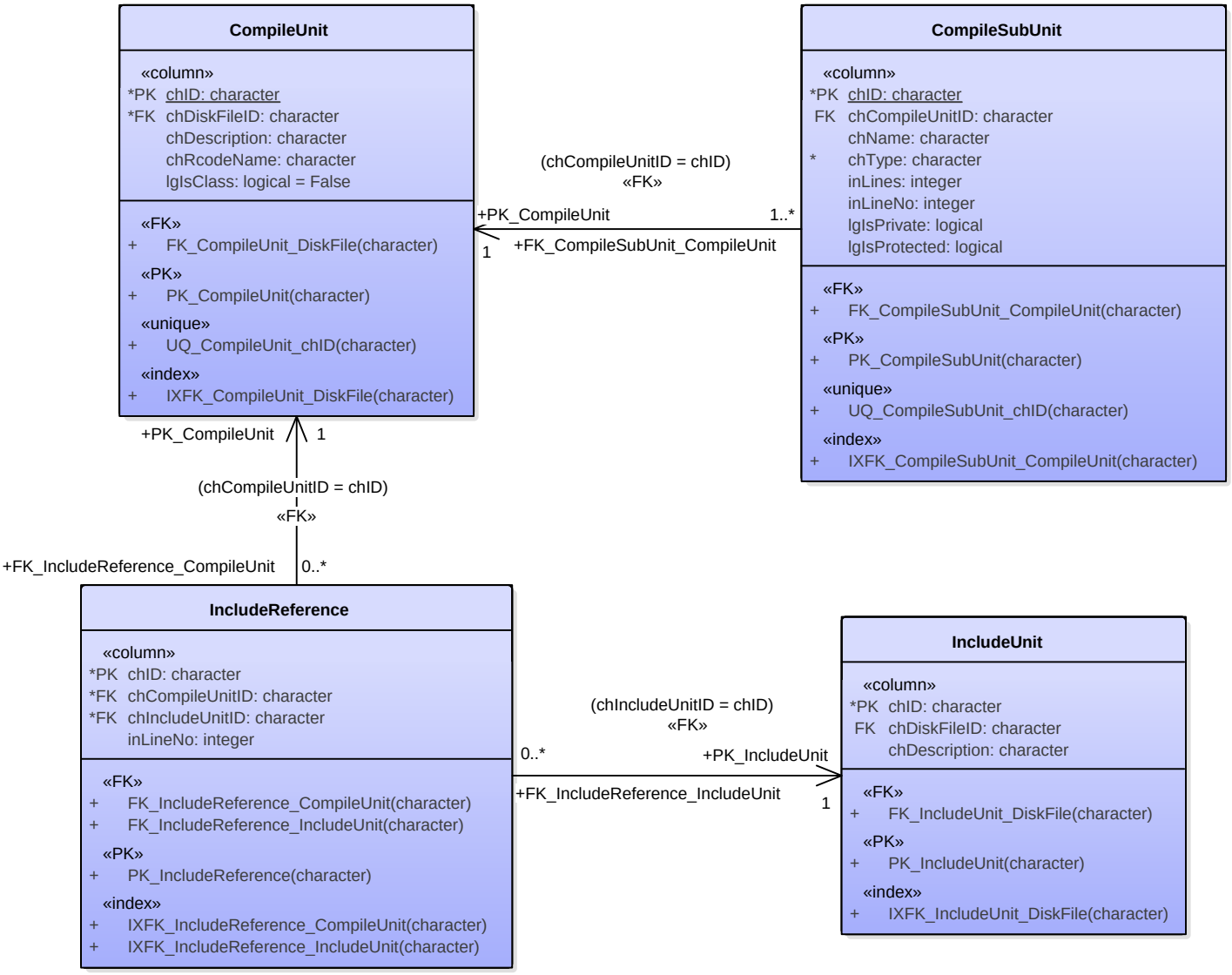

DiskFile
«column»
*PK chID: character
* chName: character
* chPackage: character
* chSourceDirectory: character
lgCompilable: logical = False
lgIsInclude: logical = False
inLines: integer
«PK»
+ PK_DiskFile(character)
IncludeUnit
«column»
*PK chID: character
FK chDiskFileID: character
chDescription: character
«FK»
+ FK_IncludeUnit_DiskFile(character)
«PK»
+ PK_IncludeUnit(character)
«index»
+ IXFK_IncludeUnit_DiskFile(character)
+PK_DiskFile
1
+FK_IncludeUnit_DiskFile
(chDiskFileID = chID)
«FK»
0..1
+PK_DiskFile
1
(chDiskFileID = chID)
«FK»
+FK_CompileUnit_DiskFile
0..*
+PK_IncludeUnit
1
(chIncludeUnitID = chID)
«FK»
+FK_IncludeReference_IncludeUnit
0..*
CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
IncludeReference
«column»
*PK chID: character
*FK chCompileUnitID: character
*FK chIncludeUnitID: character
inLineNo: integer
«FK»
+ FK_IncludeReference_CompileUnit(character)
+ FK_IncludeReference_IncludeUnit(character)
«PK»
+ PK_IncludeReference(character)
«index»
+ IXFK_IncludeReference_CompileUnit(character)
+ IXFK_IncludeReference_IncludeUnit(character)
+FK_IncludeReference_CompileUnit
+PK_CompileUnit
1
(chCompileUnitID = chID)
«FK»
0..*
+PK_CompileUnit
1
1
+PK_CompileUnit
(chCompileUnitID = chID)
«FK»
(chCompileUnitID = chID)
«FK»
+FK_CompileSubUnit_CompileUnit
1..*
1..*
+FK_CU_Block_CompileUnit
CompileSubUnit
«column»
*PK chID: character
FK chCompileUnitID: character
chName: character
* chType: character
inLines: integer
inLineNo: integer
lgIsPrivate: logical
lgIsProtected: logical
«FK»
+ FK_CompileSubUnit_CompileUnit(character)
«PK»
+ PK_CompileSubUnit(character)
«unique»
+ UQ_CompileSubUnit_chID(character)
«index»
+ IXFK_CompileSubUnit_CompileUnit(character)
CU_Block
«column»
*PK chID: character
FK chCompileUnitID: character
chBlockType: character
inLineStart: integer
inLineEnd: integer
lgIsTransaction: logical
chBlockLabel: character
«FK»
+ FK_CU_Block_CompileUnit(character)
«PK»
+ PK_CU_Block(character)
«unique»
+ UQ_CU_Block_chID(character)
«index»
+ IXFK_CU_Block_CompileUnit(character)



The next step breaks down Compile Units into sub components including procedures, methods and functions. It is this pass which also creates the links for empirical include files. For non-class Compile Units, a SubUnit corresponding to MainBlock is also created for code not otherwise enclosed in a SubUnit.



DEMO: So, here on the upper left we have the Compile Unit we identified previously. On the right we have the CompileSubUnit which are the procedures, functions, and methods. All are in the same table, identified by type. Public, Private, and Protected are recognized. Then, in the lower left is the IncludeReference which connects an IncludeUnit with each of the CompileUnits in which it is used. Note the need for the association here that is not needed with CompileSubUnit since those are limited to a single CompileUnit.


CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
CompileSubUnit
«column»
*PK chID: character
FK chCompileUnitID: character
chName: character
* chType: character
inLines: integer
inLineNo: integer
lgIsPrivate: logical
lgIsProtected: logical
«FK»
+ FK_CompileSubUnit_CompileUnit(character)
«PK»
+ PK_CompileSubUnit(character)
«unique»
+ UQ_CompileSubUnit_chID(character)
«index»
+ IXFK_CompileSubUnit_CompileUnit(character)
(chCompileUnitID = chID)
«FK»
+PK_CompileUnit
1..*
1
+FK_CompileSubUnit_CompileUnit
+PK_CompileUnit
1
(chCompileUnitID = chID)
«FK»
+FK_IncludeReference_CompileUnit
0..*
IncludeReference
«column»
*PK chID: character
*FK chCompileUnitID: character
*FK chIncludeUnitID: character
inLineNo: integer
«FK»
+ FK_IncludeReference_CompileUnit(character)
+ FK_IncludeReference_IncludeUnit(character)
«PK»
+ PK_IncludeReference(character)
«index»
+ IXFK_IncludeReference_CompileUnit(character)
+ IXFK_IncludeReference_IncludeUnit(character)
(chIncludeUnitID = chID)
«FK»
0..*
+PK_IncludeUnit
+FK_IncludeReference_IncludeUnit
1
IncludeUnit
«column»
*PK chID: character
FK chDiskFileID: character
chDescription: character
«FK»
+ FK_IncludeUnit_DiskFile(character)
«PK»
+ PK_IncludeUnit(character)
«index»
+ IXFK_IncludeUnit_DiskFile(character)

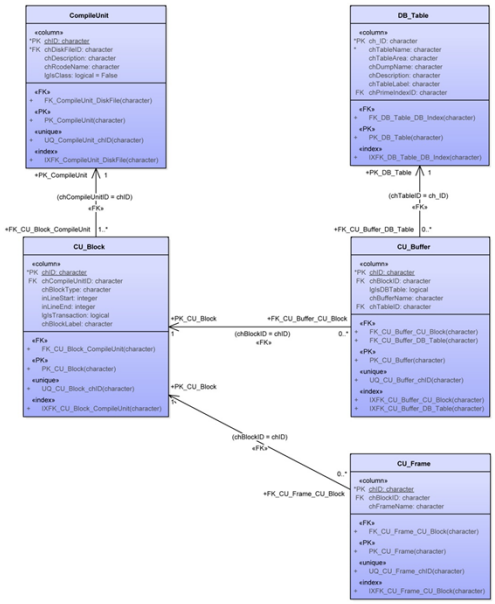
What is Captured in Current Tool?

Source Code Blocks

- All functions, procedures, and methods contained in a Compile Unit.
- All DO, FOR, REPEAT, PROCEDURE, FUNCTION, METHOD, and program main blocks.
- All FRAME and BUFFER scoping
- Transaction scopes.

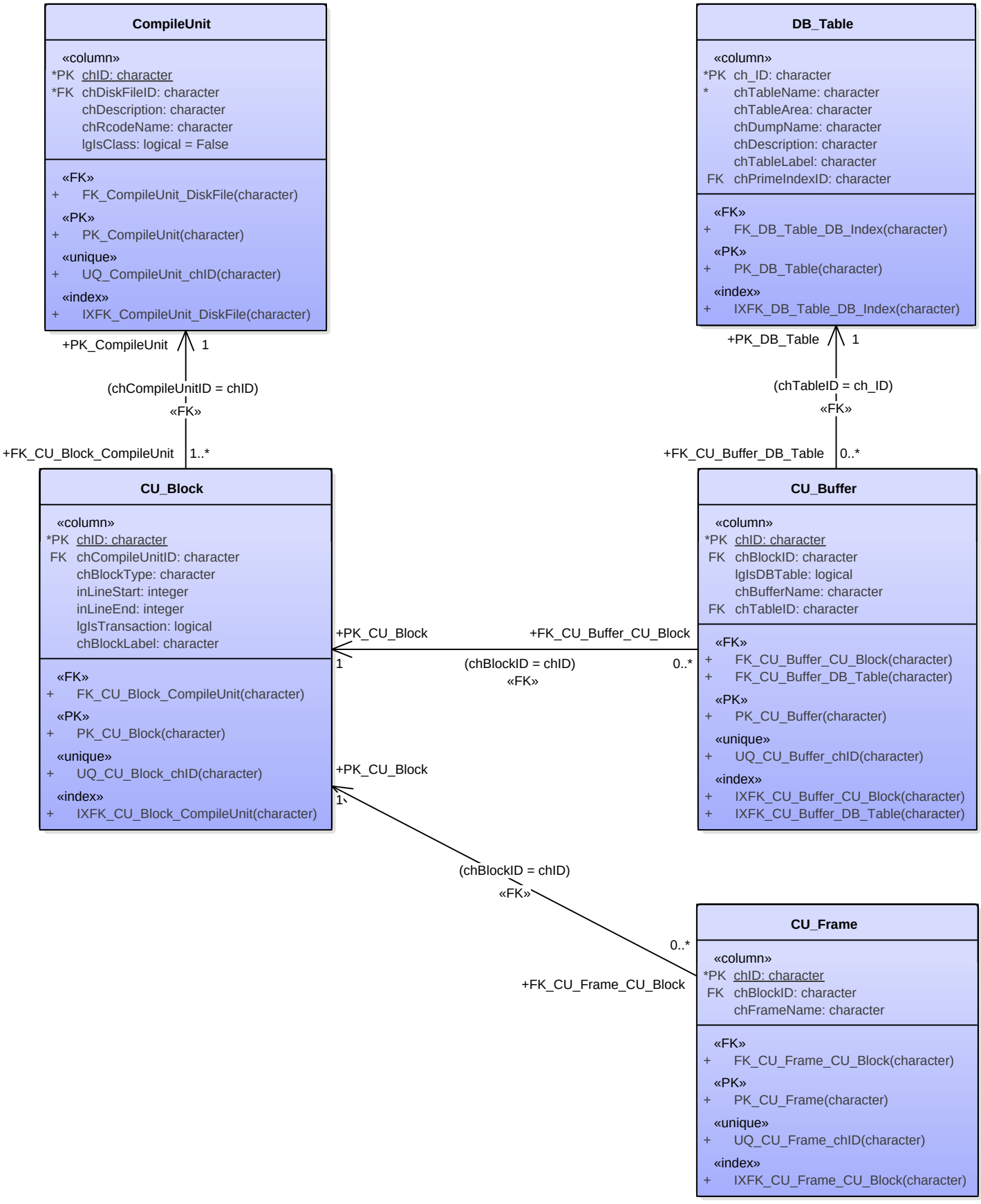
In addition to the functional decomposition of Compile SubUnits, a scan is also made to record all blocks of any type including DO, FOR, REPEAT, PROCEDURE, FUNCTION , and METHOD. In the process, it is recorded whether a transaction is scoped to the block and any frames and buffers scoped to the block.

What is Captured in Current Tool?



Demo

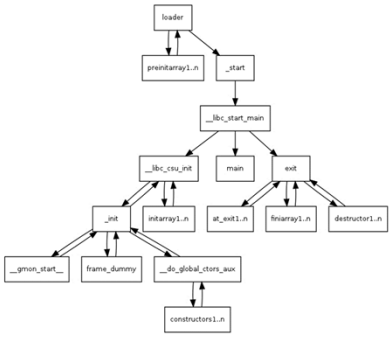
DEMO: In the upper left we have the CompileUnit from before. Below this we have the block. Some of these blocks will be Compile SubUnits as well, but many will not. These will be connected in a later analysis. When there is a buffer scoped to the block, you can see here on the right the buffer and its name. When the buffer is for a database table, the buffer is linked to that table. Finally, in the lower right we have any frames scoped to the block.


CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
DB_Table
«column»
*PK ch_ID: character
* chTableName: character
chTableArea: character
chDumpName: character
chDescription: character
chTableLabel: character
FK chPrimeIndexID: character
«FK»
+ FK_DB_Table_DB_Index(character)
«PK»
+ PK_DB_Table(character)
«index»
+ IXFK_DB_Table_DB_Index(character)
+PK_CompileUnit
1
(chCompileUnitID = chID)
«FK»
+FK_CU_Block_CompileUnit
1..*
+PK_DB_Table
1
(chTableID = ch_ID)
«FK»
+FK_CU_Buffer_DB_Table
0..*
CU_Block
«column»
*PK chID: character
FK chCompileUnitID: character
chBlockType: character
inLineStart: integer
inLineEnd: integer
lgIsTransaction: logical
chBlockLabel: character
«FK»
+ FK_CU_Block_CompileUnit(character)
«PK»
+ PK_CU_Block(character)
«unique»
+ UQ_CU_Block_chID(character)
«index»
+ IXFK_CU_Block_CompileUnit(character)
CU_Buffer
«column»
*PK chID: character
FK chBlockID: character
lgIsDBTable: logical
chBufferName: character
FK chTableID: character
«FK»
+ FK_CU_Buffer_CU_Block(character)
+ FK_CU_Buffer_DB_Table(character)
«PK»
+ PK_CU_Buffer(character)
«unique»
+ UQ_CU_Buffer_chID(character)
«index»
+ IXFK_CU_Buffer_CU_Block(character)
+ IXFK_CU_Buffer_DB_Table(character)
+PK_CU_Block
+FK_CU_Buffer_CU_Block
1
(chBlockID = chID)
«FK»
0..*
+PK_CU_Block
1
(chBlockID = chID)
«FK»
0..*
+FK_CU_Frame_CU_Block
CU_Frame
«column»
*PK chID: character
FK chBlockID: character
chFrameName: character
«FK»
+ FK_CU_Frame_CU_Block(character)
«PK»
+ PK_CU_Frame(character)
«unique»
+ UQ_CU_Frame_chID(character)
«index»
+ IXFK_CU_Frame_CU_Block(character)

What is Captured in Current Tool?

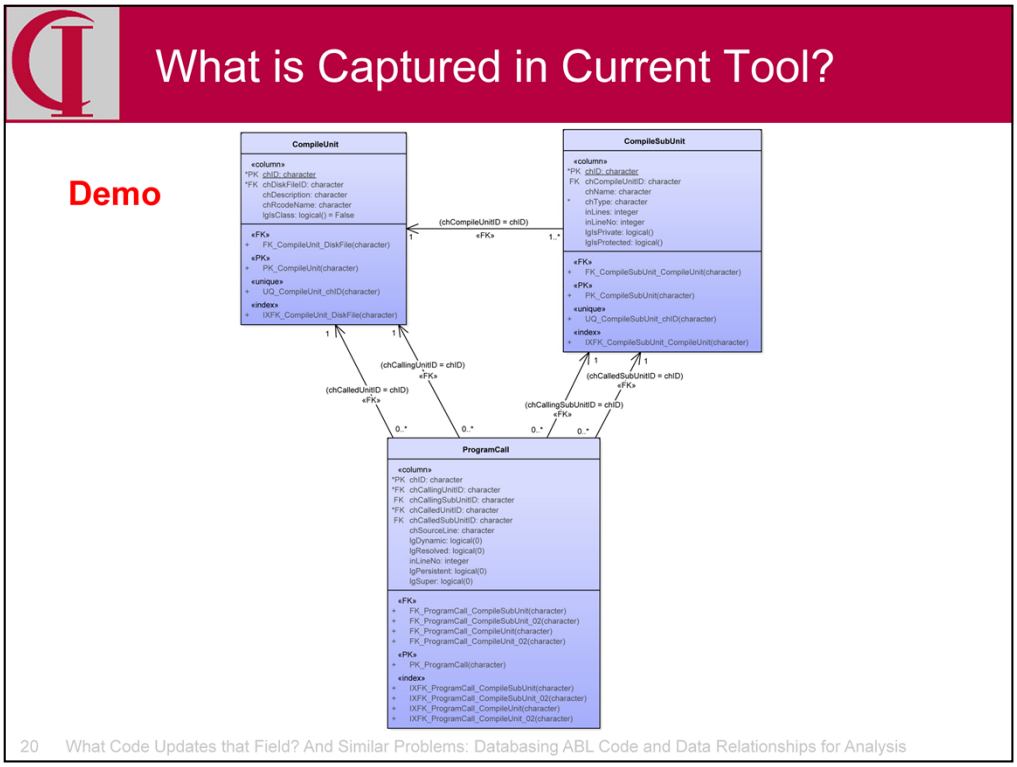
Run Links

- All Compile Unit to Compile Unit RUN and METHOD invocation links.
- Dynamic calls will require separate resolution.
- Plan to expand to Compile SubUnit links, but that also requires separate resolution or an alternate tool because the information is not in COMPILE XREF.

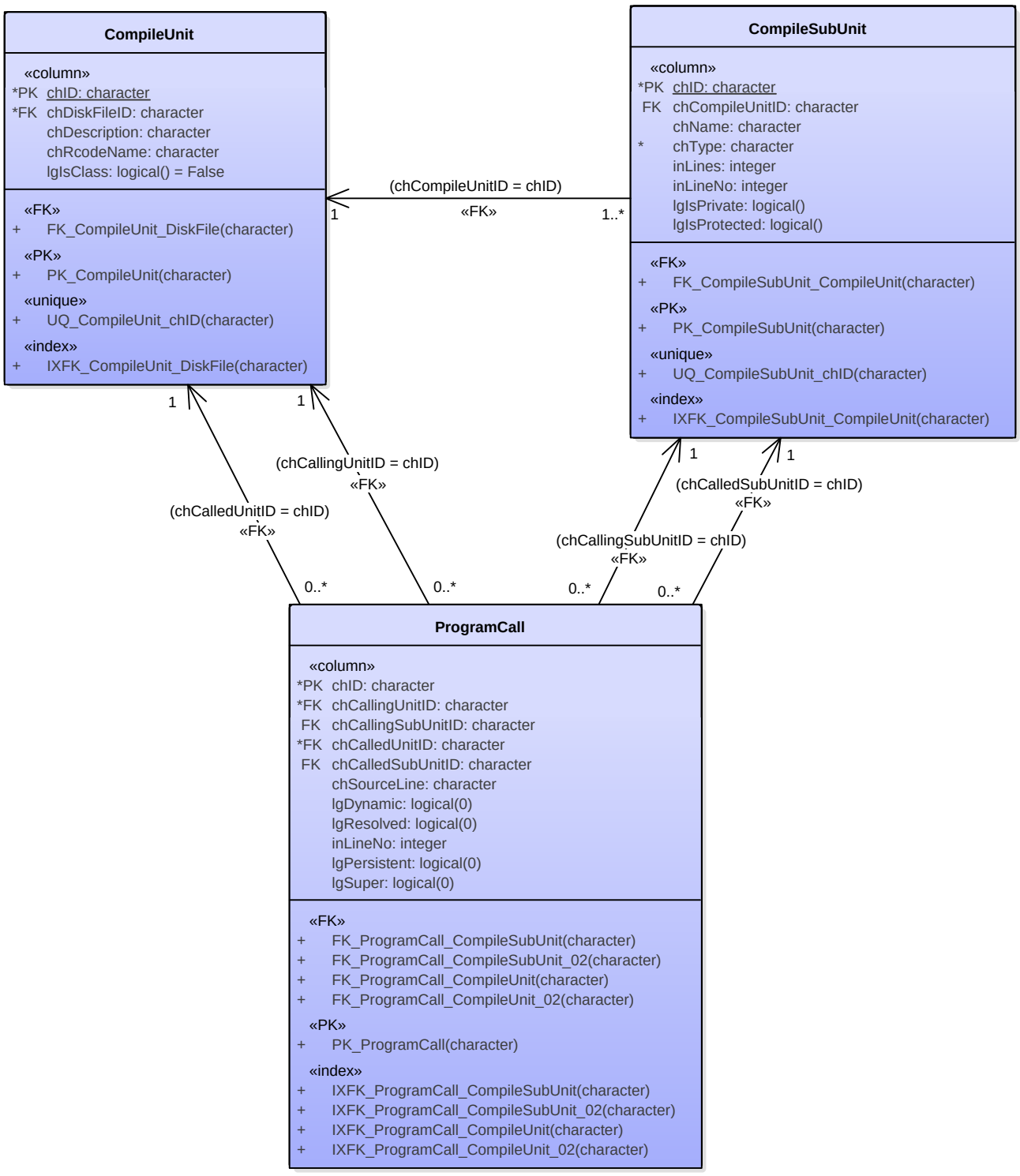


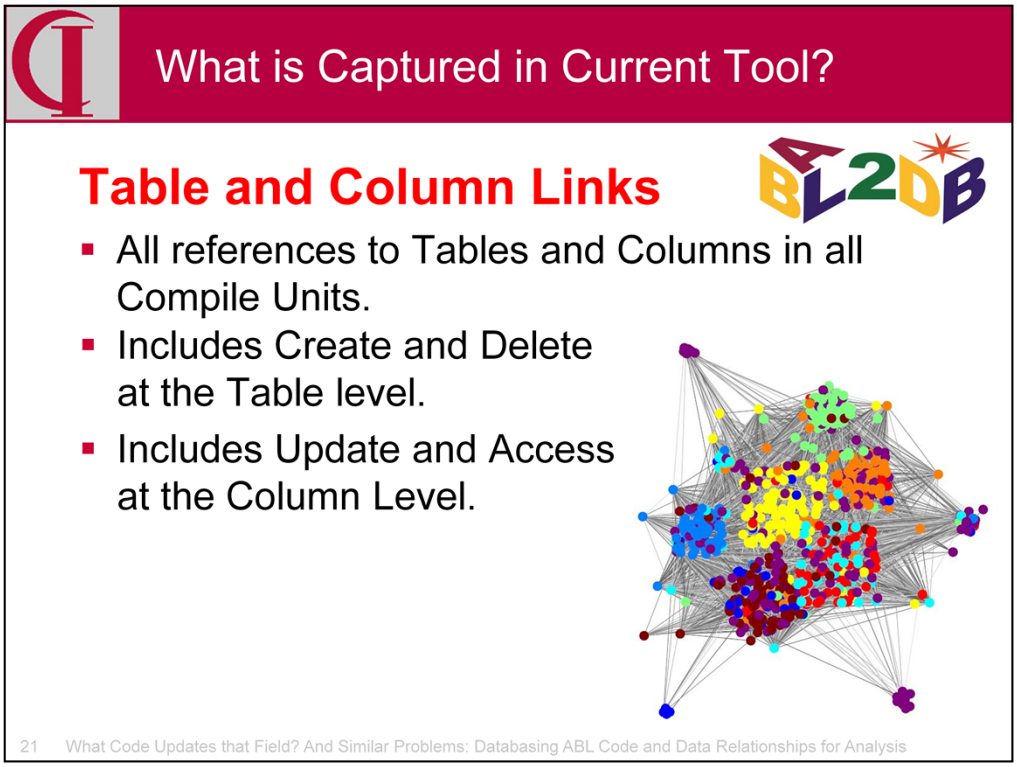
19 What Code Updates that Field? And Similar Problems: Databasing ABL Code and Data Relationships for Analysis

Now that we have identified all the static components, it is time to connect them. The first step is all the RUN relationships between pieces of the source code. The initial pass identifies all static Compile Unit to Compile Unit run connections. Not all can be resolved at this point because some are dynamic such as RUN VALUE() or RUN x IN handle. These dynamic calls will be resolved later. The data structure also provides for links between Compile SubUnits, i.e., a call located in an Internal Procedure, Method, or Function or a call to an Internal Procedure, Method, or Function from another Compile Unit, e.g., such as Method calls on classes or Internal Procedure calls to persistent procedures. Unfortunately, the information needed to make these connections is not in the COMPILE XREF data, so these connections will have to be identified by subsequent analysis or adding a new tool.

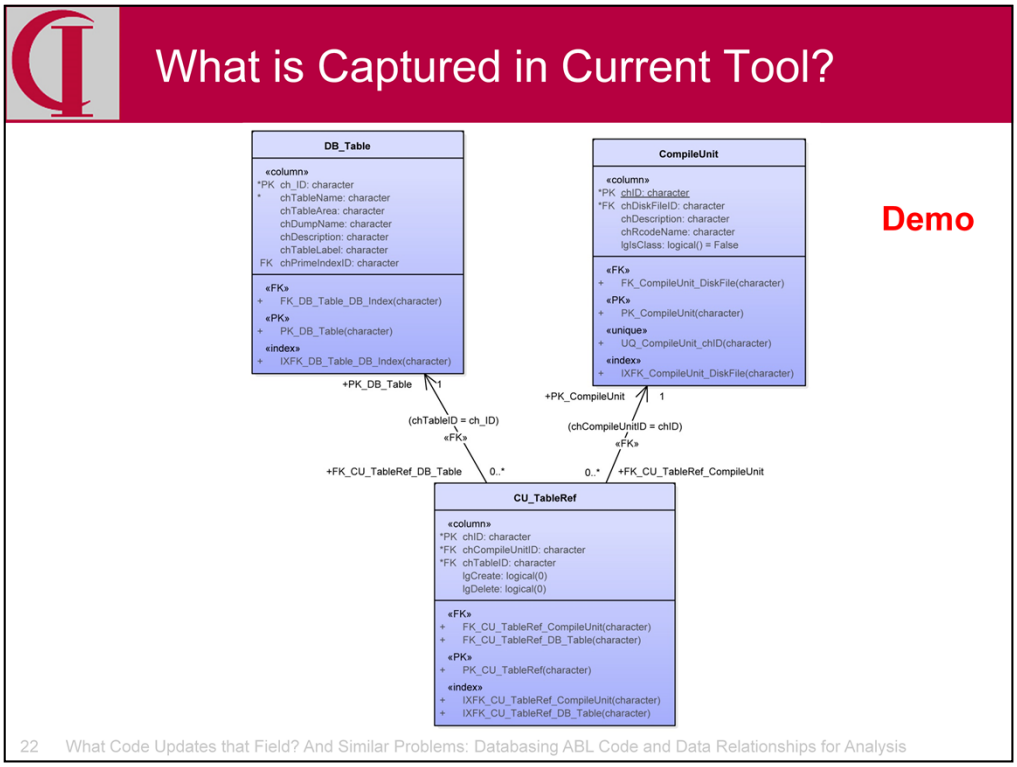


DEMO: In the upper right we have the usual Compile Unit and to the right Compile SubUnits. These get connected by Program Call links which associate caller with called. There is a copy of the call for aid in later resolution of dynamic calls and flags for Dynamic, Resolved, Persistent, and Super.

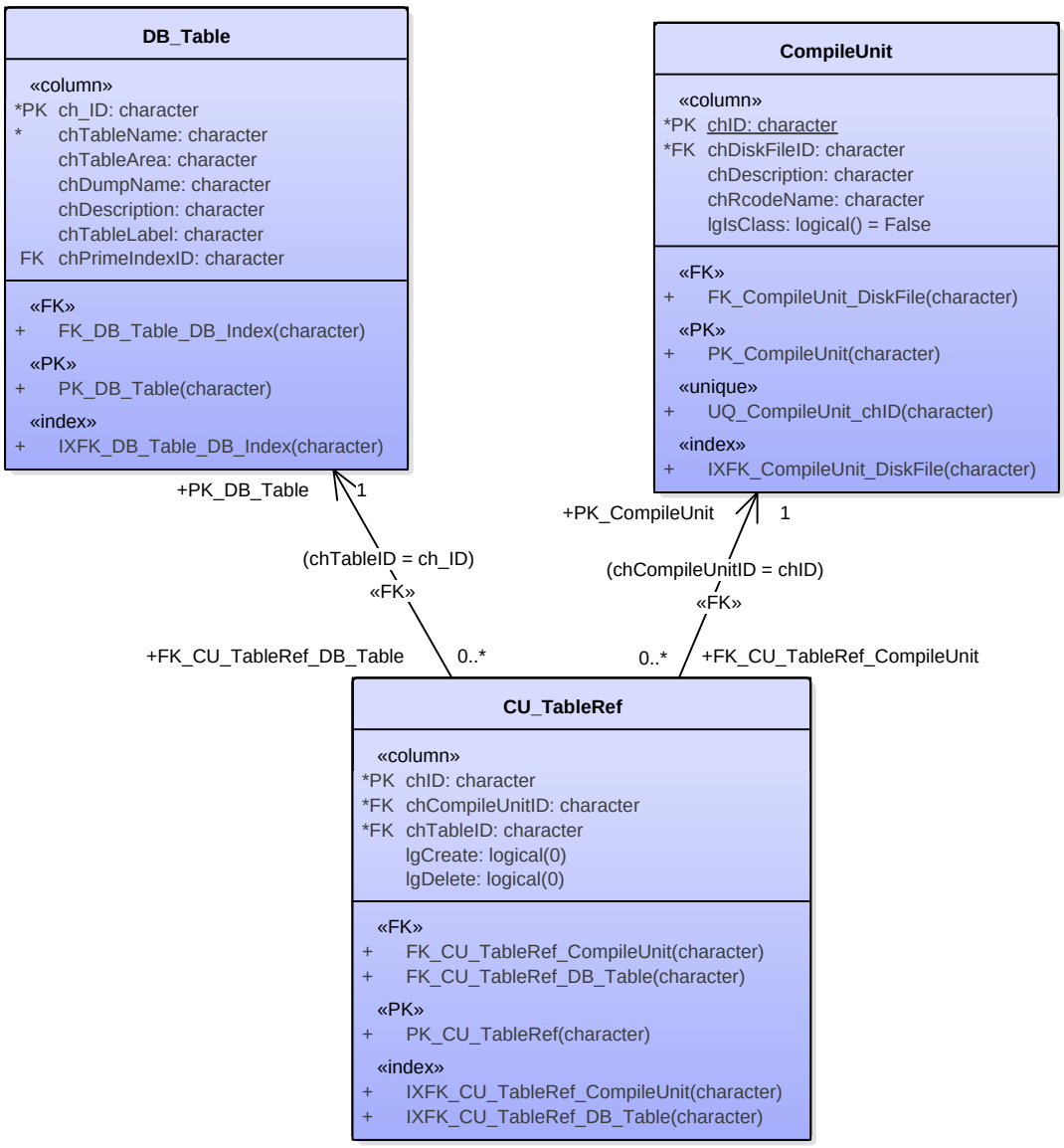

CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical() = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
CompileSubUnit
«column»
*PK chID: character
FK chCompileUnitID: character
chName: character
* chType: character
inLines: integer
inLineNo: integer
lgIsPrivate: logical()
lgIsProtected: logical()
«FK»
+ FK_CompileSubUnit_CompileUnit(character)
«PK»
+ PK_CompileSubUnit(character)
«unique»
+ UQ_CompileSubUnit_chID(character)
«index»
+ IXFK_CompileSubUnit_CompileUnit(character)
(chCompileUnitID = chID)
«FK»
1
1..*
1
1
(chCallingUnitID = chID)
«FK»
(chCalledUnitID = chID)
«FK»
1
1
(chCalledSubUnitID = chID)
«FK»
(chCallingSubUnitID = chID)
«FK»
0..*
0..*
0..*
0..*
ProgramCall
«column»
*PK chID: character
*FK chCallingUnitID: character
FK chCallingSubUnitID: character
*FK chCalledUnitID: character
FK chCalledSubUnitID: character
chSourceLine: character
lgDynamic: logical(0)
lgResolved: logical(0)
inLineNo: integer
lgPersistent: logical(0)
lgSuper: logical(0)
«FK»
+ FK_ProgramCall_CompileSubUnit(character)
+ FK_ProgramCall_CompileSubUnit_02(character)
+ FK_ProgramCall_CompileUnit(character)
+ FK_ProgramCall_CompileUnit_02(character)
«PK»
+ PK_ProgramCall(character)
«index»
+ IXFK_ProgramCall_CompileSubUnit(character)
+ IXFK_ProgramCall_CompileSubUnit_02(character)
+ IXFK_ProgramCall_CompileUnit(character)
+ IXFK_ProgramCall_CompileUnit_02(character)



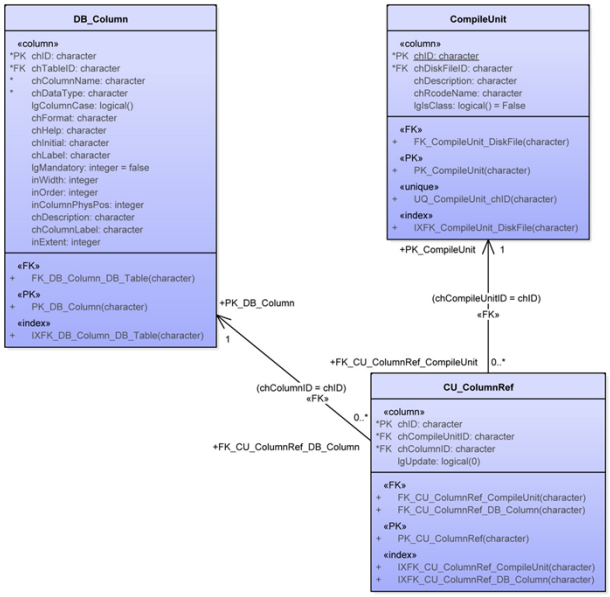
Next, we have links between the code and the data. This includes all references to tables and columns in all compile units. At the table level, it also includes flags for when the record is created or deleted. At the column level it has a flag for update, i.e., the value of the column is changed in that unit. Any column not changed is merely accessed. Any column in the table, but which is not referenced, has no a link.



DEMO: First we will look at the Table links. In the upper right we have the Compile Unit and in the upper left the Table. Connecting these is an association table which connects the two and has the flags for create and delete.

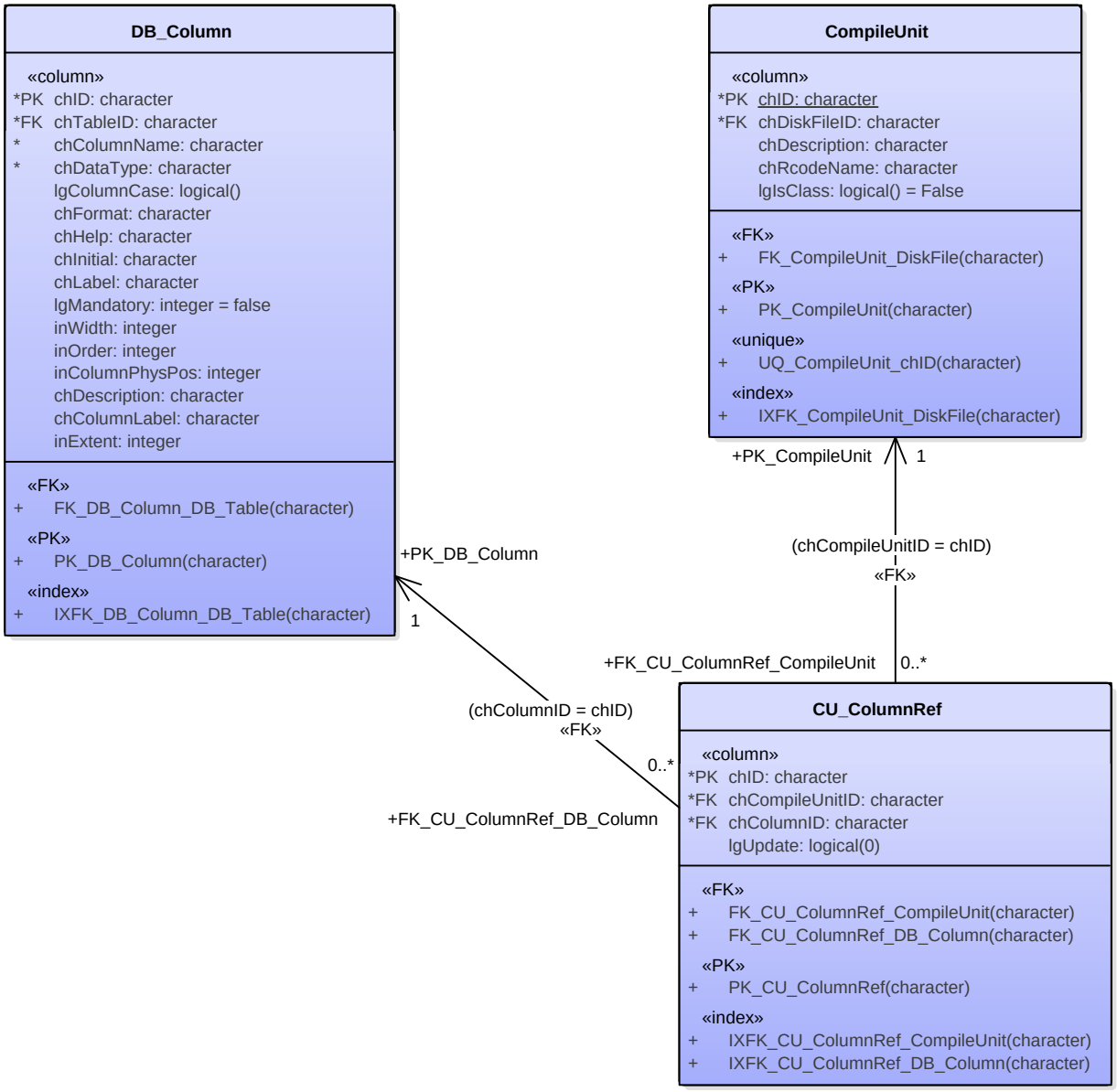

DB_Table
«column»
*PK ch_ID: character
* chTableName: character
chTableArea: character
chDumpName: character
chDescription: character
chTableLabel: character
FK chPrimeIndexID: character
«FK»
+ FK_DB_Table_DB_Index(character)
«PK»
+ PK_DB_Table(character)
«index»
+ IXFK_DB_Table_DB_Index(character)
CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical() = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
+PK_DB_Table
1
(chTableID = ch_ID)
«FK»
+PK_CompileUnit
1
(chCompileUnitID = chID)
«FK»
+FK_CU_TableRef_DB_Table
0..*
0..*
+FK_CU_TableRef_CompileUnit
CU_TableRef
«column»
*PK chID: character
*FK chCompileUnitID: character
*FK chTableID: character
lgCreate: logical(0)
lgDelete: logical(0)
«FK»
+ FK_CU_TableRef_CompileUnit(character)
+ FK_CU_TableRef_DB_Table(character)
«PK»
+ PK_CU_TableRef(character)
«index»
+ IXFK_CU_TableRef_CompileUnit(character)
+ IXFK_CU_TableRef_DB_Table(character)

What is Captured in Current Tool?



Demo

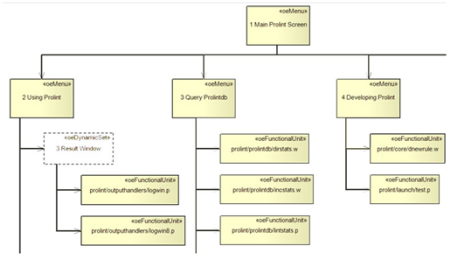
DEMO: Next we have the Compile Unit in the upper right and the Column in the upper left. Connecting these is an association table which contains the flag for Update.


DB_Column
«column»
*PK chID: character
*FK chTableID: character
* chColumnName: character
* chDataType: character
lgColumnCase: logical()
chFormat: character
chHelp: character
chInitial: character
chLabel: character
lgMandatory: integer = false
inWidth: integer
inOrder: integer
inColumnPhysPos: integer
chDescription: character
chColumnLabel: character
inExtent: integer
«FK»
+ FK_DB_Column_DB_Table(character)
«PK»
+ PK_DB_Column(character)
«index»
+ IXFK_DB_Column_DB_Table(character)
CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical() = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
+PK_CompileUnit
1
(chCompileUnitID = chID)
«FK»
+PK_DB_Column
1
+FK_CU_ColumnRef_CompileUnit
0..*
(chColumnID = chID)
«FK»
0..*
+FK_CU_ColumnRef_DB_Column
CU_ColumnRef
«column»
*PK chID: character
*FK chCompileUnitID: character
*FK chColumnID: character
lgUpdate: logical(0)
«FK»
+ FK_CU_ColumnRef_CompileUnit(character)
+ FK_CU_ColumnRef_DB_Column(character)
«PK»
+ PK_CU_ColumnRef(character)
«index»
+ IXFK_CU_ColumnRef_CompileUnit(character)
+ IXFK_CU_ColumnRef_DB_Column(character)

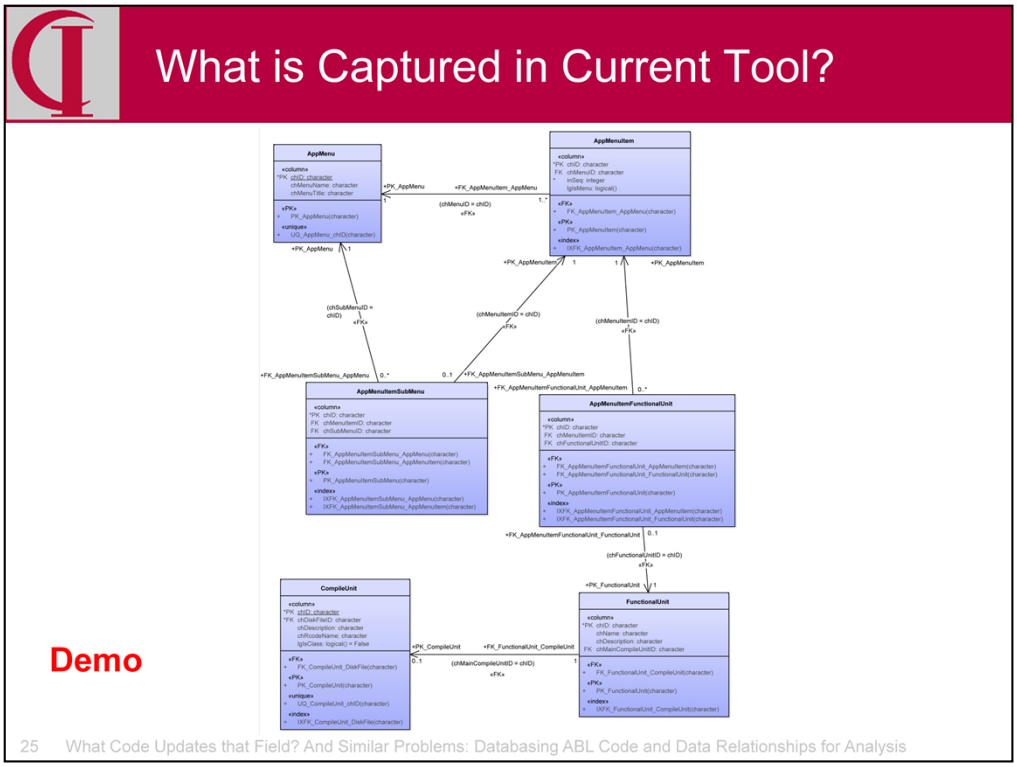
What is Captured in Current Tool?

Menus and Functional Units

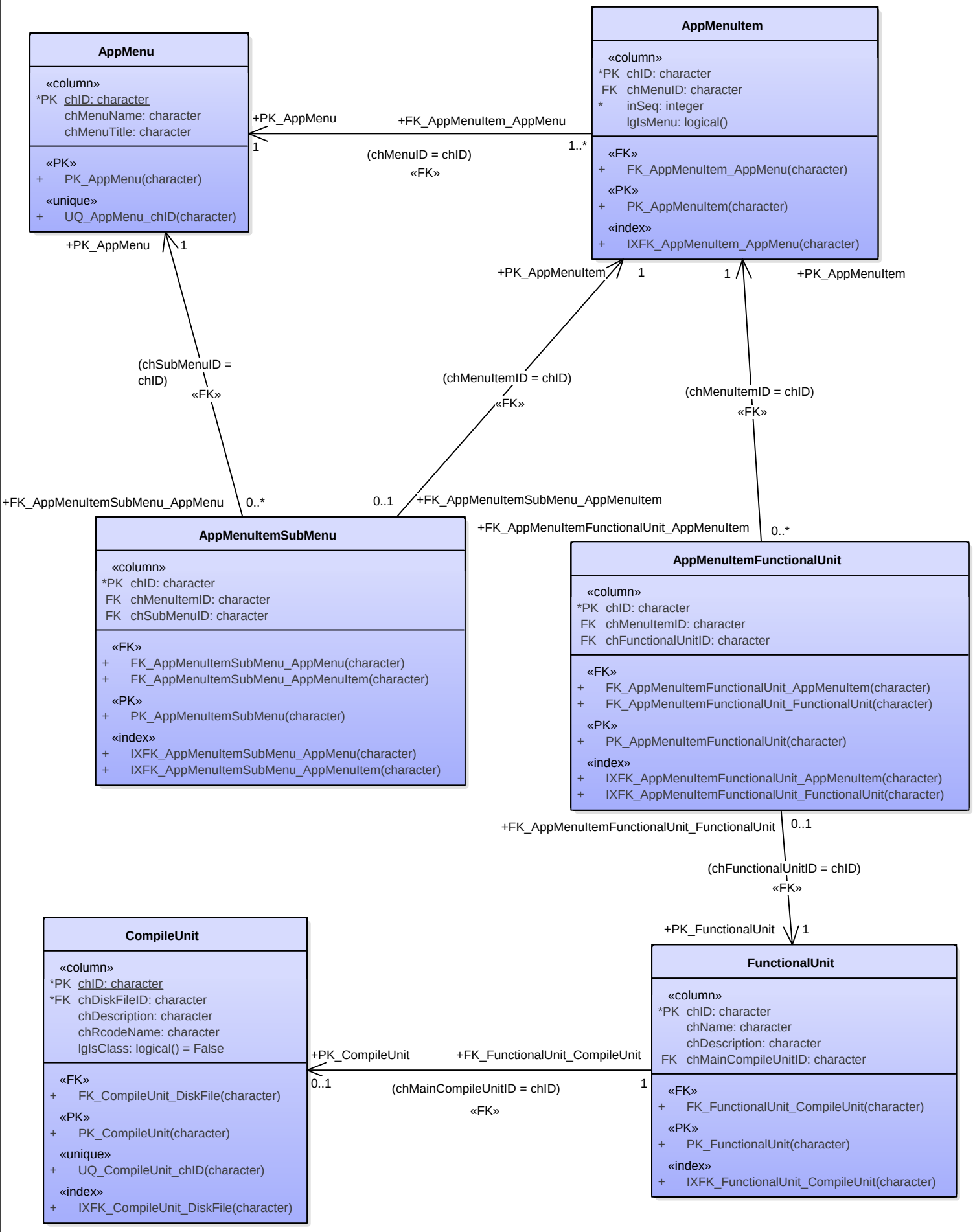
- To relate code to actual use, it is useful to know the structure by which the code is used. To provide this we have a structure for the application menu.
- Each menu item calls a specific Compile Unit and all code reachable by that Compile Unit is known as a Functional Unit.

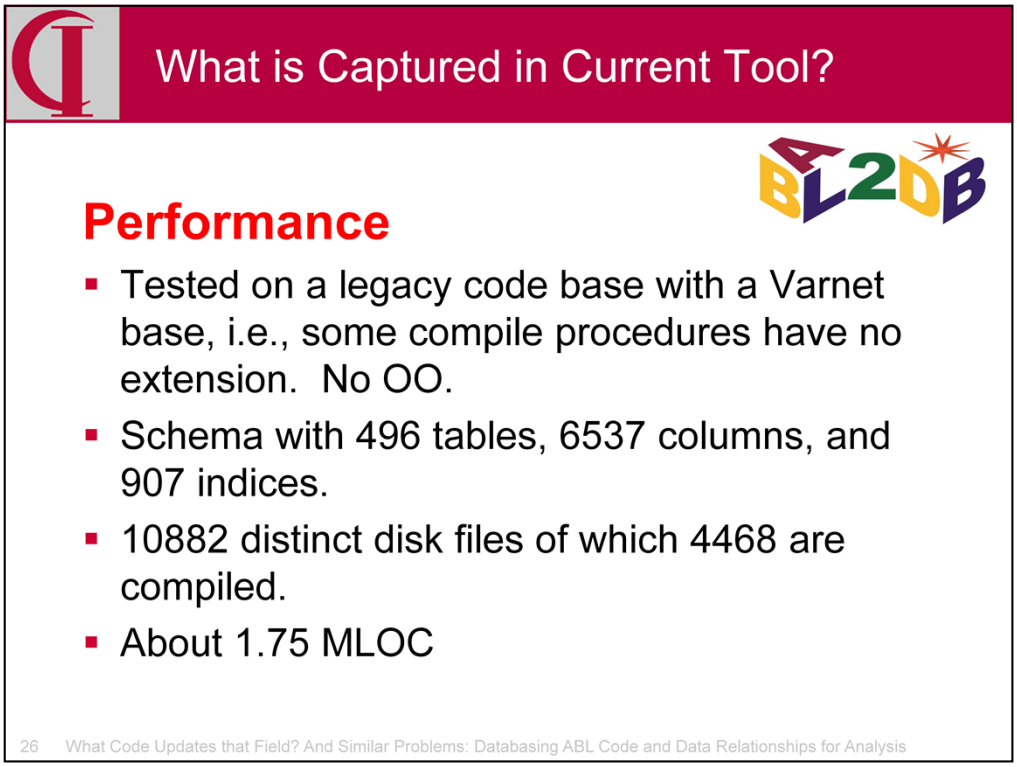


Finally, for the current version of ABL2DB, there are two additional stages. One loads descriptions for each disk file. Many code bases have some kind of standard description embedded in the source file and a tool is provided for extracting these according to the standards of the specific code base. The other is the loading of Menus and Functional Units. Menus provide us with a structure for the application as seem by the user, each selection of which runs a particular Compile Unit. From that Compile Unit, there is a body of code which can be reached including all sub programs, persistent procedures, and include files. We call this a Functional Unit. Functional Units may, of course, overlap.



DEMO: Finally, we have the AppMenu in the upper left, one entry per menu. To its right is the AppMenuItem, i.e., one specific item on that menu. That item can be either another menu, in which case an association is made via the AppMenuItemSubMenu table, or it can be an executable item, in which case association is made via the AppMenuItemFunctionalUnit table to the corresponding Functional Unit. The Functional Unit is in turn linked to the main Compile Unit which it runs.


AppMenu
«column»
*PK chID: character
chMenuName: character
chMenuTitle: character
«PK»
+ PK_AppMenu(character)
«unique»
+ UQ_AppMenu_chID(character)
AppMenuItem
«column»
*PK chID: character
FK chMenuID: character
* inSeq: integer
lgIsMenu: logical()
«FK»
+ FK_AppMenuItem_AppMenu(character)
«PK»
+ PK_AppMenuItem(character)
«index»
+ IXFK_AppMenuItem_AppMenu(character)
+PK_AppMenu
1
+FK_AppMenuItem_AppMenu
1..*
(chMenuID = chID)
«FK»
+PK_AppMenu
1
+PK_AppMenuItem
1
1
+PK_AppMenuItem
(chSubMenuID = chID)
«FK»
(chMenuItemID = chID)
«FK»
(chMenuItemID = chID)
«FK»
+FK_AppMenuItemSubMenu_AppMenu
0..*
0..1
+FK_AppMenuItemSubMenu_AppMenuItem
AppMenuItemSubMenu
«column»
*PK chID: character
FK chMenuItemID: character
FK chSubMenuID: character
«FK»
+ FK_AppMenuItemSubMenu_AppMenu(character)
+ FK_AppMenuItemSubMenu_AppMenuItem(character)
«PK»
+ PK_AppMenuItemSubMenu(character)
«index»
+ IXFK_AppMenuItemSubMenu_AppMenu(character)
+ IXFK_AppMenuItemSubMenu_AppMenuItem(character)
+FK_AppMenuItemFunctionalUnit_AppMenuItem
0..*
AppMenuItemFunctionalUnit
«column»
*PK chID: character
FK chMenuItemID: character
FK chFunctionalUnitID: character
«FK»
+ FK_AppMenuItemFunctionalUnit_AppMenuItem(character)
+ FK_AppMenuItemFunctionalUnit_FunctionalUnit(character)
«PK»
+ PK_AppMenuItemFunctionalUnit(character)
«index»
+ IXFK_AppMenuItemFunctionalUnit_AppMenuItem(character)
+ IXFK_AppMenuItemFunctionalUnit_FunctionalUnit(character)
+FK_AppMenuItemFunctionalUnit_FunctionalUnit
0..1
(chFunctionalUnitID = chID)
«FK»
+PK_FunctionalUnit
1
CompileUnit
«column»
*PK chID: character
*FK chDiskFileID: character
chDescription: character
chRcodeName: character
lgIsClass: logical() = False
«FK»
+ FK_CompileUnit_DiskFile(character)
«PK»
+ PK_CompileUnit(character)
«unique»
+ UQ_CompileUnit_chID(character)
«index»
+ IXFK_CompileUnit_DiskFile(character)
+PK_CompileUnit
0..1
+FK_FunctionalUnit_CompileUnit
1
(chMainCompileUnitID = chID)
«FK»
FunctionalUnit
«column»
*PK chID: character
chName: character
chDescription: character
FK chMainCompileUnitID: character
«FK»
+ FK_FunctionalUnit_CompileUnit(character)
«PK»
+ PK_FunctionalUnit(character)
«index»
+ IXFK_FunctionalUnit_CompileUnit(character)



The existing system was tested on my old ERP package called Integrity/Solutions. This application suite has a Varnet base and thus has some residual compile units with no extension. Its vintage predates OO, but it does have some super procedures and the like. The schema has 496 tables, 6537 columns, and 907 indices. This schema includes some metaschema tables since some of the I/S code refers to those tables. There are 10822 distinct disk files, 4468 of which are compilable. There is a fair bit of orphan junk in there which this tool could help identify and remove. Altogether, it is about 1.75 million lines of ABL code.

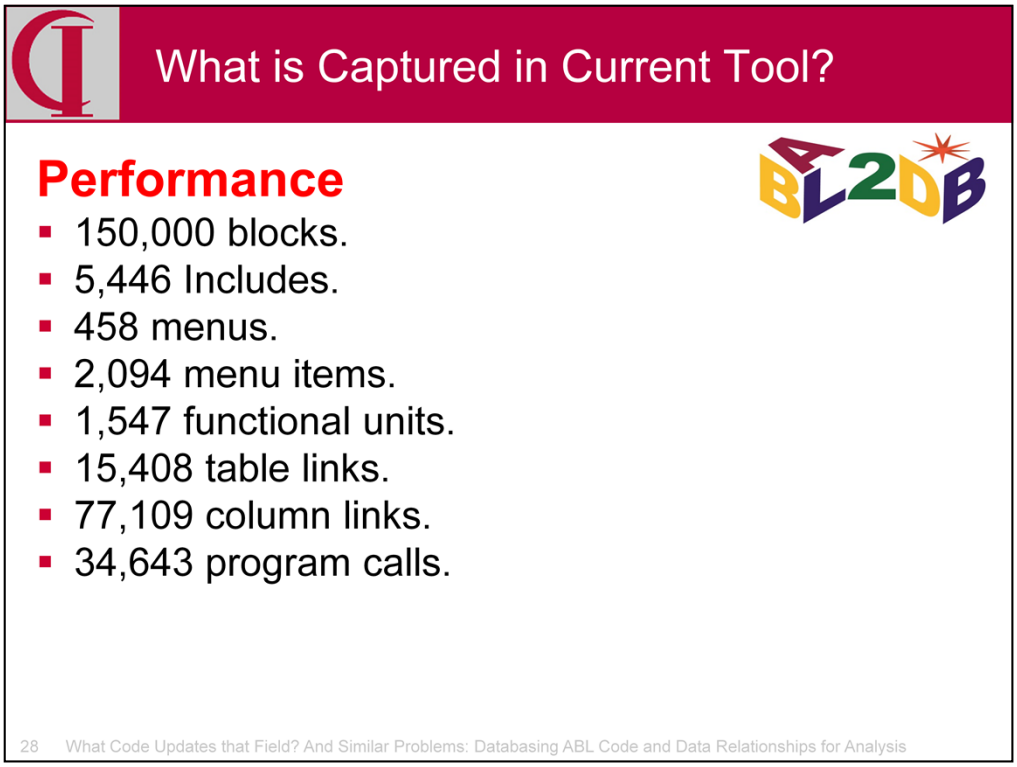
What is Captured in Current Tool?

Performance

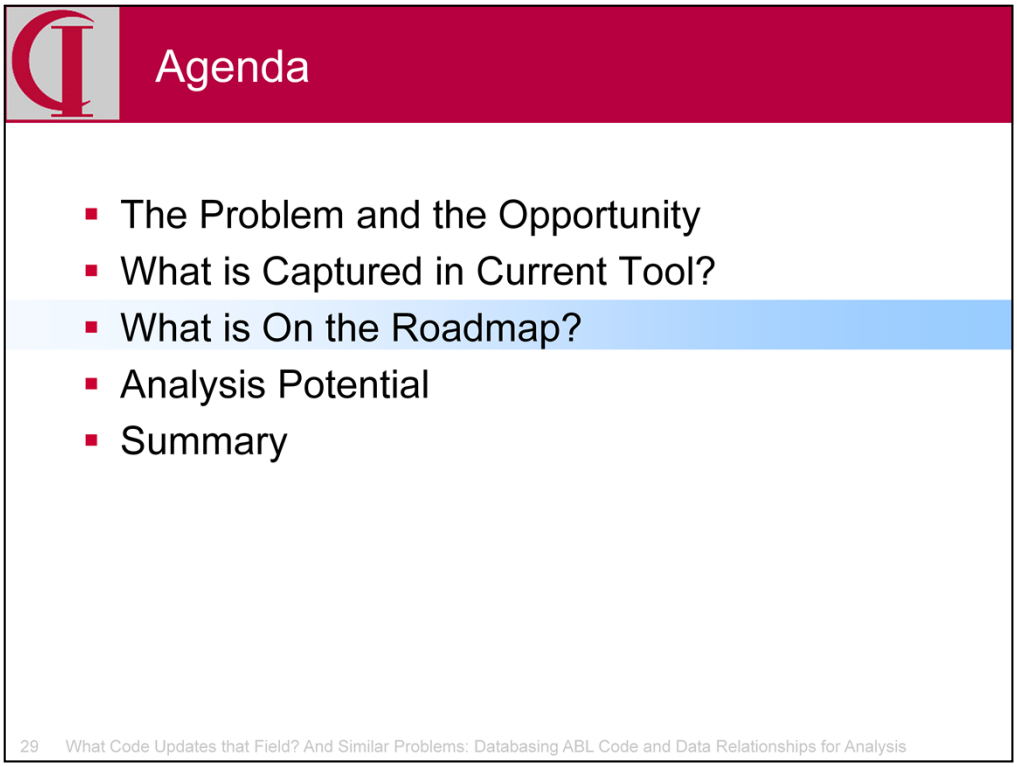
- 9 minutes for empirical compile.
- 10 seconds to load schema
- 20 seconds for scan of disk files.
- <20 minutes for compile sub units.
- 7 minutes for blocks, buffers, and frames.
- <20 minutes for run links.
- <20 minutes for table links.
- <20 seconds for descriptions.
- 3 seconds for menus and functional units.

TOTAL: 1 hour 10 minutes .

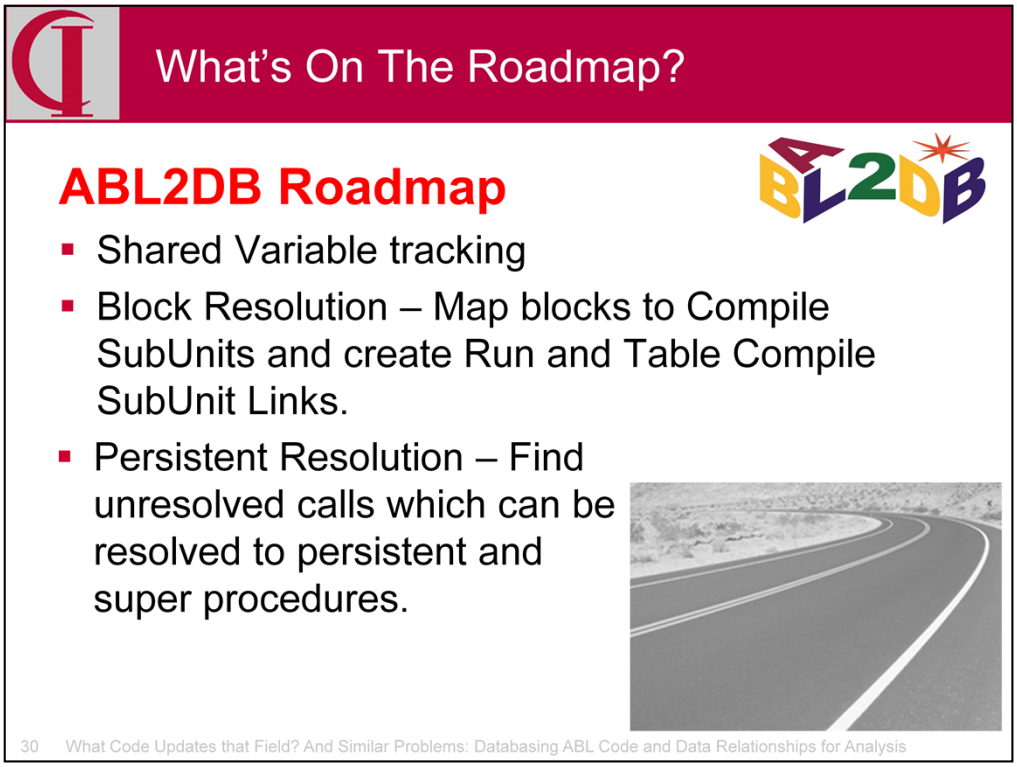
These are the times for each step on my desktop. On my laptop with an SSD disk, the times are slightly faster, but not dramatically different.



And, here are some of the counts of the kind of thing found during the analysis. The sheer volume makes it clear why no one can actually hold all of this in their head, no matter how long they have worked with the code.



That's what we have so far, so let's look ahead a bit.



The XREF file keeps track of Shared Variables and their use. While one knows they shouldn't be used any more, they can be an important part of understanding many legacy code bases.

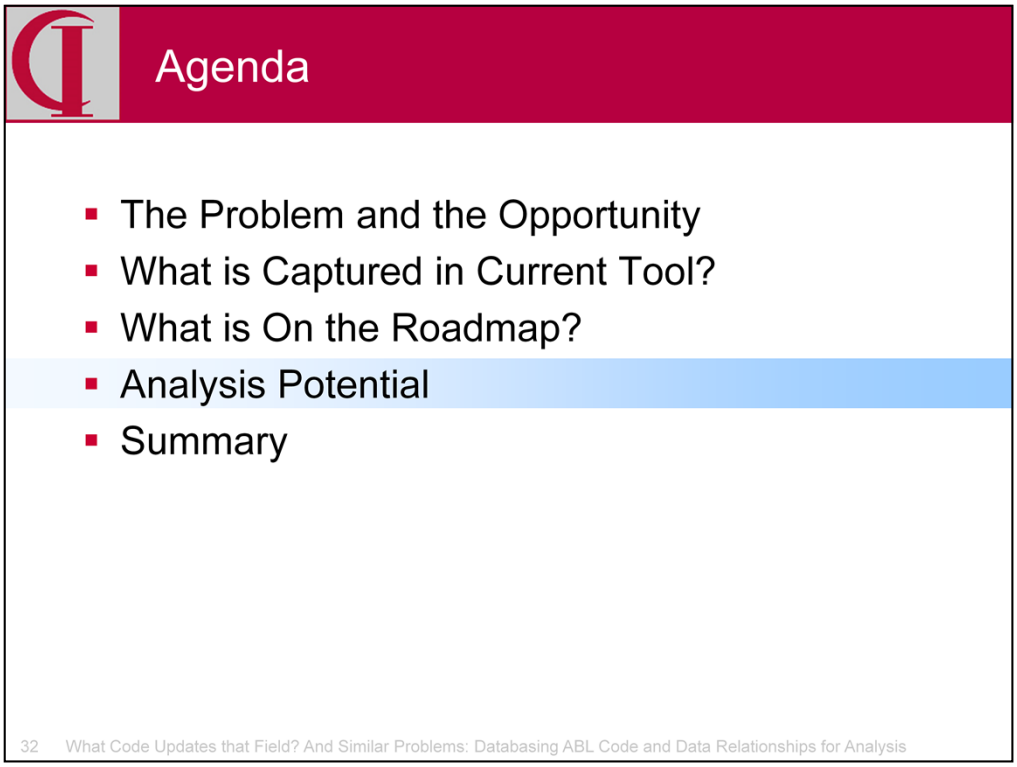
Block resolution will map blocks to compile sub units and create the Run and Table links to Compile SubUnits.

Persistent Resolution refers to a limited kind of Call Graph Analysis by identifying persistent procedures and super procedures which can be running and which might contain an unresolved reference. Some of this actually occurs in the existing version of ABL2DB, but it is not complete because of the nature of the information in the XREF file.

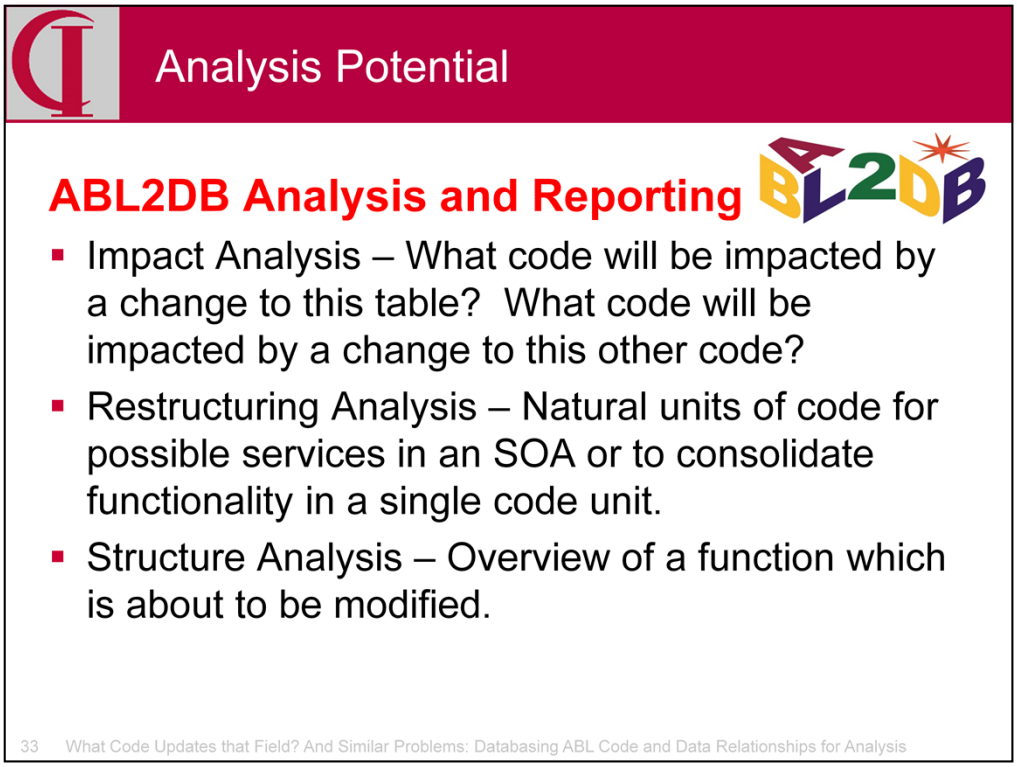
We will provide an Unresolved Dynamic Call Report for identifying those calls which cannot be automatically solved and a mechanism for manual reconciliation of those calls in a way which can be stored and reused for subsequent builds.

I am thinking about an Unused Code report. This would be very handy for cleaning up the unused older portions in my current sample code base.

There is lots more potential and I am sure many things I haven't thought of yet. In particular, I am exploring the possible use of other tools, including the Parser technology in the tool by Gilles Querret who is talking here at 1300 today.



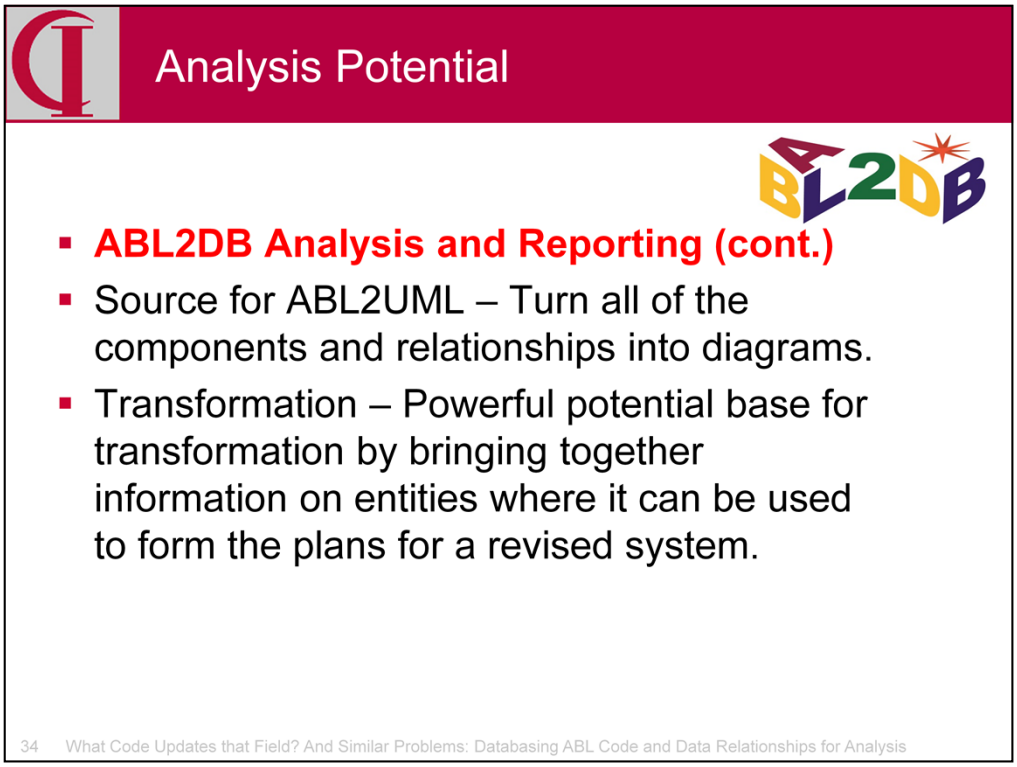
Finally, let's talk briefly about the potential for analysis.



Clearly, one of the prime uses for a code base under maintenance will be impact analysis, i.e., identifying what code or data is impacted by a possible change. While some of this category of question may be best addressed by a simple custom query, there are a couple of obvious simple reports that will fulfill many needs.

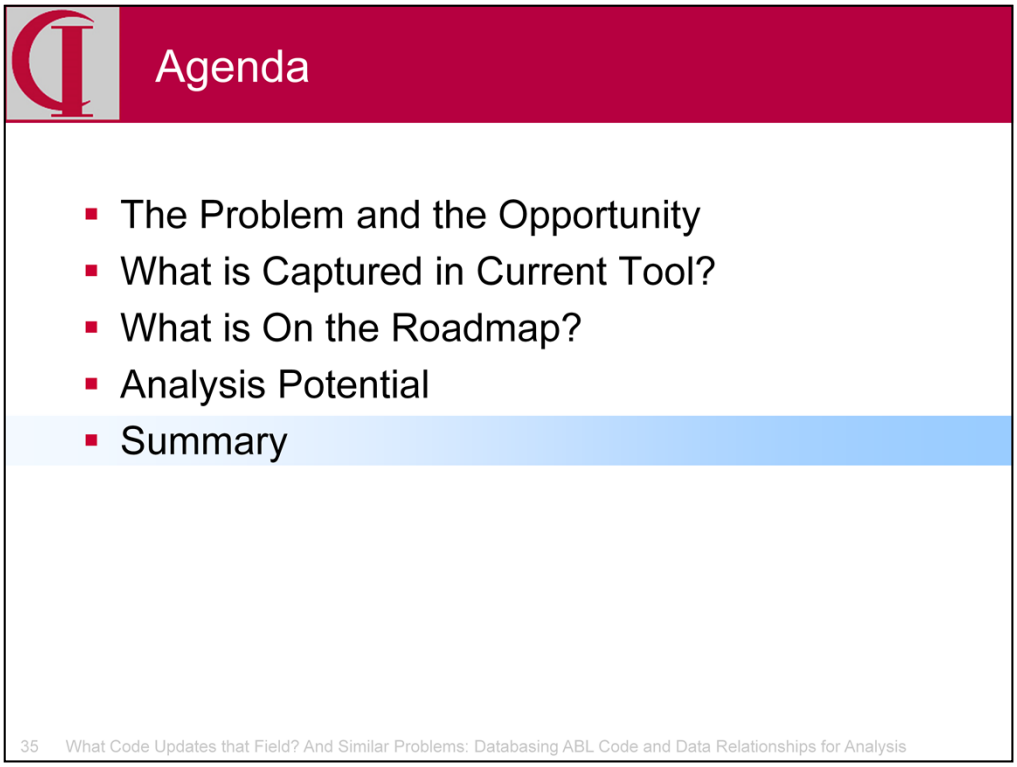
A second kind of analysis I have wanted to do for a long time is analysis of natural groups in the code. By this I mean identifying related clusters of code, excluding common service elements, which might be good candidates for a new package such as one might want to create if implementing services in a Service Oriented Architecture or simply refactoring code to avoid duplication and fractionation of related services.

And, of course, one of the simple, but very useful reports would simply to create a map of some Functional Unit which one was about to work on with indications of what parts of that unit impacted other code in other Functional Units.

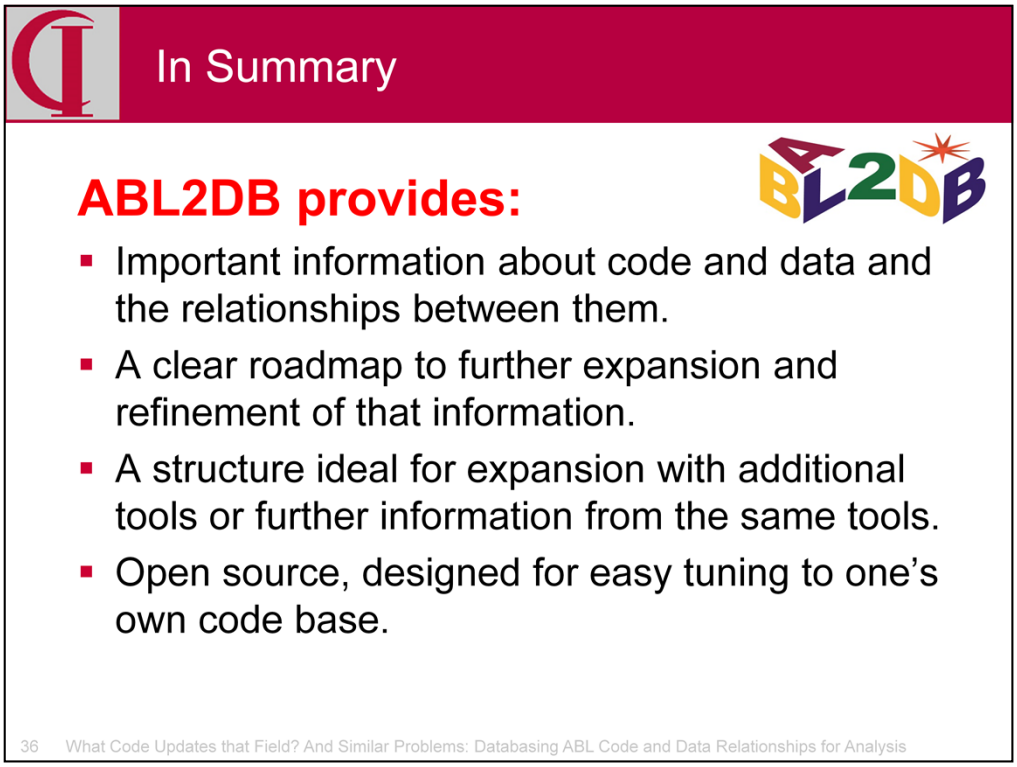


And, of course, I plan to use ABL2DB as a new source of input to an updated version of ABL2UML so that one can produce a UML Component diagram of the as-built application for a visual representation.

And, all of this can be used as a tool for transformation by bringing together information on the components of the existing system where they can be more easily analyzed for refactoring and restructuring for a new version of the application.



So, to summarize.

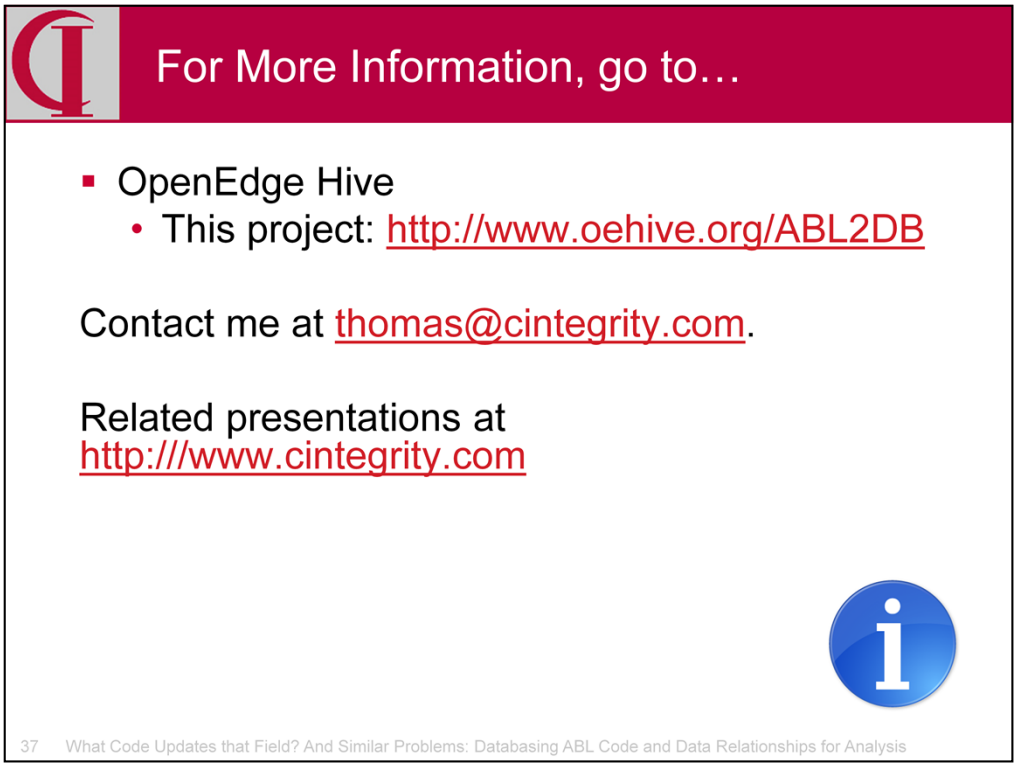


This initial version of ABL2DB covers a lot of information about the code and data in an application and their relationships. It is readily available and easy to use in building queries and reports.

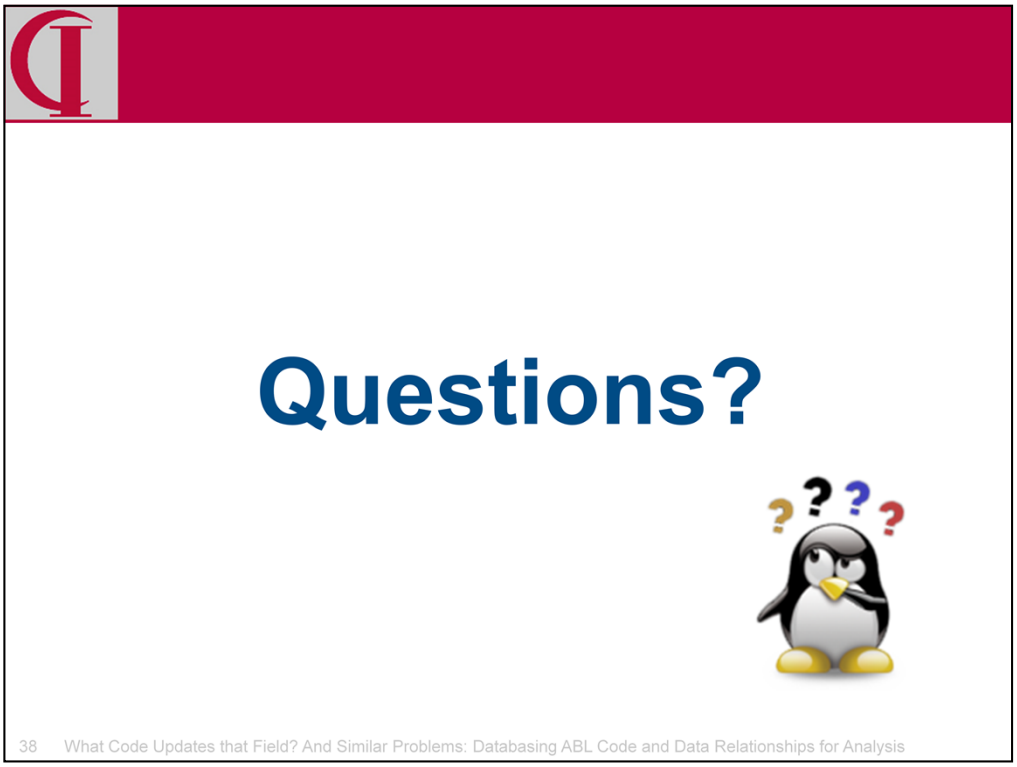
We have a clear initial roadmap for adding more information and functionality and refining the information we have.

The design of the system is very modular and easy to expand with new functions or to expand the information currently collected.

The tool is open sourced and has been designed to make the adjustments needed to fit it to the characteristics of the local code base.



Here are some links for more information.



Questions?

Thank You

Dr. Thomas Mercer-Hursh
thomas@cintegrity.com